

# UACS Developer Manual

This manual discusses configuring UACS default astronaut and cargo implementations, as well as the C++ API.

## Standard Resource Names

These standard resource names should be used by vessels and cargoes. The name must be lowercase, and spaces can be included.

***fuel, ramjet fuel, oxygen, food, water, hydrogen, helium***

Note: for all ramjet engine variations, use 'fuel' or 'ramjet fuel' (i.e., if the vessel has SCRAM engines, use 'ramjet fuel', not 'scram fuel').

## Standard Astronaut Roles

These names should be by vessels and astronauts. The role must be lowercase, and spaces can be included.

***commander, pilot, mission specialist, flight engineer***

## Stations

Stations are normal Orbiter vessels that can provide breathable habitat for astronauts, unlimited resources for other vessels, or both.

Stations are created by modifying a vessel configuration file by creating a to-parent attachment point with a label that UACS uses to identify the station. UACS\_R is used for resource stations, UACS\_B for breathable stations, and UACS\_RB for both. The attachment point position, direction, and rotation have no effect, only the label is used to identify the station.

If the station supports resources, a UACS\_RESOURCES option can be added to specify resources. Use the standard resource names detailed above. If the option isn't added, the station is assumed to support all resources. A vessel can drain resources if it's within the station size (mean radius) plus the drain range (set by the vessel).

To create a station, open the vessel you want to be a station configuration file which can be found in the 'Config\Vessels' folder, and find the attachment sections.

BEGIN\_ATTACHMENT

.....

END\_ATTACHMENT

If you find it, add the station attachment before END\_ATTACHMENT line. Change the attachment point label (the text at the end) to the appropriate type.

```
BEGIN_ATTACHMENT  
  
P 0 0 0 0 0 1 0 1 0 UACS_RB  
  
END_ATTACHMENT
```

Otherwise, add the following lines to the end of the file. Change the attachment label as required.

```
BEGIN_ATTACHMENT  
  
.....  
  
P 0 0 0 0 0 1 0 1 0 UACS_RB  
  
END_ATTACHMENT
```

If the station supports resources, add the resource names **separated by commas with no spaces between the commas**, at the top of the file. If not added, the station will support all resources.

```
UACS_RESOURCES = fuel,ramjet fuel,oxygen
```

## Astronauts

### Configuration File

Astronaut configuration files are located in 'Config\Vessels\UACS\Astronauts' folder. If one of the required options is missing, Orbiter will crash when the cargo is loaded with a runtime error and an error message in the Orbiter.log file with the missing option.

The astronaut suit mesh and body mesh aren't displayed at the same time. Suit mesh is displayed only when the suit is on, and body mesh when the suit is off. That means the suit mesh should contain a body inside it (i.e., the astronaut face). The suit mass and body mass are both added plus the fuel and oxygen mass to form the astronaut's total weight.

Only the name, role, body mass options are specific to each astronaut instance in a scenario and can be modified for each instance. The other parameters are global to all astronaut instances and can't be changed for each instance.

The body height and suit height are the distance from the zero Y level to the astronaut feet for both the suit mesh and body mesh. If the mesh is positioned so that the astronaut feet are at zero Y for example, the value should be zero. If the mesh is positioned so that the astronaut head is at zero Y, the value should be the astronaut's actual height. If the mesh is positioned so that the astronaut waist is at zero Y, the value should be half the astronaut height, etc.

Rough values can be obtained by using Shipedit with the body and suit mesh, as explained in the cargo chapter. Cross sections and inertia tensor values can be obtained from Shipedit as well. The suit mesh should be used to calculate the cross sections and inertia tensor.

The camera offset is used to set the proper position for the cockpit view. It should be the coordinates of the astronaut head center.

Since the packed cargo attachment point position is at the cargo center, UACS needs the cargo holding direction to position the grappled cargo properly. If the cargo is held from below, the Y value is -1 (so the direction is 0 -1 0). If held from above, Y value is 1. From right, X value is 1. From left, X value is -1. From front, Z value is 1. From rear, Z value is -1. The vector must be normalized.

Each astronaut has two to-children attachment points to hold cargoes: one with a suit, the other without it. The first attachment point is the suit one, the second is the body one. Normally there is no need to change the attachment point direction and rotation. The direction and position must be normalized and be perpendicular to each other.

