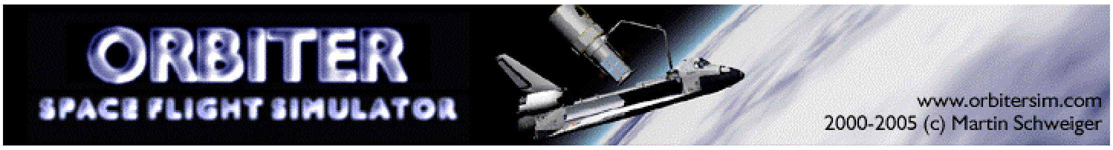
ORBITER
API Reference Manual

 16 January 2005

Orbiter home: www.medphys.ucl.ac.uk/~martins/orbit/orbit.html or www.orbitersim.com



Contents

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
- Multiplayer modules
- Custom calculation of planetary positions

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++, the use of other compilers may be possible, if they conform to the MS stack calling convention.)

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*	*SDK documentation*
Orbitersdk\include	
Orbitersdk.h	*The interface header file*
OrbiterAPI.h	*General interface functions*
VesselAPI.h	*Vessel interface*
Orbitersdk\lib	
Orbitersdk.lib	*The DLL auxiliary library*
Orbiter.lib	*The Orbiter API library*
Orbitersdk\tools*	*Tools for model and texture generation*
Orbitersdk\samples*	*Sample source code*

5 Compatibility issues

Orbiter will change its addon compatibility strategy beginning with the next release. In the future, each Orbiter release will run only addons which have been compiled with the SDK of that release. To migrate an addon to a new Orbiter release will therefore require a recompilation with the new SDK. This should help to keep addons up to date and reduce compatibilty problems. At the same time, this will allow me to purge obsolete API functions.

Latest release

- The latest release introduces a new more realistic atmospheric flight model. As a result, some aerodynamics-related vessel functions have become obsolete and are retained for backward compatibility only.

  SetWingAspect
  GetWingAspect
  SetWingEffectiveness
  GetWingEffectiveness
  SetLiftCoeffFunc

  The old atmospheric flight model will be dropped in a future version, so developers should migrate to the new model if they want to compile vessel addons against future API versions.

6 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 *opcOpenRenderViewport* 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 *opcOpenRenderViewport*, then this memory should be deallocated in *opcCloseRenderViewport*. 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.

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.

DialogTemplate
A trivial example demonstrating the use of Windows-style dialog boxes and custom functions in Orbiter.

Rcontrol
A more sophisticated dialog example. This plugin opens a dialog which allows to switch between spacecraft and remotely control the engines.

FlightData
Opens a dialog which allows to monitor vessel flight data.

CustomMFD
An example for an MFD plugin. This implements the Ascent profile MFD.

Deltaglider
Orbiter's standard implementation of the vessel module for the Delta-glider.

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, 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.

MESHHANDLE
A handle for object meshes.

SURFHANDLE
A handle for a bitmap surface. Surfaces are currently used for drawing instrument panel areas.

THRUSTER_HANDLE

Handle for (logical) thruster definitions.

THGROUP_HANDLE

Handle for thruster groups.

PROPELLANT_HANDLE

Handle for propellant resources.

NAVHANDLE

Handle for a navigation radio transmitter (VOR, ILS, IDS, XPDR)

VECTOR3

Double precision vector $\in R^3$

Synopsis:

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

MATRIX3

Double precision matrix $\in R^{3x3}$

Synopsis:

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

ELEMENTS

Keplerian orbital elements.

Synopsis:

```
typedef struct {
  double a;         semi-major axis [m]
  double e;         eccentricity
  double i;         inclination [rad]
  double theta;     longitude of ascending node [rad]
  double omegab;    longitude of periapsis [rad]
  double L;         mean longitude at epoch
} ELEMENTS;
```

ATMPARAM

Atmospheric parameters.

Synopsis:

```
typedef struct {
  double T;         temperature [K]
  double p;         pressure [Pa]
  double rho;       density [kg/m^3]
} ATMPARAM;
```

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;
```

ENGINETYPE

Enumerates thruster types

Synopsis:

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

EXHAUSTTYPE

Enumerates engine groups for exhaust rendering.

Synopsis:

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

PARTICLESTREAMSPEC

Defines the parameters of a particle stream.

Synopsis:

```
typedef struct {
  DWORD flags;
  double srcsize;
  double srcrate;
  double v0;
  double srcspread;
  double lifetime;
  double growthrate;
  double atmslowdown;
  enum LTYPE { EMISSIVE, DIFFUSE } ltype;
  enum LEVELMAP { LVL_FLAT, LVL_LIN, LVL_SQRT,
     LVL_PLIN, LVL_PSQRT } levelmap;
  double lmin, lmax;
  enum ATMSMAP { ATM_FLAT, ATM_PLIN } atmsmap;
  double amin, amax;
  SURFHANDLE tex;
} PARTICLESTREAMSPEC;
```

flags	currently not used
srcsize	particle size at creation [m]
srcrate	average particle generation rate [Hz]
v0	average particle emission velocity [m/s]
srcspread	emission velocity distribution randomisation
lifetime	average particle lifetime [s]
growthrate	particle growth rate [m/s]
atmslowdown	deceleration rate β in atmosphere, defined as $v = v_0\ e^{-\beta dt}$
ltype	lighting type (EMISSIVE or DIFFUSE)

levelmap	mapping between level parameter and particle opacity.
lmin, lmax	minimum and maximum levels for alpha mapping.
atmsmap	mapping between atmospheric parameters and particle opacity.
amin, amax	minimum and maximum atmospheric values for alpha mapping.

See the Programmer's Guide for more details on these parameters.

VESSELSTATUS

Defines vessel status parameters at a given time. This is version 1 of the vessel status interface. It is retained for backward compatibility, but new modules should use VESSELSTATUS2 instead to exploit the latest vessel capabilities such as individual thruster and propellant resource settings.

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;
  VECTOR3 vdata[10];
  double fdata[10];
  DWORD flag[10]
} VESSELSTATUS;
```

`rpos`	position relative to reference body in ecliptic frame
`rvel`	velocity relative to reference body in ecliptic frame
`vrot`	rotation velocity about principal axes in ecliptic frame
`arot`	vessel orientation against ecliptic frame (see notes)
`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
`vdata`	vector buffer for future extensions. Currently used: vdata[0] contains landing parameters if status==1: vdata[0].x = longitude [rad], vdata[0].y = latitude [rad] of landing site, vdata[0].z = orientation of vessel [rad].
`fdata`	Not currently used.
`flag[0]&1`	0: ignore `eng_main` and `eng_hovr` entries, do not change thruster settings 1: set `THGROUP_MAIN` and `THGROUP_RETRO` thruster groups from `eng_main`, and `THGROUP_HOVER` from `eng_hovr`.
`flag[0]&2`	0: ignore fuel entry, do not change fuel levels 1: set fuel level of first propellant resource from `fuel`.
`flag[1]-flag[9]`	Not currently used.

VESSELSTATUS2

Version 2 of the vessel status interface. This interface has been introduced in post-020419 versions.

Synopsis:

```
typedef struct {
  DWORD version;
  DWORD flag;
  OBJHANDLE rbody;
  OBJHANDLE base;
  int port;
  int status;
  VECTOR3 rpos;
  VECTOR3 rvel;
  VECTOR3 vrot;
  VECTOR3 arot;
  double surf_lng;
  double surf_lat;
  double surf_hdg;
  DWORD nfuel;
  struct FUELSPEC {
    DWORD idx;
    double level;
  } *fuel;
  DWORD nthruster;
  struct THRUSTSPEC {
    DWORD idx;
    double level;
  } *thruster;
  DWORD ndockinfo;
  struct DOCKINFOSPEC {
    DWORD idx;
    DWORD ridx;
    OBJHANDLE rvessel;
  } *dockinfo;
  DWORD xpdr;
} VESSELSTATUS2;
```

Parameters:

`version`	interface version (2)
`flag`	bitflags (see below)
`rbody`	handle of reference body
`base`	handle of docking or landing target
`port`	designated docking or landing port
`status`	0=active, 1=landed (inactive)
`rpos`	position relative to reference body (rbody) in ecliptic frame
`rvel`	velocity relative to reference body in ecliptic frame
`vrot`	rotation velocity about principal axes in ecliptic frame
`arot`	vessel orientation against ecliptic frame
`surf_lng`	longitude: vessel position in equatorial coordinates of rbody [rad]
`surf_lat`	latitude: vessel position in equatorial coordinates of rbody [rad]
`surf_hdg`	heading: vessel orientation on the ground
`nfuel`	number of entries in the `fuel` list
`fuel`	propellant resource list
`fuel[i].idx`	propellant resource index (0 ≤ i < nfuel)
`fuel[i].level`	propellant resource level [0..1]
`nthruster`	number of entries in the `thruster` list
`thruster`	thruster definition list
`thruster[i].idx`	thruster index (0 ≤ i < nfuel)
`thruster[i].level`	thruster level [0..1]
`ndockinfo`	number of entries in the dockinfo list
`dockinfo[i].idx`	dock index (0 ≤ i < ndockinfo)

`dockinfo[i].ridx`	dock index of docked vessel
`dockinfo[i].rvessel`	handle of docked vessel
`xpdr`	transponder setting (in steps of 0.05kHz from 108.00kHz)

flag

The meaning of the bitflags in `flag` depends on whether the VESSELSTATUS2 structure is used to get (GetStatus) or set (SetStatus) a vessel status. The following flags are currently defined:

- `VS_FUELRESET`
  Get – not used
  Set – reset all fuel levels to zero, independent of the `fuel` list.
- `VS_FUELLIST`
  Get – request a list of current fuel levels in `fuel`. The module is responsible for deleting the list after use.
  Set – set fuel levels for all resources listed in `fuel`.
- `VS_THRUSTRESET`
  Get – not used
  Set – reset all thruster levels to zero, independent of the `thruster` list
- `VS_THRUSTLST`
  Get – request a list of current thrust levels in `thruster`. The module is responsible for deleting the list after use.
  Set – set thrust levels for all thrusters listed in `thruster`.
- `VS_DOCKINFOLIST`
  Get – request a docking port status list in `dockinfo`. The module is responsible for deleting the list after use.
  Set – initialise docking status for all docking ports in `dockinfo`.

Notes:

- The `version` specification is an *input* parameter for all function calls (including GetStatus) and must be set by the user to tell Orbiter which interface to use.
- `surf_lng`, `surf_lat` and `surf_hdg` are currently only defined if the vessel is landed (status=1)
- arot=(α,β,γ) contains angles of rotation [rad] around x,y,z axes in ecliptic frame to produce this rotation matrix **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

Navmode constants

`NAVMODE_KILLROT`	*engage attitude thrusters to kill rotation*
`NAVMODE_HLEVEL`	*engage attitude thrusters to keep level with horizon*
`NAVMODE_PROGRADE`	*engage attitude thrusters to turn prograde*
`NAVMODE_RETROGRADE`	*engage attitude thrusters to turn retrograde*
`NAVMODE_NORMAL`	*engage attitude thrusters to turn orbit-normal*
`NAVMODE_ANTINORMAL`	*engage attitude thrusters to turn orbit-antinormal*
`NAVMODE_HOLDALT`	*engage hover thrusters to maintain altitude*

HUD mode constants

```
HUD_NONE
HUD_ORBIT
HUD_SURFACE
HUD_DOCKING
```

MFD mode constants

```
MFD_NONE
MFD_ORBIT
MFD_SURFACE
MFD_MAP
MFD_HSI
MFD_LANDING
MFD_DOCKING
MFD_OPLANEALIGN
MFD_OSYNC
MFD_TRANSFER
MFD_USERTYPE
```

MFD identifier constants

```
MFD_LEFT
MFD_RIGHT
MFD_USER1
MFD_USER2
MFD_USER3
```

10 Vessel modules

Vessel modules are dynamic link libraries (DLL) which contain the code to manage a vessel class. Orbiter loads a vessel library if the class configuration file of a vessel loaded during the simulation contains a MODULE entry. Only one instance of the library is loaded for each vessel class, even if multiple vessels of that class are present in the simulation. However, the library callback functions are called for each vessel. This means that global and static variables should not be used for vessel-specific parameters, to avoid conflicts between vessels. Instead, all vessel-specific data should be stored in the derived *VESSEL* instance (see below).

In general, a vessel module will create an instance of a vessel class derived from the base *VESSEL* class (see Section 11) during the vessel instance initialisation (*ovcInit*). All further interaction will then be performed through this class instance, either by Orbiter invoking callback functions to notify the vessel of various events, or by the module setting and querying vessel parameters.

In previous versions of the API, Orbiter communicated with the vessel module via nonmember callback functions (*ovcXXX*). In the current version, these have been replaced by virtual *VESSEL2* member functions (*VESSEL2::clbkXXX*) which can be overloaded by the derived class to invoke non-default actions. The only nonmember callback functions that should still be used are the instance entry and exit points (*ovcInit* and *ovcExit*), to create and delete the *VESSEL* class instance.

Vessel modules should link the Orbiter API libraries (*orbiter.lib* and *orbitersdk.lib*). The main source file should contain the

```
#define ORBITER_MODULE
```

directive.

The following list contains the callback functions used by Orbiter to communicate with the module. Many of these have become obsolete with the latest API interface and may not be supported in future versions. Developers should migrate to *VESSEL2* member callback functions to ensure future compatibility.

For a sample vessel module implementation, see for example Orbitersdk\samples\DeltaGlider.

Vessel module nonmember callback functions

InitModule

This is the module entry point. It is called *once* when the module is loaded, even if multiple vessels of this class are present. It can be used for global (vessel instance-independent) initialisations such as GDI resource allocation.

Synopsis:

```
DLLCLBK void InitModule (HINSTANCE hModule)
```

Parameters:

hModule DLL instance handle

Notes:

- This function will only be called if the `ORBITER_MODULE` preprocessor directive has been defined in the source code, and orbitersdk.lib has been linked.

ExitModule

Module exit point. This is called *once* before the module is removed from memory (usually at the end of a simulation run). It can be used to free resources allocated during `InitModule`.

Synopsis:

```
DLLCLBK void ExitModule (HINSTANCE hModule)
```

Parameters:

hModule DLL instance handle

Notes:

- This function will only be called if the `ORBITER_MODULE` preprocessor directive has been defined in the source code, and orbitersdk.lib has been linked.

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 identfying the newly created vessel.
flightmodel	level of flight model realism (0=simple, 1=complex)

Return value:

Module-generated instance of *VESSEL* or a derived class.

Notes:

- The *flightmodel* value depends on user selection in the launchpad dialog. The module can use this parameter to define two different sets of vessel parameters – a simplified one for novice users, and a realistic one for advanced users.
- 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

Obsolete. Use *VESSEL2::clbkSetClassCaps* instead.
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

Obsolete. Use *VESSEL2::clbkSetStateEx* instead.
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)*

ovcSetStateEx

Obsolete. Use *VESSEL2::clbkSetStateEx* instead.
This callback function is invoked by Orbiter when a vessel is created during the simulation with a call to *oapiCreateVesselEx*. It allows the vessel to initialise its state according to the provided *VESSELSTATUSx* interface (version x ≥ 2). To allow default initialisation, the status can be passed to *VESSEL::DefSetStateEx*.

Synopsis:

```
DLLCLBK void ovcSetStateEx (
     VESSEL *vessel,
     const void *status)
```

Parameters:

vessel	vessel interface
status	pointer to a *VESSELSTATUSx* structure

Notes:

- This callback function receives the *VESSELSTATUSx* structure passed to *oapiCreateVesselEx*. It must therefore be able to process the interface version used by those functions.
- This function remains valid even if future versions of Orbiter introduce new *VESSELSTATUSx* interfaces.
- A typical implementation may look like this:

```
DLLCLBK void ovcSetStateEx (VESSEL *vessel, const void *status)
{
    // specialised vessel initialisations
    // ...

    // default initialisation:
    vessel->DefSetStateEx (status);
}
```

ovcLoadState

Obsolete. Use *VESSEL2::clbkLoadStateEx* instead.
Called when the vessel must read its initial status from a scenario file. New modules should use ovcLoadStateEx instead.

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 non-standard 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!)

See also:

ovcLoadStateEx

ovcLoadStateEx

Obsolete. Use *VESSEL2::clbkLoadStateEx* instead.
Called when the vessel must read its initial status from a scenario file.

Synopsis:

```
DLLCLBK void ovcLoadStateEx (
     VESSEL *vessel,
     FILEHANDLE scn,
     void *vs)
```

Parameters:

vessel	vessel interface
scn	scenario file handle
vs	pointer to a *VESSELSTATUSx* struct ($x \geq 2$)

Notes:

- This callback function allows to read module-specific status parameters from a 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::ParseScenarioLineEx*, to allow the processing of generic options.
- Orbiter will always pass the latest supported *VESSELSTATUSx* version to *ovcLoadStateEx*. This is currently *VESSELSTATUS2*, but may change in future versions. To maintain compatibility, *vs* should therefore not be used other than to pass it on to *ParseScenarioLineEx*.
- A typical parser implementation may look like this:

```
DLLCLBK void ovcLoadStateEx (VESSEL *vessel, FILEHANDLE scn,
    void *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 on to Orbiter
            vessel->ParseScenarioLineEx (line, vs);
        }
    }
}
```

See also:

VESSEL::ParseScenarioLineEx

ovcSaveState

Obsolete. Use *VESSEL2::clbkSaveState* instead.
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 non-standard 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.

ovcPostCreation

Obsolete. Use *VESSEL2::clbkPostCreation* instead.
Called after a vessel has been created and its state has been set.

Synopsis:

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

Parameters:

vessel	vessel interface

Notes:

- This function can be used to perform the final setup steps for the vessel, such as animation states and instrument panel states. When this function is called, the vessel state (e.g. position, thruster levels, etc.) have been defined.

ovcFocusChanged

Obsolete. Use *VESSEL2::clbkFocusChanged* instead.
Called after a vessel gained or lost input focus.

Synopsis:

```
DLLCLBK void ovcFocusChanged (
     VESSEL *vessel,
     bool getfocus,
     OBJHANDLE hNewVessel,
     OBJHANDLE hOldVessel)
```

Parameters:

vessel	vessel interface
getfocus	true if *vessel* gained focus, false if it lost focus
hNewVessel	handle of vessel gaining focus
hOldVessel	handle of vessel losing focus

Notes:

- If *getfocus* is true, then *vessel* is the interface to *hNewVessel*, otherwise it is the interface to *hOldVessel*.
- This is also called at the beginning of the simulation to the initial focus object. In this case *hOldVessel* is NULL.

ovcVisualCreated

Obsolete. Use *VESSEL2::clbkVisualCreated* instead.
Called after a 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

Obsolete. Use *VESSEL2::clbkVisualDestroyed* instead.
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

Obsolete. Use *VESSEL2::clbkPreStep* or *VESSEL2::clbkPostStep* instead.
Called at each simulation time step after the vessel has updated its position and velocity for the current simulation time.

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.

ovcRCSmode

Obsolete. Use *VESSEL2::clbkRCSMode* instead.
Called when the RCS (reaction control system) mode changes.

Synopsis:

```
DLLCLBK void ovcRCSmode (VESSEL *vessel, int mode)
```

Parameters:

vessel	vessel interface
mode	new RCS mode: 0=disabled, 1=rotational, 2=linear

Notes:

- This callback function is invoked when the user switches RCS mode via the keyboard ("/" or "Ctrl-/" on numerical keypad) or after a call to *VESSEL::SetAttitudeMode* or *VESSEL::ToggleAttitudeMode*.

ovcADCtrlmode

Obsolete. Use *VESSEL2::clbkADCtrlMode* instead.
Called when user input mode for aerodynamic control surfaces (elevator, rudder, aileron) changes.

Synopsis:

```
DLLCLBK void ovcADCtrlmode (VESSEL *vessel, DWORD mode)
```

Parameters:

vessel	vessel interface
mode	control mode

Notes:

- The returned control mode contains bit flags as follows:
  bit 0: elevator enabled/disabled
  bit 1: rudder enabled/disabled
  bit 2: ailerons enabled/disabled
  Therefore, mode=0 indicates control surfaces disabled, mode=7 indicates fully enabled.

ovcNavmode

Obsolete. Use *VESSEL2::clbkNavMode* instead.
Called at activation/deactivation of a navmode (see also *VESSEL::ActivateNavmode*)

Synopsis:

```
DLLCLBK void ovcNavmode (
     VESSEL *vessel,
     int mode,
     bool active)
```

Parameters:

vessel	vessel interface
mode	navmode constant (see section 9)
active	*true* for activation, *false* for deactivation.

ovcHUDmode

Obsolete. Use *VESSEL2::clbkHUDMode* instead.
Called after a change of the vessel's HUD (head up display) mode.

Synopsis:

```
DLLCLBK void ovcHUDmode (VESSEL *vessel, int mode)
```

Parameters:

vessel	vessel interface
mode	new HUD mode

Notes:

- For currently supported HUD modes see *HUD_xxx* constants in section 9.
- mode *HUD_NONE* indicates that the HUD has been turned off.

ovcMFDmode

Obsolete. Use *VESSEL2::clbkMFDMode* instead.
Called after the display mode of one of the MFDs (multifunctional displays) has changed.

Synopsis:

```
DLLCLBK void ovcMFDmode (VESSEL *vessel, int mfd, int mode)
```

Parameters:

vessel	vessel interface
mfd	MFD identifier (see Section 9).
mode	new MFD mode (see Section 9).

ovcDockEvent

Obsolete. Use *VESSEL2::clbkDockEvent* instead.
Called after a docking or undocking event at one of the vessel's docking ports.

Synopsis:

```
void ovcDockEvent (
     VESSEL *vessel,
     int dock,
     OBJHANDLE connected)
```

Parameters:

vessel	vessel interface
dock	docking port index
connected	handle to docked vessel, or NULL for undocking event

ovcAnimate

Obsolete. Use *VESSEL2::clbkAnimate* instead.
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	vessel 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

Obsolete. Use *VESSEL2::clbkConsumeDirectKey* instead.
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,
     char *keystate)
```

Parameters:

vessel	vessel interface
keystate	keyboard state

Return value:

A nonzero return value will completely disable default processing of the key state for the current time step. To disable the default processing of selected keys only, use the *RESETKEY* macro (see orbitersdk.h) and return 0.

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)) {
  // perform action
  RESETKEY (keystate, OAPI_KEY_F10);
  // optional: prevent default processing of the key
}
```

- This function should be used where a key *state*, rather than a key *event* is required, for example when engaging thrusters or similar. To test for key events (key pressed, key released) use *ovcConsumeBufferedKey* instead.

ovcConsumeBufferedKey

Obsolete. Use *VESSEL2::clbkConsumeBufferedKey* instead.
This callback function notifies the module of a buffered key event (key pressed or key released).

Synopsis:

```
DLLCLBK int ovcConsumeBufferedKey (
     VESSEL *vessel,
     DWORD key,
     bool down,
     char *kstate)
```

Parameters:

vessel	vessel interface
key	key scan code (see *OAPI_KEY_xxx* constants in orbitersdk.h)
down	true if key was pressed, false if key was released
kstate	current keyboard state

Return value:

The function should return 1 if Orbiter's default processing of the key should be skipped, 0 otherwise.

Notes:

- The key state (kstate) can be used to test for key modifiers (Shift, Ctrl, etc.). The *KEYMOD_xxx* macros defined in orbitersdk.h are useful for this purpose.
- This function may be called repeatedly during a single frame, if multiple key events have occurred in the last time step.

ovcLoadPanel

Obsolete. Use *VESSEL2::clbkLoadPanel* instead.
Called when Orbiter needs to load a custom instrument panel from the module.

Synopsis:

```
DLLCLBK bool ovcLoadPanel (VESSEL *vessel, int id)
```

Parameters:

vessel	vessel interface
id	panel identifier

Return value:

false indicates failure.

Notes:

- In the body of this function the module should define the panel background bitmap, and panel capabilities, e.g. the position of MFDs and other instruments, active areas (mouse hotspots) etc.
- A vessel which implements panels must at least support panel id 0 (the main panel. If any panels register neighbour panels (see *oapiSetPanelNeighbours*), all the neighbours must be supported, too.

See also:

oapiRegisterPanelBackground, oapiRegisterPanelArea, oapiRegisterMFD.

ovcPanelMouseEvent

Obsolete. Use *VESSEL2::clbkPanelMouseEvent* instead.
Called when a previously registered panel area receives a mouse button event.

Synopsis:

```
DLLCLBK bool ovcPanelMouseEvent (
     VESSEL *vessel,
     int id,
     int event,
     int mx,
     int my)
```

Parameters:

vessel	vessel interface
id	panel area identifier
event	mouse event (see *PANEL_MOUSE_xxx* constants in orbitersdk.h)
mx, my	relative mouse position in area at event

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Notes:

- Mouse events are only sent for areas which requested notification during definition (see *oapiRegisterPanelArea*).

ovcPanelRedrawEvent

Obsolete. Use *VESSEL2::clbkPanelRedrawEvent* instead.
Called when a panel area receives a redraw event.

Synopsis:

```
DLLCLBK bool ovcPanelRedrawEvent (
     VESSEL *vessel,
     int id,
     int event,
     SURFHANDLE surf)
```

Parameters:

vessel	vessel interface
id	panel area identifier
event	redraw event (see *PANEL_REDRAW_xxx* constants in orbitersdk.h)
surf	area surface handle.

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Notes:

- This callback function is only called for areas which were not registered with the *PANEL_REDRAW_NEVER* flag.
- All redrawable panel areas receive a *PANEL_REDRAW_INIT* redraw notification when the panel is created, in addition to any registered redraw notification events.
- The surface handle surf contains either the *current area state*, or the *area background*, depending on the flags passed during area registration.
- The surface handle may be used for blitting operations, or to receive a Windows device context (DC) for Windows-style redrawing operations.

See also:

oapiGetDC, oapiReleaseDC, oapiTriggerPanelRedrawArea

11 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. Examples for various applications of the *VESSEL* class can be found in the sample vessel module implementations in the Orbitersdk\samples folder.

Public member functions

11.1 Construction/creation

VESSEL

Constructor. Creates a vessel interface instance from a vessel handle.

Synopsis:

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

Parameters:

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

Notes:

- This function creates an interface to an *existing* vessel. It does not create a new vessel. New vessels are created with the *oapiCreateVessel* and *oapiCreateVesselEx* functions.
- The *VESSEL* constructor (or the constructor of a derived specialised vessel class) will normally be invoked in the *ovcInit* callback function of a vessel module:

```
class MyVessel: public VESSEL
{
    // MyVessel interface definition
};

DLLCLBK VESSEL *ovcInit (OBJHANDLE hvessel, int flightmodel)
{
    return new MyVessel (hvessel, flightmodel);
}

DLLCLBK void ovcExit (VESSEL *vessel)
{
    delete (MyVessel*)vessel;
}
```

- The *VESSEL* interface instance created in ovcInit should be deleted in *ovcExit*.

See also:

oapiCreateVessel, *oapiCreateVesselEx*, *ovcInit*

Create

Obsolete. This function has been replaced by *oapiCreateVessel* and *oapiCreateVesselEx*.

GetHandle

Returns a handle to the vessel object.

Synopsis:

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

Return value:

vessel handle, as passed to the *VESSEL* constructor.

Notes:

- The handle is useful for various API function calls.

11.2 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

GetEnableFocus

Returns *true* if the vessel can receive the input focus, *false* otherwise.

Synopsis:

```
bool GetEnableFocus (void) const
```

Return value:
Focus enabled status.

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. Equivalent to the *oapiGetEmptyMass* API function.

Synopsis:
```
double GetEmptyMass (void) const
```

Return value:
Vessel empty mass [kg].

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 [m^2]

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 [m^2]

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]

GetCameraDefaultDirection

Returns the default camera direction for internal (cockpit) view.

Synopsis:

```
void GetCameraDefaultDirection (VECTOR3 &dir0) const
```

Parameters:

dir0 default camera direction in vessel coordinates

Notes:

- The default camera direction may change when the user selects a different instrument panel or virtual cockpit position.
- The returned direction vector is normalised to length 1.

SetEnableFocus

Set the vessel's ability to receive the input focus.

Synopsis:

```
void SetEnableFocus (bool enable) const
```

Parameters:

enable focus enabled status

Notes:

- The default focus status before the first call to *SetEnableFocus* is *true*, unless overridden by the config file.

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. Equivalent to the *oapiSetEmptyMass* API function.

Synopsis:

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

Parameters:

m vessel empty mass [kg]

SetCOG_elev

Obsolete. 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 *pt3-pt1* x *pt2-pt1* defines the horizon UP direction for the landed vessel (given a left-handed coordinate system).

SetSurfaceFrictionCoeff

Sets the coefficients of surface friction which define the deceleration forces during taxiing. mu_lng is the coefficient acting in longitudinal (forward) direction, mu_lat the coefficient acting in lateral (sideways) direction. The friction forces are proportional to the coefficient and the weight of the vessel:

$$F_{friction} = \mu G$$

Synopsis:

```
void SetSurfaceFrictionCoeff (
     double mu_lng,
     double mu_lat) const
```

Parameters:

mu_lng	friction coefficient in longitudinal direction
mu_lat	friction coefficient in lateral direction

Notes:

- The higher the coefficient, the faster the vessel will come to a halt.
- Typical parameters for a spacecraft equipped with landing wheels would be *mu_lng* = 0.1 and *mu_lat* = 0.5. If the vessel hasn't got wheels, *mu_lng* = 0.5.

- The coefficients should be adjusted for belly landings when the landing gear is retracted.
- The longitudinal and lateral directions are defined by the touchdown points:

$$\vec{s}_{\ln g} = \vec{p}_0 - \frac{1}{2}(\vec{p}_1 + \vec{p}_2), \quad \vec{s}_{lat} = \vec{p}_2 - \vec{p}_1$$

See also:

SetTouchdownPoints

SetCrossSections

Sets the vessel's cross sections projected in the direction of the vessel's principal axes [m^2].

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]

SetPitchMomentScale

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

Synopsis:

```
void SetPitchMomentScale (double scale) const
```

Parameters:

scale scale factor for pitch moment

SetBankMomentScale

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

Synopsis:

```
void SetBankMomentScale (double scale) const
```

Parameters:

scale scale factor for bank moment

SetPMI

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

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 `shipedit` 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).

SetCameraDefaultDirection

Sets the default camera direction for internal (cockpit) view.

Synopsis:

```
void SetCameraDefaultDirection (const VECTOR3 &cd) const
```

Parameters:

cd	new camera direction in vessel coordinates

Notes:

- By default, the default direction is (0,0,1), i.e. forward.
- The supplied direction vector must be normalised to length 1.
- Calling this function automatically sets the current actual view direction to the default direction.
- This function can either be called during *ovcSetClassCaps*, to define the default camera direction globally for the vessel, or during *ovcGenericCockpit*, *ovcLoadPanel* and *clbkLoadVC*, to define different default directions for different instrument panels or virtual cockpit positions.

- In Orbiter, the user can return to the default direction by pressing the "Home" key on the cursor key pad.

SetCameraRotationRange

Sets the range over which the cockpit camera can be rotated from its default direction.

