SASL Reference Manual

April 18, 2024

Contents

Chapter 1

Introduction

SASL - is a plugin/framework for X-Plane which connects Lua scripts and runs them in a virtual machine.

It is a very powerful, yet easy to understand framework that allows designers to write complex or simple plugins for their products (**global** plugins, **aircraft** plugins, **scenery** plugins). SASL can basically do everything the native X-Plane SDK can do, and even more. SASL has many preprogrammed features and sub-systems for convenient add-ons developing process: modular components/scripts system, advanced graphics sub-system, user interaction system, windowing system and many more other built-in features/systems.

Current SASL version is compatible with X-Plane 11.25+ and runs on 64-bit systems. Versions 3.2.5 and lower are compatible with both X-Plane 10 and X-Plane 11. X-Plane versions lower than X-Plane 10 are not supported.

There is no limitation on how many SASL-driven projects may be installed and run simultaneously on user side - every SASL project is self-contained and independent.

Current SASL version supports following platforms:

- Microsoft Windows (7+)
- Linux (Ubuntu 14.04+ LTS or compatible)
- Mac OS (10.11+)

1.1 Plugin/Structure

SASL plugin must be placed in **plugins** folder as an **aircraft** plugin, **scenery** plugin or **global** plugin (standard plugin installation layout for X-Plane plugins system). The folder containing SASL plugin can have any appropriate name - this is especially useful for **global** plugin creation, because all global plugins are installed next to each other and thus all global plugins folders should have unique names. SASL plugin distribution is an engine that runs SASL **projects** inside X-Plane environment. The basic SASL plugin distribution is shown on Figure 1.1.

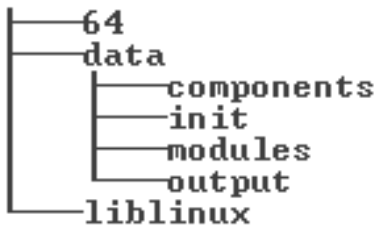

Figure 1.1: SASL Plugin distribution structure

Folders **64** and **liblinux** contains plugin binaries and additional libraries. **data** folder contains all internal SASL engine files and the place for SASL project-specific files - **modules** folder.

- **components** - standard SASL components folder (shouldn't be modified).
- **init** - SASL engine scripts folder (shouldn't be modified).
- **output** - empty folder. SASL-specific log file and other output will be dumped there while plugin is running in X-Plane.
- **modules** - folder which may contain project definition files.

modules folder is first of two places, where SASL will perform lookup for project to run. Such project classified as **Inner Project**. The second place is the **modules** folder that might be placed next to **plugins** folder (for aircraft and scenery projects). Projects in this location classified as **Outer Project**. Developer is free to choose the project place and corresponding project class (**Outer** or **Inner**) with only one limitation: **Global** SASL projects must be **Inner** because of X-Plane global plugins installation layout. More information about **Global** project and other project types will follow in the next section.

Inner Project is preferred class of project by SASL, so SASL will try to find **Outer Project** only in case if there is no valid **Inner Project**.Certain products may even consist of multiple SASL plugins (global product) or include more than one SASL plugin (for aircraft or scenery products).

SASL distribution folder also may include additional files such as version file, changelog, licenses etc.

1.2 Project

SASL project can be of three types:

- Aircraft project.
- Scenery project.
- Global project.

```
│   main.lua
│
├──3rd-modules
├──configuration
│   │   configuration.ini
│   │
│   └──widgetResources
│           DejaVuSans.ttf
│
└──Custom Module
        cameras.lua
        engines.lua
        reflections.lua
```

Figure 1.2: Project structure example in **modules** folder

Every SASL project should have specific structure. Project files should be placed in **modules** folder. Figure 1.2 shows an example of how SASL project files tree might look like. SASL project consists from Lua scripts, SASL project **configuration** files and may include resources files (images, fonts, sounds, shaders etc).

The core of the project is the **main.lua** root script. It's the only Lua file with pre-defined name in SASL project. Location and name of this file is fixed. Writing SASL Lua scripts (including **main.lua**) will be discussed later. **Custom Module** folder contains other project's Lua scripts and may contain nested sub-folders, allowing developer to define any project files structure.

The vital part of project setup is project **configuration**. For project validation there must be a special project configuration file **configuration.ini** inside **configuration** folder. This file is used by SASL engine to configure project inside simulator environment. Project type (**aircraft, scereny, global**) is one of configuration parameters.

Inside **configuration** folder there is also an optional folder called **widgetResources**. This folder and its contents will be described in next section.

In case if SASL project uses some of 3rd-party Lua libraries, these libraries might be placed inside **3rd-modules** folder to separate them from main project files. This folder is set as default path for additional Lua modules lookup performed by standard **require** function. Examples of such 3rd modules are: LuaSocket, LuaUnit etc. **3rd-modules** folder will be ignored by Commercial SASL encryption tools and modules files will be unchanged. Do not place SASL-specific scripts in this folder, it's not supported for Commercial SASL. Use standard **require** function to load such modules.

Note that SASL project structures and location rules gives the ability to run multiple separate projects even for single aircraft (in case if all these projects are located as **Inner Project** and configured as **Aircraft Project**). Or one of them can be **Outer Project**, still without any conflict.

More structure examples for different project types and classes will follow on figures 1.3,

1.4, 1.5.

```
└──plugins
    └──Aircraft Example Project
        ├──64
            lin.xpl
            mac.xpl
            win.xpl

        ├──data
            ├──components
            ├──init
            ├──modules
                main.lua

                ├──3rd-modules
                ├──configuration
                    configuration.ini

                    └──widgetResources
                        DejaVuSans.ttf

                └──Custom Module
                    altimeter.lua
                    checklists.lua
                    engines.lua

            └──output
        └──liblinux
            libc++.so
            libc++.so.1
            libc++.so.1.0
```

Figure 1.3: Aircraft Inner Project

```
├──modules
│   main.lua
│
├──3rd-modules
├──configuration
│   │   configuration.ini
│   │
│   └──widgetResources
│           DejaVuSans.ttf
│
└──Custom Module
        altimeter.lua
        checklists.lua
        engines.lua

└──plugins
    └──Aircraft Example Project
        ├──64
        │       lin.xpl
        │       mac.xpl
        │       win.xpl
        │
        ├──data
        │   ├──components
        │   ├──init
        │   ├──modules
        │   └──output
        └──liblinux
                libc++.so
                libc++.so.1
                libc++.so.1.0
```

Figure 1.4: Aircraft Outer Project

```
└──Resources
    └──plugins
        ├──DataRefEditor
        ├──Global Example SASL Project
        │   ├──64
        │       lin.xpl
        │       mac.xpl
        │       win.xpl
        │
        │   ├──data
        │   │   ├──components
        │   │   ├──init
        │   │   ├──modules
        │   │   │   main.lua
        │   │   │
        │   │   │   ├──3rd-modules
        │   │   │   ├──configuration
        │   │   │   │   configuration.ini
        │   │   │   │
        │   │   │   └──widgetResources
        │   │   │       DejaVuSans.ttf
        │   │   │
        │   │   │   └──Custom Module
        │   │   │       customcamera.lua
        │   │   │       tools.lua
        │   │   │       tweaker.lua
        │   │   │
        │   │   └──output
        │   └──liblinux
        │       libc++.so
        │       libc++.so.1
        │       libc++.so.1.0
        │
        └──PluginAdmin
```

Figure 1.5: Global Inner Project

1.3 Configuration

SASL project must be properly configured to run in simulator environment. SASL engine starts from lookup and reading configuration files and simply won't start in case of missing or incorrect configuration.

You can find **configurationExample.ini** file distributed with SASL plugin in **data/modules/configuration** folder. This is an example of project configuration that shows different options configured and corresponding syntax.

Other example of how project can be configured through configuration file is listed below:

configuration.ini

```
[project]
    id=1
    name=MyProject
    type=2
    startDisabled=0
    widget=1
```

```
[widget]
    font=SourceCodePro.ttf
    fontSize=12
    ptComponentColorR=0.0
    ptComponentColorG=1.0
    ptComponentColorB=0.0
    ptComponentColorA=1.0
    ptStandardComponentColorR=0.729
    ptStandardComponentColorG=0.729
    ptStandardComponentColorB=0.0
    ptStandardComponentColorA=1.0
    ptLabelComponentColorR=0.5
    ptLabelComponentColorG=1.0
    ptLabelComponentColorB=0.5
    ptLabelComponentColorA=1.0
    logWarnColorR=1.0
    logWarnColorG=0.76
    logWarnColorB=0.0
    logWarnColorA=1.0
    logErrorColorR=1.0
    logErrorColorG=0.0
    logErrorColorB=0.0
    logErrorColorA=1.0
```

Along with **configuration.ini** file, **ovrdconfiguration.ini** file can be used to override any option provided in **configuration.ini** – this can be useful in case of VCS usage for project to have different widget setups for different users.