Each astronaut supports only one spotlight (light emitter) and unlimited beacons for the headlight. The spotlight position and direction as well as one beacon position at least must be defined to enable the headlight. The direction vector must be normalized.

### Configuration File Options Reference

Option	Description
<b>DefaultName</b>	The astronaut person default name.
<b>DefaultRole</b>	The astronaut person default role. Use standard astronaut roles.
<b>SuitMesh</b>	The suit mesh file path from 'Meshes' folder without '.msh'.
<b>BodyMesh</b>	The body mesh file path from 'Meshes' folder without '.msh'.
<b>SuitMass</b>	The suit mass in kilograms.
<b>DefaultBodyMass</b>	The default body mass in kilograms.
<b>SuitHeight</b>	The height when the suit is on in meters.
<b>BodyHeight</b>	The height when the suit is off in meters.
<b>Size</b>	The mean radius in meters.
<b>PropellantResource1</b>	The maximum fuel mass in kilograms.
<b>PropellantResource2</b>	The maximum oxygen mass in kilograms.
<b>BEGIN_ATTACHMENT</b> <b>END_ATTACHMENT</b>	The astronaut cargo attachment point position, direction, and rotation. The first attachment point is when suit is on, the second is when suit is off.
<b>SuitHoldDir</b>	The cargo holding direction when suit is on.
<b>BodyHoldDir</b>	The cargo holding direction when suit is off.
<b>CameraOffset</b>	Optional: The cockpit camera offset.
<b>CrossSections</b>	Optional: The cross sections.
<b>Inertia</b>	Optional: The inertia tensor.
<b>SpotLightPos</b>	Optional: The headlight spotlight position.
<b>SpotLightDir</b>	Optional: The headlight spotlight direction.
<b>Beacon[n]Pos</b>	Optional: The headlight beacon position. Replace [n] by the beacon index starting from 1 (e.g. Beacon1Pos).

ImageBmp	Optional: the astronaut image in scenario editor.
----------	---

## Cargoes

### Modelling

The packed cargo box dimensions must be 1.3mx1.3mx1.3m to follow the standard size. If not followed, the cargo won't fit into cargo containers on vessels, and the touchdown points won't be calculated correctly. The box bottom must be at -0.65m, so the cargo center is at (0 0 0), which is the attachment point position. You can use the default containers if you don't want to design one yourself.

The packed cargo polygons should be as low as possible, as vessels can carry a lot of cargoes. Since a lot of polygons will greatly affect performance, the polygons shouldn't exceed 200. Use a 512p x 512p texture.

For unpacked cargoes, the polygons shouldn't exceed 1000 polygons with a 1024p x 1024p texture.

### Skin/Texture Creation

To create a skin or texture for a cargo, create a new cargo with a different texture. Make a copy of the cargo config file with a unique name. This unique name should preferably be used for the mesh and texture filename as well.

Open the config file and find the mesh you want to texture (either PackedMesh or UnpackedMesh). Mesh files are in the 'Meshes' folder in Orbiter root folder.

Make a copy of the mesh file and open it. Scroll down to the textures section and change it as required. Textures must be placed in the 'Textures' folder in Orbiter root folder. After that, change the mesh in the config file to the new mesh.

As stated above, the recommended texture size is 512p x 512p for packed meshes, and 1024p x 1024p for unpacked meshes.

### Configuration File

The cargo configuration file must be saved in the 'Config\Vessels\UACS\Cargoes' folder. The easiest way to make a configuration file is to copy a similar cargo configuration file and edit it.

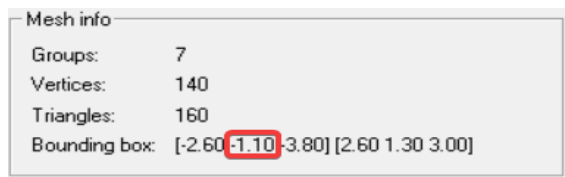
If the cargo contains a resource, see the standard resource names above. The mesh, texture, and configuration file should preferably have the same name, but it's not required.

If one of the required options is missing, Orbiter will crash when the cargo is loaded with a runtime error and an error message in the Orbiter.log file with the missing option.

To align the unpacked cargoes on ground properly, UACS needs either 3 positions (front, right, and left), or a single height. The advantage of using 3 positions is that the cargo will follow terrain slopes, which isn't possible when using a single height. The 3 positions should be used unless the unpacked cargo mesh is simple and doesn't need to follow the terrain slope.