Synopsis:

```
void SetCameraRotationRange (
     double left,
     double right,
     double up,
     double down) const
```

Parameters:

left	rotation range to the left [rad]
right	rotation range to the right [rad]
up	rotation range up [rad]
down	rotation range down [rad]

Notes:

- All ranges must be ≥ 0. The left and right ranges should be $< \pi$. The up and down ranges should be $< \pi/2$.
- The default values are 0.8π for left and right ranges, and 0.4π for up and down ranges.

SetCameraShiftRange

Set the linear movement range for the cockpit camera. Defining a linear movement allows the user to move the head forward or sideways, e.g. to get a better look out of a window.

Synopsis:

```
void SetCameraShiftRange (
     const VECTOR3 &forward,
     const VECTOR3 &left,
     const VECTOR3 &right) const
```

Parameters:

forward	offset vector when leaning forward
left	offset vector when leaning left
right	offset vector when leaning right

Notes:

- If a linear movement range is defined with this function, the user can 'lean' forward or sideways using the 'cockpit slew' keys. Supported keys are:

Name	default	action
CockpitCamDontLean	`Ctrl+Alt+Down`	return to default position
CockpitCamLeanForward	`Ctrl+Alt+Up`	lean forward
CockpitCamLeanLeft	`Ctrl+Alt+Left`	lean left
CockpitCamLeanRight	`Ctrl+Alt+Right`	lean right

- The movement vectors are taken relative to the default cockpit position defined via *SetCameraOffset*.
- This function should be called when initialising a cockpit mode (e.g. in *clbkLoadPanel* or *clbkLoadVC*). By default, Orbiter resets the linear movement range to zero whenever the cockpit mode changes.

ParseScenarioLine

Process an input line from a scenario file by updating a *VESSELSTATUS* status struct.

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 *ovcLoadState* to allow Orbiter to process any generic status parameters which are not processed by the module.
- This function is retained for backward compatibility. New modules should use the *ovcLoadStateEx* and *ParseScenarioLineEx* functions.

ParseScenarioLineEx

Process an input line from a scenario file by updating a *VESSELSTATUSx* status struct ($x \geq 2$).

Synopsis:

```
void ParseScenarioLineEx (char *line, void *status) const
```

Parameters:

line	line to be interpreted
status	status parameters (points to a *VESSELSTATUSx* variable).

Notes:

- This function should be used within the body of *ovcLoadStateEx*.
- The parser in *ovcLoadStateEx* should forward all lines not recognised by the module to Orbiter via *ParseScenarioLineEx* to allow processing of standard vessel settings.
- *ovcLoadStateEx* currently provides a *VESSELSTATUS2* status definition. This may change in future versions, so `status` should not be used within *ovcLoadStateEx* other than passing it to *ParseScenarioLineEx*.

See also:

ovcLoadStateEx

11.3 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.

GetStatusEx

Returns vessel's current status parameters in a VESSELSTATUSx structure (version $x \geq 2$).

Synopsis:

```
void GetStatusEx (void *status) const
```

Parameters:

status   pointer to a VESSELSTATUSx structure

Notes:

- This method can be used with any VESSELSTATUSx interface version supported by Orbiter. Currently only VESSELSTATUS2 is supported.
- The `version` field of the VESSELSTATUSx structure must be set by the caller prior to calling the method, to tell Orbiter which interface version is required.
- In addition, the caller must set the VS_FUELLIST, VS_THRUSTLIST and VS_DOCKINFOLIST bits in the `flag` field, if the corresponding lists are required. Otherwise Orbiter will not produce these lists.
- If VS_FUELLIST is specified and the `fuel` field is NULL, Orbiter will allocate memory for the list. The caller is responsible for deleting the list after use. If the `fuel` field is not NULL, Orbiter assumes that a list of sufficient length to store all propellant resources has been allocated by the caller.
- The same applies to the `thruster` and `dockinfo` lists.

See also:

SetStateEx, VESSELSTATUS2

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.

DefSetStateEx

Calls the default Orbiter vessel state initialisation with the provided VESSELSTATUSx interface (version $x \geq 2$).

Synopsis:

```
void DefSetStateEx (const void *status) const
```

Parameters:

status   pointer to a VESSELSTATUSx structure

Notes:

- status must point to a VESSELSTATUSx structure. Currently only VESSELSTATUS2 is supported, but future Orbiter versions may introduce new interfaces.
- Typically, this function will be called in the body of ovcSetStateEx to enable default state initialisation.

SaveDefaultState

Obsolete. Use a call to the base class VESSEL::clbkSaveState from within the overloaded callback function instead.

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.

GroundContact

Flag indicating contact with a planetary surface.

Synopsis:

```
bool GroundContact (void) const
```

Return value:

true indicates ground contact (at least one of the vessel's touchdown reference points is in contact with a planet surface).

OrbitStabilised

Flag indicating whether orbit stabilisation is used for the vessel at the current time step.

Synopsis:

```
bool OrbitStabilised (void) const
```

Return value:

true indicates that the vessels uses its osculating orbital elements to update its state vectors, assuming an unperturbed Keplerian 2-body calculation to account for the gravitational effect of the primary gravity source.

Notes:

- A vessel reverts to orbit stabilisation only if
  - the user has enabled it in the launchpad dialog, and
  - the user-defined perturbation and time step limits are satisfied, and
  - no non-gravitational forces (thrusters, aerodynamics, etc) are active

NonsphericalGravityEnabled

Flag indicating whether the vessel uses perturbations in gravity fields due to nonspherical planet shapes to update its state vectors for the current time step.

Synopsis:

```
bool NonsphericalGravityEnabled (void) const
```

Return value:

true indicates that gravity perturbations are taken into account.

Notes:

- Nonspherical gravity is applied if
  - the user has enabled it in the launchpad dialog
  - the vessel's orbit is not stabilised at the current time step.

GetMass

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

Synopsis:

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

Return value:

Current vessel mass [kg].

GetAttitudeMode

Returns the current RCS (reaction control system) thruster mode.

Synopsis:

```
int GetAttitudeMode (void) const
```

Return value:

Current RCS mode: *RCS_NONE*, *RCS_ROT*, or *RCS_LIN*.

Notes:

- The reaction control system consists of a set of small thrusters arranged around the vessel. They can be fired in pre-defined configurations to provide either a change in angular velocity (in *RCS_ROT* mode) or in linear velocity (in *RCS_LIN* mode).
- *RCS_NONE* indicates that the RCS is disabled or not available.
- Currently Orbiter doesn't allow simultaneous linear and rotational RCS control via keyboard or joystick. The user has to switch between the two. However, simultaneous operation is possible via the "RControl" plugin module.
- Not all vessel classes may define a complete RCS.

SetAttitudeMode

Set the vessel's attitude thruster mode.

Synopsis:

```
bool SetAttitudeMode (int mode) const
```

Parameters:

mode   attitude mode (RCS_NONE, RCS_ROT, or RCS_LIN).

Return value:

Error flag; *false* indicates error (requested mode not available)

GetADCtrlMode

Returns current input mode for aerodynamic control surfaces (elevator, rudder, ailerons).

Synopsis:

```
DWORD GetADCtrlMode (void) const
```

Return value:

Current control mode

Notes:

- The returned control mode contains bit flags as follows:
  bit 0:   elevator enabled/disabled
  bit 1:   rudder enabled/disabled
  bit 2:   ailerons enabled/disabled
  Therefore, mode=0 indicates control surfaces disabled, mode=7 indicates fully enabled.

SetADCtrlMode

Set input mode for aerodynamic control surfaces.

Synopsis:

```
void SetADCtrlMode (DWORD mode) const
```

Parameters:

mode   control mode

Notes:

- See `GetADCtrlMode()` for the meaning of the bit-flags in mode,.

GetAttitudeRotLevel

Returns the current thrust level for attitude thruster groups in rotational mode.

Synopsis:

```
void VESSEL::GetAttitudeRotLevel (VECTOR3 &th) const
```

Parameters:

th   vector containing thrust levels (-1 to 1)

Notes:

- The components of th are:
  th.x – attitude thrusters rotating around lateral axis
  th.y – attitude thrusters rotating around vertical axis
  th.z – attitude thrusters rotating around longitudinal axis
- To obtain the actual thrust force magnitudes [N], the absolute values must be multiplied with the max. attitude thrust (see GetMaxThrust())

See also:

VESSEL::GetMaxThrust(), VESSEL::GetAttitudeLinLevel(), VESSEL::GetAttitudeMode()

SetAttitudeRotLevel (1)

Set attitude thruster levels for rotation in all 3 axes.

Synopsis:

```
void SetAttitudeRotLevel (const VECTOR3 &th) const
```

Parameters:

th   attitude thruster levels for rotation around x,y,z axes

Notes:

- Thruster levels must be in the range [-1...1]
- This function works even if manual attitude mode is set to linear.

SetAttitudeRotLevel (2)

Set attitude thruster level for rotation around a single axis.

Synopsis:

```
void SetAttitudeRotLevel (int axis, double th) const
```

Parameters:

axis   rotation axis (0=x, 1=y, 2=z)
th   attitude thruster level

Notes:

- Thruster levels must be in the range [-1..1]

- This function works even if manual attitude mode is set to linear.

SetAttitudeLinLevel (1)

Set attitude thruster levels for linear translation in all 3 axes.

Synopsis:

```
void SetAttitudeLinLevel (const VECTOR3 &th) const
```

Parameters:

th	attitude thruster levels for translation along x,y,z

Notes:

- Thruster levels must be in the range [-1..1]
- This function works even if manual attitude mode is set to rotational.

SetAttitudeLinLevel (2)

Set attitude thruster level for linear translation along a single axis.

Synopsis:

```
void SetAttitudeLinLevel (int axis, double th) const
```

Parameters:

axis	translation axis (0=x, 1=y, 2=z)
th	attitude thruster level

Notes:

- Thruster levels must be in the range [-1..1]
- This function works even if manual attitude mode is set to rotational.

GetAttitudeLinLevel

Returns the current thrust level for attitude thrusters groups in linear mode.

Synopsis:

```
void VESSEL::GetAttitudeLinLevel (VECTOR3 &th) const
```

Parameters:

th	vector containing thrust levels (-1 to 1)

Notes:

- The components of th are:
  th.x – attitude thrusters for lateral (sideways) translation
  th.y – attitude thrusters for vertical (up/down) translation
  th.z – attitude thrusters for longitudinal (forward/backward) translation
- To obtain the actual thrust force magnitudes [N], the absolute values must be multiplied with the max. attitude thrust (see GetMaxThrust())

See also:

VESSEL::GetMaxThrust(), VESSEL::GetAttitudeRotLevel(), VESSEL::GetAttitudeMode()

ActivateNavmode

Activates a navmode.

Synopsis:

```
bool ActivateNavmode (int mode)
```

Parameters:

mode	navmode id to be activated.

Return value:

True if the specified navmode could be activated, false if not available or active already.

Notes:

- Navmodes are high-level navigation modes which involve e.g. the simultaneous and timed engagement of multiple attitude thrusters to get the vessel into a defined state. Some navmodes terminate automatically once the target state is reached (e.g. killrot), or they remain active until explicitly terminated (hlevel). Navmodes may also terminate if a second conflicting navmode is activated.
- For navmodes currently defined in Orbiter see the NAVMODE_xxx constants.

DeactivateNavmode

Deactivates a navmode.

Synopsis:

```
bool DeactivateNavmode (int mode)
```

Parameters:

mode navmode id to be deactivated.

Return value:

True if the specified navmode could be deactivated, false if not available or if deactivated already.

ToggleNavmode

Toggles a navmode on/off.

Synopsis:

```
bool ToggleNavmode (int mode)
```

Parameters:

mode navmode to be toggled.

Return value:

True if the navmode could be changed, false if it remains unchanged.

GetNavmodeState

Returns current state (on/off) of a navmode.

Synopsis:

```
bool GetNavmodeState (int mode)
```

Parameters:

mode navmode id to be checked.

Return value:

True if navmode is active, false otherwise.

AddForce

Add a custom body force.

Synopsis:

```
void AddForce (const VECTOR3 &F, const VECTOR3 &r) const
```

Parameters:

F	force vector (N)
r	radius vector (m)

Notes:

- This function can be used to implement custom forces (braking chutes, tethers, etc.) It should not be used for standard forces such as thrusters which are handled internally.
- The force is applied only for the next time step. AddForce() will therefore usually be used inside the VESSEL2::clbkPreStep() callback function.

11.4 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)

GetAngularVel

Returns vessel's current angular velocity components around its three principal axes.

Synopsis:

```
void GetAngularVel (VECTOR3 &avel) const
```

Parameters:

avel	vector receiving angular velocity components [rad/s]

Notes:

- The velocity components ω are calculated from angular moments M by Euler's equations for rigid body motion:

  $$J_z \omega_z - (J_x - J_y)\omega_x \omega_y = M_z$$

  where J are the principal moments of inertia (J=PMI*mass). Note that the differential equations are coupled which leads to a transfer of rotational energy between the rotation axes.

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).

11.5 Fuel management

CreatePropellantResource

Creates a new propellant resource ("tank") to be used for powering thrusters.

Synopsis:

```
PROPELLANT_HANDLE CreatePropellantResource (
    double maxmass,
```

```
double mass=-1.0,
double efficiency=1.0) const
```

Parameters:

maxmass	maximum propellant capacity of the resource [kg]
mass	current propellant mass of the resource [kg]
efficiency	fuel efficiency factor (> 0)

Return value:

propellant resource identifier

Notes:

- Orbiter doesn't distinguish between propellant and oxidant. A "propellant resource" is assumed to be a combination of fuel and oxidant resources.
- The interpretation of a propellant resource (liquid or solid propulsion system, ion drive, etc.) is up to the vessel developer.
- The rate of fuel consumption depends on the thrust level and Isp of the thrusters attached to the resource.
- The fuel efficiency rating, together with a thruster's Isp rating, determines how much fuel is consumed per second to obtain a given thrust:

  $$R = \frac{F}{e \cdot Isp}$$

  R: fuel rate [kg/s], F: thrust [N], e: efficiency, Isp: fuel-specific impulse [m/s]
- If mass < 0 then mass=maxmass is assumed.

DelPropellantResource

Remove a propellant resource and disable all thrusters which were linked to this resource.

Synopsis:

```
void DelPropellantResource (PROPELLANT_HANDLE &ph) const
```

Parameters:

ph	propellant resource identifier (NULL on return)

ClearPropellantResources

Remove all propellant resources and unlink all thrusters from their resources.

Synopsis:

```
void ClearPropellantResources (void) const
```

Notes:

- After a call to this function, all the vessel's thrusters will be disabled until they are linked to new resources.

GetPropellantHandleByIndex

Returns the handle of a propellant resource for a given index.

Synopsis:

```
PROPELLANT_HANDLE GetPropellantHandleByIndex (
    DWORD idx) const
```

Parameters:

idx	propellant resource index

Return value:

propellant resource handle

Notes:

- The index must be in the range between 0 and npropellant-1, where npropellant is the number of propellant resources defined for the vessel (use GetPropellantCount to obtain this value). If the index is out of range, the returned handle is NULL.
- The index of a given propellant resource may change if any resources are deleted. The handle remains valid until the corresponding resource is deleted.

GetPropellantCount

Returns the number of propellant resources currently defined for the vessel.

Synopsis:

```
DWORD GetPropellantCount (void) const
```

Return value:

Number of propellant resources currently defined for the vessel.

SetDefaultPropellantResource

Define a “default” propellant resource. This is used for the various legacy fuel-related API functions, and for the “Fuel” indicator in the generic panel-less HUD display.

Synopsis:

```
void SetDefaultPropellantResource (
     PROPELLANT_HANDLE ph) const
```

Parameters:

ph	propellant resource identifier

Notes:

- If this function is not used, the first propellant resource is used as default.

See also:

GetFuelMass(), GetFuelRate(), SetFuelMass(), SetMaxFuelMass(), GetMaxFuelMass()

SetPropellantMaxMass

Reset the maximum capacity [kg] of a fuel resource.

Synopsis:

```
void SetPropellantMaxMass (
     PROPELLANT_HANDLE ph,
     double maxmass) const
```

Parameters:

ph	propellant resource identifier
maxmass	max. fuel capacity (≥ 0) [kg]

SetPropellantEfficiency

Reset the efficiency factor of a fuel resource.

Synopsis:

```
void SetPropellantEfficiency (
     PROPELLANT_HANDLE ph,
     double efficiency) const
```

Parameters:

ph	propellant resource identifier
efficiency	fuel efficiency factor (> 0)

Notes:

- See `CreatePropellantResource()` for an explanation of the fuel efficiency factor.

SetPropellantMass

Set current mass of a propellant resource.

Synopsis:

```
void SetPropellantMass (
     PROPELLANT_HANDLE ph,
     double mass) const
```

Parameters:

ph	propellant resource identifier
mass	propellant mass [kg]

Notes:

- $0 \leq$ mass $\leq$ maxmass is required.
- This method should be used to simulate refuelling, fuel leaks, cross-feeding between tanks, etc. but *not* for normal fuel consumption by thrusters (which is handled internally by the Orbiter core).

GetPropellantMass

Returns the current mass of a propellant resource.

Synopsis:

```
double GetPropellantMass (PROPELLANT_HANDLE ph) const
```

Parameters:

ph	propellant resource identifier

Return value:

current propellant mass [kg]

GetPropellantMaxMass

Returns the maximum capacity [kg] of a fuel resource.

Synopsis:

```
double GetPropellantMaxMass (PROPELLANT_HANDLE ph) const
```

Parameters:

ph	propellant resource identifier

Return value:

max. fuel capacity [kg]

GetPropellantEfficiency

Returns the efficiency factor of a fuel resource.

Synopsis:

```
double GetPropellantEfficiency (PROPELLANT_HANDLE ph) const
```

Parameters:

ph	propellant resource identifier

Return value:

fuel efficiency factor

GetPropellantFlowrate

Returns the mass flow rate of a fuel resource.

Synopsis:

```
double GetPropellantFlowrate (PROPELLANT_HANDLE ph) const
```

Parameters:

ph propellant resource identifier

Return value:

Propellant mass flow rate [kg/s].

GetTotalPropellantMass

Returns the vessel's current total propellant mass.

Synopsis:

```
double GetTotalPropellantMass (void) const
```

Return value:

Current total propellant mass [kg]

GetTotalPropellantFlowrate

Returns the current total mass flow rate, summed over all propellant resources.

Synopsis:

```
double GetTotalPropellantFlowrate (void) const
```

Return value:

Total propellant mass flow rate [kg/s]

See also:

GetPropellantFlowrate(), GetFuelRate()

GetFuelMass

Returns the current mass of the vessel's default propellant resource.

Synopsis:

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

Return value:

Current fuel mass of default propellant resource [kg]

See also:

GetPropellantMass(), SetDefaultPropellantResource()

GetFuelRate

Returns the vessel's current propellant mass flow rate for the default propellant resource.

Synopsis:

```
double GetFuelRate (void) const
```

Return value:

Propellant mass flow rate for default propellant resource [kg/s]

See also:

GetPropellantFlowrate()

SetFuelMass

Sets the current fuel mass of the vessel's default propellant resource [kg].

Synopsis:

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

Parameters:

m Current fuel mass [kg].

Notes:

- If the vessel has not defined any propellant resources then this function has no effect.

See also:

SetPropellantMass(), SetDefaultPropellantResource()

SetMaxFuelMass

Sets the maximum fuel capacity of the vessel's default propellant resource, or creates a new resource if none exists.

Synopsis:

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

Parameters:

m Maximum fuel mass [kg].

Notes:

- If the vessel already contains propellant resources, this function resets the maximum capacity of the vessel's default resource, otherwise it creates a new resource with this capacity, and makes it the default resource.

See also:

SetPropellantMaxMass(), SetDefaultPropellantResource()

GetMaxFuelMass

Returns the maximum fuel capacity of the vessel's default propellant resource.

Synopsis:

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

Return value:

Maximum fuel mass of default propellant resource [kg].

Notes:

- The function returns 0 if no fuel resources are defined.

See also:

GetPropellantMaxMass(), SetDefaultPropellantResource()

11.6 Thruster management

CreateThruster

Add a logical thruster definition for the vessel.

Synopsis:

```
THRUSTER_HANDLE CreateThruster (
    const VECTOR3 &pos,
    const VECTOR3 &dir,
    double maxth0,
    PROPELLANT_HANDLE hp=NULL,
    double isp0=0.0,
    double isp_ref=0.0,
    double p_ref=101.4e3) const;
```

Parameters:

pos	thrust force attack point (vessel coordinates)
dir	thrust force direction (vessel coordinates)
maxth0	max. vacuum thrust rating [N]
hp	propellant resource for the thruster
isp0	vacuum Isp (fuel-specific impulse) rating [m/s]
isp_ref	Isp rating at ambient pressure p_ref [m/s]
p_ref	reference pressure for Isp rating [Pa]

Return value:

thruster identifier

Notes:

- The fuel-specific impulse defines how much thrust is produced by burning 1kg of fuel per second. If the Isp level is not specified or is ≤ 0, a default value is used (see `SetISP()`).
- To define the thrust and Isp ratings to be pressure-dependent, specify an `isp_ref` value > 0, and set `p_ref` to the corresponding atmospheric pressure. Thrust and Isp at pressure p will then be calculated as $Isp(p) = Isp_0(1 - p\eta), \quad Th(p) = Th_0(1 - p\eta)$, where p_{ref} Isp_0
- If `isp_ref` ≤ 0 then no pressure-dependence is assumed ($\eta = 0$)
- If no propellant resource is specified, the thruster is disabled until it is linked to a resource by `SetThrusterResource()`.
- Thrusters can now create simultaneous linear and angular moments, depending on the attack point and direction.
- Use `CreateThrusterGroup()` to assemble thrusters into logical groups.

See also:

DelThruster(), CreateThrusterGroup(), AddExhaust(), SetISP(), SetThrusterISP(), SetThrusterResource()

DelThruster

Delete a logical thruster definition.

Synopsis:

```
bool DelThruster (THRUSTER_HANDLE &th) const
```

Parameters:

th	thruster identifier (NULL on return)

Return value:

true on success. The function will fail if the handle is invalid.

Notes:

- Deleted thrusters will be automatically removed from all groups they have been assigned to.
- All exhaust render definitions which refer to the deleted thruster will be removed.

See also:
CreateThruster(), AddExhaust(), CreateThrusterGroup()

ClearThrusterDefinitions

Removes all thruster and thruster group definitions.

Synopsis:
```
void ClearThrusterDefinitions () const
```

Notes:
- This also removes all previously defined exhaust render definitions.

GetThrusterHandleByIndex

Returns the handle of a thruster specified by its index.

Synopsis:
```
THRUSTER_HANDLE GetThrusterHandleByIndex (DWORD idx) const
```

Parameters:

idx	thruster index

Return value:
Thruster handle

Notes:
- The index must be between 0 and nthruster-1, where nthruster is the thruster count returned by VESSEL::GetThrusterCount. If the index is out of range, the returned handle is NULL.
- Note that the thruster indices change if vessel thrusters are deleted. A thruster handle remains valid until the corresponding thruster is deleted.

GetThrusterCount

Returns the number of thrusters currently defined for the vessel.

Synopsis:
```
DWORD GetThrusterCount (void) const
```

Return value:
Number of thrusters defined for the vessel.

SetThrusterResource

Connects the thruster to a fuel resource (tank).

Synopsis:
```
void SetThrusterResource (
     THRUSTER_HANDLE th,
     PROPELLANT_HANDLE ph) const
```

Parameters:

th	thruster identifier
ph	fuel resource identifier

Notes:
- To disconnect the thruster from its current tank, use ph=NULL.

SetThrusterRef

Reset the thrust force attack point of a thruster.

Synopsis:

```
void SetThrusterRef (
     THRUSTER_HANDLE th,
     const VECTOR3 &pos) const
```

Parameters:

th	thruster identifier
pos	new attack point

Notes:

- This function should be used whenever a thruster has been physically moved in the vessel's local frame of reference.

GetThrusterRef

Returns the thrust force attack point of a thruster.

Synopsis:

```
void GetThrusterRef (
     THRUSTER_HANDLE th,
     VECTOR3 &pos) const
```

Parameters:

th	thruster identifier
pos	attack point

SetThrusterDir

Reset the force direction of a thruster.

Synopsis:

```
void SetThrusterDir (
     THRUSTER_HANDLE th,
     const VECTOR3 &dir) const
```

Parameters:

th	thruster identifier
dir	new thrust direction

Notes:

- This function should be used to reflect a tilt of the thruster (e.g. for an implementation of thrust vectoring)

GetThrusterDir

Returns the force direction of a thruster.

Synopsis:

```
void GetThrusterDir (
     THRUSTER_HANDLE th,
     VECTOR3 &dir) const
```

Parameters:

th	thruster identifier
dir	thrust direction

SetThrusterMax0

Reset the maximum vacuum thrust rating of a thruster.

Synopsis:

```
void SetThrusterMax0 (THRUSTER_HANDLE th, double maxth0)
const
```

Parameters:

th	thruster identifier
maxth0	new maximum vacuum thrust rating [N]

Notes:

- The max. thrust rating in the presence of atmospheric ambient pressure may be lower if a pressure-dependent Isp value has been defined.

See also:

CreateThruster, SetThrusterIsp

GetThrusterMax0

Returns the maximum vacuum thrust rating of a thruster.

Synopsis:

```
double GetThrusterMax0 (THRUSTER_HANDLE th) const
```

Parameters:

th	thruster identifier

Return value:

Maximum vacuum thrust rating [N]

Notes:

- To retrieve the actual current maximum thrust rating (which may be lower in the presence of ambient atmospheric pressure) use GetThrusterMax.

GetThrusterMax (1)

Returns the current maximum thrust rating of a thruster.

Synopsis:

```
double GetThrusterMax (THRUSTER_HANDLE th) const
```

Parameters:

th	thruster identifier

Return value:

maximum thrust rating at the current atmospheric pressure [N]

Notes:

- This function will return the vacuum max thrust rating, unless a pressure-dependent Isp value has been defined for the thruster.

See also:

CreateThruster, SetThrusterIsp

GetThrusterMax (2)

Returns maximum thrust rating of a thruster for a specific ambient pressure.

Synopsis:

```
double GetThrusterMax (
    THRUSTER_HANDLE th,
    double p_ref) const
```

Parameters:

th	thruster identifier
p_ref	reference pressure [Pa]

Return value:

maximum thrust rating [N] at atmospheric pressure p_ref.

SetThrusterIsp (1)

Reset the fuel-specific impulse rating of a thruster, assuming no pressure-dependence.

Synopsis:

```
void SetThrusterIsp (THRUSTER_HANDLE th, double isp) const
```

Parameters:

th	thruster identifier
isp	new Isp rating [m/s]

Notes:

- The specified Isp value is assumed to be independent of ambient atmospheric pressure. To define a pressure-dependent Isp value, use SetThrusterIsp (2).

See also:

SetISP, SetThrusterIsp (2)

SetThrusterIsp (2)

Reset pressure-dependent fuel-specific impulse rating of a thruster.

Synopsis:

```
void SetThrusterIsp (
     THRUSTER_HANDLE th,
     double isp0,
     double isp_ref,
     double p_ref=101.4e3) const
```

Parameters:

th	thruster identifier
isp0	new vacuum Isp rating [m/s]
isp_ref	Isp rating at ambient pressure p_ref [m/s]
p_ref	reference pressure for Isp rating [Pa]

Notes:

- See CreateThruster for equations of pressure-dependent thrust and Isp.

See also:

CreateThruster, SetISP, SetThrusterIsp (1)

GetThrusterIsp (1)

Returns current fuel-specific impulse (Isp) rating of a thruster.

Synopsis:

```
double GetThrusterIsp (THRUSTER_HANDLE th) const
```

Parameters:

th	thruster identifier

Return value:

Current fuel-specific impulse [m/s]

Notes:

- The return value will depend on the current ambient atmospheric pressure if a pressure-dependent Isp rating has been defined for this thruster.

See also:

SetThrusterIsp, GetThrusterIsp (2)

GetThrusterIsp (2)

Returns Isp rating for a thruster at a specific ambient pressure.

Synopsis:

```
double GetThrusterIsp (
     THRUSTER_HANDLE th,
     double p_ref) const
```

Parameters:

th	thruster identifier
p_ref	reference pressure [Pa]

Return value:

Fuel-specific impulse [m/s] at ambient pressure p_ref.

Notes:

- Unless a pressure-dependent Isp rating has been defined for this thruster, it will always return the vacuum rating, independent of the specified pressure.
- To obtain vacuum Isp rating, set p_ref to 0.
- To obtain the Isp rating at (Earth) sea level, set p_ref to 101.4e3.

GetThrusterIsp0

Returns vacuum Isp rating for a thruster.

Synopsis:

```
double GetThrusterIsp0 (THRUSTER_HANDLE th) const
```

Parameters:

th	thruster identifier

Return value:

Fuel-specific impulse in vacuum [m/s].

Notes:

- This function is equivalent to GetThrusterIsp (th, 0)

SetThrusterLevel

Set the current thrust level [0..1] for a thruster.

Synopsis:

```
void SetThrusterLevel (
     THRUSTER_HANDLE th,
     double level) const
```

Parameters:

th	thruster identifier
level	thrust level [0..1].

Notes:

- At level 1 the thruster generates maximum force, as defined by its maxth parameter.

- Certain thrusters are controlled directly by Orbiter via primary input controls (e.g. joystick throttle control for main thrusters), which may override this function.

SetThrusterLevel_SingleStep

Set thrust level for the current time step only.

Synopsis:

```
SetThrusterLevel_SingleStep (
    THRUSTER_HANDLE th,
    double level) const
```

Parameters:

th	thruster identifier
level	thrust level [0..1]

Notes:

- At level 1 the thruster generates maximum force, as defined by its maxth parameter.
- This method is applied only to the current time step, so it should normally only be used in the body of the VESSEL2::clbkPreStep() callback function.

IncThrusterLevel_SingleStep

Increment thrust level for the current time step only.

Synopsis:

```
void IncThrusterLevel_SingleStep (
    THRUSTER_HANDLE th,
    double dlevel) const
```

Parameters:

th	thruster identifier
dlevel	delta thrust level [0..1]

Notes:

- This method is applied only to the current time step, so it should normally only be used in the body of the VESSEL2::clbkPreStep() callback function.
- This function may be overridden by manual user input via keyboard and joystick, or by automatic attitude sequences.
- The resulting thrust level is clamped to range [0..1]

GetThrusterLevel

Returns the current thrust level for a thruster.

Synopsis:

double GetThrusterLevel (THRUSTER_HANDLE th) const

Parameters:

th	thruster identifier

Return value:

Current thrust level [0..1]

Notes:

- To obtain the actual force [N] generated by the thruster in vacuum, multiply the thrust level with its maximum thrust rating. However, the thrust force in the presence of ambient atmospheric pressure may be lower if `SetThrustPressureDependency` has been applied.

GetThrusterMoment

Returns the linear moment (force) and angular moment (torque) currently generated by a thruster.

Synopsis:

```
void GetThrusterMoment (
     THRUSTER_HANDLE th,
     VECTOR3 &F,
     VECTOR3 &T) const
```

Parameters:

th	thruster identifier
F	force (linear moment)
T	torque (angular moment)

Notes:

- The returned values include the influence of ambient pressure on the thrust generated by the engine.

CreateThrusterGroup

Combine thrusters into a logical group.