1.3.1 [project] section

project section in **configuration.ini** file has following configurable parameters:

- id
- name
- type
- startDisabled
- widget

This section in the the only mandatory section. Other sections are optional and depends on **project** section values.

id is an integer number, greater than 0. It's not used currently for any purpose, but reserved for future use. **name** is a string, which identifies project name. This option will be used for project logging, internal identifying etc. **type** is an integer number, which identifies project type and can be equal to the following values:

- 0 - aircraft project
- 1 - scenery project
- 2 - global project

startDisabled is an integer number, which identifies required project state after loading and can be equal to following values:

- 0 - start enabled
- 1 - start disabled

In general, in most of the cases SASL project must be configured to start enabled. Use second option only when you want to enable SASL project later externally through other plugin.

widget is an integer number, which identifies whether you need active SASL Developer Widget for the project. Can be equal to following values:

- 0 - Widget Off
- 1 - Widget On

SASL Developer Widget - is a special widget which designed to make add-on developing process more convenient. Generally, widget should be enabled only in development process and should be disabled in release version of the project for better performance. More information about SASL Developer Widget can be found in **Appendix A**.

1.3.2 [sceneryProject] section

This section in **configuration.ini** file is used for **Scenery Project** specific configuration and must be specified for this type of project. Section has following parameters:

- centerLatitude
- centerLongitude
- maxElevation
- radius

centerLatitude is a floating point number, which identifies the latitude of scenery project center.

centerLongitude is a floating point number, which identifies the longitude of scenery project center.

maxElevation is a floating point number that identifies the maximum aircraft elevation (in meters), at which scenery project will be active.

radius is a floating point number, which identifies project radius in nautical miles.

These values defines a virtual zone, where scenery project will be active. If the user aircraft is outside of this zone, scenery project will become inactive. Activating and deactivating processes will be performed automatically by SASL internally, based on current user aircraft location.

1.3.3 [widget] section

This section in **configuration.ini** file is used for configuring SASL Developer Widget visual options in case if SASL Widget is enabled. Section has following parameters:

- font
- fontSize
- ptComponentColorR
- ptComponentColorG
- ptComponentColorB
- ptComponentColorA
- ptStandardComponentColorR
- ptStandardComponentColorG
- ptStandardComponentColorB
- ptStandardComponentColorA
- ptLabelComponentColorR
- ptLabelComponentColorG
- ptLabelComponentColorB
- ptLabelComponentColorA
- logWarnColorR
- logWarnColorG
- logWarnColorB
- logWarnColorA
- logErrorColorR
- logErrorColorG
- logErrorColorB
- logErrorColorA

These parameters can be used to adapt widget for developer: change actual text size and font, select colors for logger output etc.

font is a string, which identifies file name of font that must be used in widget. This font must be located inside **configuration/widgetResources** folder. There is a default font shipped with SASL plugin distribution.

fontSize is an integer number which identifies font size, used in SASL widget. Use any value in $[10; 25]$ range for the font size. Values outside of this range will be automatically clamped. Used only as a default value.

Other parameters are used for configuring colors of different interface elements inside SASL Developer Widget tabs.

1.4 SASL Concepts

With the help of different sub-systems, SASL gives the access to almost all manageable entities in X-Plane - datarefs, commands, cameras, menus, internal communications, objects, gauges, windows, sounds and others. This reference manual is not focused on deep details of certain X-Plane concepts (such as datarefs), but will describe the way how SASL can be used to work with them.

As any other plugin for X-Plane, SASL is callback-driven. It means that SASL won't decide when to do one thing or another, but will react on specific events in internal X-Plane processing pipeline. All such events SASL internally will pass to the Lua virtual machine and adapt these events for convenient usage on Lua code side, allowing developer to define project functionality.

Different callbacks will be called during different stages of the SASL project life cycle. Some of them will be called once at specific stages (loading/unloading), some of them will be called repeatedly as scheduled (drawing, updating), and some of them will be called in respond to some other event (for example, mouse click).

SASL project uses single update loop callback to implement update logic and can draw in 3 graphics contexts:

- Aircraft 2D/3D panel
- 2D windows
- 3D world

Some graphics context may be disabled for specific project types - for example, drawing to aircraft 2D/3D panel is disabled for **Scenery** projects.

Essentially, SASL project is a nested tree of many components, which combined in specific way and working together. SASL project is tunable in many areas and its behaviour may be altered in many ways. More information on this will follow further in the manual.

Chapter 2

Components

The basic concept and building block of SASL is - the **Component**. In a gist, every SASL project is a tree of nested components and every component has a number of **properties** and **callbacks**. Some of them are default ones, and others can be defined by developer.

Properties are used to customize components and their behaviour. They can be changed outside of components definitions in every moment. **Callbacks** are used for defining graphics representation of component and defining how component responds on different events (such as mouse click or keyboard button press).

To create a component in SASL, developer must write its definition script. Such script can include properties definition and callbacks definition. Name of this script defines the component name. For example, **needle.lua** script defines component called "needle" and **reflections.lua** defines component called "reflections".

As already mentioned, every component can have subcomponents inside. It's up to developer how to organize particular project structure and how to define components hierarchy. Each SASL project has a root component, this component defined in project's root script **main.lua**.

For adding subcomponents to component, a special table named **components** must be created. Subcomponents definitions will be searched by SASL in **Custom Module** directory and other search paths. By default, **Custom Module** directory is the only search path for custom developer components. Let's take a look at project root example for **Aircraft Project** and describe how it will be processed by SASL.

main.lua, components hierarchy definition

```
size = { 2048, 2048 }

components = {
    altimeter {
        position = { 0, 0, 200, 200 };
    },
    engines {},
    reflections {}
}
```

This simple root component contains only **size** specification and definition of its subcomponents (**altimeter**, **engines**, **reflections**). Scripts for these subcomponents will be searched in default search path, so there must be an **altimeter.lua**, **engines.lua** and **reflections.lua**

scripts in **Custom Module** folder. There is also definition of **position** property for **altimeter**. This definition is used for specific configuration of component and such definitions overrides default properties values if they are present. Note the specific values for root **size** - it's recommended to set this value to the size of the panel texture in case if SASL project uses aircraft panel. Size assignment can be omitted for components that do not draw on the aircraft panel or in pop-ups.

Important: Scripts with components definitions may be searched in multiple paths. Use **addSearchPath** function to add new search path.

Each component has a set of default fields, **properties** and **callbacks**, which will be described in next sections. Fields is just an ordinary variables. Some of the properties affect only aircraft panels and pop-up panels, but most are common for all components.

2.1 Default variables (fields)

Property: **size**
Type: array of 2 numbers
Description: size of the component. The first number is the width and the second number is the height. Value of field can be specified in corresponding component script. When not explicitly specified, **size** is inherited from parent component. This value is used for aircraft panel or pop-up components.

example.lua, size definition

```
size = { 150, 150 }
```

Property: **name**
Type: string
Description: contains the name of this component.

example.lua, printing component name to the log

```
print(name)
```

2.2 Common properties

Property: **position**
Type: array of 4 numbers
Description: component's position, contains 4 numbers: { x, y, width, height }. Used

for aircraft panel or pop-ups, it specifies the position where the component is located.

main.lua, setting positions

```
components = {
    component1 {
        position = { 20, 20, 130, 170 }
    },
    component2 {
        position = { 345, 17, 200, 200 }
    },
}
```

Property: **visible**
Type: boolean
Description: it is false if the component is hidden and true if the component is visible.

main.lua, setting component initial visibility

```
components = {
    pushButton {
        position = { 10, 10, 50, 30 },
        visible = false
    }
}
```

Property: **movable**
Type: boolean
Description: it is true if the component can be dragged with the mouse. By default it is false for all of the components except pop-up panels.

Property: **resizable**
Type: boolean
Description: it is true if the component can be resized with the mouse. By default it is false for all components except pop-up panels.

Property: **resizeProportional**
Type: boolean
Description: if true, SASL will try to keep the component proportions during the resize. By default it is true. This property is used only if component is resizable.

Property: **focused**
Type: boolean
Description: it is true if the component is an active one (the user clicked on it recently). Only focused components receive keyboard input events.

Property: **clip**
Type: boolean
Description: if true, then the component will be clipped during draw event. By default it is false.

someComponent.lua, setting clipping

```
components = {
    artificialH {
        position = { 20, 20, 100, 100},
        clip = true
    }
}
```

Property: **clipSize**
Type: array of 4 numbers
Description: contains 4 numbers - x, y, width, height. It specifies the clipping area for component. If **clip** property is set to true and **clipSize** is not specified, then component will be clipped by its position. Note that in case of nested components hierarchy, even if children component's **clip** property is false, it will be clipped by all his parents components clip areas.

someComponent.lua, setting clipping with defined clip area

```
components = {
    artificialH {
        position = { 20, 20, 100, 100},
        clip = true,
        clipSize = { 10, 10, 50, 50 }
    }
}
```

Property: **fbo**
Type: boolean
Description: if true, the special FrameBuffer Object will be associated with the component. This will allow you to use additional graphics features during drawing component, such as using custom masks shapes and defining FPS limit for component. Note that any component with **fbo** value equal to true will be automatically clipped by the edges of the component, i.e. simple clipping will be applied to the component, and the component will be clipped by its position. Use with caution, FrameBuffer Object creation and rendering requires additional resources, use **clip** property if you need simple clipping only.