The 3 positions are the coordinates of the center bottom front, bottom right, and bottom left of the cargo mesh. The height is the mesh's lowest point.

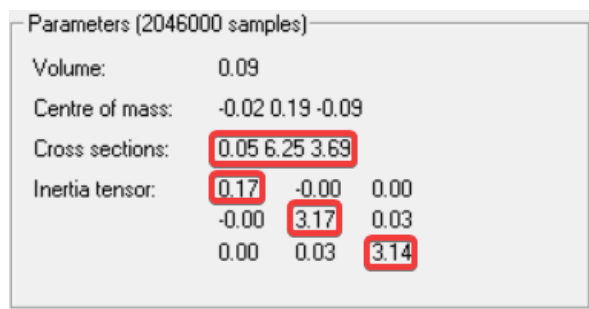
The best way to get these values is to open the mesh in a 3D design software, but you can get rough values from the Shippedit tool in the 'Orbitersdk\utils' folder. Open the unpacked cargo mesh and check the mesh bounding box. The mesh's lowest point is the negative Y value (the center value in the negative group).



The height is a positive value, so the height for the picture above is 1.1m. The 3 positions are: Front (0 -1.1 3.8) Right (2.6 -1.1 -3.8) Left (-2.6 -1.1 -3.8). Keep in mind these are the bounding box values and may deviate widely from proper values, especially if there are protrusions.

The unpacked size is the cargo unpacked mesh mean radius which is simply the absolute value of the biggest number in the bounding box, so the size for the picture is 3.8m.

The unpacked attachment position should be the lowest point of the mesh front, so the cargo is carried from the bottom. It's roughly the same as the unpacked ground front position. To get the unpacked cross sections and inertia tensor, click on 'Calc -> Start/continue MC integration'. Wait until the numbers at the 'Parameters' section stabilizes (around 500,000 samples), then stop it from 'Calc -> Stop MC integration'.



The cross sections for the picture are (0.05 6.25 3.69). The inertia tensor is diagonal as marked in the picture (0.17 3.17 3.14).

## Configuration File Options Reference

All options must be specified unless stated otherwise.

General Options	
Option	Description
PackedMesh	The packed cargo mesh file path from 'Meshes' folder without '.msh'.
PayloadMass	The payload mass in kilograms.
ContainerMass	The container mass in kilograms.
CargoType	0 = Static, 1 = Unpackable.
UnpackOnly	Optional: if the cargo is unpackable only and cannot be packed again. Applicable only if the cargo is unpackable. If not specified, 'FALSE' is assumed. If 'UnpackingType' is Orbiter vessel or 'UnpackedCount' is set, it's forced set to 'TRUE'.

<b>CargoResource</b>	Optional: the resource name (must be lowercase). Use standard resource names. Specify if the cargo is a resource.
----------------------	--

### Unpackable Options

Option	Description
<b>UnpackingType</b>	0 = UACS module, 1 = Orbiter vessel. If set to 1, 'UnpackOnly' is forced set to 'TRUE'.
<b>UnpackingMode</b>	Optional: 0 = Manual, 1 = Released, 2 = Delayed, 3 = Landed. If not specified, 'Manual' is assumed.
<b>UnpackingDelay</b>	The delay period in seconds, for 'Delayed' mode. Applicable only if 'UnpackingMode' is 'Delayed'.
<b>UnpackedMesh</b>	The unpacked cargo mesh file path from 'Meshes' folder without .msh.
<b>UnpackedSize</b>	The unpacked cargo mean radius in meters.
<b>UnpackedHeight*</b>	The unpacked height in meters for release on ground.
<b>UnpackedFrontPos*</b>	The unpacked cargo front touchdown point position.
<b>UnpackedRightPos*</b>	The unpacked cargo right touchdown point position.
<b>UnpackedLeftPos*</b>	The unpacked cargo left touchdown point position.
<b>UnpackedAttachPos</b>	Optional: the unpacked cargo attachment point position. If not specified, '0 0 0' is assumed.
<b>UnpackedCrossSections</b>	Optional: the unpacked cargo cross sections.
<b>UnpackedInertia</b>	Optional: the unpacked cargo inertia tensor.
<b>UnpackedBreathable</b>	Optional: if the cargo can be breathed in when unpacked. If not specified, 'FALSE' is assumed.
<b>UnpackedCount</b>	Optional: the count of cargoes that will be spawned when unpacking the cargo. If set, 'UnpackOnly' is forced set to 'TRUE'.
<b>UnpackedVesselName</b>	The unpacked vessel name in the scenario. It's adjusted automatically for multiple instances. Applicable only if 'UnpackingType' is 'Orbiter vessel'.
<b>UnpackedVesselModule</b>	The unpacked vessel config file path from 'Config\Vessels' folder without '.cfg'. Applicable only if 'UnpackingType' is 'Orbiter vessel'.
<b>ImageBmp</b>	Optional: the cargo image in scenario editor.