Synopsis:

```
THGROUP_HANDLE CreateThrusterGroup (
     THRUSTER_HANDLE *th,
     int nth,
     THGROUP_TYPE thgt) const
```

Parameters:

th	array of thruster identifiers, as returned by CreateThruster()
nth	number of thrusters in the array
thgt	thruster group type (see notes)

Return value:

thruster group identifier

Notes:

- The following group types are defined:

THGROUP_MAIN	main thrusters
THGROUP_RETRO	retro thrusters
THGROUP_HOVER	hover thrusters
THGROUP_ATT_PITCHUP	rotation: pitch up
THGROUP_ATT_PITCHDOWN	rotation: pitch down
THGROUP_ATT_YAWLEFT	rotation: yaw left
THGROUP_ATT_YAWRIGHT	rotation: yaw right
THGROUP_ATT_BANKLEFT	rotation: bank left
THGROUP_ATT_BANKRIGHT	rotation: bank right
THGROUP_ATT_RIGHT	translation: move right
THGROUP_ATT_LEFT	translation: move left
THGROUP_ATT_UP	translation: move up
THGROUP_ATT_DOWN	translation: move down
THGROUP_ATT_FORWARD	translation: move forward
THGROUP_ATT_BACK	translation: move back
THGROUP_USER	user-defined group

- Thruster groups (except for user-defined groups) are engaged by Orbiter as a result of user input. For example, pushing the stick backward in rotational

attitude mode will engage the thrusters in the `THGROUP_ATT_PITCHUP` group.
- It is the responsibility of the vessel designer to make sure that the thruster groups are designed so that they behave in a sensible way.
- Thrusters can be added to more than one group. For example, an attitude thruster can be simultaneously grouped into `THGROUP_ATT_PITCHUP` and `THGROUP_ATT_UP`.
- Rotational thrusters should be designed so that they don't induce a significant linear momentum. This means rotational groups require at least 2 thrusters each.
- Linear thrusters should be designed such that they don't induce a significant angular momentum.
- If a vessel does not define a complete set of attitude thruster groups, certain navmode sequences (e.g. KILLROT) may fail.

See also:

CreateThruster()

DelThrusterGroup (1)

Delete a thruster group and (optionally) all associated thrusters.

Synopsis:

```
bool DelThrusterGroup (
     THGROUP_HANDLE &thg,
     THGROUP_TYPE thgt,
     bool delth = false) const
```

Parameters:

thg	thruster group identifier (NULL on return)
thgt	thruster group type (see CreateThrusterGroup)
delth	thruster destruction flag

Return value:

true on success.

Notes:

- If delth==true, all thrusters associated with the group will be destroyed. Note that this can have side effects if the thrusters were associated with multiple groups, since they are removed from all those groups as well.

DelThrusterGroup (2)

Delete a default thruster group and (optionally) all associated thrusters.

Synopsis:

```
bool DelThrusterGroup (
     THGROUP_TYPE thgt,
     bool delth = false) const
```

Parameters:

thgt	thruster group type (excluding `THGROUP_USER`)
delth	thruster destruction flag

Return value:

true on success

Notes:

- This version can only be used for default thruster groups (< `THGROUP_USER`)

- If delth==true, all thrusters associated with the group will be destroyed. Note that this can have side effects if the thrusters were associated with multiple groups, since they are removed from all those groups as well.

GetThrusterGroupHandle

Returns the handle of one of the default thruster groups, specified by its type.

Synopsis:

```
THGROUP_HANDLE GetThrusterGroupHandle (
     THGROUP_TYPE thgt) const
```

Parameters:

thgt	thruster group type (for a list, see notes to `CreateThrusterGroup`)

Return value:

thruster group handle (or NULL if no group is defined for the specified type).

Notes:

- The thruster group type must not be THGROUP_USER. To retrieve the handle of a nonstandard thruster group, use `GetUserThrusterGroupHandleByIndex`.

GetUserThrusterGroupHandleByIndex

Returns the handle of a user-defined (nonstandard) thruster group specified by its index.

Synopsis:

```
THGROUP_HANDLE GetUserThrusterGroupHandleByIndex (
     DWORD idx) const
```

Parameters:

idx	index of user-defined thruster group

Return value:

thruster group handle

Notes:

- Use this method only to retrieve handles for nonstandard thruster groups (created with the THGROUP_USER flag). For standard groups, use GetThrusterGroupHandle instead.
- The index must be in the range between 0 and nuserthgroup-1, where nuserthgroup is the number of nonstandard thruster groups. Use GetUserThrusterGroupCount to obtain this value.

GetUserThrusterGroupCount

Returns the number of user-defined (nonstandard) thruster groups.

Synopsis:

```
DWORD GetUserThrusterGroupCount (void) const
```

Return value:

number of user-defined thruster groups.

Notes:

- The value returned by this method only includes user-defined thruster groups (created with the `THGROUP_USER` flag). It does not contain any of the standard thruster groups (such as `THGROUP_MAIN`, etc.)

SetThrusterGroupLevel (1)

Set the thrust level for all thrusters in a group.

Synopsis:

```
void SetThrusterGroupLevel (
     THGROUP_HANDLE thg,
     double level) const
```

Parameters:

thg	thruster group identifier
level	new thruster level

SetThrusterGroupLevel (2)

Set the thrust level for all thrusters in a standard group.

Synopsis:

```
void SetThrusterGroupLevel (
     THGROUP_TYPE thgt,
     double level) const
```

Parameters:

thgt	thruster group type
level	new thruster level

Notes:

- This method can only be used for standard thruster group types (the types listed in `CreateThrusterGroup` except `THGROUP_USER`).

IncThrusterGroupLevel (1)

Increment the thrust level for all thrusters in a group.

Synopsis:

```
void IncThrusterGroupLevel (
     THGROUP_HANDLE thg,
     double dlevel) const
```

Parameters:

thg	thruster group identifier
dlevel	thrust level increment

Notes:

- Thrust levels will automatically be truncated to the range [0..1]
- Use negative dlevel to decrement the thrust level.

IncThrusterGroupLevel (2)

Increment the thrust level for all thrusters in a standard group.

Synopsis:

```
void IncThrusterGroupLevel (
     THGROUP_TYPE thgt,
     double dlevel) const
```

Parameters:

thgt	thruster group type
dlevel	thrust level increment

Notes:

- This method can only be used for standard thruster group types (the types listed in `CreateThrusterGroup` except `THGROUP_USER`).
- Thrust levels will automatically be truncated to the range [0..1]
- Use negative dlevel to decrement the thrust level.

GetThrusterGroupLevel (1)

Retrieve the average thrust level for a thruster group.

Synopsis:

```
double GetThrusterGroupLevel (THGROUP_HANDLE thg) const
```

Parameters:

thg	thruster group identifier

Return value:

Average thrust level [0..1]

Notes:

- This function is probably only useful if all thrusters in the group have the same maximum thrust rating, otherwise it is difficult to interpret the average value.

GetThrusterGroupLevel (2)

Retrieve the average thrust level for a default thruster group.

Synopsis:

```
double GetThrusterGroupLevel (THGROUP_TYPE thgt) const
```

Parameters:

thgt	thruster group type

Return value:

Average thrust level [0..1]

GetManualControlLevel

Returns the thrust level of an attitude thruster group requested by the user via keyboard or joystick input.

Synopsis:

```
double VESSEL::GetManualControlLevel (
     THGROUP_TYPE thgt,
     DWORD mode = MANCTRL_ATTMODE,
     DWORD device = MANCTRL_ANYDEVICE) const
```

Parameters:

thgt	thruster group identifier
mode	attitude control mode (see notes)
device	input device (see notes)

Return value:

Manual level for the specified thruster group (0..1)

Notes:

- device can be one of the following:
  MANCTRL_KEYBOARD: retrieve keyboard thrust input
  MANCTRL_JOYSTICK: retrieve joystick thrust input
  MANCTRL_ANYDEVICE: retrieve input from any device

- mode can be one of the following:
  MANCTRL_ATTMODE: retrieve level for the vessel's current attitude mode
  MANCTRL_ROTMODE: retrieve level for rotational modes only
  MANCTRL_LINMODE: retrive level for linear modes only
  MANCTRL_ANYMODE: retrieve level for rotational and linear modes
- If mode is not *MANCTRL_ANYMODE*, only thruster groups which are of the specified mode (linear or rotational) will return nonzero values.

AddExhaust (1)

Add an exhaust render definition for a thruster.

Synopsis:

```
UINT AddExhaust (
     THRUSTER_HANDLE th,
     double lscale,
     double wscale,
     SURFHANDLE tex = 0) const
```

Parameters:

th	thruster identifier
lscale	exhaust flame size (length) [m]
wscale	exhaust flame size (width) [m]
tex	texture handle for custom exhaust flames

Return value:

Exhaust identifier

Notes:

- Thrusters defined with *CreateThruster* do not by default render exhaust effects, until an exhaust definition has been specified with *AddExhaust*.
- The size of the exhaust flame is automatically scaled by the thrust level.
- This version retrieves exhaust reference position and direction directly from the thruster setting, and will therefore automatically reflect any changes caused by *SetThrusterRef* and *SetThrusterDir*.
- To use a custom exhaust texture, set tex to a surface handle returned by *oapiRegisterExhaustTexture*. If *tex* == 0, the default texture is used.

See also:

CreateThruster, *SetThrusterRef*, *SetThrusterDir*, *SetThrusterLevel*, *oapiRegisterExhaustTexture*

AddExhaust (2)

Add an exhaust render definition for a thruster with explicit reference position and direction.

Synopsis:

```
UINT AddExhaust (
     THRUSTER_HANDLE th,
     double lscale,
     double wscale,
     const VECTOR3 &pos,
     const VECTOR3 &dir,
     SURFHANDLE tex = 0) const
```

Parameters:

th	thruster identifier
lscale	exhaust flame size (length) [m]
wscale	exhaust flame size (width) [m]

pos		reference position in the local vessel frame
dir		exhaust direction
tex		texture handle for custom exhaust flames

Return value:
Exhaust identifier

Notes:
- Unlike *AddExhaust* (1), this version uses the explicitly provided reference position and direction, rather than using the thruster parameters.
- This allows multiple exhaust render definitions to refer to a single thruster definition, e.g. where multiple thrusters have been combined into a single “logical” thruster definition. This technique can be used to simplify the description of thruster groups which are always addressed synchronously.
- The exhaust direction should be opposite to the thrust direction of the thruster it refers to.
- Exhaust positions and directions are fixed in this version, so they will not react to changes caused by *SetThrusterRef* and *SetThrusterDir*.
- To use a custom exhaust texture, set tex to a surface handle returned by *oapiRegisterExhaustTexture*. If *tex* == 0, the default texture is used.

See also:
oapiRegisterExhaustTexture

DelExhaust

Removes an exhaust render definition.

Synopsis:
```
bool DelExhaust (UINT idx) const
```

Parameters:
idx		exhaust identifier

Return value:
Error status; *false* if exhaust definition did not exist.

GetMaxThrust

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

Synopsis:
```
double GetMaxThrust (ENGINETYPE eng) const
```

Parameters:
eng		engine group identifier

Return value:
Maximum thrust rating [N]

Notes:
- This function has been replaced by `GetThrusterGroupLevel`.
- For *eng==ENGINE_ATTITUDE*, the function returns the group thrust rating for the *THGROUP_ATT_PITCHUP* group. Other attitude thrust groups may have different parameters.

SetMaxThrust

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

This function has been superseded by *CreateThruster* and *CreateThrusterGroup*. It is retained for backward compatibility and can still be used to define a simplified thruster implementation (see notes).

Synopsis:

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

Parameters:

eng	engine group identifier
th	maximum thrust rating [N]

Notes:

- This method can still be used to implement a simple, idealised thruster configuration, but it should not be mixed with the new thruster functions *CreateThruster* and *CreateThrusterGroup*.
- In the context of the new thruster interface, this function now performs the following functions:

eng	action
`ENGINE_MAIN`	`thr = CreateThruster (_V(0,0,0), _V(0,0,1), th);` `CreateThrusterGroup (&thr, 1, THGROUP_MAIN);`
`ENGINE_RETRO`	`thr = CreateThruster (_V(0,0,0), _V(0,0,-1), th);` `CreateThrusterGroup (&thr, 1, THGROUP_RETRO);`
`ENGINE_HOVER`	`thr = CreateThruster (_V(0,0,0), _V(0,1,0), th);` `CreateThrusterGroup (&thr, 1, THGROUP_HOVER);`
`ENGINE_ATTITUDE`	This creates a complete set of linear and rotational attitude thrusters and attitude thruster groups (see below)

- Calling *SetMaxThrust* for *ENGINE_ATTITUDE* will create all 12 *THGROUP_ATT_xxx* groups (see *CreateThrusterGroup*) and add one thruster to each linear group (max. rating `th`), and 2 thrusters to each rotational group (max. rating ½ `th` each), creating 18 thrusters in total. Any previous *THGROUP_ATT_xxx* definitions will be overwritten. Thrusters are mounted in an 'ideal' configuration, such that linear groups do not induce angular moments, and rotational groups do not induce linear moments. All linear thrusters are mounted in the centre of gravity, all rotational thrusters are mounted at a distance of *Size* from the centre of gravity. (This means that the vessel's size must have been set by a previous call to *SetSize*).

SetISP

Sets a default Isp value for subsequently created thrusters.

Synopsis:

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

Parameters:

isp	fuel-specific impulse [m/s].

Notes:

- The Isp defines the amount of thrust [N] obtained by burning 1 kg of fuel per second. (or conversely, the amount of fuel consumed to attain a given thrust level)
- The effect of this function has changed from v.020419: previously it redefined the global Isp value for all thrusters. Now it only takes effect for subsequently defined thrusters which do not explicitly specify their own Isp rating (see *CreateThruster*).
- Before the first call to *SetISP*, the default Isp value is $5 \cdot 10^4$ m/s.

See also:

CreateThruster, *SetThrusterISP*

GetISP

Returns vessel's current default 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.

Notes:

- The effect of this function has changed from v.020419: previously it returned the global Isp value for all thrusters. Now it returns the current default Isp value which will be used for all subsequently defined thrusters which do not define individual Isp settings.
- To obtain an actual Isp value for a thruster, use *GetThrusterISP*.

See also:

SetISP, *GetThrusterISP*

SetEngineLevel

Obsolete. Sets the thrust level for an engine group.
This function has been replaced by *SetThrusterGroupLevel*.

Synopsis:

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

Parameters:

eng	engine group identifier
level	thrust level (0..1)

Notes:

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

IncEngineLevel

Obsolete. Increase or decrease the thrust level for an engine group.
This function has been replaced by *IncThrusterGroupLevel*.

Synopsis:

```
void IncEngineLevel (ENGINETYPE eng, double dlevel) const
```

Parameters:

eng	engine group identifier
dlevel	thrust increment

Notes:

- Use negative dlevel to decrease the engine's thrust level.
- Levels are clipped to valid range.

GetEngineLevel

Obsolete. Returns the thrust level for an engine group.
This function has been replaced by *GetThrusterGroupLevel*.

Synopsis:

```
double GetEngineLevel (ENGINETYPE 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.

GetMainThrustModPtr

Obsolete. This function is no longer supported.

AddExhaustRef

Obsolete. Replaced by AddExhaust.

DelExhaustRef

Obsolete. Replaced by DelExhaust.

ClearExhaustRefs

Deletes all exhaust render definitions.

Synopsis:

```
void ClearExhaustRefs (void)
```

Notes:

- This function clears the render definitions for all thrusters, but does not affect the physical thruster behaviour. To remove thrusters physically, use ClearThrusterDefinitions instead.

AddAttExhaustRef

Obsolete. Adds an exhaust render definition for an attitude thruster. This function is only retained for backward compatibility and may be removed in a future version. Use AddExhaust instead.

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.

See also:

AddExhaust

AddAttExhaustMode

Obsolete. Assign an attitude thruster to an attitude mode. This function is only retained for backward compatibility and may be removed in a future version. Use AddExhaust instead.

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

See also:

AddExhaust

ClearAttExhaustRefs

Obsolete. Replaced by DelExhaust, DelThruster and ClearThrusterDefinitions. This function does no longer have any effect.

11.7 Docking port management

CreateDock

Create a new docking port.

Synopsis:

```
DOCKHANDLE CreateDock (
     const VECTOR3 &pos,
     const VECTOR3 &dir,
     const VECTOR3 &rot) const
```

Parameters:

pos	dock reference position in vessel coordinates
dir	approach direction in vessel coordinates
rot	longitudinal rotation alignment vector

Return value:

dock handle

Notes:

- The `dir` and `rot` vectors should be normalised to length 1.
- The `rot` vector should be perpendicular to the `dir` vector.
- When two vessels connect at their docking ports, the relative orientation of the vessels is defined such that their respective approach direction vectors (`dir`) are anti-parallel, and their longitudinal alignment vectors (`rot`) are parallel.

DelDock

Delete a previously defined docking port.

Synopsis:

```
bool DelDock (DOCKHANDLE hDock) const
```

Parameters:

hDock	dock handle

Return value:

false indicates failure (invalid dock handle)

Notes:

- Any object docked at the docking port will be undocked before the dock is deleted.

ClearDockDefinitions

Delete all docking ports defined for the vessel.

Synopsis:

```
void ClearDockDefinitions (void) const
```

Notes:

- Any docked objects will be undocked before deleting the docking ports.

DockCount

Returns number of docking ports defined for the vessel.

Synopsis:

```
UINT DockCount (void) const
```

Return value:

Number of docking ports.

SetDockParams (1)

Set the parameters for the vessel's primary docking port (port 0), or create a new dock if required.

Synopsis:

```
void SetDockParams (
     const VECTOR3 &pos,
     const VECTOR3 &dir,
```

```
        const VECTOR3 &rot) const
```

Parameters:

pos	dock reference position in vessel coordinates
dir	approach direction in vessel coordinates
rot	longitudinal rotation alignment vector

Notes:

- This function creates a new docking port if none was previously defined. Otherwise it overwrites the parameters for dock 0.
- See CreateDock for additional notes on the parameters.

SetDockParams (2)

Reset the parameters for for a vessel dock.

Synopsis:

```
void SetDockParams (
    DOCKHANDLE dock,
    const VECTOR3 &pos,
    const VECTOR3 &dir,
    const VECTOR3 &rot) const
```

Parameters:

dock	dock identifier
pos	new dock reference position
dir	new approach direction
rot	new longitudinal rotation alignment vector

Notes:

- This function should not be called while the dock is engaged.

GetDockParams

Returns the parameters of a docking port.

Synopsis:

```
void GetDockParams (
    DOCKHANDLE dock,
    VECTOR3 &pos,
    VECTOR3 &dir,
    VECTOR3 &rot) const;
```

Parameters:

dock	dock handle
pos	dock reference position
dir	approach direction
rot	longitudinal rotation alignment vector

GetDockHandle

Returns a handle to a docking port.

Synopsis:

```
DOCKHANDLE GetDockHandle (UINT n) const
```

Parameters:

n	docking port index (≥ 0)

Return value:

dock handle, or NULL if index was out of range.

GetDockStatus

Returns a handle to a docked vessel.

Synopsis:

```
OBJHANDLE GetDockStatus (DOCKHANDLE dock) const
```

Parameters:

dock	dock handle

Return value:

Handle to vessel docked at the specified port, or NULL if no vessel is docked at that port.

DockingStatus

Returns a flag indicating whether a given dock is engaged.

Synopsis:

```
UINT DockingStatus (UINT port) const
```

Parameters:

port	docking port index (≥0)

Return value:

port status: 0 = free, 1 = docked

Notes:

- This function has the same functionality as `(GetDockStatus (GetDockHandle(port)) ? 1:0)`

Undock

Release a docked vessel from a docking port.

Synopsis:

```
bool Undock (UINT n, const OBJHANDLE exclude = 0) const
```

Parameters:

n	docking port index or ALLDOCKS
exclude	optional handle of a vessel to be excluded from undocking

Return value:

true if at least one vessel was released from a port.

Notes:

- If n is set to ALLDOCKS, all docking ports are released simultaneously.
- If exclude is nonzero, this vessel will not be undocked. This is useful for implementing remote undocking in combination with ALLDOCKS.

11.8 Attachment management

Similar to docking ports, attachment points allow to connect two or more vessel objects. There are a few important differences:

- Docking ports establish peer connections, attachments establish parent-child hierarchies: A parent vessel can have multiple attached children, but each child can only be attached to a single parent.
- Attachments use a simplified physics engine: the root parent alone defines the object's trajectory (both for freespace and atmospheric flight). The children are assumed to have no influence on flight behaviour.

- Orbiter establishes docking connections automatically if the docking ports of two vessels are brought close to each other. Attachment connections are only established by API calls.
- Currently, docking connections only work in freeflight. Attachments also work for landed vessels.

Attachment connections are useful for attaching small objects to larger vessels. For example, Orbiter uses attachments to connect payload items to the Space Shuttle's cargo bay or the tip of the RMS manipulator arm (see Orbitersdk\samples\Atlantis).

Attachment points use an identifier string (up to 8 characters) which can provide a method to establish compatibility. For example, the Atlantis RMS arm tip will only connect to attachment points with an id string that contains "GS" in the first 2 characters (it ignores the last 6 characters).
Now let's assume somebody creates another Shuttle (say a Buran) with its own RMS arm. He could then allow it to

- grapple exactly the same objects as Atlantis, by checking for "GS".
- grapple a subset of objects grapplable by Atlantis, by checking additional characters, for example "GSX".
- grapple all objects grapplable by Atlantis, plus additional objects, for example by checking for "GS" *or* "GX"
- grapple entirely different objects, for example by checking for "GX".

To connect a satellite into the payload bay, Atlantis uses the id "XS" (This means that the payload bay connection can *not* be used for grappling. To allow a satellite to be grappled *and* stored in the payload bay, it must define both a "GS" and an "XS" attachment point).

CreateAttachment

Define a new attachment point for a vessel.

Synopsis:

```
ATTACHMENTHANDLE CreateAttachment (
     bool toparent,
     const VECTOR3 &pos,
     const VECTOR3 &dir,
     const VECTOR3 &rot,
     const char *id,
     bool loose = false) const
```

Parameters:

toparent	If *true*, the attachment can be used to connect to a parent (i.e. vessel acts as child). Otherwise, attachment is used to connect to a child (i.e. vessel acts a parent).
pos	attachment point position in vessel coordinates
dir	attachment direction in vessel coordinates
rot	longitudinal alignment vector in vessel coordinates
id	compatibility identifier
loose	If *true*, allow loose connections (see notes)

Return value:

Handle to the new attachment point

Notes:

- A vessel can define multiple parent and child attachment points, and can subsequently have multiple children attached, but it can only be attached to a single parent at any one time.
- the *dir* and *rot* vectors should both be normalised to length 1, and they should be orthogonal.
- The identifier string can contain up to 8 characters. It can be used to define compatibility between attachment points.

- If the attachment point is defined as loose, then the relative orientation between the two attached objects is frozen to the orientation between them at the time the connection was established. Otherwise, the two objects snap to the orientation defined by their “dir” vectors.

SetAttachmentParams

Reset attachment position and orientation for an existing attachment point.

Synopsis:

```
void SetAttachmentParams (
     ATTACHMENTHANDLE attachment,
     const VECTOR3 &pos,
     const VECTOR3 &dir,
     const VECTOR3 &rot) const
```

Parameters:

attachment	attachment handle
pos	new attachment point position in vessel coordinates
dir	new attachment direction in vessel coordinates
rot	new longitudinal alignment vector in vessel coordinates

Notes:

- If the parameters of an attachment point are changed while a vessel is attached to that point, the attached vessel will be shifted to the new position automatically.
- the *dir* and *rot* vectors should both be normalised to length 1, and they should be orthogonal.

GetAttachmentParams

Retrieve the parameters of an attachment point.

Synopsis:

```
void GetAttachmentParams (
     ATTACHMENTHANDLE attachment,
     VECTOR3 &pos,
     VECTOR3 &dir,
     VECTOR3 &rot) const
```

Parameters:

attachment	attachment handle
pos	attachment point position
dir	attachment direction
rot	longitudinal alignment vector

GetAttachmentId

Retrieve attachment identifier string.

Synopsis:

```
const char *GetAttachmentId (
     ATTACHMENTHANDLE attachment) const
```

Parameters:

attachment	attachment handle

Return value:

pointer to attachment string (8 characters)

GetAttachmentStatus

Return the current status of an attachment point.

Synopsis:
```
OBJHANDLE GetAttachmentStatus (
     ATTACHMENTHANDLE attachment) const
```

Parameters:
attachment attachment handle

Return value:
Handle of the attached vessel, or NULL if no vessel is attached to this point.

AttachmentCount

Return the number of child or parent attachment points defined for a vessel.

Synopsis:
```
DWORD AttachmentCount (bool toparent) const
```

Parameters:

toparent	If true, return the number of attachment points to parents. Otherwise, return the number of attachment points to children.

Return value:
Number of defined attachment points to connect to parents or to children.

GetAttachmentIndex

Return the list index of a vessel's attachment point defined by its handle.

Synopsis:
```
DWORD GetAttachmentIndex (
     ATTACHMENTHANDLE attachment) const
```

Parameters:
attachment attachment handle

Return value:
List index (≥ 0)

Notes:
- A vessel defines separate lists for child and parent attachment points. Therefore two different attachment points may return the same index.

GetAttachmentHandle

Return the handle of an attachment point identified by its list index.

Synopsis:
```
ATTACHMENTHANDLE GetAttachmentHandle (
     bool toparent, DWORD i) const
```

Parameters:

toparent	If true, return handle for attachment point to parent. Otherwise, return handle for attachment point to child.
i	attachment index

Return value:
Attachment handle

AttachChild

Attach a child vessel to an attachment point.

Synopsis:
```
bool AttachChild (
```

```
        OBJHANDLE child,
        ATTACHMENTHANDLE attachment,
        ATTACHMENTHANDLE child_attachment) const
```

Parameters:

child	handle of child vessel to be attached
attachment	attachment point to which the child is to be attached
child_attachment	attachment point on the child to which we want to attach

Return value:

true indicates success, false indicates failure (child refuses attachment)

Notes:

- The *attachment* handle must refer to an attachment "to child" (i.e. created with toparent=false); the *child_attachment* handle must refer to an attachment "to parent" on the child object (i.e. created with toparent=true). It is not possible to connect two parent or two child attachment points.
- A child can only be connected to a single parent at any one time. If the child is already connected to a parent, the previous parent connection is severed.
- The child may check the parent attachment's id string and, depending on the value, refuse to connect. In that case, the function returns false.

DetachChild

Break an existing attachment to a child.

Synopsis:

```
bool DetachChild (
    ATTACHMENTHANDLE attachment,
    double vel = 0.0) const
```

Parameters:

attachment	attachment handle
vel	separation velocity [m/s]

Return value:

true when detachment is successful, *false* if no child was attached, or if child refuses to detach.

11.9 Orbital elements

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

GetGravityRef

Returns a handle to the main contributor of the gravity field at the vessel's current position.

Synopsis:

```
const OBJHANDLE GetGravityRef () const
```

Return value:

Handle to gravity reference object.

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)

11.10 Surface-relative parameters

GetSurfaceRef

Returns a handle to the closest planet or moon. This is the object to which all surface-relative parameters refer.

Synopsis:

```
const OBJHANDLE GetSurfaceRef () const;
```

Return value:

Handle to surface reference object (planet or moon)

GetAltitude

Returns altitude above closest planet/moon.

Synopsis:

```
double GetAltitude (void) const
```

Return value:

altitude [m]

GetAirspeed

Returns magnitude of the freestream airflow velocity vector measured in ship-relative coordinates.

Synopsis:

```
double GetAirspeed (void) const
```

Return value:

Magnitude of airflow velocity [m/s]

Notes:

- This function also works in the absence of an atmosphere. At low altitudes, the returned value is a ground-speed equivalent. At high altitudes the value diverges from ground speed, since an atmospheric drag effect is assumed.
- This function returns the length of the vector returned by GetShipAirspeedVector.

GetHorizonAirspeedVector

Returns airspeed vector in local horizon coordinates.

Synopsis:

```
bool GetHorizonAirspeedVector (VECTOR3 &v) const
```

Parameters:

v variable receiving airspeed vector [m/s]

Return value:

false indicates error.

Notes:

- This function returns the airspeed vector in the reference frame of the local horizon. x = longitudinal component, y = vertical component, z = latitudinal component.

GetShipAirspeedVector

Returns airspeed vector in the vessel's local coordinates.

Synopsis:

```
bool GetShipAirspeedVector (VECTOR3 &v) const
```

Parameters:

v variable receiving airspeed vector [m/s]

Return value:

false indicates error

Notes:

- This function returns the airspeed vector in local ship coordinates. x = lateral component, y = vertical component, z = longitudinal component.

GetAOA

Returns AOA (angle of attack). This is the pitch angle between the velocity vector and the vessel's longitudinal axis.

Synopsis:

```
double GetAOA (void) const
```

Return value:

angle of attack [rad]

GetSlipAngle

Returns the lateral (yaw) angle between the velocity vector and the vessel's longitudinal axis.

Synopsis:

```
double GetSlipAngle (void) const
```

Return value:

lateral slip angle [rad]

GetPitch

Returns pitch angle in local horizon frame.

Synopsis:

```
double GetPitch (void) const
```

Return value:

pitch angle [rad]

GetBank

Returns bank angle in local horizon frame.

Synopsis:

```
double GetBank (void) const
```

Return value:

bank angle [rad]

11.11 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).

GetSuperstructureCG

Returns the centre of mass of the superstructure to which the vessel belongs, if applicable.

Synopsis:

```
bool GetSuperstructureCG (VECTOR3 &cg) const
```

Parameters:

cg superstructure centre of mass [m,m,m]

Return value:

true if vessel is part of a superstructure, *false* otherwise.

Notes:

- The returned vector is the position of the superstructure centre of mass, in coordinates of the local vessel frame.
- If the vessel is not part of a superstructure, *cg* returns (0,0,0).

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 $\mathbf{r}_{local}$ from local vessel coordinates to a global point $\mathbf{r}_{global}$, the following formula is used:
  $\mathbf{r}_{global} = \mathbf{R}\,\mathbf{r}_{local} + \mathbf{p}_{vessel}$,
  where $\mathbf{p}_{vessel}$ 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.

HorizonRot

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

Synopsis:

```
void HorizonRot (
     const VECTOR3 &rloc,
     VECTOR3 &rhorizon) const
```

Parameters:

rloc	vector in local vessel coordinates (input)
rhorizon	vector in local horizon coordinates (output)

Notes:

- The local horizon frame is defined as follows:
  y is "up" direction (planet centre to vessel centre)
  z is "north" direction
  x is "east" direction

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)

Notes:

- This function maps a point from the vessel's local coordinate system (centered at the vessel CG) into the global ecliptical system (centered at the solar system barycentre).
- The transform has the form
  $\mathbf{p}_{glob} = \mathbf{R}_{vessel}\mathbf{p}_{loc} + \mathbf{p}_{vessel}$
  where $\mathbf{R}_{vessel}$ is the vessel's global rotation matrix (as given by *GetRotationMatrix*), and $\mathbf{p}_{vessel}$ is the vessel position in the global frame.

Global2Local

Performs a transformation from global to local vessel coordinates.

Synopsis:

```
void Global2Local (
    const VECTOR3 &global,
    VECTOR3 &local) const
```

Parameters:

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

Notes:

- This is the inverse transform of *Local2Global*; it maps a point from global ecliptical coordinates into the vessel's local frame.
- The transform has the form
  $\mathbf{p}_{loc} = \mathbf{R}^{-1}_{vessel}(\mathbf{p}_{glob} - \mathbf{p}_{vessel})$
  where $\mathbf{R}_{vessel}$ is the vessel's global rotation matrix (as given by *GetRotationMatrix*), and $\mathbf{p}_{vessel}$ is the vessel position in the global frame.

Local2Rel

Performs a transformation from the local vessel frame to the global ecliptical frame, relative to the vessel's reference body.

Synopsis:

```
void Local2Rel (const VECTOR3 &local, VECTOR3 &rel) const
```

Parameters:

local	point in local vessel coordinates (input)
rel	transformed point in reference body-relative global coordiates (output)

Notes:

- This function maps a point from the vessel's local coordinate system (centered at the vessel CG) into an ecliptical coordinate system centered at the vessel's reference object's CG (e.g. the planet that is currently being orbited).
- A handle to the reference object can be obtained via *VESSEL::GetGravityRef*. The reference object may change if the vessel enters a different object's sphere of influence.
- The transformation has the form
  $\mathbf{p}_{rel} = \mathbf{R}_{vessel}\mathbf{p}_{loc} + \mathbf{p}_{vessel} - \mathbf{p}_{ref}$
  where $\mathbf{R}_{vessel}$ is the vessel's global rotation matrix (as given by *GetRotationMatrix*), and $\mathbf{p}_{vessel}$ and $\mathbf{p}_{ref}$ are the CG positions of the vessel and reference body in the global frame, respectively.

11.12 Atmospheric parameters

GetAtmRef

Returns a handle to the reference body for atmospheric calculations.

Synopsis:

```
const OBJHANDLE GetAtmRef (void) const
```

Return value:

Handle to the celestial body whose atmosphere the vessel is currently moving through, or NULL if the vessel is not inside an atmosphere.

GetAtmTemperature

Returns atmospheric temperature [K] at current vessel position.

Synopsis:

```
double GetAtmTemperature (void) const
```

Return value:

atmospheric temperature [K] at curremt vessel position.

Notes:

- 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/m^3] at current vessel position.

Synopsis:

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

Return value:

atmospheric density [kg/m^3] 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.

GetAtmPressure

Returns static 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.

11.13 Aerodynamics

GetDynPressure

Returns the current dynamic pressure for the vessel.

Synopsis:

```
double GetDynPressure (void) const
```

Return value:

Current vessel dynamic pressure [Pa].

Notes:

- The dynamic pressure is defined as $q = ½ \rho V^2$ with density ρ and airflow velocity V. Dynamic pressure is an important aerodynamic parameter.

GetMachNumber

Returns the vessel's current Mach number.

Synopsis:

```
double GetMachNumber (void) const
```

Return value:

Mach number – the ratio of current freestream airflow velocity over speed of sound.

Notes:

- The speed of sound depends on several parameters, e.g. atmospheric composition and temperature. The Mach number can therefore vary even if the airspeed is constant.

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.
- The cw coefficients are only used if no airfoils are defined (see CreateAirfoil), in which case the flight model reverts to legacy parasite drag calculation.

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.
- The cw coefficients are only used if no airfoils are defined (see CreateAirfoil), in which case the flight model reverts to legacy parasite drag calculation.

SetRotDrag

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

Synopsis:

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

Parameters:

rd	drag components for rotation around the 3 vessel axes

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 $a^{(\omega)}_{x,y,z} = -\omega^{(\omega)}_{x,y,z}\, q\, S_y\, r_{x,y,z}$ with angular velocity $\omega^{(\omega)}$, dynamic pressure q, and reference surface S_y, defined by the vessel's cross section projected along the vertical (y) axis.

CreateAirfoil

Define the lift and drag characteristics of an airfoil.

Synopsis:

```
void CreateAirfoil (
     AIRFOIL_ORIENTATION align,
     const VECTOR3 &ref,
     AirfoilCoeffFunc cf,
     double c,
     double S,
     double A) const
```

Parameters:

align	lift vector orientation (`LIFT_VERTICAL` or `LIFT_HORIZONTAL`)
ref	lift and drag vector attack point
cf	pointer to coefficient callback function (see notes)
c	airfoil chord length [m]
S	wing area [m^2]
A	wing aspect ratio

Notes:

- A vessel can define multiple airfoils (for wings, main body, tail stabilisators, etc.). In general, it should define at least one vertical and one horizontal component.
- Airfoil definitions for wings and horizontal stabilisers set align to `LIFT_VERTICAL`. Vertical stabilisers (vertical tail fin, etc) set align to `LIFT_HORIZONTAL`.
- The location of the attack point (together with the moment coefficient) is important for the aerodynamic stability of the vessel. Usually the attack point

will be aft of the CG, and the moment coefficient will have a negative slope around the trim angle of attack.

- The AirfoilCoeffFunc is a callback function which must be supplied by the module which calculates the lift, moment and drag coefficients for the airfoil. It has the following interface:

```
void AirfoilCoeffFunc (
      double aoa, double M, double Re,
      double *cl, double *cm, double *cd)
```

  and returns the lift coefficient (cl), moment coefficient (cm) and drag coefficient (cd) as a function of angle of attack aoa [rad], Mach number M and Reynolds number Re. Note that aoa can range over the full circle ($-\pi$ to π). For vertical lift components, aoa is the pitch angle of attack (α), while for horizontal components it is the yaw angle of attack (β). Some useful functions for calculating the coefficients can be found in Section 17.7.
- If the wing area S is set to 0, then Orbiter uses the projected vessel cross sections to define a reference area. Let $v = (v_x, v_y, v_z)$ be the unit vector of freestream air flow in vessel coordinates. Then the reference area is calculated as $S = v_z C_z + v_y C_y$ for a LIFT_VERTICAL airfoil, and as $S = v_z C_z + v_x C_x$ for a LIFT_HORIZONTAL airfoil, where C_x, C_y, C_z are the vessel cross-sections in x, y and z direction, respectively.
- The wing aspect ratio is defined as defined as $A = b^2/S$ with wing span b.
- A vessel should typically define its airfoils in the ovcSetClassCaps callback function. If no airfoils are defined, Orbiter will fall back to its legacy (pre-030601) drag calculation, using the cw coefficients defined in SetCW. Legacy lift calculation is no longer supported.
- For more details, see the Programmer's Guide.

CreateAirfoil2

Identical to CreateAirfoil, but returns a handle for the new airfoil.

Synopsis:

```
AIRFOILHANDLE CreateAirfoil2 (
     AIRFOIL_ORIENTATION align,
     const VECTOR3 &ref,
     AirfoilCoeffFunc cf,
     double c,
     double S,
     double A) const
```

Parameters:

See CreateAirfoil.

Return value:

Handle for the new airfoil.

Notes:

- Use this function if you need to reference the airfoil later (e.g. to delete it).

DelAirfoil

Delete a previously defined airfoil.

Synopsis:

```
bool DelAirfoil (AIRFOILHANDLE hAirfoil) const
```

Parameters:

hAirfoil	airfoil handle

Return value:

false indicated failure (invalid handle)

Notes:

- Avoid deleting all airfoils without creating new ones, because this will cause Orbiter to revert to the obsolete legacy atmospheric flight model.

ClearAirfoilDefinitions

Remove all airfoil definitions currently defined for the vessel.

Synopsis:

```
void ClearAirfoilDefinitions (void) const
```

Notes:

- This function is useful if a vessel needs to re-define all its airfoil definitions as a result of a structural change.
- After clearing all airfoils, you should generate new ones. Even wingless objects (such as capsules) should define their aerodynamic behaviour by airfoils (see CreateAirfoil). Vessels without airfoil definitions revert to the obsolete legacy atmospheric flight model.

CreateControlSurface

Create an airfoil control surface (elevator, rudder, aileron, flaps, etc.) which allows atmospheric flight control.

Synopsis:

```
void CreateControlSurface (
     AIRCTRL_TYPE type,
     double area,
     double dCl,
     const VECTOR3 &ref,
     int axis = AIRCTRL_AXIS_AUTO,
     UINT anim = (UINT)-1) const
```

Parameters:

type	Control type. This is a member of the `AIRCTRL_TYPE` enumeration type (see notes).
area	control surface area [m^2]
dCl	shift in lift coefficient achieved by fully extended control
ref	lift/drag force attack point for the control
axis	Control rotation axis. This is a member of the `AIRCTRL_AXIS_AUTO` enumeration type (see notes).
anim	animation reference, if applicable

Notes:

- The following control types are available:

`AIRCTRL_ELEVATOR`	elevator (pitch control)
`AIRCTRL_RUDDER`	rudder (yaw control)
`AIRCTRL_AILERON`	aileron (bank control)
`AIRCTRL_FLAP`	flaps

- The following control axis types are available:

`AIRCTRL_AXIS_AUTO`	automatic axis selection
`AIRCTRL_AXIS_YPOS`	+Y axis (vertical)
`AIRCTRL_AXIS_YNEG`	-Y axis (vertical)
`AIRCTRL_AXIS_XPOS`	+X axis (transversal)
`AIRCTRL_AXIS_XNEG`	-X axis (transversal)

where switching between positive and negative axes reverses the effect of the control. Automatic axis control will select the following axes:

Elevator:	XPOS
Rudder:	YPOS
Aileron:	XPOS if ref.x > 0, XNEG otherwise
Flap:	XPOS

- At least 2 control surfaces must be defined for ailerons (e.g. on the left and right wing) with opposite rotation axes, to obtain the angular moment for banking the vessel.
- Elevators will usually use the XPOS axis, assuming the attack point is aft of the CG. If pitch control is provided by a canard configuration *ahead* of the CG, XNEG should be used instead.
- To improve performance, multiple control surfaces may sometimes be defined by a single call to CreateControlSurface. For example, the elevator controls on the left and right wing may be combined by setting a centered attack point.
- Control surfaces can be animated, by passing an animation reference to CreateControlSurface. The animation reference is obtained from a call to CreateAnimation(). The animation should support a state in the range from 0 to 1, with neutral surface position at state 0.5.

SetControlSurfaceLevel

Modify the position of a control surface.

Synopsis:

```
void SetControlSurfaceLevel (
     AIRCTRL_TYPE type,
     double level) const
```

Parameters:

type	Control type. This is a member of the *AIRCTRL_TYPE* enumeration type.
level	new setting (-1 .. 1)

Notes:

- This function is only useful for flap and trim controls, because elevators, rudder and ailerons are normally continuously scanned from the keyboard and joystick inputs and overridden in each frame.

GetControlSurfaceLevel

Retrieve the current position of a control surface.

Synopsis:

```
double GetControlSurfaceLevel (AIRCTRL_TYPE type) const
```

Parameters:

type	Control type. This is a member of the *AIRCTRL_TYPE* enumeration type.

Return value:

Current control position (-1 to 1).

CreateVariableDragElement

Attach a drag force to the vessel whose magnitude is controlled by an external variable which may vary between 0 (no drag) and 1 (full drag). Useful for defining drag produced by movable parts such as landing gear.

Synopsis:

```
void CreateVariableDragElement (
     double *drag,
     double factor,
```

```
        const VECTOR3 &ref) const
```

Parameters:

drag	pointer to external control parameter
factor	drag magnitude scale factor
ref	drag attack point

Notes:

- The magnitude of the drag force is calculated as
  $D = d \cdot f \cdot q_\infty$
  where d is the control parameter (drag), f is the scale factor, and q_∞ is the freestream dynamic pressure.
- Depending on the attack point, the drag force may induce an angular moment.
- Control parameter d should be restricted to values between 0 and 1.

ClearVariableDragElements

Remove all drag components previously defined with CreateVariableDragElement.

Synopsis:

```
void ClearVariableDragElements () const
```

SetWingAspect

Obsolete. This function is part of the legacy aerodynamics model and is retained for backward compatibility only.
It sets the wing aspect ratio (wingspan^2 / wing area).

Synopsis:

```
void SetWingAspect (double aspect) const
```

Parameters:

aspect	wing aspect ratio [dimensionless]

Notes:

- The value defined by this function is only used in legacy mode, i.e. if the vessel does not define any airfoils via CreateAirfoil.
- Default value is 1.0

GetWingAspect

Obsolete. This function is part of the legacy aerodynamics model and is retained for backward compatibility only.
Returns the vessel’s wing aspect ratio (wingspan^2 / wing area).

Synopsis:

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

Return value:

Wing aspect ratio (wingspan^2 / wing area)

Notes:

- The value returned by this function is used by Orbiter only for legacy vessels, i.e. vessels which do not define any airfoils via CreateAirfoil.

SetWingEffectiveness

Obsolete. This function is part of the legacy aerodynamics model and is retained for backward compatibility only.
Sets the wing form factor. Used for lift and drag calculation.

Synopsis:

```
void SetWingEffectiveness (double we) const
```

Parameters:

we wing form factor.

Notes:

- The value defined by this function is only used in legacy mode, i.e. if the vessel does not define any airfoils via CreateAirfoil.
- Typical values are: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings.

GetWingEffectiveness

Obsolete. This function is part of the legacy aerodynamics model and is retained for backward compatibility only.
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:

- The value returned by this function is used by Orbiter only for legacy vessels, i.e. vessels which do not define any airfoils via CreateAirfoil.
- This form factor describes the wing's effectiveness in producing lift in an atmosphere as a function of its shape.

SetLiftCoeffFunc

Obsolete. This function is part of the legacy aerodynamics model and is retained for backward compatibility only.
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 preferred method for defining lift and drag characteristics is via the CreateAirfoil method, which is much more versatile. Orbiter ignores the SetLiftCoeffFunc function if any airfoils have been created with CreateAirfoil.
- The callback function must be able to deal with aoa values in the range $-\pi$ ... π.
- If the function is not installed, the vessel is assumed not to produce any lift.

11.14 Surface contact parameters

SetSurfaceFrictionCoeff

Set the surface friction coefficients in longitudinal and lateral direction.

Synopsis:

```
void SetSurfaceFrictionCoeff (
```

```
          double mu_lng,
          double mu_lat) const
```

Parameters:

mu_lng	longitudinal coefficient
mu_lat	lateral coefficient

Notes:

- The friction forces for each touchdown reference point which intersects the surface are calculated by
  $f = c_F M g$
  where c_F: friction coefficient, M: vessel mass: g: surface g-force
- Vessels with landing gear should define mu_lng < mu_lat. For isotropic surface friction, mu_lng = mu_lat should be used.
- The default values are mu_lng = 0.1, mu_lat = 0.5.

SetMaxWheelbrakeForce

Define the maximum force which can be provided by the vessel's wheel brake system.

Synopsis:

```
void SetMaxWheelbrakeForce (double f) const
```

Parameters:

f	maximum force [N]

SetWheelbrakeLevel

Apply the wheel brake.

Synopsis:

```
void SetWheelbrakeLevel (
     double level,
     int which = 0,
     bool permanent = true) const
```

Parameters:

level	wheelbrake level (0..1)
which	0 = both, 1 = left, 2 = right main gear
permanent	*true* sets the level permanently, *false* only applies to current time step

GetWheelbrakeLevel

Returns the current wheel brake level.

Synopsis:

```
double GetWheelbrakeLevel (int which) const
```

Parameters:

which	0 = average of both main gear levels, 1 = left, 2 = right

Return value:

wheel brake level (0..1)

11.15 Communications/radio interface

InitNavRadios

Defines the number of NAV radio receivers supported by the vessel.

Synopsis:

```
void InitNavRadios (DWORD nnav) const
```

Parameters:

nnav   number of NAV radio receivers

Notes:

- A vessel requires NAV radio receivers to obtain instrument navigation aids such as ILS or docking approach information.
- Typically, a vessel should define 2-3 NAV receivers.
- If no NAV receivers are available, then certain MFD modes such as Landing or Docking will not be supported.
- Default is 2 NAV receivers.

SetNavRecv

Set the frequency step for a NAV receiver.

Synopsis:

```
bool   SetNavRecv (DWORD n, DWORD step) const
```

Parameters:

n   NAV receiver index (≥ 0)

step   frequency step (≥ 0)

Return value:

false if n ≥ nnav (see *InitNavRadios*), otherwise *true*.

Notes:

- NAV radios can be tuned from 108.00 to 140.00 kHz in steps of 0.05 kHz. The frequency corresponding to a receiver step is given by f = 108.0 kHz + step · 0.05 kHz.

GetNavRecv

Returns the frequency step of a NAV receiver.

Synopsis:

```
DWORD  GetNavRecv (DWORD n) const
```

Parameters:

n   NAV receiver index (≥ 0)

Return value:

frequency step (≥ 0). If index n is out of range, the return value is 0.

GetNavRadioFreq

Returns the current radio frequency of a NAV receiver [kHz]

Synopsis:

```
float GetNavRadioFreq (DWORD n) const
```

Parameters:

n   NAV radio index (≥0)

Return value:

NAV radio frequency [kHz]. If index n is out of range then the return value is 0.0.

EnableTransponder

Enable/disable a vessel’s transponder. The transponder is a radio transmitter which can be used by other vessels to obtain navigation information, e.g. for docking rendezvous approaches.

Synopsis:

void EnableTransponder (bool enable) const

Parameters:

enable	flag for enabling/disabling the transponder

11.16 Visual manipulation

ClearMeshes

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

Synopsis:

```
void ClearMeshes () const
```

AddMesh (1)

Loads a new mesh from file and adds it to the vessel's visual representation.

Synopsis:

```
int 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.

Return value:

mesh index

AddMesh (2)

This version adds a preloaded mesh to the vessel's visual representation.

Synopsis:

```
void AddMesh (MESHHANDLE hMesh, const VECTOR3 *ofs=0) const
```

Parameters:

hMesh	mesh handle
ofs	optional pointer to a displacement vector which describes the offset (in meter) of the mesh origin against the vessel origin.

Return value:

mesh index

See also:

oapiLoadMesh

GetMesh

Returns a handle for a vessel mesh given by its index.

Synopsis:

```
MESHHANDLE GetMesh (UINT idx) const
```

Parameters:

idx	mesh index (≥ 0)

Return value:

mesh handle, or NULL if index out of range.

SetMeshVisibilityMode

Defines whether a mesh is visible for cockpit or external camera modes.

Synopsis:

```
void SetMeshVisibilityMode (UINT meshidx, WORD mode) const
```

Parameters:

meshidx	mesh index as returned by *AddMesh*
mode	visibility mode

Notes:

- mode can be a combination of any of the following flags:
  MESHVIS_EXTERNAL: The mesh is rendered in external camera modes (track or ground mode).
  MESHVIS_COCKPIT: The mesh is rendered in internal (cockpit) modes.
  MESHVIS_VC: The mesh is rendered only in virtual cockpit mode
- The default behaviour is *MESHVIS_EXTERNAL* (render in external modes only).
- You can use *MESHVIS_ALWAYS* as a shortcut for *MESHVIS_EXTERNAL* | *MESHVIS_COCKPIT*.
- To render a mesh only in virtual cockpit mode, but not in any other internal modes, use *MESHVIS_VC* instead of *MESHVIS_COCKPIT*.

SetMeshVisibleInternal

Obsolete. This method has been replaced by *SetMeshVisibilityMode*.
Marks a mesh as visible from internal cockpit view.

Synopsis:

```
void SetMeshVisibleInternal (
     UINT meshidx,
     bool visible) const
```

Parameters:

meshidx	mesh index as returned by *AddMesh*
visible	visibility flag

Notes:

- By default, a vessel is not rendered when the camera is in internal (cockpit) view. This function can be used to force rendering of some or all of the vessel's meshes.

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.

MeshgroupTransform

Transform a mesh group of the vessel's visual. Transformations include translation, rotation and scaling.

Synopsis:

```
bool MeshgroupTransform (
     VISHANDLE vis,
     const MESHGROUP_TRANSFORM &mt) const
```

Parameters:

vis	visual handle
mt	transformation parameters

Notes:

- The *MESHGROUP_TRANSFORM* structure is defined as follows:

```
typedef struct {
  union {
    struct {                    // rotation parameters
      VECTOR3 ref;              // rotation axis reference point
      VECTOR3 axis;             // rotation axis direction
      float angle;              // rotation angle (rad)
    } rotparam;
    struct {                    // translation parameters
      VECTOR3 shift;            // translation vector
    } transparam;
    struct {                    // scaling parameters
      VECTOR3 scale;            // scaling factors along coordinate axes
    } scaleparam;
  } P;
  int nmesh;                    // mesh id
  int ngrp;                     // group id
  enum { TRANSLATE, ROTATE, SCALE }
    transform;                  // transform type
} MESHGROUP_TRANSFORM;
```

- If *ngrp* is set to < 0 then the complete mesh is transformed.

SetReentryTexture

Select a previously registered texture to be used for rendering reentry flames.

Synopsis:

```
void SetReentryTexture (
     SURFHANDLE tex,
     double plimit=6e7,
     double lscale=1.0,
     double wscale=1.0) const
```

Parameters:

tex	texture handle
plimit	friction power limit
lscale	texture length scaling factor
wscale	texture width scaling factor

Notes:

- The texture handle is obtained by a previous call to *oapiRegisterReentryTexture*.

- If a custom texture is not explicitly set, Orbiter uses a default texture (reentry.dds) for rendering reentry flames. To suppress reentry flames altogether for a vessel, call *SetReentryTexture(NULL)*.

See also:
oapiRegisterReentryTexture

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.

CreateAnimation

Create a "semi-automatic" animation sequence. The sequence can contain multiple components (rotations, translations, scalings of mesh groups) with a fixed temporal correlation. The animation is driven by manipulating its "state", which is a number between 0 and 1 used to linearly interpolate the animation within its range. See *API User's Guide* for details.

Synopsis:
```
UINT CreateAnimation (double initial_state) const
```

Parameters:
initial_state   the animation state corresponding to the unmodified mesh

Return value:
Animation identifier

Notes:
- Once you have created an animation, use *AddAnimationComponent* to add components.
- Use *SetAnimation* to manipulate the animation state.
- *initial_state* defines at which state the animation is stored in the mesh file. Example: Landing gear animation between retracted state (0) and deployed

state (1). If the landing gear is retracted in the mesh file set *initial_state* to 0. If it is deployed in the mesh file, set *initial_state* to 1.

AddAnimationComponent

Add a component (rotation, translation or scaling of mesh groups) to an animation. Optionally, animations can be stacked hierachically, where transforming a parent recursively also transforms all its children (e.g. a wheel spinning while the landing gear is being retracted).

Synopsis:

```
ANIMATIONCOMPONENT_HANDLE AddAnimationComponent (
     UINT anim,
     double state0,
     double state1,
     MGROUP_TRANSFORM *trans,
     ANIMATIONCOMPONENT_HANDLE parent = NULL) const
```

Parameters:

anim	animation identifier, as returned by *CreateAnimation*
state0	animation cutoff state 0 for the component
state1	animation cutoff state 1 for the component
trans	transformation data (see notes)
parent	parent transformation

Return value:

Animation component handle

Notes:

- *state0* and *state1* (0..1) allow to define the temporal endpoints of the component's animation within the sequence. For example, *state0*=0 and *state1*=1 perform the animation over the whole sequence animation, while *state0*=0 and *state1*=0.5 perform the animation over the first half of the sequence animation. This allows to build complex animations where different components are animated in a defined temporal sequence.
- *MGROUP_TRANSFORM* is the base class for mesh group transforms. The following derived classes are available:

`MGROUP_ROTATE` (rotation)

Constructor:

```
MGROUP_ROTATE (UINT mesh, UINT *grp, UINT ngrp,
     const VECTOR3 &ref, const VECTOR3 &axis,
     float angle)
```

where:

mesh	mesh index (0=first mesh, etc.)
grp	array of mesh group indices
ngrp	number of mesh groups
ref	rotation reference point
axis	rotation axis
angle	angular range of rotation [rad]

`MGROUP_TRANSLATE` (translation)

Constructor:

```
MGROUP_TRANSLATE (UINT mesh, UINT *grp, UINT ngrp,
     const VECTOR3 &shift)
```

where:

mesh	mesh index
grp	array of mesh group indices
ngrp	number of mesh groups

shift   translation vector

`MGROUP_SCALE` (scaling)

Constructor:

```
MGROUP_SCALE (UINT mesh, UINT *grp, UINT ngrp,
     const VECTOR3 &ref, const VECTOR3 &scale)
```

where:

mesh	mesh index
grp	array of mesh group indices
ngrp	number of mesh groups
ref	reference point for scaling origin
scale	scaling factors in x, y and z

- To animate a complete mesh, rather than individual mesh groups, set the "*grp*" pointer to NULL in the constructor of the corresponding *MGROUP_TRANSFORM* operator. The "*ngrp*" value is then ignored.
- To define a transformation as a child of another transformation, set parent to the handle returned by the *AddAnimationComponent* call for the parent.
- Instead of adding mesh groups to an animation, it is also possible to add a local VECTOR3 array. To do this, set "*mesh*" to *LOCALVERTEXLIST*, and set "*grp*" to *MAKEGROUPARRAY(vtxptr)*, where *vtxptr* is the VECTOR3 array. "*ngrp*" is set to the number of vertices in the array. Example:

```
VECTOR3 vtx[2] = {_V(0,0,0), _V(1,0,-1)};
MGROUP_TRANSFORM *mt = new MGROUP_TRANSFORM (LOCALVERTEXLIST,
    MAKEGROUPARRAY(vtx), 2);
AddAnimationComponent (anim, 0, 1, mt);
```

Transforming local vertices in this way does not have an effect on the visual appearance of the animation, but it can be used by the module to keep track of a transformed point during animation. The Atlantis module uses this method to track a grappled satellite during animation of the RMS arm.

Bugs:

- When defining a scaling transformation as a child of a parent rotation, only homogeneous scaling is supported, i.e. *scale.x* = *scale.y* = *scale.z* is required.

DelAnimationComponent

Remove a component from an animation.

Synopsis:

```
bool DelAnimationComponent (
     UINT anim,
     ANIMATIONCOMPONENT_HANDLE hAC)
```

Parameters:

anim	animation identifier
hAC	animation component handle

Return value:

false indicates failure (anim out of range, or hAC invalid)

Notes:

- If the component has children belonging to the same animation, these will be deleted as well.

- In the current implementation, the component must not have children belonging to other animations. Trying to delete such a component will result in undefined behaviour.

SetAnimation

Set the state of an animation.

Synopsis:

```
bool SetAnimation (UINT anim, double state) const
```

Parameters:

anim	animation identifier
state	animation state (0..1)

Return value:

false indicates failure (animation identifier out of range)

Notes:

- Each animation is defined by its state, with extreme points *state*=0 and *state*=1. When setting a state between 0 and 1, Orbiter carries out the appropriate transformations to advance the animation to that state. It is the responsibility of the code developer to call *SetAnimation* in such a way as to provide a smooth movement of the animated parts.

RegisterAnimSequence

Obsolete. This method has been replaced by *CreateAnimation*. It is available for backward compatibility only and will be removed in a future version.

Synopsis:

```
UINT RegisterAnimSequence (double defstate) const
```

Parameters:

defstate	animation state stored in the mesh.

Return value:

Animation sequence identifier.

Notes:

- Unlike *RegisterAnimation*/*UnregisterAnimation*, this function allows to create animation sequences which are processed by the Orbiter core, rather than manually by the module. The user only needs to define the components of the animation sequence once after creating the vessel, using *AddAnimComp*, and can then manipulate the animation state via *SetAnimState*.
- Each animation sequence is defined by its *state*, which has a value between 0 and 1. For example, for an animated landing gear operation *state* 0 may correspond to retracted gears, *state* 1 to fully deployed gears.
- *defstate* defines at which state the animation is stored in the mesh file.

AddAnimComp

Obsolete. This method has been replaced by *AddAnimationComponent*. It is available for backward compatibility only and will be removed in a future version.

Synopsis:

```
bool AddAnimComp (UINT seq, ANIMCOMP *comp)
```

Parameters:

seq	sequence identifier, as returned by *RegisterAnimSequence*

comp	animation component description (see notes)

Return value:

false indicates failure.

Notes:

- *ANIMCOMP* is a structure defining the component's animation:

```
typedef struct {
  UINT *grp;                     // array of group indices to be included in component
  UINT ngrp;                     // number of groups in the grp array
  double state0;                 // animation cutoff state 1
  double state1;                 // animation cutoff state 2
  MESHGROUP_TRANSFORM trans;     // transformation parameters
} ANIMCOMP;
```

- For a complete description of the *MESHGROUP_TRANSFORM* structure see method *VESSEL::MeshgroupTransform*.
- Note that in this case the *angle* or *shift* fields in *MESHGROUP_TRANSFORM* describe the *range* of animation, e.g. the angle over which a landing gear is rotated from fully retracted to fully deployed.
- *state0* and *state1* (0..1) allow to define the temporal endpoints of the component's animation within the sequence. For example, *state0*=0 and *state1*=1 perform the animation over the whole sequence animation, while *state0*=0 and *state1*=0.5 perform the animation over the first half of the sequence animation.

RecordEvent

Not implemented yet.

11.17 Particle systems

AddExhaustStream (1)

Add a particle stream definition to generate an exhaust stream for a vessel. Exhaust streams can be emissive (to simulate "glowing" ionised gases) or diffuse (e.g. for simulating vapour trails).

Synopsis:

```
PSTREAM_HANDLE AddExhaustStream (
      THRUSTER_HANDLE th,
      PARTICLESTREAMSPEC *pss = 0) const
```

Parameters:

th	thruster handle to which the exhaust stream is linked.
pss	particle stream specification

Return value:

Handle to the newly created particle stream.

Notes:

- The *PARTICLESTREAMSPEC* structure is defined in section 8. More details can be found in the Programmer's Guide.
- Multiple streams can be defined for a single engine. For example, an emissive stream with short lifetime may represent the ionised exhaust gases, while a diffuse stream with longer lifetime represents the vapour trail.
- To improve performance, closely packed engines may share a single exhaust stream.
- If the user has disabled particle streams in the launchpad dialog, this function will return NULL. The module must be able to cope with this case.

AddExhaustStream (2)

Add a particle stream definition to generate an exhaust stream for a vessel. This version allows to specify an independent reference point for particle emission.

Synopsis:

```
PSTREAM_HANDLE AddExhaustStream (
     THRUSTER_HANDLE th,
     const VECTOR3 &pos,
     PARTICLESTREAMSPEC *pss = 0) const
```

Parameters:

th	thruster handle to which the exhaust stream is linked.
pos	particle emission reference point
pss	particle stream specification

Return value:

Handle to the newly created particle stream.

Notes:

- This version allows to pass an explicit particle emission reference position, independent of the engine reference point.
- If the user has disabled particle streams in the launchpad dialog, this function will return NULL. The module must be able to cope with this case.

AddReentryStream

Add a particle stream definition to generate a reentry stream for a vessel.

Synopsis:

```
PSTREAM_HANDLE AddReentryStream (
     PARTICLESTREAMSPEC *pss) const
```

Parameters:

pss	particle stream specification

Return value:

Handle to the newly created particle stream.