Property: **fpsLimit**
Type: number
Description: defines FPS limit for component, which means that the component will be redrawn only N times per second. The default value is -1. -1 means that component's redraw event will be synchronized with actual simulator frame rate. Note that setting this property to positive values initiates the creation of FrameBuffer Object, use with caution.

someComponent.lua, setting clipping

```
components = {
    nd {
        position = { 0, 0, 300, 400},
        fbo = true,
        fpsLimit = 4
    }
}
```

Property: **noRenderSignal**
Type: boolean
Description: requires FrameBuffer Object creation (**fbo** property must be set to true). Set it to true if you want to skip rendering this component in the next frame's draw event. If property is set to true, the next frame will simply redraw the render from the previous frame with no changes. The assignment will last only one frame and then property will be automatically reset to false.

2.3 Common callbacks

After project configuration and loading all project scripts, project is live inside simulator environment. During simulator events loop all components can receive and handle a set of different events, such as draw, update and user interaction events, etc. For event handling, common callback functions must be defined.

In a few words, for example, for each aircraft panel component or pop-up component draw events will be received every frame. And the special events will be received by each component in case if project is unloading or new scenery just loaded. Developer can leave events callbacks undefined in case if this event handling is not needed.

2.3.1 General callbacks

Callback: **update()**

Description: called every time when X-Plane updates its state. Use this callback

to define update logic.

Warning: if you define your own **update** callback you are responsible for calling subcomponents **update** callbacks. Use the **updateAll** function to call **update** callback of all subcomponents.

fuelTank.lua, updating

```
function update()
    local currentFuelAmount = getFuelAmount()
    ... -- update logic

    updateAll(components) -- updating for subcomponents
end

components = {
    subcomponent1 {},
    subcomponent2 {}
}
```

Callback: **draw()**

Description: called when the component should draw itself. It is important to avoid any calculation (related to "update logic") inside of this function to make your SASL project faster. Use the **update** function for update logic. Note that inside **draw** callback you can use only 2D graphics drawing. **draw** callback is guaranteed to be called only after **onModuleInit** and first **update** calls are done.

Warning: if you define your own **draw** callback you are responsible for calling subcomponents **draw** callbacks. Use the **drawAll** function to call **draw** callback of all subcomponents.

nd.lua, drawing

```
function draw()
    ... -- drawing it's own component stuff

    drawAll(components) -- drawing of subcomponents
end

components = {
    subcomponent1 { position = { 0, 0, 20, 40} },
    subcomponent2 { position = { 50, 50, 10, 10} }
}
```

Callback: **draw3D()**

Description: called when the component should draw 3D graphics. This callback shouldn't be used to draw X-Plane objects represented with **obj** format. As in **draw**

callback, update logic should be avoided. Note that inside **draw3D** callback you can use only 3D graphics drawing. **draw3D** callback is guaranteed to be called only after **onModuleInit** and first **update** calls are done.

Warning: if you define your own **draw3D** callback you are responsible for calling subcomponents **draw3D** callbacks. Use the **drawAll3D** function to call **draw3D** callback of all subcomponents.

Callback: **drawObjects()**

Description: called when the component should draw X-Plane objects. As in **draw** callback, update logic should be avoided. Note that inside **drawObjects** callback you can use only specific functions to draw X-Plane objects. **drawObjects** callback is guaranteed to be called only after **onModuleInit** and first **update** calls are done.

Warning: **drawObjects** callback responsible for calling **drawObjects** of its subcomponents. Use the **drawAllObjects** function to call **drawObjects** callback of all subcomponents.

Callback: **onModuleInit()**

Description: called right before first **update** call. Use this callback to perform deferred initialization in case if required resources/data is not ready during script load.

someComponent.lua, deferred initialization in script

```
function onModuleInit()
    ...
end
```

Callback: **onModuleShutdown()**

Description: called when the SASL project is about to be unloaded (before **onModuleDone**).

someComponent.lua, preparing for project shutdown

```
function onModuleShutdown()
    ...
end
```

Callback: **onModuleDone()**

Description: called when the SASL project is about to be unloaded (after **onModuleShutdown**). Use this callback to finalize your project (save state, configurations, preferences, etc.).

Note that this callback won't be called in case if project is currently stopped due to error condition. For example, if project is stopped on Lua error (configured with **sasl.options.setLuaErrorsHandling** function) and you're reloading your project either with **Reload** button in widget or with corresponding sim command.

someComponent.lua, saving state for future use

```
function onModuleDone()
    -- Saving our settings or whatever else
    saveState(settingsFileName)
    ...
end
```

Callback: **onAirportLoaded**(**number** flightNumber)

Description: called when the user's plane is positioned at new airport (in X-Plane 11 – at every "Start New Flight" sequence). **flightNumber** parameter indicates the count of flights started since SASL plugin enabling. For example, it can be used to handle "Start New Flight" event on project side.

Callback: **onSceneryLoaded()**

Description: called when the new scenery is loaded.

Callback: **onPlaneLoaded()**

Description: called when the user plane is loaded.

Callback: **onPlaneUnloaded()**

Description: called when the user plane is unloaded.

Callback: **onAirplaneCountChanged()**

Description: called whenever the user adjusts the number of X-Plane aircraft models.

Callback: **onPlaneCrash()**

Description: this callback available only for **Aircraft projects** and called whenever the user plane is crashed. By default, when user plane is crashed the SASL system is reloads, but you can omit this action by returning 0 from this function in your root component (**main.lua** script). Note that if this callback is absent in root component or returns 1, this callback will not be called for other components and subcomponents.

main.lua, forcing project to not reload on plane crash

```
function onPlaneCrash()
    return 0
end
```

2.3.2 Mouse Callbacks

All mouse callbacks can return boolean values. If you want to consume mouse event you need to return **true** and if you want to pass it through you need to return **false**. If **false** is returned, then passed mouse events can be handled by other plugins.

Type: **MouseButton**
Description: mouse button identifier, can be equal to one of pre-defined constants:

- MB_LEFT
- MB_RIGHT
- MB_MIDDLE

Callback: **onMouseDown**(**component** component, **number** x, **number** y, **MouseButton** button, **number** parentX, **number** parentY).

Description: called when the mouse button is pressed down. The argument **component** contains reference to the component itself. **x** and **y** are the coordinates of the mouse pointer in this component coordinate system. **parentX** and **parentY** are the coordinates of the mouse pointer in the parent component coordinate system.

radio.lua, mouse down event

```
function onMouseDown(component, x, y, button, parentX, parentY)
    if button == MB_LEFT then
        print("Handled!")
    end
    return true
end
```

Callback: **onMouseUp**(**component** component, **number** x, **number** y, **MouseButton** button, **number** parentX, **number** parentY)

Description: called when the mouse button is released. The argument **component** contains reference to the component itself. **x** and **y** are the coordinates of the mouse pointer in this component coordinate system. **parentX** and **parentY** are the coordinates of the mouse pointer in the parent component coordinate system.

Callback: **onMouseHold**(**component** component, **number** x, **number** y, **MouseButton** button, **number** parentX, **number** parentY)

Description: called when the mouse button is in clicked state (hold down). The argument **component** contains reference to the component itself. **x** and **y** are the coordinates of the mouse pointer in this component coordinate system. **parentX** and **parentY** are the coordinates of the mouse pointer in the parent component coordinate system.

Callback: **onMouseMove**(**component** component, **number** x, **number** y, **MouseButton** button, **number** parentX, **number** parentY)

Description: called when the mouse pointer is moved. The argument **component** contains reference to the component itself. **x** and **y** are the coordinates of the mouse pointer in this component coordinate system. **parentX** and **parentY** are the coordinates of the mouse pointer in the parent component coordinate system.

Callback: **onMouseEnter**()

Description: called when the mouse pointer enters the component's location. This function will be called for component only if you have specified **onMouseMove** callback for this component and return **true** from it.

Callback: **onMouseLeave**()

Description: called when the mouse pointer leaves component's location. This function will be called for component only if you have specified **onMouseMove** callback for this component and return **true** from it.

Callback: **onMouseWheel**(**component** component, **number** x, **number** y, **MouseButton** button, **number** parentX, **number** parentY, **number** value)

Description: called when the mouse wheel clicked. The argument **component** contains reference to the component itself. **x** and **y** are the coordinates of the mouse pointer in this component coordinate system. **parentX** and **parentY** are the coordinates of the mouse pointer in the parent component coordinate system. The argument **button** must be ignored. **value** argument contains the number of performed mouse wheel clicks. Positive values means up direction.

2.3.3 Keyboard Callbacks

Keyboard callback will be called only for focused components. Component become focused after clicking on it.