\*: Specify either UnpackedHeight or UnpackedFrontPos, UnpackedRightPos, and UnpackedLeftPos, not both.

## API

The API provides a C++ interface to support UACS in vessels, and to develop custom astronauts and cargoes. Use the API for astronauts and cargoes if the default implementation isn't sufficient.

The API header files are in 'Orbitersdk\include\UACS' folder. There are 4 files: 'Common.h', 'Module.h', 'Astronaut.h', and 'Cargo.h'.

'Common.h' includes definitions used in multiple APIs, and there is no need to explicitly include it.

'Module.h' contains the module API for Orbiter modules (vessels, plugins, MFDs, etc.) to interact with UACS system.

'Astronaut.h' contains the astronaut API for custom astronauts, and 'Cargo.h' contains the cargo API for custom cargoes.

2 files can be included simultaneously. For example, 'Astronaut.h' and 'Module.h' can be included for a custom astronaut that supports UACS cargoes.

Link against the API library, which is 'UACS\_API.lib' in 'Orbitersdk\lib' folder. The API requires the C++20 standard. To set it in Visual Studio, open the solution, then right click on the project -> Properties -> General -> Set the C++ Language Standard to ISO C++20 Standard (/std:c++20).

The API reference can be found in 'Orbitersdk\doc' folder.

## Module API

The module API is used by Orbiter modules (vessels, plugins, MFDs, etc.) to interact with UACS system.

For vessels, call the module API constructor and provide a pointer to the calling vessel and either a pointer to the vessel astronaut information (as the VslAstrInfo struct), or vessel cargo information (as the VslCargoInfo struct), or both. If nullptr is passed instead of either information, don't call its methods (i.e., don't call astronaut methods if astronaut information is nullptr). The structs must live until the API instance is destroyed.

For modules other than vessels, call the constructor and pass nullptr for each argument. Astronaut station and airlock and cargo slot specific methods can't be called. These methods are GetTotalAstrMass, AddAstronaut, TransferAstronaut, EgressAstronaut, GetTotalCargoMass, AddCargo, DeleteCargo, GrappleCargo, ReleaseCargo, PackCargo, UnpackCargo, DrainGrappledResource, DrainScenarioResource, and DrainStationResource.

The instance is always usable even if UACS isn't installed. Call GetUACSVersion to find out whether UACS is installed or not, as it returns an empty string view if UACS isn't installed. All methods can be called and return logical values, so there is no need to check if UACS is installed before calling them. Generally, you should check if UACS is installed after creating the instance and display a message to the user if UACS is not installed.

Vessels must call 3 API methods: ParseScenarioLine, clbkPostCreation, clbkSaveState. The ParseScenarioLine method must be called when reading scenario in the vessel clbkLoadStateEx method. While it is used currently to load astronaut information, it should be called even if the vessel doesn't support astronauts.

The `clbkPostCreation` method must be called from the vessel `clbkPostCreation` method, after defining all airlocks, stations, and slots. If the vessel supports astronauts, it must update the empty weight to include the mass of astronauts onboard. Use the `GetTotalAstrMass` method.