Notes:

- Vessels automatically define a default emissive particle stream, but you may want to add further stream to customise the appearance.

DelExhaustStream

Delete a previously added particle stream.

Synopsis:

```
bool DelExhaustStream (PSTREAM_HANDLE ch) const
```

Parameters:

ch	particle stream handle

Return value:

false indicates failure (particle stream does not exist)

Notes:

- If a thruster is deleted (with *DelThruster*), any attached particle streams are deleted automatically.
- A deleted particle stream will no longer emit particles, but existing particles persist until they expire.

12 VESSEL class extensions

Additions to the *VESSEL* interface are implemented by a chain of classes derived from *VESSEL*. Each interface in the chain inherits all methods of the previous classes. New interfaces may add addtional callback or query functions. You should always derive your own vessel class from the most recent interface in the chain. Older interfaces will remain valid for backward comparison, unless explicitly stated.

12.1 Class VESSEL2

Inheritance:

VESSEL → VESSEL2

The *VESSEL2* class adds a variety of callback functions to the *VESSEL* interface (*clbkXXX*). These are called by Orbiter to notify the vessel about different types of events and allow it to react to them. The *VESSEL2* class implements these as virtual functions which act as placeholders to be overwritten by derived classes whenever a non-default behaviour is required.
Some of the callback methods defined in this section replace *ovcXXX* vessel module callback functions defined in section 10. In those cases, the default behaviour of *VESSEL2::clbkXXX* functions will be to call the equivalent *ovcXXX* function (if it exists) for backward compatibility. Addon developers should always use the *VESSEL2::clbkXXX* methods in preference over the *ovcXXX* functions.

clbkSetClassCaps

Called after vessel creation, this function allows to set vessel class capabilities and parameters. This can include definition of physical properties (size, mass, docking ports, etc.), creation of propellant resources and engines, aerodynamic parameters, including airfoil definitions, lift and drag properties, or active control surfaces.

<u>Synopsis:</u>

```
void clbkSetClassCaps (FILEHANDLE cfg)
```

<u>Parameters:</u>

cfg	handle for the vessel class configuration file

<u>Default action:</u>

Calls module callback function *ovcSetClassCaps* if present, for backward compatibility.

<u>Notes:</u>

- This function is called after the vessel has been created, but before its state is read from the scenario file. This means that its state (position, velocity, fuel level, etc.) is undefined at this point.
- Use this function to set vessel class capabilities, not vessel state parameters.
- Orbiter will scan the vessel class configuration file for generic parameters (like mass or size) after *clbkSetClassCaps* returns. This allows to override generic caps defined in the module by editing the configuration file.
- The configuration file handle is also passed to *clbkSetClassCaps*, to allow reading of vessel class-specific parameters from file.
- The default action of calling *ovcSetClassCaps* will be dropped in future versions.

clbkLoadStateEx

Called when the vessel needs to load its initial state from a scenario file.

<u>Synopsis:</u>

```
void clbkLoadStateEx (FILEHANDLE scn, void *status)
```

Parameters:

scn	scenario file handle
status	pointer to *VESSELSTATUSx* structure ($x \geq 2$)

Default action:

Calls *ovcLoadStateEx* if defined by the module, for backward compatibility. In *ovcLoadStateEx* doesn't exist, *clbkLoadStateEx* loads the generic vessel state.

Notes:

- This callback function allows to read custom vessel status parameters from a scenario file.
- The function should define a loop which parses lines from the scenario file via *oapiReadScenario_nextline*.
- You should not call the base class *VESSEL2::clbkLoadStateEx* to parse generic parameters, because this will skip over any custom scenario entries. Instead, any lines which the module parser does not recognise should be forwarded to Orbiter's default scenario parser via *VESSEL::ParseScenarioLineEx*.
- Orbiter will always pass the latest supported *VESSELSTATUSx* version to *ovcLoadStateEx*. This is currently *VESSELSTATUS2*, but may change in future versions. To maintain compatibility, `vs` should therefore not be used other than to pass it on to *ParseScenarioLineEx*.
- A typical parser implementation may look like this:

```
void MyVessel::clbkLoadStateEx (FILEHANDLE scn, void *status)
{
    char *line;
    int my_value;

    while (oapiReadScenario_nextline (scn, line)) {
        if (!strnicmp (line, "my_option", 9)) {  // custom item
            sscanf (line+9, "%d", &my_value);
        } else if (...) { // more items
            ...
        } else {          // anything not recognised is passed on to Orbiter
            ParseScenarioLineEx (line, vs);
        }
    }
}
```

See also:

VESSELSTATUS2
VESSEL::ParseScenarioLineEx
oapiReadScenario_nextline

clbkSaveState

Called when the vessel needs to save its current status to a scenario file (typically at the end of a simulation session).

Synopsis:

```
void clbkSaveState (FILEHANDLE scn)
```

Parameters:

scn	scenario file handle

Default action:

Calls *ovcSaveState* if defined by the module, for backward compatibility. If *ovcSaveState* doesn't exist, *clbkSaveState* saves the generic vessel state.

Notes:

- This function only needs to be overloaded if the vessel must save nonstandard parameters.

- If *clbkSaveState* is overloaded, generic state parameters will only be written if the base class *VESSEL2::clbkSaveState* is called.
- To write custom parameters to the scenario file, use the *oapiWriteLine* function.
- The default action of calling *ovcSaveState* will be dropped in future versions.

clbkSetStateEx

This callback function is invoked by Orbiter when a vessel is created during the simulation with a call to *oapiCreateVesselEx*. It allows the vessel to initialise its state according to the provided *VESSELSTATUSx* interface (version x ≥ 2). To allow default initialisation, the status can be passed to *VESSEL::DefSetStateEx*.

Synopsis:

```
void clbkSetStateEx (const void *status)
```

Parameters:

status   pointer to a *VESSELSTATUSx* structure

Default action:

Calls the module's *ovcSetStateEx* callback function if present, to provide backward compatibility.

Notes:

- This callback function receives the *VESSELSTATUSx* structure passed to *oapiCreateVesselEx*. It must therefore be able to process the interface version used by those functions.
- This function remains valid even if future versions of Orbiter introduce new *VESSELSTATUSx* interfaces.
- A typical implementation may look like this:

```
void MyVessel::clbkSetStateEx (const void *status)
{
    // specialised vessel initialisations
    // ...

    // default initialisation:
    DefSetStateEx (status);
}
```

clbkPostCreation

Called after a vessel has been created and its state has been set.

Synopsis:

```
void clbkPostCreation ()
```

Default action:

Calls the module callback function *ovcPostCreation* if present, to provide backward compatibility.

Notes:

- This function can be used to perform the final setup steps for the vessel, such as animation states and instrument panel states. When this function is called, the vessel state (e.g. position, thruster levels, etc.) have been defined.
- The default action of calling *ovcPostCreation* will be dropped in future versions.

clbkPlaybackEvent

Not implemented yet.

clbkFocusChanged

Called after a vessel gained or lost input focus.

Synopsis:

```
void clbkFocusChanged (
     bool getfocus,
     OBJHANDLE hNewVessel,
     OBJHANDLE hOldVessel)
```

Parameters:

getfocus	*true* if the vessel gained focus, *false* if it lost focus
hNewVessel	handle of vessel gaining focus
hOldVessel	handle of vessel losing focus

Default action:

Calls the module callback function *ovcFocusChanged* if present, to provide backward compatibility.

Notes:

- Whenever the input focus is switched to a new vessel (e.g. via user selection F3), this method is called for both the vessel losing focus (*getfocus*=*false*) and the vessel gaining focus (*getfocus*=*true*).
- In both calls, *hNewVessel* and *hOldVessel* are the vessel handles for the vessel gaining and the vessel losing focus, respectively.
- This method is also called at the beginning of the simulation for the initial focus object. In this case *hOldVessel* is NULL.

clbkPreStep

Called at each simulation time step *before* the state is updated to the current simulation time. This function allows to define actions which need to be controlled continuously.

Synopsis:

```
void clbkPreStep (double SimT, double SimDT, double mjd)
```

Parameters:

SimT	next simulation run time (second)
SimDT	step length over which the current state will be integrated (seconds)
mjd	next absolute simulation time (days) in Modified Julian Date format

Default action:

None

Notes:

- This function is called at each frame of the simulation, after the integration step length has been determined, but before the time integration is applied to the current simulation state.
- This function is useful when the step length Δt is required in advance of the time integration, for example to apply a force that produces a given Δv, since the *AddForce* request will be applied in the next update. Using *clbkPostStep* for this purpose would be wrong, because its Δt parameter refers to the previous step length.

```
void MyVessel::clbkPreStep (double simt, double simdt, double mjd)
{
  double F = mass * dv/simdt;
  AddForce(_V(0,0,F), _V(0,0,0));
}
```

See also:

VESSEL2::clbkPostStep, *opcPreStep*, *opcPostStep*

clbkPostStep

Called at each simulation time step *after* the state has been updated to the current simulation time. This function allows to define actions which need to be controlled continuously.

Synopsis:

```
void clbkPostStep (double simt, double simdt, double mjd)
```

Parameters:

simt	current simulation run time (seconds)
simdt	last time step length (seconds)
mjd	absolute simulation time (days) in Modified Julian Date format.

Default action:

Calls the module callback function *ovcTimestep(this,simt)* if present, to provide backward compatibility.

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.
- The default action of calling *ovcTimestep* will be dropped in future versions.

See also:

VESSEL2::clbkPreStep, *opcPreStep*, *opcPostStep*

clbkVisualCreated

Called after a visual representation (a render object) has been created for the vessel.

Synopsis:

```
void clbkVisualCreated (VISHANDLE vis, int refcount)
```

Parameters:

vis	handle for the newly created visual
refcount	visual reference count

Default action:

Calls the module *ovcVisualCreated* callback function if present, for backward compatibility.

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.
- The default action of calling *ovcVisualCreated* will be dropped in future versions.

clbkVisualDestroyed

Called before the visual representation of the vessel is destroyed.

Synopsis:

```
void clbkVisualDestroyed (VISHANDLE vis, int refcount)
```

Parameters:

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

Default action:

Calls the module *ovcVisualDestroyed* callback function if present, for backward compatibility.

Notes:

- Orbiter calls this function before it destroys a visual representation of the vessel. This may be in response to the destruction of the actual vessel, but in general simply means that the vessel has moved out of visual range of the current camera location.
- The default action of calling *ovcVisualDestroyed* will be dropped in future versions.

clbkRCSMode

Called when a vessel's RCS (reaction control system) mode changes. Usually the RCS consists of a set of small thrusters arranged so as to allow controlled attitude changes. In Orbiter, the RCS can be driven in either rotational mode (to change the vessel's angular velocity) or in linear mode (to change its linear velocity), or be switched off.

Synopsis:

```
void clbkRCSMode (int mode)
```

Parameters:

mode new RCS mode: 0=disabled, 1=rotational, 2=linear

Default action:

Calls the module *ovcRCSmode* callback function if present, for backward compatibility.

Notes:

- This callback function is invoked when the user switches RCS mode via the keyboard ("/" or "Ctrl-/" on numerical keypad) or after a call to *VESSEL::SetAttitudeMode* or *VESSEL::ToggleAttitudeMode*.
- Not all vessel types may support a reaction control system. In that case, the callback function can be ignored by the module.

clbkADCtrlMode

Called when user input mode for aerodynamic control surfaces (elevator, rudder, aileron) changes.

Synopsis:

```
void clbkADCtrlMode (DWORD mode)
```

Parameters:

mode control mode

Default action:

Calls module *ovcADCtrlmode* callback function if present. Otherwise no action.

Notes:

- The returned control mode contains bit flags as follows:
  bit 0: elevator enabled/disabled
  bit 1: rudder enabled/disabled
  bit 2: ailerons enabled/disabled

Therefore, *mode*=0 indicates control surfaces disabled, *mode*=7 indicates fully enabled.

clbkNavMode

Called when an automated "navigation mode" is activated or deactivated for a vessel. Most navigation modes engage the vessel's RCS to attain a specific attitude, including pro/retrograde, normal to the orbital plane, level with the local horizon, etc.

Synopsis:

```
void clbkNavMode (int mode, bool active)
```

Parameters:

mode	navmode identifier (see Section 9).
active	true if activated, false if deactivated.

Default action:

Calls the module *ovcNavmode* callback function if present, for backward compatibility.

clbkHUDMode

Called after a change of the vessel's HUD (head-up-display) mode.

Synopsis:

```
void clbkHUDMode (int mode)
```

Parameters:

mode	new HUD mode

Default action:

Calls the module *ovcHUDmode* callback function if present, for backward compatibility.

Notes:

- For currently supported HUD modes see *HUD_xxx* constants in section 9.
- mode *HUD_NONE* indicates that the HUD has been turned off.

clbkMFDMode

Called when the user has switched one of the MFD (multi-functional display) instruments to a different display mode.

Synopsis:

```
void clbkMFDMode (int mfd, int mode)
```

Parameters:

mfd	MFD identifier (see Section 9)
mode	new MFD mode id (see Section 9)

Default action:

Calls the module *ovcMFDmode* callback function if present, for backward compatibility.

clbkDrawHUD

Called when the vessel's head-up display (HUD) needs to be redrawn (usually at each time step, unless the HUD is turned off). Overwriting this function allows to implement vessel-specific modifications of the HUD display (or to suppress the HUD altogether).

Synopsis:

```
void clbkDrawHUD (
     int mode,
     const HUDPAINTSPEC *hps,
     HDC hDC)
```

Parameters:

mode	HUD mode (see *HUD_xxx* constants in section 9).
hps	pointer to a *HUDPAINTSPEC* structure (see notes)
hDC	GDI drawing device context

Default action:

Draws a standard HUD display with Orbiter's default display layout.

Notes:

- If a vessel overwrites this method, Orbiter will draw the default HUD only if the base class *VESSEL::clbkDrawHUD* is called.
- `hps` points to a *HUDPAINTSPEC* structure containing information about the HUD drawing surface. It has the following format:

```
typedef struct {
    int W, H;
    int CX, CY;
    double Scale;
    int Markersize;
} HUDPAINTSPEC;
```

  where *W* and *H* are width and height of the HUD drawing surface in pixels, *CX* and *CY* are the x and y coordinates of the HUD centre (the position of the "forward marker", which is not guaranteed to be in the middle of the drawing surface or even within the drawing surface!), *Scale* represents an angular aperture of 1° expressed in HUD pixels, and *Markersize* is a "typical" size which can be used to scale objects like direction markers.
- The device context passed to *clbkDrawHUD* contains the appropriate settings for the current HUD display (font, pen, colours). If you need to change any of the GDI settings, make sure to restore the defaults before calling the base class *clbkDrawHUD*. Otherwise the default display will be corrupted.
- Try to avoid changing HUD display colours. Orbiter has its own internal mechanism to allow users to switch the HUD colour.
- *clbkDrawHUD* can be used to implement entirely new vessel-specific HUD modes. In this case, the module would maintain its own record of the current HUD mode, and ignore the `mode` parameter passed to *clbkDrawHUD*.

clbkConsumeDirectKey

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

Synopsis:

```
int ovcConsumeDirectKey (char *kstate)
```

Parameters:

kstate	keyboard state

Return value:

A nonzero return value will completely disable default processing of the key state for the current time step. To disable the default processing of selected keys only, use the *RESETKEY* macro (see orbiterapi.h) and return 0.

Default action:

Calls the module *ovcConsumeKey* callback function if present. Otherwise returns 0.

Notes:

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

```
if (KEYDOWN (kstate, OAPI_KEY_F10)) {
  // perform action
  RESETKEY (kstate, OAPI_KEY_F10);
  // optional: prevent default processing of the key
}
```

- This function should be used where a key *state*, rather than a key *event* is required, for example when engaging thrusters or similar. To test for key events (key pressed, key released) use *clbkConsumeBufferedKey* instead.

clbkConsumeBufferedKey

This callback function notifies the vessel of a buffered key event (key pressed or key released).

Synopsis:

```
int ovcConsumeBufferedKey (
    DWORD key,
    bool down,
    char *kstate)
```

Parameters:

key	key scan code (see *OAPI_KEY_xxx* constants in orbiterapi.h)
down	*true* if key was pressed, *false* if key was released
kstate	current keyboard state

Return value:

The function should return 1 if Orbiter's default processing of the key should be skipped, 0 otherwise.

Default action:

Calls the module *ovcConsumeBufferedKey* callback function if present. Otherwise returns 0.

Notes:

- The key state (kstate) can be used to test for key modifiers (Shift, Ctrl, etc.). The *KEYMOD_xxx* macros defined in orbiterapi.h are useful for this purpose.
- This function may be called repeatedly during a single frame, if multiple key events have occurred in the last time step.

clbkDockEvent

Called after a docking or undocking event at one of the vessel's docking ports.

Synopsis:

```
void clbkDockEvent (int dock, OBJHANDLE mate)
```

Parameters:

dock	docking port index
mate	handle to docked vessel, or NULL for undocking event

Default action:

Calls the module *ovcDockEvent* callback function if present. Otherwise no action.

Notes:

- *dock* is the index (≥ 0) of the vessel's docking port at which the docking/undocking event takes place.
- *mate* is a handle to the vessel docking at the port, or NULL to indicate an undocking event.

clbkAnimate

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

Synopsis:

```
void clbkAnimate (double simt)
```

Parameters:

simt	simulation up time (seconds since simulation start)

Default action:

Calls the module ovcAnimate callback function if present. Otherwise no action.

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.

clbkLoadGenericCockpit

Called when the vessel's generic cockpit view (consisting of two "floating" MFD instruments and a HUD, displayed on top of the 3-D render window) is selected by the user pressing F8, or by a function call.

Synopsis:

```
bool clbkLoadGenericCockpit ()
```

Return value:

The function should return *true* if it supports generic cockpit view, *false* otherwise.

Default behaviour:

Sets camera direction to "forward" (0,0,1) and returns *true*.

Notes:

- The generic cockpit view is available for all vessel types by default, unless this function is overwritten to return false.
- Only disable the generic view if the vessel supports either 2-D instrument panels (see `clbkLoadPanel`) or a virtual cockpit (see `clbkLoadVC`). If no valid cockpit view at all is available for a vessel, Orbiter will crash.
- Even if the vessel supports panels or virtual cockpits, you shouldn't normally disable the generic view, because it provides the best performance on slower computers.

clbkLoadPanel

Called when Orbiter tries to switch the cockpit view to a 2-D instrument panel.

Synopsis:

```
bool clbkLoadPanel (int id)
```

Parameters:

id — panel identifier (≥ 0)

Return value:

The function should return *true* if it supports the requested panel, *false* otherwise.

Default action:

Calls `ovcLoadPanel` if defined, for backward compatibility, otherwise returns *false*.

Notes:

- In the body of this function the module should define the panel background bitmap and panel capabilities, e.g. the position of MFDs and other instruments, active areas (mouse hotspots) etc.
- A vessel which implements panels must at least support panel id 0 (the main panel). If any panels register neighbour panels (see `oapiSetPanelNeighbours`), all the neighbours must be supported, too.
- The default action of calling `ovcLoadPanel` will be dropped in future versions.

See also:

`oapiRegisterPanelBackground`, `oapiRegisterPanelArea`, `oapiRegisterMFD`.

clbkPanelMouseEvent

Called when a mouse-activated panel area receives a mouse event.

Synopsis:

```
bool clbkPanelMouseEvent (
     int id,
     int event,
     int mx,
     int my)
```

Parameters:

id	panel area identifier
event	mouse event (see `PANEL_MOUSE_xxx` constants in orbitersdk.h)
mx, my	relative mouse position in area at event

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Default action:

Calls `ovcPanelMouseEvent` if defined, for backward compatibility, otherwise returns *false*.

Notes:

- Mouse events are only sent for areas which requested notification during definition (see `oapiRegisterPanelArea`).
- The default action of calling `ovcPanelMouseEvent` will be dropped in future versions.

clbkPanelRedrawEvent

Called when a registered panel area needs to be redrawn.

Synopsis:

```
bool clbkPanelRedrawEvent (
     int id,
     int event,
     SURFHANDLE surf)
```

Parameters:

id	panel area identifier
event	redraw event (see `PANEL_REDRAW_xxx` constants in orbitersdk.h)
surf	area surface handle

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Default action:

Calls `ovcPanelRedrawEvent` if defined, for backward compatibility, otherwise returns *false*.

Notes:

- This callback function is only called for areas which were not registered with the `PANEL_REDRAW_NEVER` flag.
- All redrawable panel areas receive a `PANEL_REDRAW_INIT` redraw notification when the panel is created, in addition to any registered redraw notification events.
- The surface handle surf contains either the *current area state*, or the *area background*, depending on the flags passed during area registration.
- The surface handle may be used for blitting operations, or to receive a Windows device context (DC) for Windows-style redrawing operations.
- The default action of calling `ovcPanelRedrawEvent` will be dropped in future versions.

See also:

`oapiGetDC`, `oapiReleaseDC`, `oapiTriggerPanelRedrawArea`

clbkLoadVC

Called when Orbiter tries to switch the cockpit view to a 3-D virtual cockpit mode (for example in response to the user switching cockpit modes with F8).

Synopsis:

```
bool clbkLoadVC (int id)
```

Parameters:

id	virtual cockpit identifier (≥ 0)

Return value:

true if the vessel supports the requested virtual cockpit, *false* otherwise.

Default action:

None, returning *false* (i.e. virtual cockpit mode not supported).

Notes:

- In the body of this function the module should define MFD display targets (with `oapiVCRegisterMFD`) and other active areas (with `oapiVCRegisterArea`) for the requested virtual cockpit.

clbkVCMouseEvent

Called when a mouse-activated virtual cockpit area receives a mouse event.

Synopsis:

```
bool clbkVCMouseEvent (int id, int event, VECTOR3 &p)
```

Parameters:

id	area identifier
event	mouse event (see `PANEL_MOUSE_xxx` constants in orbitersdk.h)
p	parameter vector (area type-dependent, see notes)

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Default action:

None, returning false.

Notes:

- To generate a mouse-activated area in a virtual cockpit, you must do the following when registering the area during `clbkLoadVC`:
  - register the area with a call to `oapiVCRegisterArea` with a mouse mode other than `PANEL_MOUSE_IGNORE`.
  - define a mouse-click area in the vessel's local frame. Use one of the `oapiVCRegisterAreaClickmode_XXX` functions. You can define spherical or quadrilateral click areas.
- Parameter p returns information about the mouse position at the mouse event. The type of information returned depends on the area type for which the event was generated:

Area type	**p**
spherical	p.x is distance of mouse event from area centre p.y and p.z not used
quadrilateral	p.x and p.y are the area-relative mouse x and y positions (top left = (0,0), bottom right = (1,1) p.z not used

clbkVCRedrawEvent

Called when a registered virtual cockpit area needs to be redrawn.

Synopsis:

```
bool clbkVCRedrawEvent (int id, int event, SURFHANDLE surf)
```

Parameters:

id	area identifier
event	redraw event (see `PANEL_REDRAW_xxx` constants in orbitersdk.h)

Return value:

The function should return *true* if it processes the event, *false* otherwise.

Default action:

None, returning *false*.

Notes:

- To allow an area of the virtual cockpit to be redrawn dynamically, the area must be registered with `oapiVCRegisterArea` during `clbkLoadVC`, using a redraw mode other than `PANEL_REDRAW_NEVER`.
- When registering the area with `oapiVCRegisterArea`, you must also provide a handle to the texture onto which the redrawn surface is mapped. This texture must be part of the virtual cockpit mesh, and it must be listed in the mesh file with the 'D' ("dynamic") flag (see 3DModel.pdf).

- “Redrawing” an area is not limited to dynamically updating textures. It may also involve mesh transforms (e.g. to animate levers and switches rendered in 3D).

13 Class MFD

This class acts as an interface for user defined MFD (multi functional display) modes. It provides control over keyboard and mouse functions to manipulate the MFD mode, and allows the module to draw the MFD display. The MFD class is a pure virtual class. Each user-defined MFD mode requires the definition of a specialised class derived from MFD. An example for a generic MFD mode implemented as a plugin module can be found in orbitersdk\samples\CustomMFD.

Public member functions

13.1 Construction/creation

MFD

Constructor. Creates a new MFD.

Synopsis:

```
MFD (DWORD w, DWORD h, VESSEL *vessel)
```

Parameters:

w	width of the MFD display (pixel)
h	height of the MFD display (pixel)
vessel	pointer to VESSEL interface associated with the MFD.

Notes:

- MFD is a pure virtual function, so it can’t be instantiated directly. It is used as a base class for specialised MFD modes.
- New MFD modes are registered by a call to oapiRegisterMFDMode. Whenever the new mode is selected by the user, Orbiter sends a OAPI_MSG_MFD_OPENED signal to the message handler, to which the module should respond by creating the MFD mode and returning a pointer to it. Orbiter will automatically destroy the MFD mode when it is turned off.

13.2 Display repaint

Update

Callback function: Orbiter calls this method when the MFD needs to update its display.

Synopsis:

```
virtual void Update (HDC hDC) = 0
```

Parameters:

hDC	Windows device context for drawing on the MFD display surface.

Notes:

- The frequency at which this function is called corresponds to the “MFD refresh rate” setting in Orbiter’s parameter settings, unless a redraw is forced by InvalidateDisplay.
- This function must be overwritten by derived classes.

InvalidateDisplay

Force a display update in the next frame. This function causes Orbiter to call the MFD’s Update method in the next frame.

Synopsis:

```
void InvalidateDisplay ()
```

Title

Displays a title string in the upper left corner of the MFD display.

Synopsis:

```
void Title (HDC hDC, const char *title) const
```

Parameters:

hDC	device context
title	title string (null-terminated)

Notes:

- This method should be called from within Update()
- The title string can contain up to approx. 35 characters when displayed in the default Courier MFD font.
- This method switches the text colour of the GDI context to white.

SelectDefaultFont

Selects a predefined MFD font into the device context.

Synopsis:

```
HFONT SelectDefaultFont (HDC hDC, DWORD i) const
```

Parameters:

hDC	Windows device context
i	font index

Return value:

Windows font handle

Notes:

- Currently supported are font indices 0-2, where
  0 = standard MFD font (Courier, fixed pitch)
  1 = small font (Arial, variable pitch)
  2 = small font, rotated 90 degrees (Arial, variable pitch)
- In principle, an MFD mode may create its own fonts using the standard Windows *CreateFont* function, but using the predefined fonts is preferred to provide a consistent MFD look.
- Default fonts are scaled automatically according to the MFD display size.

SelectDefaultPen

Selects a predefined pen into the device context.

Synopsis:

```
HPEN SelectDefaultPen (HDC hDC, DWORD i) const
```

Parameters:

hDC	Windows device context
i	pen index

Return value:

Windows pen handle

Notes:

- Currently supported are pen indices 0-5, where
  0 = solid, HUD display colour
  1 = solid, light green
  2 = solid, medium green
  3 = solid, medium yellow

4 = solid, dark yellow
5 = solid, medium grey

- In principle, an MFD mode may create its own pen resources using the standard Windows *CreatePen* function, but using predefined pens is preferred to provide a consistent MFD look.

ButtonLabel

Return the label for the specified MFD button.

Synopsis:

```
virtual char *ButtonLabel (int bt)
```

Parameters:

bt button identifier

Return value:

The function should return a 0-terminated character string of up to 3 characters, or NULL if the button is unlabelled.

ButtonMenu

Defines a list of short descriptions for the various MFD mode button/key functions.

Synopsis:

```
virtual int  ButtonMenu (const MFDBUTTONMENU **menu) const
```

Parameters:

menu on return this should point to an array of button menu items. (see notes)

Return value:

number of items in the list

Notes:

- The definition of the MFDBUTTONMENU struct is:
```
typedef struct {
  const char *line1, *line2;
  char selchar;
} MFDBUTTONMENU;
```
containing up to 2 lines of short description, and the keyboard key to trigger the function.
- Each line should contain no more than 16 characters, to fit into the MFD display.
- If the menu item only uses one line, then line2 should be set to NULL.
- menu==0 is valid and indicates that the caller only requires the number of items, not the actual list.
- A typical implementation would be
```
int MyMFD::ButtonMenu (const MFDBUTTONMENU **menu) const
{
  static const MFDBUTTONMENU mnu[2] = {
    {"Select target", 0, 'T'},
    {"Select orbit", "reference", 'R'}
  };
  if (menu) *menu = mnu;
  return 2;
}
```

13.3 Input

ConsumeKeyBuffered

MFD keyboard handler for buffered keys.

Synopsis:

```
virtual bool ConsumeKeyBuffered (DWORD key)
```

Parameters:

key key code (see OAPI_KEY_xxx constants in orbitersdk.h)

Return value:

The function should return true if it recognises and processes the key, false otherwise.

ConsumeKeyImmediate

MFD keyboard handler for immediate (unbuffered) keys.

Synopsis:

```
virtual bool ConsumeKeyImmediate (char *kstate)
```

Parameters:

kstate: keyboard state.

Return value:

The function should return true only if it wants to inhibit Orbiter's default immediate key handler for this time step completely.

Notes:

- The state of single keys can be queried by the KEYDOWN macro.
- The immediate key handler is useful where an action should take place *while a key is pressed*.

ConsumeButton

MFD button handler. This function is called when the user performs a mouse click on a panel button associated with the MFD.

Synopsis:

```
virtual bool ConsumeButton (int bt, int event)
```

Parameters:

bt button identifier.

event mouse event (see PANEL_MOUSE_xxx constants in orbitersdk.h)

Return value:

The function should return true if it processes the button event, false otherwise.

Notes:

- This function is invoked as a response to a call to `oapiProcessMFDButton` in a vessel module.
- Typically, `ConsumeButton` will call `ConsumeKeyBuffered` or `ConsumeKeyImmediate` to emulate a keyboard event.

13.4 Load/save state

WriteStatus

Called when the MFD should write its status to a scenario file.

Synopsis:

```
virtual void WriteStatus (FILEHANDLE scn) const
```

Parameters:

scn scenario file handle (write only)

Notes:

- Use the oapiWriteScenario_xxx functions to write MFD status parameters to the scenario.
- The default behaviour is to do nothing. MFD modes which need to save status parameters should overwrite this function.

ReadStatus

Called when the MFD should read its status from a scenario file.

Synopsis:

```
virtual void ReadStatus (FILEHANDLE scn)
```

Parameters:

scn	scenario file handle (read only)

Notes:

- Use a loop with oapiReadScenario_nextline to read MFD status parameters from the scenario.
- The default behaviour is to do nothing. MFD modes which need to read status parameters should overwrite this function.

StoreStatus

Called before destruction of the MFD mode, to allow the mode to save its status to static memory.

Synopsis:

```
virtual void StoreStatus (void) const
```

Notes:

- This function is called before an MFD mode is destroyed (either because the MFD switches to a different mode, or because the MFD itself is destroyed). It allows the MFD to back up its status parameters, so it can restore its last status when it is created next time.
- Since the MFD mode instance is about to be destroyed, status parameters should be backed up either in static data members, or outside the class instance.
- In principle this function could be implemented by opening a file and calling WriteStatus with the file handle. However for performance reasons file I/O should be avoided in this function.
- The default behaviour is to do nothing. MFD modes which need to save status parameters should overwrite this function.

RecallStatus

Called after creation of the MFD mode, to allow the mode to restore its status from the last save.