Type: **AsciiKeyCode**
Description: ASCII key code identifier, can contain corresponding numeric code or can be equal to one of pre-defined constants:

- SASL_KEY_RETURN
- SASL_KEY_ESCAPE
- SASL_KEY_TAB
- SASL_KEY_DELETE
- SASL_KEY_LEFT
- SASL_KEY_RIGHT
- SASL_KEY_UP
- SASL_KEY_DOWN
- SASL_KEY_0
- SASL_KEY_1

- SASL_KEY_2
- SASL_KEY_3
- SASL_KEY_4
- SASL_KEY_5
- SASL_KEY_6
- SASL_KEY_7
- SASL_KEY_8
- SASL_KEY_9
- SASL_KEY_DECIMAL

Type: **VirtualKeyCode**
Description: virtual key code identifier, can be equal to one of pre-defined constants:

- SASL_VK_BACK
- SASL_VK_TAB
- SASL_VK_CLEAR
- SASL_VK_RETURN
- SASL_VK_ESCAPE
- SASL_VK_SPACE
- SASL_VK_PRIOR
- SASL_VK_NEXT
- SASL_VK_END
- SASL_VK_HOME
- SASL_VK_LEFT
- SASL_VK_UP
- SASL_VK_RIGHT
- SASL_VK_DOWN
- SASL_VK_SELECT
- SASL_VK_PRINT

- SASL_VK_EXECUTE
- SASL_VK_SNAPSHOT
- SASL_VK_INSERT
- SASL_VK_DELETE
- SASL_VK_HELP
- SASL_VK_0
- SASL_VK_1
- SASL_VK_2
- SASL_VK_3
- SASL_VK_4
- SASL_VK_5
- SASL_VK_6
- SASL_VK_7
- SASL_VK_8
- SASL_VK_9
- SASL_VK_A
- SASL_VK_B
- SASL_VK_C
- SASL_VK_D
- SASL_VK_E
- SASL_VK_F
- SASL_VK_G
- SASL_VK_H
- SASL_VK_I
- SASL_VK_J
- SASL_VK_K
- SASL_VK_L
- SASL_VK_M

- SASL_VK_N
- SASL_VK_O
- SASL_VK_P
- SASL_VK_Q
- SASL_VK_R
- SASL_VK_S
- SASL_VK_T
- SASL_VK_U
- SASL_VK_V
- SASL_VK_W
- SASL_VK_X
- SASL_VK_Y
- SASL_VK_Z
- SASL_VK_NUMPAD0
- SASL_VK_NUMPAD1
- SASL_VK_NUMPAD2
- SASL_VK_NUMPAD3
- SASL_VK_NUMPAD4
- SASL_VK_NUMPAD5
- SASL_VK_NUMPAD6
- SASL_VK_NUMPAD7
- SASL_VK_NUMPAD8
- SASL_VK_NUMPAD9
- SASL_VK_MULTIPLY
- SASL_VK_ADD
- SASL_VK_SEPARATOR
- SASL_VK_SUBTRACT
- SASL_VK_DECIMAL

- SASL_VK_DIVIDE
- SASL_VK_F1
- SASL_VK_F2
- SASL_VK_F3
- SASL_VK_F4
- SASL_VK_F5
- SASL_VK_F6
- SASL_VK_F7
- SASL_VK_F8
- SASL_VK_F9
- SASL_VK_F10
- SASL_VK_F11
- SASL_VK_F12
- SASL_VK_F13
- SASL_VK_F14
- SASL_VK_F15
- SASL_VK_F16
- SASL_VK_F17
- SASL_VK_F18
- SASL_VK_F19
- SASL_VK_F20
- SASL_VK_F21
- SASL_VK_F22
- SASL_VK_F23
- SASL_VK_F24
- SASL_VK_EQUAL
- SASL_VK_MINUS
- SASL_VK_RBRACE

- SASL_VK_LBRACE
- SASL_VK_QUOTE
- SASL_VK_SEMICOLON
- SASL_VK_BACKSLASH
- SASL_VK_COMMA
- SASL_VK_SLASH
- SASL_VK_PERIOD
- SASL_VK_BACKQUOTE
- SASL_VK_ENTER
- SASL_VK_NUMPAD_ENT
- SASL_VK_NUMPAD_EQ

Callback: **onKeyDown**(**component** component, **AsciiKeyCode** char, **VirtualKeyCode** key, **number** shiftDown, **number** ctrlDown, **number** altOptDown).

Description: called when the keyboard button is pressed. The argument **component** contains reference to the component itself. The argument **char** contains the pressed character ASCII code if the button has corresponding code. The argument **key** contains virtual key code of the pressed button.

shiftDown, **ctrlDown**, **altOptDown** - additional arguments which defines if special buttons is pressed or not. They can be equal to 0 (up) or 1 (down).

Callback must return **true** if the key was processed and event shouldn't be passed to other handlers.

radio.lua, key down event

```
function onKeyDown(component, char, key, shDown, ctrlDown, altOptDown)
    print("Char:"..string.char(char))
end
```

Callback: **onKeyUp**(**component** component, **AsciiKeyCode** char, **VirtualKeyCode** key, **number** shiftDown, **number** ctrlDown, **number** altOptDown).

Description: called when the keyboard button is released. The argument **component** contains reference to the component itself. The argument **char** contains the pressed character ASCII code if the button has corresponding code. The argument **key**

contains virtual key code of the pressed button.

shiftDown, **ctrlDown**, **altOptDown** - additional arguments which defines if special buttons is pressed or not. They can be equal to 0 (up) or 1 (down).

Callback must return **true** if the key was processed and event shouldn't be passed to other handlers.

2.4 Aircraft Global Parameters

In case if your SASL project is configured as **Aircraft Project**, a set of global parameters must be used to configure global entities (like aircraft panel). For now, there is only 5 such global parameters:

Parameter: **panel2d**
Type: boolean
Description: set this parameter to **true**, if aircraft support 2D panel view.

Parameter: **panelWidth2d**
Type: number
Description: width of 2D aircraft panel. Will be used only in case if **panel2d = true**

Parameter: **panelHeight2d**
Type: number
Description: height of 2D aircraft panel. Will be used only in case if **panel2d = true**

Parameter: **panelWidth3d**
Type: number
Description: width of 3D aircraft panel.

Parameter: **panelHeight3d**
Type: number
Description: height of 3D aircraft panel.

All these parameters must be specified in **main** component of your **Aircraft Project**, if you are want to draw on aircraft panel and place your components there. Below you can find example of aircraft global parameters usage:

main.lua, aircraft global parameters setup

```
panel2d = false
panelWidth3d = 2048
panelHeight3d = 2048
```

There's also additional global parameters, which can be used in your processing and drawing (but you can't change their values from your project):

Parameter: **globalShowInteractiveAreas**
Type: boolean
Description: this value is **true** if the X-Plane option "Show mouse click-regions in the cockpit" enabled, and **false** otherwise. Use this value to highlight your interactive areas on 3D panel and pop-ups. This value by default used in standard component called **interactive**.

Chapter 3

Functional

3.1 Components

Namespace: global

3.1.1 subpanel

```
component comp = subpanel(table description)
```

Creates new pop-up panel component and returns it. Returns pop-up panel component. In **description** table you can specify values of following parameters: **name**, **position**, **visible**, **savePosition**, **noBackground**, **noClose**, **noMove**, **noResize**, **fbo**, **clip**, **clipSize**, **command**, **description**, **pinnedToXWindow**, **proportionalToXWindow**. Some of them has default values and not necessarily must be included in **description**.

name - name of pop-up panel. Default name is **'subpanel'**.

position - pop-up panel position, specified by table - { **x, y, width, height** }. This parameter is mandatory.

visible - when **true**, pop-up will be created in visible state. Default value is **false**.

savePosition - when **true**, size and position of this components will be saved in the file **popupsPositions.txt** and re-used next time the project is loaded. This parameter shouldn't be used in case if you are using non-standard pinning options with **pinnedToXWindow** parameter.

noBackground - when **true**, default background won't be drawn for this pop-up panel. Default value is **false**.

noClose - when **true**, close click-spot won't be added for this pop-up panel. Default value is **false**.

noMove - when **true**, pop-up panel won't be movable. Default value is **false**.

noResize - when **true**, resize click-spot won't be added for this pop-up panel. Default value is **false**. Parameter will be ignored in case if **proportinalToXWindow** is **true**.

fbo, **clip**, **clipSize** parameters will be passed for parent component in this pop-up panel. Use them as described in **Components** chapter.

command and **description** are special values that allow to automatically create command for panel pop-up action. Value of command is name of **command**, value of **description** will be shown in X-Plane commands configuration dialogue.

pinnedToXWindow - table of two booleans - { **horizontalDisplacement, verticalDisplacement** }. Use this parameter for pinning pop-up panel to X-Plane window. Default value is { **false, true** }, which means that pop-up panel position won't change after X-Plane window resizing.

proportionalToXWindow - when **true**, SASL will try to keep pop-up panel size proportional to X-Plane window size. Default value is **false**.

components field is a table of pop-up panel child components.

main.lua, popup panel creation

```
somePanel = subpanel {
    name = 'Menu';
    position = { 50, 50, 600, 600 };
    savePosition = false;
    noBackground = false;
    noClose = true;
    noMove = false;
    visible = true;
    noResize = false;
    pinnedToXWindow = { true, true } ;
    proportionalToXWindow = false;
    components = {
        popupPanelComponent {
            position = { 0 , 0 , 600, 600 },
            clip = true
        };
    };
}
```

3.1.2 contextWindow (XP11)

```
ContextWindow w = contextWindow(table description)
```