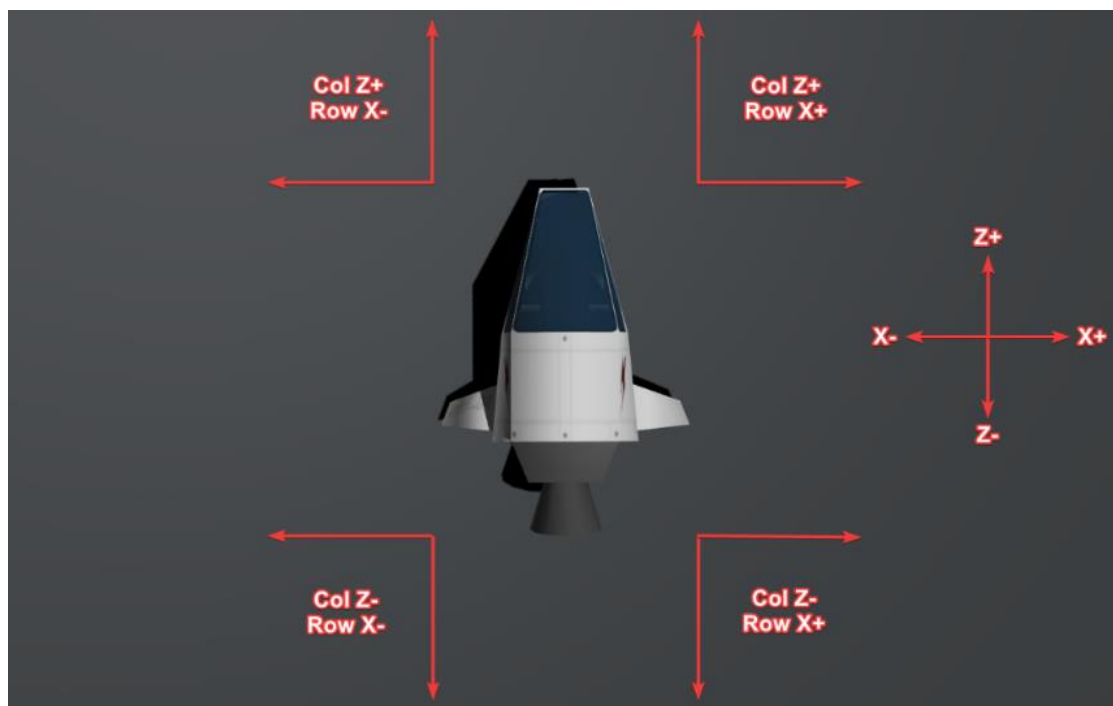
The `clbkSaveState` method must be called from the vessel `clbkSaveState` method. Currently, it is used to save astronaut information, but it should be called even if the vessel doesn't support astronauts.

Vessels that support astronauts must override Orbiter `clbkGeneric` method. It's used to inform vessels of astronaut ingress and egress, so vessels can update the vessel empty weight to add/remove the astronaut mass, and in case of astronaut ingress, to accept the astronaut into the vessel. The method is also called with the astronaut ingress flag when an astronaut is added or transferred from another vessel. It's also called with the astronaut egress flag when the vessel transfers an astronaut into another vessel.

On the ground, objects (astronauts and cargoes) are released in a table consisting of columns and rows. The `GroundInfo` struct provides the necessary information to build the table: initial position (i.e., first cell position), column direction, row direction, number of columns and rows, space between columns, and space between rows.

Objects are placed in the first row until all columns are filled, then placed in the next row. Since the object elevation follows ground elevation, the Y values in initial position and directions have no effect. It should be set to zero.

UACS can calculate the column and row directions based on the initial position. If the position absolute X value is greater than or equal to the absolute Z value, the directions are set as the picture below. If the absolute Z value is greater, the columns and rows are rotated (i.e., column direction is row direction and row direction is column direction).





Astronauts are referred to as the `AstrInfo` struct. They can be independent vessel in the scenario or stored as the `AstrInfo` struct inside vessels. Stations, which are represented by the `StationInfo` struct, are defined by a name and an `AstrInfo std::optional`. If an astronaut is currently in the station, the station `AstrInfo` is defined, otherwise it's `nullopt`.

Airlocks are the way in which astronauts can egress from or ingress into the vessel. They are defined by name, position, direction, rotation, open flag, release velocity (if released in space), ground release information if released on ground, and a docking handle. The docking handle is used to transfer astronauts between docked vessels. If 2 vessels are docked through 2 docking ports, each of which is associated with an airlock, astronauts can be transferred between the vessels.

Station and airlock information to be passed to UACS through the `VslAstrInfo` struct. All variables inside the struct can be modified at any time. The changes are applied immediately.

In the `clbkGeneric` method, the `msgid` parameter is `UACS::API::MSG`, and `prm` is based on the 'Message' enum, all defined in the 'Common.h' file. The context is a pointer to the station/action area index (as `size_t`). Cast the `void*` to `size_t*`, then dereference the pointer and access the relevant station/action area information. For astronaut ingress/egress, astronaut information is always defined when the method is called and can be accessed through the station information.