Synopsis:

```
virtual void RecallStatus (void)
```

Notes:

- This is the counterpart to the StoreStatus function. It should be implemented if and only if StoreStatus is implemented.

14 Class GraphMFD

This class is derived from MFD and provides a template for MFD modes containing 2D graphs. An example is the ascent profile recorder in the samples\CustomMFD folder.

14.1 Construction/creation

GraphMFD

Constructor. Creates a new GraphMFD.

Synopsis:

```
GraphMFD (DWORD w, DWORD h, VESSEL *vessel)
```

Parameters:

w	width of the MFD display (pixel)
h	height of the MFD display (pixel)
vessel	pointer to VESSEL interface associated with the MFD

14.2 Graph/plot management

AddGraph

Adds a new graph to the MFD.

Synopsis:

```
int AddGraph (void)
```

Return value:

graph identifier

Notes:

- This function allocates data for a new graph. To display plots in the new graph, one or more calls to AddPlot are required.

AddPlot

Adds a plot to an existing graph.

Synopsis:

```
void AddPlot (
    int g,
    float *absc,
    float *data,
    int ndata,
    int col,
    int *ofs = 0)
```

Parameters:

g	graph identifier
absc	pointer to array containing the abscissa (x-axis) values.
data	pointer to array containing the data (y-axis) values.
ndata	number of data points
col	plot colour index
ofs	pointer to data offset (optional)

Notes:

- Data arrays are not copied, so they should not be deleted after the call to AddPlot.
- col is used as an index to select a pen for the plot using the SelectDefaultPen function. Valid range is the same as for SelectDefaultPen.
- If defined, *ofs is the index of the first plot value in the data array. The plot is drawn using the points *ofs to ndata-1, followed by points 0 to *ofs-1. This allows to define continuously updated plots without having to copy blocks of data within the arrays.

SetRange

Sets a fixed range for the x or y axis of a graph.

Synopsis:

```
void SetRange (int g, int axis, float rmin, float rmax)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
rmin	minimum value
rmax	maximum value

Notes:

- The range applies to all plots in the graph.

SetAutoRange

Allows the graph to set its range automatically according to the range of the plots.

Synopsis:

```
void SetAutoRange (int g, int axis, int p = -1)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
p	plot identifier (-1=all)

Notes:

- If p ≥ 0, then p specifies the plot used for determining the graph range. If p = -1, then all of the graph's plots are used to determine the range.

FindRange

Determines the range of an array of data.

Synopsis:

```
void FindRange (
     float *d,
     int ndata,
     float &dmin,
     float &dmax) const
```

Parameters:

d	data array
ndata	number of data
dmin	minimum value on return
dmax	maximum value on return

SetAxisTitle

Sets the title for a given graph and axis.

Synopsis:

```
void SetAxisTitle (int g, int axis, char *title)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)
title	axis title

Notes:

- The MFD may append an extension of the form “x <scale>” to the title, where <scale> is a scaling factor applied to the tick labels of the axis. It is therefore a good idea to finish the title with the units applicable to the data of this axis, so that for example a title “Altitude: km” may become “Altitude: km x 1000”.

SetAutoTicks

Calculates tick intervals for a given graph and axis.

Synopsis:

```
void SetAutoTicks (int g, int axis)
```

Parameters:

g	graph identifier
axis	axis identifier (0=x, 1=y)

Notes:

- This function is called from within SetRange and normally doesn’t need to be called explicitly by derived classes.

Plot

Displays a graph.

Synopsis:

```
void Plot (
    HDC hDC,
    int g,
    int h0,
    int h1,
    const char *title = 0)
```

Parameters:

hDC	Windows device context
g	graph identifier
h0	upper boundary of plot area (pixel)
h1	lower boundary of plot area (pixel)
title	graph title

Notes:

- This function should be called from Update to paint the graph(s) into the provided device context.

15 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.

InitModule

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 InitModule (HINSTANCE hDLL)
```

Parameters:

hDLL	DLL module handle

Notes:

- To guarantee correct initialisation of your module, you must link the Orbitersdk.lib library (found in Orbitersdk\lib) with your plugin project, and add the line
`#define ORBITER_MODULE`
at the beginning of the main source file of your project.
- If Orbitersdk.lib is not linked, the standard Windows entry point `DllMain` will be called instead when the library is loaded.

ExitModule

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 ExitModule (HINSTANCE hDLL)
```

Parameters:

hDLL	DLL module handle

Notes:

- To guarantee correct cleanup of your module, you must link the Orbitersdk.lib library (found in Orbitersdk\lib) with your plugin project, and add the line
`#define ORBITER_MODULE`
at the beginning of the main source file of your project.
- If Orbitersdk.lib is not linked, the standard Windows entry point `DllMain` will be called instead when the library is unloaded.

opcDLLInit

Obsolete. Use `InitModule` instead.

opcDLLExit

Obsolete. Use `ExitModule` instead.

opcOpenRenderViewport

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 opcCloseRenderViewport method is called. Note that for windowed modes the width and height parameters may be smaller than the user-defined window size, to accommodate window borders and title line.

Synopsis:

```
DLLCLBK void opcOpenRenderViewport (
    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

opcCloseRenderViewport

Called before the simulation window is closed.

Synopsis:

```
DLLCLBK void opcCloseRenderViewport (void)
```

opcPreStep

Called at each time step of the simulation, before the state is updated to the current simulation time. This function is only called when the "physical" state of the simulation is propagated in time. opcPreStep is not called while the simulation is paused, even if the user moves the camera.

Synopsis:

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

Parameters:

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

Notes:

- This function is called by Orbiter *after* the new time step (SimDT) and simulation time (SimT) have been calculated, but *before* the simulation state is integrated to SimT. The parameters passed to opcPreStep therefore are the values that *will* be applied in the current simulation step.
- A schematic flow diagram of the frame update loop is given by

> Set $k = 0$, $T_0^{sim} = 0$ and $T_0^{sys} = \langle \text{system time} \rangle$
> Loop
> $k = k + 1$
> $T_k^{sys} = \langle \text{system time} \rangle$
> $\Delta T_k^{sys} = T_k^{sys} - T_{k-1}^{sys}$
> $\Delta T_k^{sim} = \Delta T_k^{sys} \cdot \langle \text{warp factor} \rangle$
> $T_k^{sim} = T_{k-1}^{sim} + \Delta T_k^{sim}$
>   Call opcPreStep ($T_k^{sim}, \Delta T_k^{sim}$)
>   Integrate simulation state from T_{k-1}^{sim} to T_k^{sim}
>   Call opcPostStep ($T_k^{sim}, \Delta T_k^{sim}$)
>   Render scene
> end

- See also opcPostStep.

opcPostStep

Called at each time step of the simulation, after the state has been updated to the current simulation time.

Synopsis:

```
DLLCLBK void opcPostStep (
     double SimT,
     double SimDT,
     double mjd)
```

Parameters:

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

opcTimestep

Obsolete. Replaced by opcPreStep.

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.
- This callback function is also called at the beginning of the simulation, where new_focus is the vessel receiving the initial focus, and old_focus is NULL.
- opcFocusChanged is sent to plugin modules after the vessels receiving and losing focus have been notified via `VESSEL2::clbkFocusChanged`.

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

16 Planet modules

Planet modules can be used to calculate ephemerides (position and velocity) in cases where Orbiter's standard 2-body approximation or dynamic update is not sufficient. By defining a custom module for a planet or moon, more accurate solutions, including semi-analytic perturbation codes, can be implemented. Modules also allow to implement altitude-dependent atmospheric parameters.

See the API Guide manual on how to write a planet module. Typically, during instance initialisation a planet class derived from *CELBODY* will be created, and Orbiter then communicates with the module by calling its overloaded callback functions. The module must be referenced in the planet's configuration file.

The older standalone module callback functions (*opcXXX*) are obsolete and should no longer be used.

16.1 Initialisation functions

The following global functions will be called by Orbiter during module and instance initialisation/cleanup. They require that the module is linked with Orbitersdk\lib\orbitersdk.lib, and defines #define ORBITER_MODULE in its main source file.

InitModule

Called after the DLL is loaded by Orbiter. This happens only once per Orbiter session.

Synopsis:

```
DLLCLBK void InitModule (HINSTANCE hModule)
```

Parameters:

hModule module instance handle

Notes:

- This function is optional. You can use this function to initialise global parameters, if required.
- It is called the first time Orbiter loads a planet referencing this module. It will not be called again if the user exits to the Launchpad and runs another scenario.

ExitModule

Called before Orbiter unloads the DLL. This usually happens when Orbiter is closed.

Synopsis:

```
DLLCLBK void ExitModule (HINSTANCE hModule)
```

Parameters:

hModule module instance handle

Notes:

- This function is optional. You can use it to clean up the module, e.g. by deallocating dynamic data.

InitInstance

Called when Orbiter loads a planet referencing this module.

Synopsis:

```
DLLCLBK CELBODY *InitInstance (OBJHANDLE hBody)
```

Parameters:

hBody object handle for the planet

Return value:

CELBODY-derived class instance

Notes:

- Your module *must* define this function.
- Create an instance of your planet class (derived from CELBODY) here, and return a pointer to it.

ExitInstance

Called after a simulation run when Orbiter destroys the planet.

Synopsis:

```
DLLCLBK void ExitInstance (CELBODY *body)
```

Parameters:

body pointer to planet class

Notes:

- Use this method to destruct the planet class instance created in InitInstance.
- You should cast body to your derived class when deleting the instance, e.g. delete (MyPlanet*)body.

16.2 The CELBODY class

CELBODY defines callback methods which Orbiter will call whenever it requires information from your planet module. You define the behaviour of the planet by overloading the relevant methods. Below is a list of public CELBODY methods:

bEphemeris

Returns true or false depending on whether the module supports ephemeris calculation.

Synopsis:

```
virtual bool bEphemeris() const
```

Return value:

If your module supports ephemeris calculation (that is, if it defines the clbkEphemeris and clbkFastEphemeris methods) return true. Otherwise return false.

Default action:

Returns false.

clbkInit

Called when the planet is initialised at the beginning of a simulation run. This function allows to read any parameters from the configuration file, and perform additional initialisation tasks such as reading data files.

Synopsis:

```
virtual void clbkInit (FILEHANDLE cfg);
```

Parameters:

cfg	file handle of configuration file

Default action:

None.

clbkEphemeris

Called when Orbiter requires (non-sequential) ephemeris data from the planet for a given time.

Synopsis:

```
virtual int clbkEphemeris (
     double mjd,
     int req,
     double *ret)
```

Parameters:

mjd	ephemeris date (days, in Modified Julian Date format)
req	data request bitflags (see notes)
ret	pointer to result vector

Return value:

bitflags describing returned data (see notes)

Default action:

None, returning 0.

Notes:

- The ephemeris data should be calculated with respect to the body's parent body, in the ecliptic frame (J2000 equator and equinox).

- req specifies the data that should be calculated by the callback function. This can be any combination of
  `EPHEM_TRUEPOS` (true body position)
  `EPHEM_TRUEVEL` (true body velocity)
  `EPHEM_BARYPOS` (barycentric position)
  `EPHEM_BARYVEL` (barycentric velocity)
  where the barycentre refers to the system consisting of the body itself and all its children (e.g. moons).
- `ret` is a pointer to an array of 12 doubles, to which the function should write its results:
  `ret[0-2]`: true position (if requested)
  `ret[3-5]`: true velocity (if requested)
  `ret[6-8]`: barycentric position (if requested)
  `ret[9-11]`: barycentric velocity (if requested)
- Data can be returned in either polar or cartesian format. In cartesian format, the position data blocks should contain x,y and z position (in meters), and the velocity data blocks should contain dx/dt, dy/dt and dz/dt (in m/s), where x points to the vernal equinox, y points to ecliptic zenith, and z is orthogonal to both.
  In polar format, the position data blocks should contain longitude φ [rad], latitude θ [rad] and radial distance r [AU], and the velocity data blocks should contain $d\varphi/dt$ [rad/s], $d\theta/dt$ [rad/s] and $d\,r/dt$ [AU/s].
  When returning data in polar format, include the `EPHEM_POLAR` flag in the return value.
- The return value should contain the flags for the data that were actually computed. For example, if both true and barycentric data were requested, but the module can only compute true positions, it should return `EPHEM_TRUEPOS | EPHEM_TRUEVEL`.
- If the true and barycentric positions are identical (that is, if the body has no child objects) the return value should contain the additional flag `EPHEM_TRUEISBARY`.
- If both true and barycentric data are requested, but are computationally expensive to compute (for example, if they require two separate series evaluations), the module can return true positions only. Orbiter will then calculate the barycentric data directly, after evaluating the child object positions.
- If a request can't be satisfied at all (e.g. if barycentric data were requested, but the module can only compute true positions), the module should calculate whatever data it can, and signal so via the return value. Orbiter will then try to convert these data to the required ones.
- If the returned ephemerides are computed in terms of the *barycentre of the parent body's system*, the return value should include the `EPHEM_PARENTBARY` flag. If the ephemerides are computed in terms of the parent body's true position, this flag should not be included.
- This function is not called by Orbiter to update the planet's position during the normal simulation frame update. (For that purpose, *clbkFastEphemeris* is called instead). clbkEphemeris is only called if the planet state at some arbitrary time point is required, e.g. by an instrument calculating a transfer orbit.

clbkFastEphemeris

Called by Orbiter to update the body's state to the next simulation frame.

Synopsis:

```
virtual int clbkFastEphemeris (
     double simt,
     int req,
     double *ret)
```

Parameters:

simt	simulation time (seconds)
req	data request bitflags (see notes)
ret	pointer to result vector

Return value:

bitflags describing returned data (see notes)

Default action:

None, returning 0.

Notes:

- This function should perform the same function as *clbkEphemeris*, but it will be called at each simulation frame. This means that the sampling times will be incremented in small steps, allowing for a potentially more efficient implementation, e.g. by using an interpolation scheme.
- If possible, a full evaluation of a long series of perturbation terms should be avoided here, to avoid performance hits.
- Note that the time parameter is passed in the form of simulation time (seconds) unlike clbkEphemeris, which uses absolute MJD time. This avoids rounding errors in the time variable, and allows higher temporal resolutions.

clbkAtmParam

Called by Orbiter to obtain atmospheric parameters at a given altitude.

Synopsis:

```
virtual bool clbkAtmParam (double alt, ATMPARAM *prm)
```

Parameters:

alt	altitude over planet mean radius
prm	pointer to ATMPARAM structure receiving results

Return value:

true if parameters have been retrieved sucessfully, *false* to indicate that the planet has no atmosphere, or if alt is above the cutoff limit for atmospheric calculations.

Default action:

None, returning *false*.

Notes:

- The ATMPARAM structure contains the following fields:

double T	absolute temperature [K]
double p	pressure [N/m^2]
double rho	density [kg/m^3]

- Currently, atmospheric parameters are assumed to be functions of altitude only. Local variations (“weather”) are not yet supported.

16.3 Orbital parameters

<Planet>_SetPrecision

Obsolete. Set the error limit in *CELBODY::clbkInit* instead.
Define the relative error for the calculations for <Planet>.

Synopsis:

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

Parameters:

prec module-specific

Return value:

0 if successful, < 0 otherwise

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>_Ephemeris

Obsolete. Use *CELBODY::clbkEphemeris* instead.

Calculate ecliptic positions and velocities. 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>_Ephemeris (
     double mjd,
     double *ret,
     int &format)
```

Parameters:

mjd	date in MJD format (MJD = JD-2400000.5)
ret	array of position and velocity data calculated by the function. The type of data depends on the *format* flag (see notes).
format	data format flag (see notes).

Return value:

Error code (not currently used)

Notes:

- Orbiter currently accepts the following data formats:

  EPHEMERIS_POLAR - returned values are polar coordinates and velocities:

  ret[0] = ecliptic longitude [rad]
  ret[1] = ecliptic latitude [rad]
  ret[2] = radius [m]
  ret[3] = velocity in longitude [rad/s]
  ret[4] = velocity in latitude [rad/s]
  ret[5] = radial velocity [m/s]

  EPHEMERIS_CARTESIAN - returned values are cartesian coordinates and velocities:

  ret[0] = x-coordinate (direction of vernal equinox) [m]
  ret[1] = y-coordinate (perpendicular to ecliptic) [m]
  ret[2] = z-coordinate (perpendicular to x and y) [m]
  ret[3] = velocity in x [m/s]
  ret[4] = velocity in y [m/s]

ret[5] = velocity in z [m/s]

When implementing this function, you should calculate the ephemeris data in one of these formats and set the format flag accordingly.

- 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>_FastEphemeris (see below).

<Planet>_FastEphemeris

Obsolete. Use *CELBODY::clbkFastEphemeris* instead.

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>_FastEphemeris (
     double simt,
     double *ret,
     int &format)
```

Parameters:

simt	Time (in seconds) since simulation start
ret	results (as in <Planet>_Ephemeris)
format	data format flag (see <Planet>_Ephemeris for details)

Return value:

currently 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>_Ephemeris (oapiTime2MJD (simt), ret,
  format);
  ```
  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.

16.4 Physical parameters

<Planet>_AtmPrm

Obsolete. Use *CELBODY::clbkAtmParam* instead.

If defined, this function returns atmospheric parameters as a function of altitude above zero ("sea level").

Synopsis:

```
DLLCLBK void <Planet>_AtmPrm (double alt, ATMPARAM *prm)
```

Parameters:

alt	altitude [m]
prm	structure to be filled with atmospheric parameters

Notes:

- The ATMPARAM structure contains the following fields:
  double T absolute temperature [K]
  double p pressure [N/m^2]
  double rho density [kg/m^3]

17 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.

17.1 General functions

oapiGetOrbiterInstance

Returns the instance handle for the running Orbiter application.

Synopsis:

```
HINSTANCE  oapiGetOrbiterInstance ()
```

Return value:

Orbiter instance handle

17.2 Obtaining object handles

oapiGetObjectByName

Retrieve the handle for an object from its name. Objects may be vessels, 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)

Notes:

- This function can not be used to obtain handles for surface bases. Use `oapiGetBaseByName` or `oapiGetBaseByIndex` instead.

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 <= index < *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 <= index < *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

Obsolete. Returns NULL.

Synopsis:

```
OBJHANDLE oapiGetStationByName (char *name)
```

oapiGetStationByIndex

Obsolete. Returns NULL.

Synopsis:

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

oapiGetStationCount

Obsolete. Returns 0.

Synopsis:

```
DWORD oapiGetStationCount (void)
```

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 index is out of range)

oapiGetGbodyCount

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

Synopsis:

```
DWORD oapiGetGbodyCount ()
```

Return value:

Number of objects

oapiGetBaseByName

Returns the handle of a surface base on a given planet or moon.

Synopsis:

```
OBJHANDLE oapiGetBaseByName (OBJHANDLE hPlanet, char *name)
```

Parameters:

hPlanet	handle of the planet or moon on which the base is located
name	base name (not case-sensitive)

Return value:

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

oapiGetBaseByIndex

Returns the handle of a surface base on a given planet or moon from its list index.

Synopsis:

```
OBJHANDLE oapiGetBaseByIndex (OBJHANDLE hPlanet, int index)
```

Parameters:

hPlanet	handle of the planet or moon on which the base is located.
index	object index (≥ 0)

Return value:

object handle. (NULL indicates that the index is out of range)

oapiGetBaseCount

Returns the number of surface bases located on the specified planet.

Synopsis:

```
DWORD oapiGetBaseCount (OBJHANDLE hPlanet)
```

Parameters:

hPlanet	handle of a planet or moon.

Return value:

Number of surface bases.

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 oapiSetFocusObject (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. a planet or moon) will fail.

oapiGetVesselInterface

Returns the VESSEL class interface for a vessel handle.

Synopsis:

```
VESSEL *oapiGetVesselInterface (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

Pointer to VESSEL class interface for this vessel (see section 11).

oapiGetFocusInterface

Returns the VESSEL class interface for the current focus object.

Synopsis:

```
VESSEL *oapiGetFocusInterface ()
```

Return value:

Pointer to VESSEL class interface for focus object (see section 11).

oapiCreateVessel

Creates a new vessel. This version uses the original VESSELSTATUS interface.

Synopsis:

```
OBJHANDLE  oapiCreateVessel (
     const char *name,
     const char *classname,
     const VESSELSTATUS &status)
```

Parameters:

name	vessel name
classname	vessel class name
status	status parameters

Return value:

handle of the newly created vessel

Notes:

- A configuration file for the specified vessel class must exist in the Config subdirectory.
- This function replaces VESSEL::Create().

See also:

oapiCreateVesselEx, *ovcSetState*, *VESSELSTATUS*

oapiCreateVesselEx

Creates a new vessel. This version allows to use a *VESSELSTATUSx* interface (version x ≥ 2).

Synopsis:

```
OBJHANDLE oapiCreateVesselEx (
     const char *name,
     const char *classname,
     const void *status)
```

Parameters:

name	vessel name
classname	vessel class name
status	pointer to a *VESSELSTATUSx* structure

Return value:

- A configuration file for the specified vessel class must exist in the Config subdirectory.
- status must point to a *VESSELSTATUSx* structure. Currently only *VESSELSTATUS2* is supported, but future Orbiter versions may add new interfaces.
- During the vessel creation process Orbiter will call the module's *ovcSetStateEx* callback function if it exists. Orbiter will *not* try to call the *ovcSetState* function.

See also:

oapiCreateVessel, *ovcSetStateEx*, *VESSELSTATUS2*

oapiDeleteVessel

Deletes an existing vessel.

Synopsis:

```
bool oapiDeleteVessel (
     OBJHANDLE hVessel,
     OBJHANDLE hAlternativeCameraTarget = 0)
```

Parameters:

hVessel	vessel handle
hAlternativeCameraTarget	optional new camera target

Return value:

true if vessel could be deleted.

Notes:

- If the current focus vessel is deleted, Orbiter will switch focus to the closest focus-enabled vessel. If the last focus-enabled vessel is deleted, Orbiter returns to the launchpad.
- If the current camera target is deleted, a new camera target can be provided in *hAlternativeCameraTarget*. If not specified, the focus object is used as default camera target.
- The actual vessel destruction does not occur until the end of the current frame. Self-destruct calls are therefore permitted.
- A vessel will undock all its docking ports before being destructed.

17.3 Generic object parameters

oapiGetSize

Returns the size (mean radius) of an object.

Synopsis:

```
double oapiGetSize (OBJHANDLE hObj)
```

Parameters:

hObj object handle

Return value:

Object size (mean radius) in meter.

oapiGetMass

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

Synopsis:

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

Parameters:

hObj object handle

Return value:

object mass [kg]

17.4 Vessel fuel management

oapiGetEmptyMass

Returns empty mass of a vessel, excluding fuel.

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.

oapiGetPropellantHandle

Returns an identifier of a vessel's propellant resource.

Synopsis:

```
PROPELLANT_HANDLE oapiGetPropellantHandle (
    OBJHANDLE hVessel,
    DWORD idx)
```

Parameters:

hVessel vessel handle

idx propellant resource index (≥ 0)

Return value:

propellant resource id, or NULL if idx ≥ # propellant resources

oapiGetPropellantMaxMass

Returns the maximum capacity [kg] of a propellant resource.

Synopsis:

```
double oapiGetPropellantMaxMass (PROPELLANT_HANDLE ph)
```

Parameters:

ph	propellant resource identifier

Return value:

maximum fuel capacity [kg] of the resource.

See also:

oapiGetPropellantHandle(), VESSEL::GetPropellantMaxMass()

oapiGetPropellantMass

Returns the current fuel mass [kg] of a propellant resource.

Synopsis:

```
double oapiGetPropellantMass (PROPELLANT_HANDLE ph)
```

Parameters:

ph	propellant resource identifier

Return value:

current fuel mass [kg] of the resource.

oapiGetFuelMass

Returns current fuel mass of the first propellant resource of a vessel.

Synopsis:

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

Parameters:

hVessel	vessel handle

Return value:

Current fuel mass [kg]

Notes:

- This function is equivalent to `oapiGetPropellantMass (oapiGetPropellantHandle (hVessel, 0))`
- 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 capacity of the first propellant resource of a vessel.

Synopsis:

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

Parameters:

hVessel	vessel handle

Return value:

Maximum fuel mass [kg]

Notes:

- This function is equivalent to `oapiGetPropellantMaxMass (oapiGetPropellantHandle (hVessel, 0))`
- 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)

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.

17.5 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.

oapiGetBarycentre

Returns the global position of the barycentre of a complete planetary system or a single planet-moons system.

Synopsis:

```
void oapiGetBarycentre (OBJHANDLE hObj, VECTOR3 *bary)
```

Parameters:

hObj	celestial body handle
bary	pointer to vector receiving barycentre data

Notes:

- The barycentre is the centre of mass of a distribution of objects. In this case, all involved celestial bodies are considered point masses, and the barycentre is defined as

$$\vec{\mathbf{r}}^B = \left(\sum_i m_i\right)^{-1} \sum_i m_i \vec{\mathbf{r}}_i$$

- *hObj* must be the handle of a celestial body.
- The summation involves the body itself and all its secondaries, e.g. a planet and its moons.
- The barycentre of a star (0^{th} level object) is always the origin (0,0,0).
- The barycentre of an object without associated secondaries is identical to its position.

17.6 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, π/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.

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)

17.7 Aerodynamics

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. The 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/m^3]

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]

oapiGetInducedDrag

This is a helper function which is useful when implementing the callback function calculating the aerodynamics coefficients for an airfoil (see *VESSEL::CreateAirfoil*). It computes the lift-induced component $c_{D,i}$ of the drag coefficient as a function of lift coefficient c_L, wing aspect ratio A, and wing efficiency factor e, as

$$c_{D,i} = \frac{c_L^2}{\pi A e}$$

Synopsis:

```
double oapiGetInducedDrag (double cl, double A, double e)
```

Parameters:

cl	lift coefficient
A	wing aspect ratio
e	wing efficiency factor

Return value:

Induced drag coefficient $c_{D,i}$

Notes:

- The full drag coefficient required by the airfoil callback function consists of several components: profile drag $c_{D,e}$, induced drag $c_{D,i}$ and wave drag $c_{D,w}$
  $$c_D = c_{D,e} + c_{D,i} + c_{D,w}$$
  where $c_{D,e}$ is caused by skin friction and pressure components, and $c_{D,w}$ is a result of the shock wave and flow separation in transonic and supersonic flight.
- The wing aspect ratio is defined as b^2/S, where b is the wing span, and S is the wing area.
- The efficiency factor depends on the wing shape. The most efficient wings are elliptical, with $e = 1$. For all other shapes, $e < 1$.
- This function can be interpreted slightly differently by moving the angle of attack-dependency of the profile drag into the induced drag component:
  $$c_D = c_{D,0} + c'_{D,i} + c_{D,w}$$
  where $c_{D,0}$ is the zero-lift component of the profile drag, and $c'_{D,i}$ is a modified induced drag obtained by replacing the shape factor e with the *Oswald efficiency factor*. See Programmer's Guide for more details.

oapiGetWaveDrag

This is a helper function which is useful when implementing the callback function calculating the aerodynamics coefficients for an airfoil (see *VESSEL::CreateAirfoil*). It uses a simple model to compute the wave drag component of the drag coefficient, $c_{D,w}$. Wave drag significantly affects the vessel drag around Mach 1, and falls off towards lower and higher airspeeds.

This function uses the following model:

$$c_{D,w} = \begin{cases} 0 & \text{if } M < M_1 \\ c_m \dfrac{M - M_1}{M_2 - M_1} & \text{if } M_1 < M < M_2 \\ c_m & \text{if } M_2 < M < M_3 \\ c_m \dfrac{(M_3^2 - 1)^{1/2}}{(M^2 - 1)^{1/2}} & \text{if } M > M_3 \end{cases}$$

where $0 < M_1 < M_2 < 1 < M_3$ are characteristic Mach numbers, and cm is the maximum wave drag coefficient at transonic speeds.

Synopsis:

```
double oapiGetWaveDrag (
    double M,
    double M1, double M2, double M3,
    double cmax)
```

Parameters:

M	current Mach number
M1, M2, M3	characteristic Mach numbers
cmax	maximum wave drag coefficient

Return value:

Wave drag coefficient $c_{D,w}$

Notes:

- The model underlying this function assumes a piecewise linear wave drag profile for $M < M_3$, and a decay with $(M^2-1)^{-1/2}$ for $M > M_3$. If this profile is not suitable for a given airfoil, the programmer must implement wave drag manually.

17.8 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 current attitude thruster mode.

Synopsis:

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

Parameters:

hVessel	vessel handle

Return value:

Current attitude mode (0=disabled or not available, 1=rotational, 2=linear)

Notes:

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

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 (1=rotational, 2=linear, 0=unchanged disabled)

Notes:

- The handle must refer to a vessel. This function does not support other object types.
- This function flips between linear and rotational, but has no effect if attitude thrusters were disabled.

oapiSetAttitudeMode

Set a vessel's attitude thruster mode.

Synopsis:

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

Parameters:

hVessel	vessel handle

mode attitude mode (0=disable, 1=rotational, 2=linear)

Return value:
Error flag; *false* indicates failure (requested mode not available)

Notes:
- The handle must refer to a vessel. This function does not support 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=disabled or not available, 1=rotational, 2=linear)

oapiToggleFocusAttitudeMode

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

Synopsis:
```
int oapiToggleFocusAttitudeMode ()
```

Return value:
The new attitude mode (1=rotational, 2=linear, 0=unchanged disabled)

Notes:
- This function flips between linear and rotational, but has no effect if attitude thrusters were disabled.

oapiSetFocusAttitudeMode

Set the current focus vessel's attitude thruster mode.

Synopsis:
```
bool oapiSetFocusAttitudeMode (int mode)
```

Parameters:
mode attitude mode (0=disable, 1=rotational, 2=linear)

Return value:
Error flag; *false* indicates error (requested mode not available)

oapiRegisterExhaustTexture

Request a custom texture for vessel exhaust rendering.

Synopsis:
```
SURFHANDLE oapiRegisterExhaustTexture (char *name)
```

Parameters:
name exhaust texture file name (without path and extension)

Return value:
texture handle

Notes:
- The exhaust texture must be stored in DDS format in Orbiter's default texture directory.
- If the texture is not found the function returns NULL.

- The texture can be used to define custom textures in *VESSEL::AddExhaust*.

See also:
VESSEL::AddExhaust

oapiRegisterReentryTexture

Request a custom texture for vessel reentry flame rendering.

Synopsis:
```
SURFHANDLE oapiRegisterReentryTexture (char *name)
```

Parameters:

name   reentry texture file name (without path and extension)

Return value:
texture handle

Notes:
- The exhaust texture must be stored in DDS format in Orbiter's default texture directory.
- If the texture is not found the function returns NULL.
- The texture can be used to define custom textures in *VESSEL::SetReentryTexture*.

See also:
VESSEL::SetReentryTexture

17.9 Functions for planetary bodies

All *OBJHANDLE* function parameters used in this section must refer to planetary bodies (planets, moons, astereoids, etc.) unless stated otherwise. Invalid handles may lead to crashes.
Currently, the orientation of planetary rotation axes is assumed time-invariant. Precession, nutation and similar effects are not currently simulated.

oapiGetPlanetPeriod

Returns the rotation period (the length of a siderial day) of a planet.