Creates new modern SASL context window. Returns **ContextWindow** object. SASL Context Window is based on modern X-Plane 11 windows, which support multi-monitor systems, popping out in the first-class OS window, and VR. There are many different

options that defines window depiction and functionality. In **description** table you can specify values of following parameters: **name**, **position**, **minimumSize**, **maximumSize**, **visible**, **proportional**, **gravity**, **noBackground**, **layer**, **noDecore**, **noFocus**, **customDecore**, **decoration**, **noResize**, **noMove**, **fbo**, **clip**, **clipSize**, **command**, **description**, **resizeCallback**, **saveState**. Some of them has default values and not necessarily must be included in **description**.

name - name of context window. When context window is created in **XP decorated** mode (**noDecore = false, customDecore = false**), or if the window is in OS pop-out state, this name will appear as the window title.

position - context window position, specified by table - { **x, y, width, height** }. This parameter is mandatory.

minimumSize - minimum window size, specified by table - { **width, height** }. Default value is { 100, 100 }. In case if resizing is allowed, window will automatically maintain its size limits.

maximumSize - maximum window size, specified by table - { **width, height** }. Default value is { 2048, 2048 }. In case if resizing is allowed, window will automatically maintain its size limits.

visible - when **true**, context window will be created in visible state. Default value is **false**.

proportional - when **true**, window will automatically maintain initial window proportion after resize. Default value is **true**.

gravity - table with gravity values for context window edges - { **left, top, right, bottom** }. Window gravity controls how the window shifts as the whole simulator window resizes. A gravity value of 1 means the window maintains its positioning relative to the right or top edges, 0 the left/bottom, and 0.5 keeps it centered. Default gravity table is { **0, 1, 0, 1** }, meaning your window will maintain its position relative to the top left and will not change size as its containing window grows.

noBackground - when **true**, default background won't be drawn for this context window. Default value is **false**.

noFocus = when **true**, context window won't be able to acquire interactive focus. This option may be useful if window is not meant to be interactive in any way and shouldn't affect currently focused window when this window visibility changes.

layer - specified the window layer. There are four window layer identifiers, which you can use:

- SASL_CW_LAYER_FLIGHT_OVERLAY - lowest layer, used for HUD-like displays

- SASL_CW_LAYER_FLOATING_WINDOWS - default layer, most of X-Plane modern windows live in this layer
- SASL_CW_LAYER_MODAL - interruptive modal that covers the screen with a transparent black overlay
- SASL_CW_LAYER_GROWL_NOTIFICATIONS - highest level notifications layer

noDecore - when **true**, window won't enable decoration (window header, title, close and OS pop-out buttons). Default value is **false**.

customDecore - when **true** (and **noDecore = false**), window will be created with custom SASL decoration.

decoration - optional table that customize window decoration depiction and events handling (when **customDecore = true**). You can specify only those callbacks or values which you want to override.

```
decoration = {
    headerHeight = ..., -- Default header height is 25px.
    draw = function(w, h) -- draw window header
        ...
    end,
    onMouseDown = function(x, y, w, h, button)
        ... -- return true to consume event
    end,
    onMouseUp = function(x, y, w, h, button)
        ... -- return true to consume event
    end,
    onMouseHold = function(x, y, w, h, button)
        ... -- return true to consume event
    end,
    onMouseMove = function(x, y, w, h)
        ...
    end,
    onMouseWheel = function(x, y, w, h, clicks)
        ...
    end,
    main = {
        draw = function(w, h) -- draw window header
            ...
        end,
        onMouseDown = function(x, y, w, h, button)
            ... -- return true to consume event
        end,
        onMouseUp = function(x, y, w, h, button)
            ... -- return true to consume event
        end,
        onMouseHold = function(x, y, w, h, button)
            ... -- return true to consume event
        end,
        onMouseMove = function(x, y, w, h)
            ...
        end,
        onMouseWheel = function(x, y, w, h, clicks)
            ...
        end
    }
}
```

noResize - when **true**, context window won't enable resize spots on the window edges. On **XP decorated** windows this parameter will only force window resize limits to be equal to the original size. Default value is **false**.

resizeMode - specifies resize mode for window. There are two values can be specified:

- SASL_CW_RESIZE_ALL_BOUNDS - window will be resizable by all edges. Default value
- SASL_CW_RESIZE_RIGHT_BOTTOM - window will be resizable only with right bottom click spot (15x15). Use custom decoration to define its graphic representation

noMove - when **true**, context window won't be movable. This parameter is taken into account only if the window is not **XP decorated**. Default value is **false**.

vrAuto - when **true**, context window will automatically handle transferring to VR and from VR. Default value is **false**.

fbo, **clip**, **clipSize** parameters will be passed for parent component in this context window. Use them as described in **Components** chapter.

command and **description** are special values that allow to automatically create command for context window show action. Value of command is name of **command**, value of **description** will be shown in X-Plane commands configuration dialogue.

callback - is an optional callback function which will be called during changing of window (windowing system) state. **id** is a window unique identifier, and **event** can be equal to one of these predefined values:

- SASL_CW_EVENT_VISIBILITY - visibility of window changed
- SASL_CW_EVENT_MODE - window mode changed
- SASL_CW_EVENT_POSITION - window position changed
- SASL_CW_EVENT_SCR_BOUNDS - global screen bounds changed

```
callback = function(id, event)
   ...
end
```

resizeCallback is an optional callback function. It will be called whenever context window is resized. Default resize callback will simply change the size of the higher level component for this context window, but you can alter this behaviour by providing your own resize function. This should be a function, which takes 5 arguments: higher level component, new width and height of resized window, current window **mode** and current **proportional** identifier. The function should return 4 values - x, y, width and height of resized higher level component.

```
resizeCallback = function(c, w, h, mode, proportional)
    ...
end
```

saveState - when **true**, context window will try to save its state (position, mode, etc) between loadings of SASL project. Windows VR state currently can't be saved. Default value is **false**.

components field is a table of context window child components.

main.lua, creation of context window

```
newWindow = contextWindow {
    name = 'Window';
    position = { 50, 50, 600, 600 };
    noBackground = false;
    minimumSize = { 300, 300 };
    maximumSize = { 1200, 1200 };
    gravity = { 0, 1, 0, 1 };
    visible = true;
    components = {
        myComponent {
            position = { 0 , 0 , 600, 600 }
        };
    };
}
```

Returned **ContextWindow** object can be used further for window manipulation.

3.1.2.1 ContextWindow object (XP11)

3.1.2.1.1 setSizeLimits

```
ContextWindow:setSizeLimits(number minW, number minH, number maxW, number
    maxH)
```

Applies provided size limits for the context window.

```
myWindow = contextWindow {
    ...
}

myWindow:setSizeLimits(50, 50, 650, 600)
```

3.1.2.1.2 getSizeLimits

```
number minW, number minH, number maxW, number maxH = ContextWindow:
    getSizeLimits()
```

Returns current size limits for the context window.

```
myWindow = contextWindow {
    ...
}

print(myWindow:getSizeLimits())
```

3.1.2.1.3 setIsVisible

```
ContextWindow:setIsVisible(boolean isVisible)
```

Changes context window visibility state.

```
myWindow = contextWindow {
    ...
}

myWindow:setIsVisible(false)
```

3.1.2.1.4 isVisible

```
boolean visible = ContextWindow:isVisible()
```

Returns context window visibility state.

```
myWindow = contextWindow {
    ...
}

print(myWindow:isVisible())
```

3.1.2.1.5 setProportional

```
ContextWindow:setProportional(boolean isProportional)
```

Enables/disables proportional mode for context window contents.

```
myWindow = contextWindow {
    ...
}

myWindow:setProportional(true)
```

3.1.2.1.6 setMovable

```
ContextWindow:setMovable(boolean isMovable)
```

Enables/disables movable mode for context window. This function only affects context windows that are not created with **X-Plane decoration**.

```
myWindow = contextWindow {
    ...
}

myWindow:setMovable(true)
```

3.1.2.1.7 setResizable

```
ContextWindow:setResizable(boolean isResizable)
```

Enables/disables ability to resize window. This function only affects context windows that are not created with **X-Plane decoration**.

```
myWindow = contextWindow {
    ...
}

myWindow:setResizable(true)
```

3.1.2.1.8 setTitle

```
ContextWindow:setTitle(string title)
```

Changes window title. Note that window title is only shown in **XP decorated** mode or in OS pop-out mode.

```
myWindow = contextWindow {
    ...
}

myWindow:setTitle("Configuration")
```

3.1.2.1.9 setGravity

```
ContextWindow:setGravity(number left, number top, number right, number
    bottom)
```

Changes window gravity values. The meaning of the parameters is the same as in **gravity** parameter for **contextWindow** function.

```
myWindow = contextWindow {
    ...
}

myWindow:setGravity(1, 1, 0, 1)
```

3.1.2.1.10 setPosition

```
ContextWindow:setPosition(number x, number y, number width, number height)
```

Sets the position of the context window. Note that you need to take into account the current window **mode** (in-sim, OS pop-out, VR), when you're setting window position, because different coordinate systems are used in different modes.

```
myWindow = contextWindow {
    ...
}

myWindow:setPosition(0, 0, 600, 400)
```

3.1.2.1.11 getPosition

```
number x, number y, number width, number height = ContextWindow:getPosition
    ()
```

Retunrs the position of the context window, depending on current window **mode** (in-sim, OS pop-out, VR).

```
myWindow = contextWindow {
    ...
}

-- Print window position if the window is popped out in OS window
if myWindow:isPoppedOut() then
    local x, y, w, h = myWindow:getPosition()
    print(x, y, w, h)
end
```

3.1.2.1.12 setMode

```
ContextWindow:setMode(CwMode mode, number monitor)
```

Sets current **mode** for window. Acceptable values for **mode**:

- SASL_CW_MODE_FREE - in-sim default mode
- SASL_CW_MODE_POPOUT - first-class OS window mode (pop-out)
- SASL_CW_MODE_VR - VR mode
- SASL_CW_MODE_MONITOR_CENTER - mode, which will keep window centered on specified monitor
- SASL_CW_MODE_MONITOR_FULL - mode, which will keep window full-screen on specified monitor

monitor is a numeric monitor identifier and can be used to select specific monitor for window. This parameter can be omitted and only applies for specific **modes**. You can pass SASL_CW_MONITOR_MAIN constant to identify that you want to use main monitor for the mode setting.