For astronaut ingress, update the vessel empty weight and return 1. To reject the astronaut ingress, return 0. Note that UACS already checks if the station is free, airlock open, etc.

For astronaut egress, update the vessel empty weight. The return value is ignored, but 1 should be returned. The reason the vessel is notified of astronaut egress is to have the weight change logic one place, instead of being scattered across multiple methods.

For action area trigger, do the action as required and return 1. If the trigger is rejected, return 0.

There are 2 ways to get scenario astronaut information: using UACS astronaut index or Orbiter vessel `OBJHANDLE`. The first way is done by getting the total astronaut count and looping through each index, using `GetScnAstrCount` and `GetAstrInfoByIndex` methods. It should be used if there is no need to loop all scenario vessels, only astronauts, as it is more efficient.

The second way is by using Orbiter vessel `OBJHANDLE` using the `GetAstrInfoByHandle` method. It should be used if you already have an `OBJHANDLE`, or you are already looping through all scenario vessels.

Astronaut information can be set using `SetAstrInfoByIndex` and `SetAstrInfoByHandle`. Don't create an instance of the `AstrInfo` struct and pass it. Rather, get the astronaut information using `GetAstrInfoByIndex` or `GetAstrInfoByHandle` and modify it as required. This is to preserve the astronaut class name (which mustn't be changed) and custom data.

Vessels can add astronauts by getting the available astronaut count using `GetAvailAstrCount`, get his name if required using `GetAvailAstrName`, and add the astronaut to an empty station using `AddAstronaut`. If manually passing astronaut information, don't set the class name as it's set by UACS. Modules can get the information but can't call the `AddAstronaut` method.

The astronaut information of any vessel can be obtained using the `GetVslAstrInfo` method. This information can be used to specify a target station when transferring astronauts from one vessel to another.

Cargoes are independent Orbiter vessels. They are referred to in UACS as the CargoInfo struct, which is used to identify both grappled cargoes and other scenario cargoes.

The CargoInfo struct contains the following information: the cargo OBJHANDLE, attachment flag (useful when getting information about scenario cargoes), type, unpack only flag, unpacked flag, breathable flag, and resource.

For scenario cargoes, there are 2 ways to get their information: using UACS cargo index or Orbiter vessel OBJHANDLE.

The first way is done by getting the total cargo count and looping through each index, using GetScnCargoCount and GetCargoInfoByIndex methods. It should be used if there is no need to loop all scenario vessels, only cargoes, as it is more efficient.

The second way is by using Orbiter vessel OBJHANDLE using the GetCargoInfoByHandle method. It should be used if you already have an OBJHANDLE, or you are already looping through all scenario vessels.

Each slot is defined using the SlotInfo struct by an attachment point, holding direction, open flag, release velocity if released in space, ground information for release on ground, and an std::optional of the slot cargo information. If no cargo is attached to the slot, the optional is a nullopt.

Since the packed cargo attachment point position is at the cargo center, UACS needs the cargo holding direction to position the grappled cargo properly. If the cargo is held from below, the Y value is -1 (so the direction is 0 -1 0). If held from above, Y value is 1. From right, X value is 1. From left, X value is -1. From front, Z value is 1. From rear, Z value is -1.

Slot information along with vessel-specific settings is passed to UACS through the VslCargoInfo struct. These settings are the astronaut mode flag, grapple range, packing/unpacking range, drainage range, max single cargo mass, and max total cargo max.

If the astronaut mode is on:

1. Unpacked cargoes can be grappled.
2. On ground, cargoes are released in a single position (slot attachment point position or ground information position), not in a table. All of the slot ground information except position is ignored.
3. On ground, cargo heading once released is the same as its heading when grappled. If astronaut mode is off, the heading is set to the vessel heading.

The best way to manage UACS cargo information is to have a member variable of the VslCargoInfo, set the options in the constructor, then set the slots in clbkSetClassCaps. All variables inside the VslCargoInfo struct can be modified at any time. The changes will apply immediately.

Vessels can add cargoes by getting the available cargo count using GetAvailCargoCount, get its name if required using GetAvailCargoName, and add the cargo to an empty slot using AddCargo. Modules can get the information but can't call the AddCargo method.