Synopsis:
```
double oapiGetPlanetPeriod (OBJHANDLE hPlanet)
```

Parameters:

hPlanet   planet handle

Return value:
planet rotation period [seconds]

oapiGetPlanetObliquity

Returns the obliquity of the planet's rotation axis (the angle between the rotation axis and the ecliptic zenith).

Synopsis:
```
double oapiGetPlanetObliquity (OBJHANDLE hPlanet)
```

Parameters:

hPlanet   planet handle

Return value:
obliquity [rad]

Notes:

- In Orbiter, the ecliptic zenith (at epoch J2000) is the positive y-axis of the global frame of reference.

oapiGetPlanetTheta

Returns the longitude of the ascending node of the equatorial plane (denoted by θ), that is, the angle between the vernal equinox and the ascending node of the equator w.r.t. the ecliptic.

Synopsis:

```
double oapiGetPlanetTheta (OBJHANDLE hPlanet)
```

Parameters:

hPlanet	planet handle

Return value:

longitude of ascending node of the equator [rad]

Notes:

- For Earth, this function will return 0. (The ascending node of Earth's equatorial plane is the *definition* of the vernal equinox).

oapiGetPlanetObliquityMatrix

Returns a rotation matrix which performs the transformation from the planet's tilted coordinates into global coordinates.

Synopsis:

```
void oapiGetPlanetObliquityMatrix (
     OBJHANDLE hPlanet,
     MATRIX3 *mat)
```

Parameters:

hPlanet	planet handle
mat	pointer to a matrix receiving the rotation data

Notes:

- The returned matrix is given by

$$R_A = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix}$$

  where θ is the longitude of the ascending node of the equator, as returned by *oapiGetPlanetTheta*, and φ is the obliquity as returned by *oapiGetPlanetObliquity*.
- R_A does not include the current rotation of the planet around its axis. R_A is therefore time-independent.

oapiGetPlanetCurrentRotation

Returns the current rotation angle of the planet around its axis.

Synopsis:

```
double oapiGetPlanetCurrentRotation (OBJHANDLE hPlanet)
```

Parameters:

hPlanet	planet handle

Return value:

Rotation angle [rad]

Notes:

- The complete rotation matrix from planet local to global (ecliptic) coordinates is given by

$$R = R_A \begin{bmatrix} \cos\omega & 0 & -\sin\omega \\ 0 & 1 & 0 \\ \sin\omega & 0 & \cos\omega \end{bmatrix}$$

where R_A is the obliquity matrix as returned by *oapiGetPlanetObliquityMatrix*, and ω is the rotation angle returned by *oapiGetPlanetCurrentRotation*.

oapiPlanetHasAtmosphere

Test for existence of planetary atmosphere.

Synopsis:

```
double oapiPlanetHasAtmosphere (OBJHANDLE hPlanet)
```

Parameters:

hPlanet planet handle

Return value:

true if an atmosphere has been defined for the planet, false otherwise.

oapiGetPlanetAtmConstants

Returns atmospheric constants for a planet.

Synopsis:

```
const ATMCONST *oapiGetPlanetAtmConstants (
    OBJHANDLE hPlanet)
```

Parameters:

hPlanet planet handle

Return value:

pointer to *ATMCONST* structure containing atmospheric coefficients for the planet (see notes)

Notes:

- *ATMCONST* has the following components:

```
typedef struct {
  double p0;          // pressure at mean radius ('sea level') [Pa]
  double rho0;        // density at mean radius [kg/m3]
  double R;           // specific gas constant [J/(K kg)]
  double gamma;       // ratio of specific heats, c_p/c_v
  double C;           // exponent for pressure equation (temporary)
  double O2pp;        // partial pressure of oxygen
  double altlimit;    // atmosphere altitude limit [m]
  double radlimit;    // radius limit (altlimit + mean radius)
  double horizonalt;  // horizon rendering altitude
  VECTOR3 color0;     // sky colour at sea level during daytime
} ATMCONST;
```

- If the specified planet does not have an atmosphere, return value is NULL.

oapiGetPlanetAtmParams

Returns atmospheric parameters as a function of distance from the planet centre.

Synopsis:

```
void oapiGetPlanetAtmParams (
    OBJHANDLE hPlanet,
    double rad,
    ATMPARAM *prm)
```

Parameters:

hPlanet	planet handle
rad	radius from planet centre [m]
prm	pointer to *ATMPARAM* structure receiving parameters

Notes:

- See section 8 for definition of *ATMPARAM* structure.
- If the planet has no atmosphere, or if the defined radius is beyond the defined upper atmosphere limit, all parameters are set to 0.

oapiGetPlanetJCoeffCount

Returns the number of perturbation coefficients defined for a planet to describe the latitude-dependent perturbation of its gaviational potential. A return value of 0 indicates that the planet is considered to have a spherically symmetric gravity field.

Synopsis:

```
DWORD oapiGetPlanetJCoeffCount (OBJHANDLE hPlanet)
```

Parameters:

hPlanet	planet handle

Return value:

Number of perturbation coefficients.

Notes:

- Even if a planet defines perturbation coefficients, its gravity perturbation may be ignored, if the user disabled nonspherical gravity sources, or if orbit stabilisation is active at a given time step. Use the *VESSEL::NonsphericalGravityEnabled* function to check if a vessel uses the perturbation terms in the update of its state vectors.
- Depending on the distance to the planet, Orbiter may use fewer perturbation terms than defined, if their contribution is negligible:

  If $J_n\left(\frac{R}{r}\right)^n < \varepsilon, \quad (n \geq 2)$, ignore all terms $\geq n$,

  where R is the planet radius, r is the distance from the planet, and J_n is the n-2nd perturbation term defined for the planet. Orbiter uses $\varepsilon = 10^{-10}$.

oapiGetPlanetJCoeff

Returns a perturbation coefficient for the calculation of a planet's gravitational potential.

Synopsis:

```
double oapiGetPlanetJCoeff (OBJHANDLE hPlanet, DWORD n)
```

Parameters:

hPlanet	planet handle
n	coefficient index

Return value:

Perturbation coefficient J_{n+2}.

Notes:

- Valid indices n are 0 to *oapiGetPlanetJCoeffCount*()-1.
- Orbiter calculates the planet's gravitational potential U for a given distance r and latitude ϕ by

$$U(r,\phi) = \frac{GM}{r}\left[1 - \sum_{n=2}^{N} J_n \left(\frac{R}{r}\right)^2 P_n(\sin\phi)\right]$$

where R is the planet's equatorial radius, M is its mass, G is the gravitational constant, and P_n is the Legendre polynomial of order n.

- Orbiter currently considers perturbations to be only a function of latitude (polar), not of longitude.
- The first coefficient, $n = 0$, returns J_2, which accounts for the ellipsoid shape of a planet (flattening). Higher perturbation terms are usually small compared to J_2 (and not known for most planets).

17.10 Surface base functions

oapiGetBaseEquPos

Returns the equatorial coordinates (longitude, latitude and radius) of the location of a surface base.

Synopsis:

```
void oapiGetBaseEquPos (
     OBJHANDLE hBase,
     double *lng,
     double *lat,
     double *rad = 0)
```

Parameters:

hBase	surface base handle
lng	pointer to variable to receive longitude value [rad]
lat	pointer to variable to receive latitude value [rad]
rad	pointer to variable to receive radius value [m]

Notes:

- *hBase* must be a valid base handle (e.g. from *oapiGetBaseByName*)
- The radius pointer can be omitted if not required.
- Currently, rad will always return the planet mean radius.

oapiGetBasePadCount

Returns the number of VTOL landing pads owned by the base.

Synopsis:

```
DWORD oapiGetBasePadCount (OBJHANDLE hBase)
```

Parameters:

hBase	surface base handle

Return value:

Number of landing pads

Notes:

- *hBase* must be a valid base handle (e.g. from *oapiGetBaseByName*)
- This function only counts VTOL pads, not runways.

oapiGetBasePadEquPos

Returns the equatorial coordinates (longitude, latitude and radius) of the location of a VTOL landing pad.

Synopsis:

```
bool oapiGetBasePadEquPos (
     OBJHANDLE hBase,
     DWORD pad,
     double *lng,
     double *lat,
     double *rad = 0)
```

Parameters:

hBase	surface base handle
pad	pad index
lng	pointer to variable to receive longitude value [rad]
lat	pointer to variable to receive latitude value [rad]
rad	pointer to variable to receive radius value [m]

Return value:

false indicates failure (pad index out of range). In that case, the return values are undefined.

Notes:

- *hBase* must be a valid base handle (e.g. from *oapiGetBaseByName*)
- 0 ≤ *pad* < *oapiGetBasePadCount*() is required.
- The radius pointer can be omitted if not required.

oapiGetBasePadStatus

Returns the status of a VTOL landing pad (free, occupied or cleared).

Synopsis:

```
bool oapiGetBasePadStatus (
     OBJHANDLE hBase,
     DWORD pad,
     int *status)
```

Parameters:

hBase	surface base handle
pad	pad index
status	pointer to variable to receive pad status

Return value:

false indicates failure (pad index out of range)

Notes:

- hBase must be a valid base handle (e.g. from *oapiGetBaseByName*)
- 0 ≤ *pad* < *oapiGetBasePadCount*() is required.
- status can be one of the following:
  0 = pad is free
  1 = pad is occupied
  2 = pad is cleared for an incoming vessel

oapiGetBasePadNav

Returns a handle to the ILS transmitter of a VTOL landing pad, if available.

Synopsis:

```
NAVHANDLE oapiGetBasePadNav (OBJHANDLE hBase, DWORD pad)
```

Parameters:

hBase	surface base handle
pad	pad index

Return value:

Handle of a ILS transmitter, or NULL if the pad index is out of range or the pad has no ILS.

Notes:

- hBase must be a valid base handle (e.g. from *oapiGetBaseByName*)
- 0 ≤ *pad* < *oapiGetBasePadCount*() is required.

17.11 Navigation radio transmitter functions

oapiGetNavPos

Returns the current position of a NAV transmitter (in global coordinates, i.e. heliocentric ecliptic).

Synopsis:

```
void oapiGetNavPos (NAVHANDLE hNav, VECTOR3 *gpos)
```

Parameters:

hNav	NAV transmitter handle
gpos	pointer to variable to receive global position

oapiGetNavChannel

Returns the channel number of a NAV transmitter.

Synopsis:

```
DWORD oapiGetNavChannel (NAVHANDLE hNav)
```

Parameters:

hNav	NAV transmitter handle

Return value:

channel number

Notes:

- Channel numbers range from 0 to 639.
- To convert a channel number *ch* into a frequency, use $f = (108.0 + 0.05\,ch)$ kHz

oapiGetNavFreq

Returns the frequency of a NAV transmitter.

Synopsis:

```
float oapiGetNavFreq (NAVHANDLE hNav)
```

Parameters:

hNav	NAV transmitter handle

Return value:

Transmitter frequency [kHz]

Notes:

- In Orbiter, NAV transmitter frequencies range from 108.0 to 139.95 kHz and are incremented in 0.05 kHz steps.

oapiGetNavRange

Returns the range of a NAV transmitter.

Synopsis:

```
float oapiGetNavRange (NAVHANDLE hNav)
```

Parameters:

hNav	NAV transmitter handle

Return value:

Transmitter range [m]

Notes:

- A NAV receiver will only receive a signal when within the range of a transmitter.
- Variable receiver sensitivity is not currently implemented.
- Shadowing of a transmitter by obstacles between transmitter and receiver is not currently implemented.

oapiNavInRange

Determines whether a given global coordinate is within the range of a NAV transmitter.

Synopsis:

```
bool oapiNavInRange (NAVHANDLE hNav, const VECTOR3 &gpos)
```

Parameters:

hNav	NAV transmitter handle
gpos	Global coordinates [m,m,m] of a point (cartesian heliocentric ecliptic)

Return value:

true if the point is within range of the transmitter.

17.12 Simulation time

oapiGetSimTime

Retrieve simulation 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 simulation time step (from previous to current frame) in seconds.

Synopsis:

```
double oapiGetSimStep ()
```

Return value:

Simulation time step (seconds)

Notes:

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

oapiGetSysTime

Retrieve system (real) time since simulation start.

Synopsis:

```
double oapiGetSysTime ()
```

Return value:

Real-time simulation up time (seconds)

Notes:

- This function measures the real time elapsed since the simulation was started. Unlike *oapiGetSimTime*, it doesn't take into account time acceleration.

oapiGetSysStep

Retrieve length of last system time step in seconds.

Synopsis:

```
OAPIFUNC double oapiGetSysStep ()
```

Return value:

System time step (seconds)

Notes:

- Unlike *oapiGetSimStep*, this function does not include the time compression factor. It is useful to control actions which do not depend on the simulation time acceleration.

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.

oapiGetFrameRate

Returns current simulation frame rate (frames/sec).

Synopsis:

```
double oapiGetFrameRate (void)
```

Return value:

Current frame rate (fps)

17.13 Camera functions

oapiCameraInternal

Returns flag to indicate internal/external camera mode.

Synopsis:

```
bool oapiCameraInternal (void)
```

Return value:

true indicates an internal camera mode, i.e. the camera is located inside a vessel cockpit. In this case, the camera target is always the current focus object.
false indicates an external camera mode, i.e. the camera points toward an object from outside. The camera target may be a vessel, planet, spaceport, etc.

oapiCameraMode

Returns the current camera view mode.

Synopsis:

```
int oapiCameraMode ()
```

Return value:

CAM_COCKPIT	cockpit (internal) mode
CAM_TARGETRELATIVE	tracking mode (relative direction)
CAM_ABSDIRECTION	tracking mode (absolute direction)
CAM_GLOBALFRAME	tracking mode (global frame)
CAM_TARGETTOOBJECT	tracking mode (target to object)
CAM_TARGETFROMOBJECT	tracking mode (object to target)
CAM_GROUNDOBSERVER	ground observer mode

oapiCockpitMode

Returns the current cockpit display mode.

Synopsis:

```
int oapiCockpitMode ()
```

Return value:

COCKPIT_GENERIC (generic cockpit mode: left+right MFD and HUD)
COCKPIT_PANELS (2D panel mode)
COCKPIT_VIRTUAL (virtual cockpit mode)

Notes:

- This function also works if the camera is not currently in cockpit mode.

oapiCameraTarget

Returns a handle to the current camera target.

Synopsis:

```
OBJHANDLE  oapiCameraTarget (void)
```

Return value:

Handle to the current camera target (i.e. the object the camera is pointing at in external mode, or the handle of the vessel in cockpit mode)

Notes:

- The camera target is not necessarily a vessel, and if it is a vessel, it is not necessarily the focus object (the vessel receiving user input).

oapiCameraGlobalPos

Returns current camera position in global coordinates.

Synopsis:

```
void oapiCameraGlobalPos (VECTOR3 *gpos)
```

Parameters:

gpos	pointer to vector to receive global camera coordinates

Notes:

- The global coordinate system is the heliocentric ecliptic frame at epoch J2000.0.

oapiCameraGlobalDir

Returns current camera direction in global coordinates.

Synopsis:

```
void oapiCameraGlobalDir (VECTOR3 *gdir)
```

Parameters:

gdir	pointer to vector to receive global camera direction

oapiCameraTargetDist

Returns the distance between the camera and its target [m].

Synopsis:

```
double oapiCameraTargetDist (void)
```

Return value:

Distance between camera and camera target [m].

oapiCameraAzimuth

Returns the current camera azimuth angle with respect to the target.

Synopsis:

```
double oapiCameraAzimuth ()
```

Return value:

Camera azimuth angle [rad]. Value 0 indicates that the camera is behind the target.

Notes:

- This function is useful only in external camera mode. In internal mode, it will always return 0.

oapiCameraPolar

Returns the current camera polar angle with respect to the target.

Synopsis:

```
double oapiCameraPolar ()
```

Return value:

Camera polar angle [rad]. Value 0 indicates that the camera is at the same elevation as the target.

Notes:

- This function is useful only in external camera mode. In internal mode, it will always return 0.

oapiCameraAperture

Returns the current camera aperture (the field of view) in rad.

Synopsis:

```
double oapiCameraAperture (void)
```

Return value:

camera aperture [rad]

Notes:

- Orbiter defines the the aperture as ½ of the vertical field of view, between the viewport centre and the top edge of the viewport.

oapiCameraSetAperture

Change the camera aperture (field of view).

Synopsis:

```
void oapiCameraSetAperture (double aperture)
```

Parameters:

aperture	new aperture [rad]

Notes:

- Orbiter restricts the aperture to the range from RAD*5 to RAD*80 (i. e. field of view between 10° and 160°. Very wide angles (> 90°) should only be used to implement specific optical devices, e.g. wide-angle cameras, not for standard observer views.
- The Orbiter user interface does not accept fields of view > 90°. As soon as the user manipulates the aperture manually, it will be clamped back to the range from 10° to 90°.

oapiCameraScaleDist

Moves the camera closer to the target or further away.

Synopsis:

```
void oapiCameraScaleDist (double dscale)
```

Parameters:

dscale	distance scaling factor

Notes:

- Setting dscale < 1 will move the camera closer to its target. dscale > 1 will move it further away.
- This function is ignored if the camera is in internal mode.

oapiCameraRotAzimuth

Rotate the camera around the target (azimuth angle).

Synopsis:

```
void oapiCameraRotAzimuth (double dazimuth)
```

Parameters:

dazimuth	change in azimuth angle [rad]

Notes:

- This function is ignored if the camera is in internal mode.

oapiCameraRotPolar

Rotate the camera around the target (polar angle).

Synopsis:

```
void oapiCameraRotPolar (double dpolar)
```

Parameters:

dpolar	change in polar angle [rad]

Notes:

- This function is ignored if the camera is in internal mode.

oapiCameraSetCockpitDir

Set the camera direction in cockpit mode.

Synopsis:

```
void oapiCameraSetCockpitDir (
     double polar,
     double azimuth,
     bool transition = false)
```

Parameters:

polar	polar angle [rad]
azimuth	azimuth angle [rad]
transition	transition flag (see notes)

Notes:

- This function is ignored if the camera is not currently in cockpit mode.
- The polar and azimuth angles are relative to the default view direction (see *VESSEL::SetCameraDefaultDirection*)
- The requested direction should be within the current rotation ranges (see *VESSEL::SetCameraRotationRange*), otherwise the result is undefined.
- If *transition==false*, the new direction is set instantaneously; otherwise the camera swings from the current to the new direction (not yet implemented).

oapiCameraAttach

Attach the camera to a new target, or switch between internal and external camera mode.

Synopsis:

```
void oapiCameraAttach (OBJHANDLE hObj, int mode)
```

Parameters:

hObj	handle of the new camera target
mode	camera mode (0=internal, 1=external, 2=don't change)

Notes:

- If the new target is not a vessel, the camera mode is always set to external, regardless of the value of mode.

17.14 Keyboard input

oapiAcceptDelayedKey

Obsolete. This function is should no longer be used. See ovcConsumeBufferedKey for handling buffered key events. *May be removed in a future version.*

17.15 Mesh management

oapiLoadMesh

Loads a mesh from file and returns a handle to it.

Synopsis:

```
MESHHANDLE oapiLoadMesh (const char *fname)
```

Parameters:

fname	mesh file name

Return value:

Handle to the loaded mesh. (NULL indicates load error)

Notes:

- The file name should not contain a path or file extension. Orbiter appends extension .msh and searches in the default mesh directory.
- Meshes should be deallocated with *oapiDeleteMesh* when no longer needed.

See also:

oapiDeleteMesh, *VESSEL::AddMesh*

oapiLoadMeshGlobal

Retrieves a mesh handle from the global mesh manager. When called for the first time for any given file name, the mesh is loaded from file and stored as a system resource. Every further request of the same mesh directly returns a handle to the stored mesh without further file I/O.

Synopsis:

```
const MESHHANDLE oapiLoadMeshGlobal (const char *fname)
```

Parameters:

fname	mesh file name

Return value:

mesh handle

Notes:

- Once a mesh is globally loaded it remains in memory until the user closes the simulation window.

- This function can be used to pre-load meshes to avoid load delays during the simulation. For example, parent objects may pre-load meshes for any child objects they may create later.
- Do *NOT* delete any meshes obtained by this function with *oapiDeleteMesh*! Orbiter takes care of deleting globally managed meshes.

oapiDeleteMesh

Removes a mesh from memory.

Synopsis:

```
void oapiDeleteMesh (MESHHANDLE hMesh)
```

Parameters:

hMesh	mesh handle

oapiMeshGroupCount

Returns the number of mesh groups defined in a mesh.

Synopsis:

```
DWORD oapiMeshGroupCount (MESHHANDLE hMesh)
```

Parameters:

hMesh	mesh handle

Return value:

number of mesh groups defined in the mesh

Notes:

- Each mesh is subdivided into mesh groups, defining a part of the 3-D object represented by the mesh.
- A group consists of a list of vertex coordinates and vertex indices, representing its geometry, and optionally a material and a texture reference.
- See *3DModel* document for details of the mesh format.

oapiMeshGroup

Returns a pointer to the group specification of a mesh group.

Synopsis:

```
MESHGROUP *oapiMeshGroup (MESHHANDLE hMesh, DWORD idx)
```

Parameters:

hMesh	mesh handle
idx	group index (≥ 0)

Return value:

pointer to mesh group specification (or NULL if idx out of range)

Notes:

- MESHGROUP is a structure defined as follows:

```
typedef struct {        // mesh group definition
  NTVERTEX *Vtx;        // vertex list
  WORD *Idx;            // index list
  DWORD nVtx;           // vertex count
  DWORD nIdx;           // index count
  DWORD MtrlIdx;        // material index (>= 1, 0=none)
  DWORD TexIdx;         // texture index (>= 1, 0=none)
  DWORD UsrFlag;        // user-defined flag
  WORD zBias;           // z bias
  WORD Flags;           // internal flags
} MESHGROUP;
```

where NVERTEX defines a vertex with normals and texture coordinates:

```
typedef struct {       // vertex definition including normals and texture coordinates
  float x, y, z;       // position
  float nx, ny, nz;    // normal
  float tu, tv;        // texture coordinates
} NVERTEX;
```

- This method can be used to edit the a mesh group directly (for geometry animation, texture animation, etc.)

oapiGetTextureHandle

Retrieve a surface handle for a mesh texture.

Synopsis:

```
OAPIFUNC SURFHANDLE oapiGetTextureHandle (
      MESHHANDLE hMesh,
      DWORD texidx)
```

Parameters:

hMesh	mesh handle
texidx	texture index (≥ 1)

Return value:

surface handle

Notes:

- This function can be used for dynamically updating textures during the simulation.
- the texture index is given by the order in which the textures appear in the texture list at the end of the mesh file.
- Important: Any textures which are to be dynamically modified should be listed with the 'D' flag ("dynamic") in the mesh file. This causes Orbiter to decompress the texture when it is loaded. Blitting operations to compressed surfaces is very inefficient on most graphics hardware.

17.16 Particle stream management

oapiParticleSetLevelRef

Reset the reference pointer used by the particle stream to calculate the intensity (opacity) of the generated particles.

Synopsis:

```
void oapiParticleSetLevelRef (
      PSTREAM_HANDLE ph,
      double *lvl)
```

Parameters:

ph	particle stream handle
lvl	pointer to variable defining particle intensity

Notes:

- The variable pointed to by lvl should be set to values between 0 (lowest intensity) and 1 (highest intensity).
- By default, exhaust streams are linked to the thrust level setting of the thruster they are associated with. Reentry streams are set to a fixed level of 1 by default.
- This function allows to customise the appearance of the particle streams directly by the module.
- Other parameters besides the intensity level, such as atmospheric density can also have an effect on the particle intensity.

17.17 HUD, panel, virtual cockpit and MFD management

oapiSetHUDMode

Set HUD (head up display) mode.

Synopsis:

```
bool oapiSetHUDMode (int mode)
```

Parameters:

mode   new HUD mode

Return value:

true if mode has changed, false otherwise.

Notes:

- Mode *HUD_NONE* will turn off the HUD display.
- See constants *HUD_xxx* (section 9) for currently supported HUD modes.

oapiGetHUDMode

Query current HUD (head up display) mode.

Synopsis:

```
int oapiGetHUDMode (void)
```

Return value:

Current HUD mode

oapiToggleHUDColour

Switch the HUD display to a different colour.

Synopsis:

```
void oapiToggleHUDColour (void)
```

Notes:

- Orbiter currently defines 3 HUD colours: green, red, white. Calls to *oapiToggleHUDColour* will cycle through these.

oapiIncHUDIntensity

Increase the brightness of the HUD display.

Synopsis:

```
void oapiIncHUDIntensity (void)
```

Notes:

- Calling this function will increase the intensity (in virtual cockpit modes) or brightness (in other modes) of the HUD display up to a maximum value.
- This function should be called repeatedly (e.g. while the user presses a key).

oapiDecHUDIntensity

Decrease the brightness of the HUD display.

Synopsis:

```
void oapiDecHUDIntensity (void)
```

Notes:

- Calling this function will decrease the intensity (in virtual cockpit modes) or brightness (in other modes) of the HUD display down to a minimum value.

- This function should be called repeatedly (e.g. while the user presses a key).

oapiOpenMFD

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

Synopsis:

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

Parameters:

mode	MFD mode (see Section 9)
id	MFD identifier (see Section 9)

Notes:

- mode *MFD_NONE* will turn off the MFD.
- For the on-screen instruments, only *MFD_LEFT* and *MFD_RIGHT* are supported. Custom panels may support (up to 3) additional MFDs.

oapiGetMFDMode

Get the current mode of the specified MFD.

Synopsis:

```
int oapiGetMFDMode (int id)
```

Parameters:

id	MFD identifier (see Section 9)

Return value:

MFD mode (see Section 9)

oapiSendMFDKey

Sends a keystroke to an MFD.

Synopsis:

```
int oapiSendMFDKey (int id, DWORD key)
```

Parameters:

id	MFD identifier (see Section 9)
key	key code (see *OAPI_KEY_xxx* constants in orbitersdk.h)

Return value:

nonzero if the MFD understood and processed the key.

Notes:

- This function can be used to interact with the MFD as if the user had pressed Shift-*key*, for example to select a different MFD mode, to select a target body, etc.

oapiProcessMFDButton

Requests a default action as a result of a MFD button event.

Synopsis:

```
virtual bool ProcessMFDButton (
     int mfd,
     int bt,
     int event) const
```

Parameters:

mfd	MFD identifier (see Section 9)
bt	button number (≥ 0)

event	mouse event (a combination of *PANEL_MOUSE_xxx* flags)

Return value:

Returns true if the button was processed, false if no action was assigned to the button.

Notes:

- Orbiter assigns default button actions for the various MFD modes. For example, in *Orbit* mode the action assigned to button 0 is *Select reference*. Calling *oapiProcessMFDButton* (for example as a reaction to a mouse button event) will execute this action.

oapiMFDButtonLabel

Retrieves a default label for an MFD button.

Synopsis:

```
const char *oapiMFDButtonLabel (int mfd, int bt)
```

Parameters:

mfd	MFD identifier (see Section 9)
bt	button number (≥ 0)

Return value:

pointer to static string containing the label, or NULL if the button is not assigned.

Notes:

- Labels contain 1 to 3 characters.
- This function can be used to paint the labels on the MFD buttons of a custom panel.
- The labels correspond to the default button actions executed by *VESSEL::ProcessMFDButton*.

oapiRegisterMFD

Registers an MFD position for a custom panel.

Synopsis:

```
void oapiRegisterMFD (int id, const MFDSPEC &spec)
```

Parameters:

id	MFD identifier (see Section 9)
spec	MFD parameters (see below)

Notes:

- Should be called in the body of *ovcLoadPanel* for panels which define MFDs.
- Defining more than 2 or 3 MFDs per panel can degrade performance.
- *MFDSPEC* is a struct with the following fields:

```
typedef struct {
  RECT pos;              // position of MFD in panel (pixel)
  int nbt_left;          // number of buttons on left side of MFD display
  int nbt_right;         // number of buttons on right side of MFD display
  int bt_yofs;           // y-offset of top button from top display edge (pixel)
  int bt_ydist;          // y-distance between buttons (pixel)
} MFDSPEC;
```

oapiRegisterPanelBackground

Register the background bitmap for a custom panel.

Synopsis:

```
void oapiRegisterPanelBackground (
```

```
        HBITMAP hBmp,
        DWORD flag = PANEL_ATTACH_BOTTOM|PANEL_MOVEOUT_BOTTOM,
        DWORD ck = (DWORD)-1)
```

Parameters:

hBmp	bitmap handle
flag	property bit flags (see notes)
ck	transparency colour key

Notes:

- This function will normally be called in the body of *ovcLoadPanel*.
- Typically the bitmap will be stored as a resource in the DLL and obtained by a call to the Windows function *LoadBitmap*(...).
- flag defines panel properties and can be a combination of the following bitmasks:
  PANEL_ATTACH_{LEFT/RIGHT/TOP/BOTTOM}
  PANEL_MOVOUT_{LEFT/RIGHT/TOP/BOTTOM}
  where *PANEL_ATTACH_BOTTOM* means that the bottom edge of the panel cannot be scrolled above the bottom edge of the screen (other directions work equivalently) and *PANEL_MOVEOUT_BOTTOM* means that the panel can be scrolled downwards out of the screen (other directions work equivalently)
- The colour key, if defined, specifies a colour which will appear transparent when displaying the panel. The key is in (hex) 0xRRGGBB format. If no key is specified, the panel will be opaque. It is best to use black (0x000000) or white (0xffffff) as colour keys, since other values may cause problems in 16bit screen modes. Of course, care must be taken that the keyed colour does not appear anywhere in the opaque part of the panel.

oapiRegisterPanelArea

Defines a rectangular area within a panel to receive mouse or redraw notifications.

Synopsis:

```
void oapiRegisterPanelArea (
    int aid,
    const RECT &pos,
    int draw_event = PANEL_REDRAW_NEVER,
    int mouse_event = PANEL_MOUSE_IGNORE,
    int bkmode = PANEL_MAP_NONE)
```

Parameters:

aid	area identifier
pos	bounding box of the marked area
draw_event	defines redraw events
mouse_event	defines mouse events
bkmode	redraw background mode

Notes:

- Each panel area must be defined with an identifier *aid* which is unique within the panel.
- draw_event can have the following values:
  PANEL_REDRAW_NEVER: do not generate redraw events.
  PANEL_REDRAW_ALWAYS: generate a redraw event at every time step.
  PANEL_REDRAW_MOUSE: mouse events trigger redraw events.
- For possible values of mouse_event see orbitersdk.h.
  PANEL_MOUSE_IGNORE prevents mouse events from being triggered.
- bkmode defines the bitmap handed to the redraw callback:
  PANEL_MAP_NONE: provides an undefined bitmap. Should be used if the whole area is repainted.
  PANEL_MAP_CURRENT: provides a copy of the current area.

PANEL_MAP_BACKGROUND: provides a copy of the panel background (as defined by oapiRegisterPanelBackground).
PANEL_MAP_BGONREQUEST: like *PANEL_MAP_BACKGROUND*, this stores the area background, but the user must request it explicitly with a call to *oapiBltPanelAreaBackground*. This can improve performance if the area does not need to be updated at each call of the repaint callback function.

oapiSetPanelNeighbours

Defines the neighbour panels of the current panels. These are the panels the user can switch to via Ctrl-Arrow keys.