Use functional from **Windows** section to obtain monitors numeric identifiers and/or their bounds in different coordinate systems (**getMonitorsIDsGlobal**, **getMonitorsIDsOS**, **getMonitorBoundsGlobal**, **getMonitorBoundsOS**).

Note that you need to maintain current modes for your windows in different possible scenarios. For example, if you wan't some window to appear in VR, you need to handle this - window will not change its mode automatically when you're entering/leaving VR.

```
myWindow = contextWindow {
    ...
}

-- Pop-out context window
myWindow:setMode(SASL_CW_MODE_POPOUT)
```

3.1.2.1.13 getMode

```
CwMode mode, number monitor = ContextWindow:getMode()
```

Returns current **mode** for window and current **monitor** identifier if the mode is associated with specific monitor.

3.1.2.1.14 isPoppedOut

```
boolean isPoppedOut = ContextWindow:isPoppedOut()
```

Returns **true**, if the context window is currently in first-class OS window mode, and **false** otherwise.

```
myWindow = contextWindow {
    ...
}

if myWindow:isPoppedOut() then
    ...
end
```

3.1.2.1.15 isInVR

```
boolean inVr = ContextWindow:isInVR()
```

Returns **true**, if the context window is currently in VR mode, and **false** otherwise.

```
myWindow = contextWindow {
    ...
}

if myWindow:isInVR() then
    ...
end
```

3.1.2.1.16 setResizeMode

```
ContextWindow:setResizeMode(CWResizeMode mode)
```

Sets resize **mode** for context window.

3.1.2.1.17 setVrAutoHandling

```
ContextWindow:setVrAutoHandling(boolean auto)
```

Enables/disables VR auto handling mode for context window.

3.1.2.1.18 setCallback

```
ContextWindow:setCallback(function callback)
```

Sets callback function for context window.

3.1.2.1.19 destroy

```
ContextWindow:destroy()
```

Destroys context window object by freeing all associated resources, leaving dummy window object which should be removed on user side by assigning a **nil** to it.

```
myWindow = contextWindow {
    ...
}
...
myWindow:destroy()
myWindow = nil
```

3.1.3 updateAll

```
updateAll(table components)
```

Calls **update** callback for all components in specified table.

main.lua, dispatching update calls for subcomponents

```
function update()
    ...

    updateAll(components)
end

components = {
    first {},
    second {}
}
```

3.1.4 drawAll

```
drawAll(table components)
```

Calls **draw** callback for all components in specified table.

main.lua, dispatching draw calls for subcomponents

```
function draw()
    ...

    drawAll(components)
end

components = {
    first {},
    second {}
}
```

3.1.5 drawAll3D

```
drawAll3D(table components)
```

Calls **draw3D** callback for all components in specified table.

main.lua, dispatching 3D drawing calls for subcomponents

```
function draw3D()
    ...

    drawAll3D(components)
end

components = {
    first {},
    second {}
}
```

3.1.6 drawAllObjects

```
drawAllObjects(table components)
```

Calls **drawObjects** callback for all components in specified table.

main.lua, dispatching objects drawing calls for subcomponents

```
function drawObjects()
    ...

    drawAllObjects(components)
end

components = {
    first {},
    second {}
}
```

3.1.7 Search paths

There are two lists of search path for SASL project. One is for searching components definitions (scripts) and resources (such as textures, sounds, shaders, etc). And second only for resources.

3.1.7.1 addSearchPath

```
addSearchPath(string path)
```

Adds **path** to search paths table. New path will be inserted at the beginning of table.

3.1.7.2 popSearchPath

```
popSearchPath()
popSearchPath(string path)
```

Removes top search path from the stack, or removes specific search **path**.

3.1.7.3 addSearchResourcesPath

```
addSearchResourcesPath(string path)
```

Adds **path** to search resources paths table. New path will be inserted at the beginning of table.

altimeter.lua, adding search path for resources

```
addSearchPath(moduleDirectory.."/backgroungs/")
image = loadImage("altbackground.png")
```

3.1.7.4 popSearchResourcesPath

```
popSearchResourcesPath()
popSearchResourcesPath(string path)
```

Removes top search resources path from the stack, or removes specific search resources **path**.

3.1.8 Logging

3.1.8.1 logInfo

```
logInfo(...)
```

Writes data into simulator and SASL log with "info"-level. New log entry will include component's name.

3.1.8.2 print

```
print(...)
```

Writes data into simulator and SASL log with "info"-level. New log entry will include component's name.

3.1.8.3 logWarning

```
logWarning(...)
```

Writes data into simulator and SASL log with "warning"-level. New log entry will include component's name.

3.1.8.4 logError

```
logError(...)
```

Writes data into simulator and SASL log with "error"-level. New log entry will include component's name.

3.1.8.5 logDebug

```
logDebug(...)
```

Writes data into simulator and SASL log with "debug"-level. New log entry will include component's name.

3.2 Properties

Namespace: global

Along with internal components properties in SASL, developer can also use so called **Simulator Properties** or **Global Properties** (as opposed to internal). Those properties allows interacting with X-Plane DataRefs and are created using specific functions.

Simulator properties can hold following datarefs types: int, float, double, string(data), int array, float array. Functions, which work with array types, uses standard Lua tables for getting and setting properties values.

3.2.1 Components Properties

3.2.1.1 defineProperty

```
defineProperty(string name, value inValue)
```

Defines internal component property with specified **name** and assigns **inValue** value to it. You can use references to simulator properties as values. Name must use the same name conventions as all Lua variables. This function defines only internal properties, but internal properties can hold references to simulator properties as well.

someComponent.lua, internal properties definition

```
defineProperty("angle", 0.0)
defineProperty("framerate", globalPropertyf("sim/operation/misc/
    frame_rate_period"))
```

3.2.1.2 createProperty

```
property prop = createProperty(value)
```

Creates new property and assigns **value** to it. **get** and **set** functions may be called to set/retrieve the value of the property.

3.2.1.3 isProperty

```
boolean result = isProperty(any p)
```

Returns **true** if passed argument is valid **property** and **false** otherwise.

3.2.2 Simulator Properties

3.2.2.1 globalProperty

```
property p = globalPropertyi(string name)
property p = globalPropertyf(string name)
property p = globalPropertyd(string name)
property p = globalPropertys(string name)
property p = globalPropertyia(string name)
property p = globalPropertyfa(string name)

property p = globalPropertyi(string name, boolean suppCastsWarn)
property p = globalPropertyf(string name, boolean suppCastsWarn)
property p = globalPropertyd(string name, boolean suppCastsWarn)
property p = globalPropertys(string name, boolean suppCastsWarn)
property p = globalPropertyia(string name, boolean suppCastsWarn)
property p = globalPropertyfa(string name, boolean suppCastsWarn)

property p = globalProperty(string name)

property p = globalPropertyiae(string name, number index)
property p = globalPropertyfae(string name, number index)

property p = globalPropertyiae(string name, number index, boolean
    suppCastsWarn)
property p = globalPropertyfae(string name, number index, boolean
    suppCastsWarn)
```

Returns reference to simulator property of type (**i**nt, **f**loat, **d**ouble, **s**tring(data), **i**nt **a**rray, **f**loat **a**rray) with corresponding **name**. **suppCastsWarn** is a boolean optional argument that changes function verbosity. If it's **true**, then no warning will be generated in case of allowable type casts.