UACS provides 3 methods to drain resources: DrainGrappledResource, DrainScenarioResource, and DrainStationResource. They should preferably be called in that order (i.e., drain from grappled resources first. If not successful, drain from scenario resource, etc.).

Start by including the module API header making variables of it and the astronaut and cargo information structs. Override `clbkLoadStateEx`, `clbkSaveState`, `clbkPostCreation`, and `clbkGeneric` methods if not already overridden. Override `clbkGeneric` only if the vessel supports astronauts.

```
#include <UACS/Module.h>
.....
public:
    void clbkLoadStateEx(FILEHANDLE scn, void* status);
    void clbkSaveState(FILEHANDLE scn);

    void clbkPostCreation();
    int clbkGeneric(int msgid, int prm, void* context);

private:
    UACS::Module uacs;
    UACS::VslAstrInfo vslAstrInfo; // Add if the vessel supports astronauts
    UACS::VslCargoInfo vslCargoInfo; // Add if the vessel supports cargoes
```

In the constructor, initialize the API instance in the initializer list. You can set the cargo options in the constructor as well. If the vessel doesn't support astronaut or cargoes, pass `nullptr` instead of the relevant struct.

```
Vessel::Vessel(OBJHANDLE hVessel, int flightmodel) : VESSEL4(hVessel, flightmodel),
uacs(this, &vslAstrInfo, &vslCargoInfo)
```

In the `clbkSetClassCaps` method, set the airlocks and cargo slots. The airlock and cargo slot position below are based on the UACS Carrier. Set action areas here as necessary.

```
void Vessel::clbkSetClassCaps(FILEHANDLE cfg)
{
    .....
    UACS::AirlockInfo airInfo;
    airInfo.name = "Airlock";
    airInfo.pos = { 0,-0.74, 3.5 };
    airInfo.dir = { 0,0,-1 };
    airInfo.rot = { -1,0,0 };
    airInfo.hDock = CreateDock({ 0,-1,-1 }, { 0,-1,0 }, { 0,0,-1 });
    airInfo.gndInfo.pos = { 4,0,-1.3 };

    vslAstrInfo.airlocks.push_back(airInfo);
    vslAstrInfo.stations.emplace_back("Pilot");

    UACS::SlotInfo slotInfo;
    slotInfo.hAttach = CreateAttachment(false, { 0,1.3,-1 }, { 0,1,0 }, { 0,0,1 },
"UACS");
    slotInfo.holdDir = { 0, -1, 0 };
    slotInfo.relVel = 0.05;
    slotInfo.gndInfo.pos = { -4,0,-1.3 };
    vslCargoInfo.slots.push_back(slotInfo);
}
```

In the `clbkLoadStateEx` method, call the vessel API `ParseScenarioLine` method as appropriate. Use the implementation below if the method isn't already implemented.

```
void Vessel::clbkLoadStateEx(FILEHANDLE scn, void* status)
{
    char* line;

    while (oapiReadScenario_nextline(scn, line))
        if (!uacs.ParseScenarioLine(line)) ParseScenarioLineEx(line, status);
}
```

In the `clbkSaveState` method, call the vessel API `clbkSaveState` method.

```
void Vessel::clbkSaveState(FILEHANDLE scn)
{
    VESSEL4::clbkSaveState(scn);

    uacs.clbkSaveState(scn);
}
```

In the `clbkPostCreation` method, call the vessel API `clbkPostCreation` method. Update the vessel empty mass to include mass of astronauts onboard.

```
void Vessel::clbkPostCreation()
{
    uacs.clbkPostCreation();

    SetEmptyMass(GetEmptyMass() + uacs.GetTotalAstrMass());
}
```

In the `clbkGeneric` method (only if the vessel supports astronauts), update the vessel empty weight based on the received message. If the vessel has action areas, implement their logic as appropriate.

```
int Vessel::clbkGeneric(int msgid, int prm, void* context)
{
    if (msgid == UACS::MSG)
    {
        switch (prm)
        {
            case UACS::ASTR_INGRS:
            {
                auto astrIdx = *static_cast<size_t*>(context);
                auto& astrInfo = vslAstrInfo.stations.at(astrIdx).astrInfo;

                SetEmptyMass(GetEmptyMass() + astrInfo->mass);
                return 1;
            }
            case UACS::ASTR_EGRS:
            {
                auto astrIdx = *static_cast<size_t*>(context);
                auto& astrInfo = vslAstrInfo.stations.at(astrIdx).astrInfo;

                SetEmptyMass(GetEmptyMass() - astrInfo->mass);
                return 1;
            }
            default:
                return 0;
        }
    }

    return 0;
}
```

For action areas, simply add `UACS::ACTN_TRIG` to the switch statement, get the action area index, and handle as required.