Synopsis:

```
void oapiSetPanelNeighbours (
     int left,
     int right,
     int top,
     int bottom)
```

Parameters:

left	panel id of left neighbour (or –1 if none)
right	panel id of right neighbour (or –1 if none)
top	panel id of top neighbour (or –1 if none)
bottom	panel id of bottom neighbour (or –1 if none)

Notes:

- This function should be called during panel registration (in *ovcLoadPanel*) to define the neighbours of the registered panel.
- Every panel (except panel 0) must be listed as a neighbour by at least one other panel, otherwise it is inaccessible.

oapiTriggerPanelRedrawArea

Triggers a redraw notification for a panel area.

Synopsis:

```
void oapiTriggerPanelRedrawArea (int panel_id, int area_id)
```

Parameters:

panel_id	panel identifier (≥0)
area_id	area identifier (≥0)

Notes:

- The redraw notification is ignored if the requested panel is not currently displayed.

oapiBltPanelAreaBackground

Copies the stored background of a panel area into the provided surface. This function should only be called from within the repaint callback function of an area registered with the *PANEL_MAP_BGONREQUEST* flag.

Synopsis:

```
bool oapiBltPanelAreaBackground (
     int aid,
     SURFHANDLE surf)
```

Parameters:

aid	area identifier
surf	surface handle

Notes:

- Areas defined with the *PANEL_MAP_BGONREQUEST* receive a surface with undefined contents when their repaint callback is called. They can use *oapiBltPanelAreaBackground* to copy the area background into the surface.
- For areas not registered with the *PANEL_MAP_BGONREQUEST*, this function will do nothing.
- Using *PANEL_MAP_BGONREQUEST* is more efficient than *PANEL_MAP_BACKGROUND* if the area doesn't need to be repainted at each call of the callback function, because it delays blitting the background until the module requests the background. This is particularly significant for areas which are updated at each time step.

oapiSwitchPanel

Switch to a neighbour instrument panel in 2-D panel cockpit mode.

Synopsis:

```
int oapiSwitchPanel (int direction)
```

Parameters:

direction neighbour direction (see notes)

Return value:

Identifier of the newly selected panel (≥ 0) or -1 if the requested panel does not exist.

Notes:

- direction can be one of the following:
  `PANEL_LEFT` (switch to panel left of current)
  `PANEL_RIGHT` (switch to panel right of current)
  `PANEL_UP` (switch to panel up from current)
  `PANEL_DOWN` (switch to panel down from current)
- The neighbourhood status between panels is established by the *oapiSetPanelNeighbours* function.
- This function has no effect if the current view is not in 2-D panel cockpit mode.

oapiSetPanel

Switch to a different instrument panel in 2-D panel cockpit mode.

Synopsis:

```
int oapiSetPanel (int panel_id)
```

Parameters:

panel_id panel identifier (≥ 0)

Return value:

panel_id if the panel was set successfully, or -1 if failed (camera not in 2-D panel cockpit mode, or requested panel does not exist for the current vessel)

Notes:

- This function has no effect if the current view is not in 2-D panel cockpit mode.

oapiVCRegisterHUD

Define a render target for the head-up display (HUD) in a virtual cockpit.

Synopsis:

```
void oapiVCRegisterHUD (const VCHUDSPEC *spec)
```

Parameters:

spec	hud specification (see notes)

Notes:

- This function should be placed in the body of the *ovcLoadVC* vessel module callback function.
- *VCHUDSPEC* is a structure defined as

```
struct VCHUDSPEC {
  DWORD nmesh;          // mesh index
  DWORD ngroup;         // group index
  VECTOR3 hudcnt;       // HUD centre in vessel frame
  double size;          // physical size of the HUD [m]
};
```

- The mesh group specified by nmesh and ngroup should be a square panel in front of the camera position in the virtual cockpit. This group is rendered separately from the rest of the mesh and should therefore have FLAG 2 set in the mesh file. The group material and texture can be set to 0.
- The HUD centre position and size are required to allow Orbiter to correctly scale the display.
- Orbiter renders the HUD with completely transparent background. Rendering the glass pane, brackets, etc. is up to the vessel designer.

oapiVCRegisterMFD

Define a render target for rendering an MFD display in a virtual cockpit.

Synopsis:

```
void oapiVCRegisterMFD (int mfd, const VCMFDSPEC *spec)
```

Parameters:

mfd	MFD identifier
spec	render target specification (see notes)

Notes:

- The render target specification is defined as a structure:
  `struct VCMFDSPEC { DWORD nmesh, ngroup };`
  where nmesh is the mesh index (≥ 0), and ngroup is the group index (≥ 0) defining the render target.
- This function should be placed in the body of the *ovcLoadVC* vessel module callback function.
- The addressed mesh group should define a simple square (4 vertices, 2 triangles). The group materials and textures can be set to 0.

oapiVCRegisterArea (1)

Define an active area in a virtual cockpit. Active areas can be repainted. This function is similar to *oapiRegisterPanelArea*.

Synopsis:

```
void oapiVCRegisterArea (
     int aid,
     const RECT &tgtrect,
     int draw_event,
     int mouse_event,
     int bkmode,
     SURFHANDLE tgt)
```

Parameters:

aid	area identifier

tgtrect	bounding box of the active area in the target texture (pixels)
draw_event	redraw condition (see *oapiRegisterPanelArea*)
mouse_event	mouse event (see *oapiRegisterPanelArea*)
bkmode	background mode (see *oapiRegisterPanelArea*)
tgt	target texture to be updated

Notes:

- The target texture can be retrieved from a mesh by using the *oapiGetTextureHandle* method. Dynamic textures must be marked with flag 'D' in the mesh file.
- Redraw events can be used not only to update mesh textures dynamically, but also to animate mesh groups, or edit mesh vertices or texture coordinates.
- If no dynamic texture repaints are required during redraw events, use the alternative version of *oapiVCRegisterArea* instead.
- To define a mouse-sensitive volume in the virtual cockpit, use one of the *oapiVCSetAreaClickmode_XXX* functions.

oapiVCRegisterArea (2)

Define an active area in a virtual cockpit. This version is used when no dynamic texture update is required during redraw events.

Synopsis:

```
void oapiVCRegisterArea (
     int aid,
     int draw_event,
     int mouse_event)
```

Parameters:

aid	area identifier
draw_event	redraw condition (see oapiRegisterPanelArea)
mouse_event	mouse event (see oapiRegisterPanelArea)

Notes:

- This function is equivalent to
```
oapiVCRegisterArea (aid, _R(0,0,0,0), draw_event,
      mouse_event, PANEL_MAP_NONE, NULL)
```

oapiVCTriggerRedrawArea

Triggers a redraw notification for a virtual cockpit area.

Synopsis:

```
void oapiVCTriggerRedrawArea (int vc_id, int area_id)
```

Parameters:

vc_id	virtual cockpit identifier
area_id	area identifier (as specified during area registration)

Notes:

- This function triggers a call to the *ovcVCRedrawEvent* callback function in the vessel module.
- The request is ignored if the specified virtual cockpit is not currently active.

oapiVCSetAreaClickmode_Spherical

Associate a spherical region in the virtual cockpit with a registered area to receive mouse events.

Synopsis:

```
void oapiVCSetAreaClickmode_Spherical (
     int id,
     const VECTOR3 &cnt,
     double rad)
```

Parameters:

id	area identifier (as specified during area registration)
cnt	centre of active area in the local vessel frame
rad	radius of active area [m]

Notes:

- The area identifier must refer to an area which has previously been registered with a call to *oapiVCRegisterArea*, with the required mouse event modes.
- This function can be called repeatedly, to change the mouse-sensitive area.

oapiVCSetAreaClickmode_Quadrilateral

Associate a quadrilateral region in the virtual cockpit with a registered area to receive mouse events.

Synopsis:

```
void oapiVCSetAreaClickmode_Quadrilateral (
     int id,
     const VECTOR3 &p1,
     const VECTOR3 &p2,
     const VECTOR3 &p3,
     const VECTOR3 &p4)
```

Parameters:

id	area identifier (as specified during area registration)
p1	top left corner of region
p2	top right corner
p3	bottom left corner
p4	bottom right corner

Notes:

- This function will trigger mouse events when the user clicks within the projection of the quadrilateral region on the render window. The mouse event handler will receive the relative position within the area at which the mouse event occurred, where the top left corner has coordinates (0,0), and the bottom right corner has coordinates (1,1). See also *VESSEL2::clbkVCMouseEvent*.
- The area can define any flat quadrilateral in space. It is not limited to rectangles, but all 4 points should be in the same plane.

oapiTriggerRedrawArea

Triggers a redraw notification to either a 2D panel or a virtual cockpit.

Synopsis:

```
void oapiTriggerRedrawArea (
     int panel_id,
     int vc_id,
     int area_id)
```

Parameters:

panel_id	identifier for the panel to receive the redraw message
vc_id	identifier for the virtual cockpit to receive the redraw message
area_id	area identifier

Notes:

- This function can be used to combine the functionality of the *oapiTriggerPanelRedrawArea* and *oapiVCTriggerRedrawArea* methods. Depending on the current cockpit mode, Orbiter sends the redraw request to either *ovcPanelRedrawEvent* or *ovcVCRedrawEvent*.
- This method can only be used if the panel and virtual cockpit areas share a common area identifier.

oapiGetDC

Obtain a Windows device context handle (HDC) for a surface.

Synopsis:

```
HDC oapiGetDC (SURFHANDLE surf)
```

Parameters:

surf	surface handle

Return value:

device context handle for the surface

Notes:

- The device context can be used to perform standard Windows drawing operations (such as *LineTo*, *Rectangle*, *TextOut*, etc.) on the surface.
- When the context is no longer needed it must be released with a call to *oapiReleaseDC*.

oapiReleaseDC

Release a previously acquired device context for a surface.

Synopsis:

```
void oapiReleaseDC (SURFHANDLE surf, HDC hDC)
```

Parameters:

surf	surface handle
hDC	device context to be released

Notes:

- Use this function to release a device context previously acquired with oapiGetDC.
- Standard Windows device context rules apply. For example, any custom device objects loaded via *SelectObject* must be unloaded before calling *oapiReleaseDC*.

oapiGetColour

Returns a colour value adapted to the current screen colour depth for given red, green and blue components.

Synopsis:

DWORD oapiGetColour (DWORD red, DWORD green, DWORD blue)

Parameters:

red	red component (0-255)
green	green component (0-255)
blue	blue component (0-255)

Return value

colour value

Notes:

- Colour values are required for some surface functions like *oapiClearSurface* or *oapiSetSurfaceColourKey*. The colour key for a given RGB triplet depends on the screen colour depth. This function returns the colour value for the closest colour match which can be displayed in the current screen mode.
- In 24 and 32 bit modes the requested colour can always be matched. The colour value in that case is (red << 16) + (green << 8) + blue.
- For 16 bit displays the colour value is calculated as
((red*31)/255) << 11 + ((green*63)/255 << 5 + (blue*31)/255
assuming a "565" colour mode (5 bits for red, 6, for green, 5 for blue). This means that a requested colour may not be perfectly matched.
- These colour values should not be used for Windows (GDI) drawing functions where a *COLORREF* value is expected.

oapiCreateSurface (1)

Create a surface of the specified dimensions.

Synopsis:

```
SURFHANDLE oapiCreateSurface (int width, int height)
```

Parameters:

width	width of surface bitmap (pixels)
height	height of surface bitmap (pixels)

Return value

Handle to the new surface.

Notes:

- The bitmap contents are undefined after creation, so the surface must be repainted fully before mapping it to the screen.
- If you want to use the surface as a texture, use *oapiCreateTextureSurface* instead.
- Surfaces should be destroyed by calling *oapiDestroySurface* when they are no longer needed.

See also:

oapiDestroySurface()

oapiCreateSurface (2)

Create a surface from a bitmap. Bitmap surfaces are typically used for blitting operations during instrument panel redraws.

Synopsis:

```
SURFHANDLE oapiCreateSurface (
     HBITMAP hBmp,
     bool release_bmp = true)
```

Parameters:

hBmp	bitmap handle
release_bmp	flag for bitmap release

Return value:

Handle to the new surface.

Notes:

- The easiest way to access bitmaps is by storing them as resources in the module, and loading them via a call to *LoadBitmap*.

- Do not use this function with a bitmap generated by *CreateBitmap*. To create a surface of specified dimensions, use *oapiCreateSurface (width, height)* instead.
- If *release_bmp == true*, then *oapiCreateSurface* will destroy the bitmap after creating a surface from it (i.e. the *hBmp* handle will be invalid after the function returns), otherwise the module is responsible for destroying the bitmap by a call to *DestroyObject* when it is no longer needed.
- Surfaces should be destroyed by calling *oapiDestroySurface* when they are no longer needed.

oapiCreateTextureSurface

Create a surface that can be used as a texture for a 3-D object.

Synopsis:

```
SURFHANDLE oapiCreateTextureSurface (
     int width,
     int height)
```

Parameters:

width	width of surface bitmap (pixels)
height	height of surface bitmap (pixels)

Return value:

handle of new texture surface

Notes:

- Use this function instead of *oapiCreateSurface* if you want the surface to be used as a surface texture for a 3-D object, for example via a call to *oapiSetTexture*.
- For maximum compatibility, the surface should be square, and dimensions powers of 2, for example 64x64, 128x128, 256x256, etc. Note that older video cards may not support textures larger than 256x256.
- Surfaces should be destroyed by calling *oapiDestroySurface* when they are no longer needed.

oapiDestroySurface

Destroy a surface previously created with oapiCreateSurface.

Synopsis:

```
void oapiDestroySurface (SURFHANDLE surf)
```

Parameters:

surf	surface handle

oapiSetSurfaceColourKey

Define a colour key for a surface to allow transparent blitting.

Synopsis:

```
void oapiSetSurfaceColourKey (SURFHANDLE surf, DWORD ck)
```

Parameters:

surf	surface handle
ck	colour key (0xRRGGBB)

Notes:

- Defining a colour key and subsequently calling oapiBlt with the SURF_PREDEF_CK flag is slightly more efficient than passing the colour key explicitly to oapiBlt each time, if the same colour key is used repeatedly.

See also:

oapiClearSurfaceColourKey(), oapiBlt()

oapiClearSurfaceColourKey

Clear a previously defined colour key.

Synopsis:

```
void oapiClearSurfaceColourKey (SURFHANDLE surf)
```

Parameters:

surf	surface handle

See also:

oapiSetSurfaceColourKey(), oapiBlt()

oapiBlt

Copy a surface into another surface.

Synopsis:

```
void oapiBlt (
    SURFHANDLE tgt, SURFHANDLE src,
    int tgtx, int tgty,
    int srcx, int srcy,
    int w, int h,
    DWORD ck = SURF_NO_CK)
```

Parameters:

tgt	target surface
src	source surface
tgtx, tgty	coordinates of upper left corner of copied area in target bitmap.
srcx, srcy	coordinates of upper left corner of copied area in source bitmap.
w, h	width, height of copied rectangle (pixel)
ck	colour key (see notes)

Notes:

- Typically, this function is used to update panel instruments during processing of ovcPanelRedrawEvent.
- This function must not be used while a device context is acquired for the target surface (i.e. between oapiGetDC and oapiReleaseDC calls).
- If a blitting operation is necessary between oapiGetDC and oapiReleaseDC, you may use the standard Windows BitBlt function. However this does not use hardware acceleration and should therefore be avoided.
- Transparent blitting can be performed by specifying a colour key in ck. The transparent colour can either be passed explicitly in ck, or ck can be set to SURF_PREDEF_CK to use the key previously defined with oapiSetSurfaceColourKey().

See also:

oapiSetSurfaceColourKey()

oapiColourFill

Fill an area of the target surface with a uniform colour.

Synopsis:

```
void oapiColourFill (
    SURFHANDLE tgt,
    DWORD fillcolor,
    int tgtx = 0, int tgty = 0,
    int w = 0, int h = 0)
```

Parameters:

tgt	target surface
tgtx, tgty	coordinates of upper left corner of area to fill.
w, h	width, height, of area to fill.

Notes:

- The fill colour should be acquired with oapiGetColour(), to ensure compatibility with 16-bit colour modes.
- This function must not be used while a device context is acquired for the target surface (i.e. between oapiGetDC and oapiReleaseDC calls).
- If w and h are zero (the default) the whole surface is filled. The tgtx and tgty values are ignored in that case and can be omitted.

17.18 Custom MFD modes

oapiRegisterMFDMode

Register a custom MFD mode.

Synopsis:

```
int oapiRegisterMFDMode (MFDMODESPEC &spec)
```

Parameters:

spec	MFD specs (see notes below)

Return value:

MFD mode identifier

Notes:

- This function registers a custom MFD mode with Orbiter. There are two types of custom MFDs: generic and vessel class-specific. Generic MFD modes are available to all vessel types, while specific modes are only available for a single vessel class. Generic modes should be registered in the `opcDLLInit` callback function of a plugin module. Vessel class specific modes are not implemented yet.
- MFDMODESPEC is a struct defining the parameters of the new mode:

```
typedef struct {
  char *name;            // points to the name of the new mode
  int (*msgproc)(UINT,UINT,WPARAM,LPARAM);
                         // address of MFD message parser
} MFDMODESPEC;
```

- See orbitersdk\samples\CustomMFD for a sample MFD mode implementation.

oapiUnregisterMFDMode

Unregister a previously registered custom MFD mode.

Synopsis:

```
bool oapiUnregisterMFDMode (int mode)
```

Parameters:

mode	mode identifier, as returned by RegisterMFDMode

Return value:

true on success (mode could be unregistered).

oapiDisableMFDMode

Disable an MFD mode.

Synopsis:

```
void oapiDisableMFDMode (int mode)
```

Parameters:

mode MFD mode to be disabled.

Notes:

- The list of disabled MFDs is cleared whenever the focus switches to a new vessel. To disable MFD modes permanently for a particular vessel type, oapiDisableMFDMode should be called from within the ovcFocusChanged callback function.
- For builtin MFD modes, mode can be any of the MFD_xxx constants. For MFD modes defined in plugin modules, the mode id must be obtained by a call to oapiGetMFDModeSpec.

oapiGetMFDModeSpec

Returns the mode identifier and spec for an MFD mode defined by its name.

Synopsis:

```
int oapiGetMFDModeSpec (
    char *name,
    MFDMODESPEC **spec = NULL)
```

Parameters:

name	MFD name (as defined in MFDMODESPEC::name during oapiRegisterMFDMode)
spec	If defined, this will return a pointer to the MFDMODESPEC structure for the mode.

Return value:

MFD mode identifier.

Notes:

- This function returns the same value as oapiRegisterMFDMode for the given mode.
- The mode identifiers for custom MFD modes can not be assumed to persist across simulation runs, since they will change if the user loads or unloads MFD plugins.
- This function can also be used for built-in MFD modes, which are defined as follows:

Name	Mode identifier
Orbit	MFD_ORBIT
Surface	MFD_SURFACE
Map	MFD_MAP
HSI	MFD_HSI
VOR/VTOL	MFD_LANDING
Docking	MFD_DOCKING
Align Planes	MFD_OPLANEALIGN
Sync Orbit	MFD_OSYNC
Transfer	MFD_TRANSFER
COM/NAV	MFD_COMMS

17.19 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:

scn	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:

scn	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:

scn	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:

scn	file handle
item	item id
vec	vector to be written

oapiReadScenario_nextline

Reads an item from a scenario 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.

17.20 User input

oapiOpenDialog

Open a dialog box defined as a Windows resource.

Synopsis:

```
HWND oapiOpenDialog (
     HINSTANCE hDLLInst,
     int resourceId,
     DLGPROC msgProc,
     void *context = 0)
```

Parameters:

hDLLInst	module instance handle (as obtained from *opcDLLInit*)
resourceId	dialog resource identifier
msgProc	pointer to Windows message handler
context	optional user-defined pointer

Return value:

handle of the new dialog box, or NULL if the dialog was open already.

Notes:

- Use *oapiOpenDialog* instead of standard Windows methods such as *CreateWindow* or *DialogBox*, to make sure the dialog works in fullscreen mode.
- Only one instance of a dialog box can be open at a time. A second call to *oapiOpenDialog* with the same dialog id will fail and return NULL.
- The interface of the message handler is as follows:

  ```
  BOOL CALLBACK MsgProc (
          HWND hDlg, UINT uMsg,
          WPARAM wParam, LPARAM lParam)
  ```

  See standard Windows documentation for usage of the dialog message handler.
- The context pointer can be set to user-defined data which can be retrieved via the *oapiGetDialogContext* function. This allows to pass data into the message handler.
- Note that *oapiGetDialogContext* can not be used when processing the WM_INITDIALOG message. In this case, the context pointer can be acessed via lParam instead.

oapiFindDialog

Returns the window handle of an open dialog box, or NULL if the specified dialog box is not open.

Synopsis:

```
HWND oapiFindDialog (HINSTANCE hDLLInst, int resourceId)
```

Parameters:

hDLLInst	module instance handle (as obtained from opcDLLInit)
resourceId	dialog resource identifier

Return value:

Window handle of dialog box, or NULL if the dialog was not found.

oapiCloseDialog

Close a dialog box.

Synopsis:

```
void oapiCloseDialog (HWND hDlg)
```

Parameters:

hDlg	dialog window handle (as obtained by oapiOpenDialog)

Notes:

- This function should be called in response to an IDCANCEL message in the dialog message handler to close a dialog which was opened by *oapiOpenDialog*.

oapiGetDialogContext

Retrieves the context pointer of a dialog box which has been defined during the call to *oapiOpenDialog*.

Synopsis:

```
void *oapiGetDialogContext (HWND hDlg)
```

Parameters:

hDlg	dialog window handle

Notes:

- This function returns NULL if no context pointer was specified in *oapiOpenDialog*.

oapiDefDialogProc

Default Orbiter dialog message handler. This function should be called from the message handler of all dialogs created with *oapiOpenDialog* to perform default actions for any messages not processed in the handler.

Synopsis:

```
BOOL oapiDefDialogProc (
     HWND hDlg,
     UINT uMsg,
     WPARAM wParam,
     LPARAM lParam)
```

Parameters:

The parameters passed to the message handler.

Return value:

The value returned by *oapiDefDialogProc* should be returned by the message handler.

Notes:

- Typical usage:

```
BOOL CALLBACK MsgProc (HWND hDlg, UINT uMsg,
  WPARAM wParam, LPARAM lParam)
{
  switch (uMsg) {
  case WM_COMMAND:
    switch (LOWORD (wParam)) {
    case IDCANCEL: // dialog closed by user
      CloseDlg (hDlg);
      return TRUE;
    }
    break;
    // add more messages to be processed here
  }
  return oapiDefDialogProc (hDlg, uMsg, wParam, lParam);
}
```

- *oapiDefDialogProc* currently only processes the `WM_SETCURSOR` message, and always returns FALSE.

oapiRegisterCustomCmd

Register a custom function. Custom functions can be accessed in Orbiter by pressing Ctrl-F4. A common use for custom functions is opening plugin dialog boxes.

Synopsis:

```
DWORD oapiRegisterCustomCmd (
      char *label,
      char *desc,
      CustomFunc func,
      void *context)
```

Parameters:

label	label to appear in the custom function list.
desc	a short description of the function
func	pointer to the function to be executed
context	pointer to custom data which will be passed to func

Return value:

function identifier

Notes:

- The interface of the custom function is defined as follows:
  `typedef void (*CustomFunc)(void *context)`
  where context is the pointer passed to `oapiRegisterCustomCmd`.

oapiUnregisterCustomCmd

Unregister a previously defined custom function.

Synopsis:

```
bool oapiUnregisterCustomCmd (int cmdId)
```

Parameters:

cmdId	custom function identifier (as returned by *oapiRegisterCustomCmd*)

Return value:

false indicates failure (cmdId not recognised)

oapiOpenInputBox

Opens a modal input box requesting a string from the user.

Synopsis:

```
void oapiOpenInputBox (
     char *title,
     bool (*Clbk)(void*,char*,void*),
     char *buf = 0,
     int vislen = 20,
     void *usrdata = 0)
```

Parameters:

title	input box title
Clbk	callback function receiving the result of the user input (see notes)
buf	initial state of the input string
vislen	number of characters visible in input box
usrdata	user-defined data passed to the callback function

Notes:

- Format for callback function:
  `bool InputCallback (void *id, char *str, void *usrdata)`
  where *id* identifies the input box, *str* contains the user-supplied string, and *usrdata* contains the data specified in the call to oapiOpenInputBox. The callback function should return true if it accepts the string, false otherwise (the box will not be closed if the callback function returns false).
- The box can be closed by the user by pressing Enter ("OK") or Esc ("Cancel"). The callback function is only called in the first case.
- The input box is modal, i.e. all keyboard input is redirected into the dialog box. Normal key functions resume after the box is closed.

17.21 Debugging

oapiDebugString

Returns a pointer to a string which will be displayed in the lower left corner of the viewport.

Synopsis:

```
char *oapiDebugString ()
```

Return value:

Pointer to debugging string.

Notes:

- This function should only be used for debugging purposes. Do not use it in published modules!
- The returned pointer refers to a global char[256] in the Orbiter core. It is the responsibility of the module to ensure that no overflow occurs.
- If the string is written to more than once per time step (either within a single module or by multiple modules) the last state before rendering will be displayed.
- A typical use would be:

```
sprintf (oapiDebugString(), "my value is %f", myvalue);
```

18 Custom dialog controls

Orbiter defines custom dialog control classes which may come useful when defining dialog box interfaces. To make use of the controls, you must include the Orbitersdk\include\DlgCtrl.h header in your plugin code, and link with Orbitersdk\lib\DlgCtrl.lib.

In order to use Orbiter custom dialog controls, your code must call the *oapiRegisterCustomControls* function, usually inside the *opcDLLInit* callback function. During cleanup (e.g. in *opcDLLExit*) you must call *oapiUnregisterCustomControls*.

oapiRegisterCustomControls

This allows to use Orbiter's custom controls in dialog boxes. See section 18.

Synopsis:

```
#include "DlgCtrl.h"
void oapiRegisterCustomControls (HINSTANCE hInst)
```

Parameters:

hInst	module instance handle

Notes:

The module should call *oapiUnregisterCustomControls* before exiting.

oapiUnregisterCustomControls

Unregister Orbiter custom dialog controls.

Synopsis:

```
void oapiUnregisterCustomControls (HINSTANCE hInst)
```

Parameters:

hInst	module instance handle

18.1 Gauge control

This is similar to a standard scrollbar control. It consists of a horizontal or vertical bar with a level indicator and arrow buttons on either end. The user can manipulate the control by either pressing the arrow buttons, or by clicking and dragging the level indicator.

Unlike standard Windows scroll bars, the gauge control does not block the simulation while a mouse button is pressed over the control. You should always use the gauge control in preference to scroll bars to avoid jumps in the simulation.

The Rcontrol code in the SDK sample directory demonstrates the use of gauge controls.

Defining a gauge control in the dialog template

Place a custom control in the dialog window and sets its class to *OrbiterCtrl_Gauge*. The control can be horizontal or vertical.

Addressing gauge controls from the module code

oapiSetGaugeParams

Initialises a gauge control once the dialog box has been opened (e.g. with oapiOpenDialog).

Synopsis:

```
void oapiSetGaugeParams (
     HWND hCtrl,
     GAUGEPARAM *gp,
     bool redraw = true)
```

Parameters:

hCtrl	window handle of the control
gp	parameter list (see notes)
redraw	if true, the gauge is redrawn to reflect the parameter changes

Notes:

- The GAUGEPARAM struct has the following entries:
  `int rangemin, rangemax`
  min. and max. gauge values
  `enum GAUGEBASE { LEFT, RIGHT, TOP, BOTTOM } base`
  gauge orientation: LEFT: left to right, RIGHT: right to left, etc.
  `enum GAUGECOLOR { BLACK, RED } color`
  gauge indicator colour

oapiSetGaugeRange

Set minimum and maximum gauge values.

Synopsis:

```
void oapiSetGaugeRange (
     HWND hCtrl,
     int rmin, int rmax,
     bool redraw)
```

Parameters:

hCtrl	window handle of the control
rmin	minimum gauge value
rmax	maximum gauge value
redraw	if true, the gauge is redrawn to reflect the range change

oapiSetGaugePos

Set the current gauge value.

Synopsis:

```
int oapiSetGaugePos (
     HWND hCtrl,
     int pos,
     bool redraw = true)
```

Parameters:

hCtrl	window handle of control
pos	new gauge value
redraw	if true, the gauge is redrawn to reflect the value change

Return value:

The new gauge value, clamped to the gauge range.

oapiIncGaugePos

Increment/decrement the current gauge value.

Synopsis:

```
int oapiIncGaugePos (
     HWND hCtrl,
     int dpos,
     bool redraw = true)
```

Parameters:

hCtrl	window handle of control
dpos	value change
redraw	if true, the gauge is redrawn to reflect the value change

Return value:

The new gauge value, clamped to the gauge range.

oapiGetGaugePos

Returns the current gauge value.

Synopsis:

```
int oapiGetGaugePos (HWND hCtrl)
```

Parameters:

hCtrl window handle of control

Return value:

Current gauge value.

Control messages

Gauge controls send the following messages to the message queue of the owning dialog box:

WM_HSCROLL

Scrolling notification. This is sent while the user left-clicks and drags the gauge indicator, or continuously (at a rate of 100Hz) while the left mouse button is held down on one of the arrow buttons. Both horizontal and vertical gauges send the WM_HSCROLL message to simplify message handling.

Message parameters:

LOWORD(wParam)	event type
HIWORD(wParam)	gauge value
(HWND)lParam	window handle of control

Notes:

The event type can be one of the following:

SB_LINELEFT: The user has pressed an arrow button to decrement the gauge value.

SB_LINERIGHT: The user has pressed an arrow button to increment the gauge value.

SB_THUMBTRACK: The user is dragging the gauge indicator with the mouse.

19 Standard ORBITER modules

19.1 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 16 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.

19.2 Moon

Moon.dll is Orbiter's driver module for controlling Earth's moon. It contains a partial implementation of the *Lunar Solution ELP 2000-82B* algorithm by M. Chapront-Touze and J. Chapront[2]. This is a semi-analytical calculation of lunar ephemerides consisting of trigonometric and Poisson series, with constants fitted to JPL's ephemerides DE200/LE200. The original version calculates cartesian geocentric lunar coordinates in the mean dynamical ecliptic and inertial equinox of J2000. The code has been adapted to Orbiter by additionally calculating and returning the time derivatives of the coordinates. Moon.dll requires data file ELP82.dat containing a table of perturbation terms to be present in the Config\Moon\Data directory.
The number of terms used by Orbiter can be controlled by setting the ErrorLimit parameter in Moon.cfg. Valid range is 1e-2 to 1e-8 (default 1e-5). The current error limit and number of terms can be found in Orbiter.log under entry *ELP82*.
The current version does not include tidal, relativistic or solar eccentricity perturbation terms, to avoid inconsistencies with Orbiter's dynamic model.

20 Index

P

R

S

[1] P. Bretagnon and G. Francou, Bureau des Longitudes, CNRS URA 707, Planetary Solution VSOP87

[2] M. Chapront-Touze and J. Chapront, Bureau des Longitudes, CNRS URA 707, Lunar Solution ELP 2000-82B