There is a special version of this function, which automatically determine type and returns corresponding created property: **globalProperty**.

globalPropertyiae and **globalPropertyfae** functions variants allows to use index binding for specific element in array dataref. Returned **property** is associated with this single element in such case.

someComponent.lua, lookup for simulator standard datarefs

```
planeX = globalPropertyf("sim/flightmodel/position/local_x")
planeY = globalPropertyf("sim/flightmodel/position/local_y")
planeZ = globalPropertyf("sim/flightmodel/position/local_z")

wFixed = globalProperty("sim/flightmodel/weight/m_fixed")
```

3.2.2.2 createGlobalProperty

```
property p = createGlobalPropertyi(string name, value default)
property p = createGlobalPropertyf(string name, value default)
property p = createGlobalPropertyd(string name, value default)
property p = createGlobalPropertys(string name, value default)
```

```
property p = createGlobalPropertyia(string name, value default)
property p = createGlobalPropertyfa(string name, value default)

property p = createGlobalPropertyia(string name, number size)
property p = createGlobalPropertyfa(string name, number size)

property p = createGlobalPropertyi(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertyf(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertyd(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertys(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertyia(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertyfa(string name, value default, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)

property p = createGlobalPropertyia(string name, number size, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
property p = createGlobalPropertyfa(string name, number size, boolean
    isNotPublished, boolean isShared, boolean isReadOnly)
```

Creates new simulator property of type (**i**nt, **f**loat, **d**ouble, **s**tring(data), **i**nt **a**rray, **f**loat **a**rray) with corresponding **name** and assign **default** value to it. There is optional boolean parameters **isNotPublished**, **isShared** and **isReadOnly**. If **isNotPublished** is true, then such property will be not accessible in DataRefEditor plugin. If **isShared** is true, then property will be shared by all plugins and not will be owned by SASL project. This means that such properties will not be destroyed when SASL project is unloaded if any other plugin still use them. Shared properties owned by X-Plane itself. If **isReadOnly** is true, then property wont be modifiable by other plugins. By default **isNotPublished**, **isShared** and **isReadOnly** is **false**.

Array variants of function (**ia**, **fa**) can also accept initial array size instead of array value. Property will be initialized by array of 0 values in this case.

Returns reference to created property.

someComponent.lua, creating our project's datarefs

```
panelBrt = createGlobalPropertyf("project/panel/brightness", 1.0)
panelR = createGlobalPropertyf("project/panel/red_component", 0.0, true,
    false)

trafficLat = createGlobalPropertyfa("project/traffic/position/latitude", { 0
    .0, 0.0, 0.0, 0.0})
trafficLon = createGlobalPropertyfa("project/traffic/position/longitude", 4)

defineProperty("menuTab", createGlobalPropertyi("project/menu/tab", 0))
```

3.2.2.3 createFunctionalProperty

```
property p = createFunctionalPropertyi(string name, function getter,
    function setter)
```

```
property p = createFunctionalPropertyf(string name, function getter,
    function setter)
property p = createFunctionalPropertyd(string name, function getter,
    function setter)
property p = createFunctionalPropertys(string name, function getter,
    function setter)
property p = createFunctionalPropertyia(string name, function getter,
    function setter)
property p = createFunctionalPropertyfa(string name, function getter,
    function setter)

property p = createFunctionalPropertyi(string name, function getter,
    function setter, boolean isNotPublished)
property p = createFunctionalPropertyf(string name, function getter,
    function setter, boolean isNotPublished)
property p = createFunctionalPropertyd(string name, function getter,
    function setter, boolean isNotPublished)
property p = createFunctionalPropertys(string name, function getter,
    function setter, boolean isNotPublished, function sizeGetter)
property p = createFunctionalPropertyia(string name, function getter,
    function setter, boolean isNotPublished, function sizeGetter)
property p = createFunctionalPropertyfa(string name, function getter,
    function setter, boolean isNotPublished, function sizeGetter)
```

Creates new functional simulator property of type (**i**nt, **f**loat, **d**ouble, **s**tring or data, **i**nt **a**rray, **f**loat **a**rray) with corresponding **name**. There is optional boolean parameter **isNotPublished**. If **isNotPublished** is true, then such property will be not accessible in DataRefEditor plugin. By default **isNotPublished** is false.

For **int**, **float**, **double** properties: **getter** callback is a function without arguments that returns value of property type. **setter** function takes one argument of property type.

```
function getterFunction()
    return something
end

function setterFunction(value inValue)
    something = inValue
end
```

For **int array**, **float array** and **string** properties: **getter** callback is a function which takes 2 arguments and returns value of property type. **setter** function takes 2 arguments. For these types of properties additional callback **sizeGetter** may be specified. **sizeGetter** is a function without arguments which returns current size of array-like property.

```
function getterFunction(number offset, number numValues)
    ...
    return ...
end

function setterFunction(value inValue, number offset)
    ...
end

function sizeGetterFunction()
    return ...
end
```

Returns reference to created functional property.

someComponent.lua, functional property creation

```
-- Will be called when some plugin changes dataref value
```

```
function setMenuTabCallback(inValue)
    setTabID(inValue)
end

-- Will be called when some plugin wants to get dataref value
function getMenuTabCallback()
    return currentTabID()
end

tabIDProp = createFunctionalPropertyi("project/menu/tab", getMenuTabCallback
    , setMenuTabCallback)
```

3.2.2.4 size

```
number size = property.size()
```

Returns size of simulator property (array size or string length). Can be used only for array or string properties.

someComponent.lua, dataref size

```
instType = globalPropertyia("sim/aircraft/panel/acf_ins_type")
instTypeSize = instType.size()
```

3.2.2.5 raw

```
userdata ref = property.raw()
```

Returns raw pointer to simulator property object (DataRef).

3.2.3 Get and Set

get and **set** are families of functions that used for accessing and changing values of properties (both component internal properties and global simulator properties)

3.2.3.1 get

```
value val = get(property prop)
```

Returns value of property **prop** (internal component property or global simulator property).

someComponent.lua, getting components position

```
currentPosition = get(position)
```

```
value val = get(property prop, number firstIndex, number numElements)
```

Returns specific part of array simulator property **prop**. This part is defined by **firstIndex** and **numElements**.

someComponent.lua, getting first 10 values of array dataref

```
instType = globalPropertyia("sim/aircraft/panel/acf_ins_type")
local values = get(instType, 1, 10)
```

```
number value = get(property prop, number index)
```

Returns single value of array simulator property **prop** with specified **index**.

someComponent.lua, getting 5-th element of array dataref

```
instType = globalPropertyia("sim/aircraft/panel/acf_ins_type")
local values = get(instType, 5)
```

3.2.3.2 set

```
set(property prop, value val)
```

Sets **val** as current **prop** property value (internal component property or global simulator property).

someComponent.lua, setting component's property

```
set(image, altBackground)
```

```
set(property prop, table value, number firstIndex, number numElements)
```

Sets specific part of array simulator property **prop**. This part is defined by **firstIndex** and **numElements**.

someComponent.lua, partial setting of dataref value

```
instType = globalPropertyia("sim/aircraft/panel/acf_ins_type")
set(instType, {0, 0, 0}, 4, 3)
```

```
set(property prop, number value, number index)
```

Sets single value of array simulator property **prop**, specified by **index**.

someComponent.lua, setting array dataref element

```
instType = globalPropertyia("sim/aircraft/panel/acf_ins_type")
set(instType, 0, 4)
```

3.3 Options

Use functional from this section to customize behaviour and internal processing inside SASL engine.

3.3.1 setUpdatePhase

```
sasl.options.setUpdatePhase(UpdatePhaseIdentifier ID)
```

Changes the behaviour on when to call 'update' callback for SASL project, before or after simulator flight model calculation. Calling 'update' after sim flightmodel calculation should be used when SASL project calculates some values based on current flightmodel values. **ID** can be one of the following pre-defined constants:

- SASL_UPDATE_BEFORE_FM - default option.
- SASL_UPDATE_AFTER_FM - call 'update' after flightmodel calculation.

3.3.2 Performance

While it's not mandatory, using this functional is good for better over-all simulator performance.

As an example, consider SASL project, which has a simple 2D user interface and operates some simulator data. Such project don't need 3D rendering capabilities and aircraft panel rendering, so it's better to disable these parts of SASL engine via corresponding functions. Similar actions may be applied in case if you don't need other parts of SASL engine.

3.3.2.1 setAircraftPanelRendering

```
sasl.options.setAircraftPanelRendering(boolean isOn)
```

Enables/disables rendering on aircraft panel (for non-scenery SASL project types). **isOn** defines whether the rendering should be enabled or disabled. By default, aircraft panel rendering is on.

main.lua, disabling aircraft panel rendering

```
-- Disabling aircraft panel rendering stage in SASL engine
sasl.options.setAircraftPanelRendering(false)
```

3.3.2.2 set3DRendering

```
sasl.options.set3DRendering(boolean isOn)
```

Enables/disables 3D rendering for SASL project. **isOn** defines whether the rendering should be enabled or disabled. By default, 3D rendering is off (since SASL 3.14).

main.lua, disabling 3D rendering

```
-- Disabling 3D rendering
sasl.options.set3DRendering(false)
```

3.3.2.3 setInteractivity

```
sasl.options.setInteractivity(boolean isOn)
```

Enables/disables some default interactive abilities for SASL project. **isOn** defines whether the subset of interactivity should be enabled or disabled. When interactivity is partially disabled using this option, mouse and keyboard callbacks won't be called for components on 3D aircraft panel, and for components placed inside deprecated **subpanel** windows. Also, with disabled interactivity, functions **getCSMouseIsOnPanel** and **getCSPanelMousePos** won't return valid data. **contextWindow** and all their functional will work even if this option is set to **false**, so you can freely set this option to **false** if you only need **contextWindow** functional and their interactive capabilities. By default, interactivity is on.

main.lua, disabling interactivity

```
-- Disabling interactivity
sasl.options.setInteractivity(false)
```

3.3.3 Rendering Customization

3.3.3.1 setRenderingMode2D

```
sasl.options.setRenderingMode2D(RenderingMode2DIdentifier ID)
```

Changes current 2D rendering mode based on the passed **ID** value. **ID** can be one of the following pre-defined constants:

- SASL_RENDER_2D_DEFAULT - default 2D rendering mode using single draw pass.
- SASL_RENDER_2D_MULTIPASS - advanced 2D rendering mode using separate draw calls for lit and non-lit drawing.