The API setup is now complete. You can call all API methods as required.

## Astronaut API

Astronauts that need functions outside of config file interface can use C++ API to integrate into UACS framework. Astronauts can implement both the UACS astronaut and vessel APIs.

The configuration file must be saved in 'Config\Vessels\UACS\Astronauts' folder alongside normal UACS astronauts. Fill the configuration file as any normal Orbiter vessel configuration file.

Inherit from UACS::Astronaut class, which inherits from Orbiter VESSEL4 class. Implement normal Orbiter vessel methods as necessary.

The AstrInfo struct contains the information UACS needs from the astronaut to integrate it into UACS system, which are name, role, mass, height, fuel level, oxygen level, alive flag, custom data, and class name.

The clbkSetAstrInfo and clbkGetAstrInfo methods must be implemented by astronauts. The best way is to have an AstrInfo struct member variable.

In clbkGetAstrInfo, remember to set the current fuel and oxygen levels and any custom data if not set already elsewhere, then return the address of the member variable.

In the clbkSetAstrInfo method, set the member variable to the passed variable, then set the astronaut status as required. Return true if the information was set.

There are 2 ways to get scenario astronaut information: using UACS astronaut index or Orbiter vessel OBJHANDLE. The first way is done by getting the total astronaut count and looping through each index, using GetScnAstrCount and GetAstrInfoByIndex methods. It should be used if there is no need to loop all scenario vessels, only astronauts, as it is more efficient. Note that your astronaut is included.

The second way is by using Orbiter vessel OBJHANDLE using the GetAstrInfoByHandle method. It should be used if you already have an OBJHANDLE, or you are already looping through all scenario vessels.

Other methods can be used as necessary. At minimum, the astronaut should provide means to ingress by calling the ingress method. The default parameters should be sufficient for basic astronauts. Otherwise, use GetVslAstrInfo to obtain a vessel astronaut information and specify a target airlock and station.

If the astronaut has a removable suit, it's useful to call InBreathable method to prevent removing the suit when not in a breathable area. The method can either check whether the astronaut is inside a breathable vessel only, or if inside a breathable vessel or atmosphere.

## Cargo API

A custom cargo is a normal Orbiter vessel that uses the C++ API to act like normal cargoes. Follow the instructions and restrictions in the cargo section above (packed dimensions must be 1.3m x 1.3m x 1.3m, unpacked height and 3 positions, etc.).

The configuration file must be saved in 'Config\Vessels\UACS\Cargoes' folder alongside normal UACS cargoes. Fill the configuration file as any normal Orbiter vessel configuration file.

Cargoes must inherit from UACS::Cargo class, which in turn inherits from Orbiter VESSEL4 class. Implement normal Orbiter vessel methods as required.

Cargo information is handled via the CargoInfo struct that UACS gets a constant pointer to via the clbkGetCargoInfo method, which must be implemented by cargoes.

This information is attachment point handle, type, unpack only flag, unpacked flag, breathable flag, ground release positions, and resource.

The struct the pointer references must live until the cargo is destroyed, so don't return an address of a temporary variable. The best way is to have a CargoInfo member variable and return its address in that method. Methods other than clbkGetCargoInfo should be implemented based on the cargo type and functionality.

Information not related to cargo status (usually type, unpackOnly, breathable, and resource) should be set in the vessel constructor. Other information should be set in clbkSetClassCaps or clbkLoadStateEx as required. It's important to note that the cargo is solely responsible for changing the information as required, such as setting the unpacked flag and the attachment point based on the cargo status. UACS never modifies cargo information.

The packed cargo attachment position should be at the cargo center (0 0 0), not at top or bottom. The direction should be (0 -1 0) and rotation (0 0 1). This is important so vessels can properly position cargoes in their slots depending on the slot design.

The unpacked attachment position should be the center position of the mesh front, with same direction and rotation as packed cargoes, for astronauts to carry the cargo properly.