Warning: in case of using SASL_RENDER_2D_MULTIPASS mode you are responsible for correct handling of lit and non-lit drawing. When **draw** function of component will be called for the Aircraft Panel, use **isLitStage** and **isNonLitStage** functions from **Graphics** section for determining current rendering stage.

main.lua, setting advanced 2D rendering mode

```
sasl.options.setRenderingMode2D(SASL_RENDER_2D_MULTIPASS)

function draw()
    if sasl.gl.isNonLitStage() then
        -- ...
    end
    if sasl.gl.isLitStage() then
        -- ...
    end

    drawAll(components)
end

components = {
    myPanelComponent { position = { 0, 0, 300, 200 } },
    myPanelComponent2 { position = { 430, 10, 200, 200 } }
}
```

3.3.3.2 setPanelRenderingMode

```
sasl.options.setPanelRenderingMode(PanelRenderingModeIdentifier ID)
```

Changes current rendering mode of Aircraft Panel based on the passed **ID** value. **ID** can be one of the following pre-defined constants:

- SASL_RENDER_PANEL_DEFAULT - default panel rendering mode using single Aircraft Panel draw call after X-Plane
- SASL_RENDER_PANEL_BEFORE_AND_AFTER - advanced panel rendering mode using separate draw calls before X-Plane rendering on panel and after

Warning: in case of using SASL_RENDER_PANEL_BEFORE_AND_AFTER mode you are responsible for correct handling of drawing before and after X-Plane. When **draw** function of component will be called for the Aircraft Panel, use **isPanelBeforeStage** and **isPanelAfterStage** functions from **Graphics** section for determining current rendering stage.

main.lua, setting advanced panel rendering mode

```
sasl.options.setPanelRenderingMode(SASL_RENDER_PANEL_BEFORE_AND_AFTER)

function draw()
    if sasl.gl.isPanelBeforeStage() then
        -- ...
    end
    if sasl.gl.isPanelAfterStage() then
        -- ...
    end

    drawAll(components)
end

components = {
    myPanelComponent { position = { 0, 0, 300, 200 } },
    myPanelComponent2 { position = { 430, 10, 200, 200 } }
}
```

3.3.3.3 setCustomBlending

```
sasl.options.setCustomBlending(boolean isOn)
```

Enables/disables ability to alter blending parameters when drawing to 2D targets. By default it is enabled.

main.lua, custom blending option

```
sasl.options.setCustomBlending(false)
```

3.3.3.4 setUpdateDrawingReady

```
sasl.options.setUpdateDrawingReady(boolean isOn)
```

Enables/disables possible optimization when SASL project uses rendering into texture targets in **update** callback. When this option is on, during **update** function calls drawing subsystem will be correctly initialized before the call, and multiple consecutive calls to **setRenderTarget** and **restoreRenderTarget** won't cause re-initialization. By default it is disabled, in order to not initialize drawing subsystem when it's not used at all.

main.lua

```
sasl.options.setUpdateDrawingReady(true)
```

3.3.4 Debugging

3.3.4.1 setLuaErrorsHandling

```
sasl.options.setLuaErrorsHandling(ErrorsHandlingModeID ID)
```

Sets different modes for handling Lua errors during project development, depending on the passed **ID** value. **ID** can be one of the following pre-defined constants:

- SASL_KEEP_PROCESSING - default errors handling mode.
- SASL_STOP_PROCESSING - mode, which stops SASL processing when error occures. This mode can be useful during development stage, when you don't want same error stack trace dumped every drawing frame or update cycle. This mode is unlikely very useful in release versions of SASL projects, so make sure that proper mode is used for release versions.

main.lua, setting errors handling mode

```
sasl.options.setLuaErrorsHandling(SASL_STOP_PROCESSING)
```

3.3.4.2 setLuaStackTraceLimit

```
sasl.options.setLuaStackTraceLimit(number limit)
```

Sets limit for Lua stack trace entries, which is used during warning or error message dump. Default stack trace limit is 6.

main.lua, setting stack trace limit

```
sasl.options.setLuaStackTraceLimit(5)
```

3.3.4.3 setLuaWarningsAsErrors

```
sasl.options.setLuaWarningsAsErrors(boolean isOn)
```

Enables/disables mode, where all Lua warnings are treated as errors.

3.4 Windows (XP11)

Functions from this section are can be used for gathering general information about simulator desktop window, monitors count, and monitors bounds in different coordinate systems (Global simulator desktop bounds and OS bounds).

For the same monitor, monitor IDs, which will be returned from **getMonitorsIDsGlobal** and **getMonitorsIDsOS** functions will match. But for the same monitor ID, global bounds and OS bounds may not match, because of different coordinates systems (since the X-Plane global desktop may not match the operating system's global desktop and due to UI scaling applied).

3.4.1 getMonitorsIDsGlobal

```
table ids = sasl.windows.getMonitorsIDsGlobal()
```

Returns the table of monitor IDs, which are covered with global simulator desktop window (only monitors with simulator in full-screen mode are included). Monitors with only an X-Plane window (not in full-screen mode) will not be included.

3.4.2 getMonitorsIDsOS

```
table ids = sasl.windows.getMonitorsIDsOS()
```

Returns the table of all monitor IDs in the OS. This may include monitors that are not covered by the simulator window.

3.4.3 getMonitorBoundsGlobal

```
number x, number y, number width, number height =
    sasl.windows.getMonitorBoundsGlobal(number id)
```

Returns the bounds (taking scaling into account) of each full-screen X-Plane window within the X-Plane global desktop space. **id** is a numeric identifier of particular monitor. Use **getMonitorsIDsGlobal** function to get all monitors IDs.

If X-Plane is running in full-screen and your monitors are of the same size and configured contiguously in the OS, then the combined global bounds of all full-screen monitors will match the total global desktop bounds, as returned by **getScreenBoundsGlobal** function. (Of course, if X-Plane is running in windowed mode, this will not be the case.

Likewise, if you have differently sized monitors, the global desktop space will include wasted space.)

test.lua, traversing global monitors bounds

```
local ids = sasl.windows.getMonitorsIDsGlobal()
for i = 1, #ids do
    local x, y, w, h = sasl.windows.getMonitorBoundsGlobal(ids[i])
    print(ids[i], x, y, w, h)
end
```

3.4.4 getMonitorBoundsOS

```
number x, number y, number width, number height =
    sasl.windows.getMonitorBoundsOS(number id)
```

Returns the bounds of the monitor in OS pixels. **id** is a numeric identifier of particular monitor. Use **getMonitorsIDsOS** function to get all monitors IDs.

test.lua, traversing OS monitors bounds

```
local ids = sasl.windows.getMonitorsIDsOS()
for i = 1, #ids do
    local x, y, w, h = sasl.windows.getMonitorBoundsOS(ids[i])
    print(ids[i], x, y, w, h)
end
```

3.4.5 getScreenBoundsGlobal

```
number x, number y, number width, number height =
    sasl.windows.getScreenBoundsGlobal()
```

This routine returns the bounds of the "global" X-Plane desktop. This function is multi-monitor aware and takes into account current UI scaling option.

If the user is running X-Plane in full-screen on two or more monitors (typically configured using one full-screen window per monitor), the global desktop will be sized to include all X-Plane windows.

The origin of the screen coordinates is not guaranteed to be (0, 0). Suppose the user has two displays side-by-side, both running at 1080p. Suppose further that they've configured their OS to make the left display their "primary" monitor, and that X-Plane is running in full-screen on their right monitor only. In this case, the global desktop bounds would be the rectangle from (1920, 0) to (3840, 1080). If the user later asked X-Plane to draw on their primary monitor as well, the bounds would change to (0, 0) to (3840, 1080).

If the usable area of the virtual desktop is not a perfect rectangle (for instance, because the monitors have different resolutions or because one monitor is configured in the operating system to be above and to the right of the other), the global desktop will include any wasted space. Thus, if you have two 1080p monitors, and monitor 2 is configured to have its bottom left touch monitor 1's upper right, your global desktop area would be the rectangle from (0, 0) to (3840, 2160).

Note that popped-out windows (windows drawn in their own operating system windows, rather than "floating" within X-Plane) are not included in these bounds.

3.5 Commands

Commands interface provides ability to manipulate simulator commands in any possible way: find and execute already existing commands, and create new project-specific commands. In order to create functioning custom command, developer need to create new command first with **createCommand** function and specify the callback function (or multiple callback functions) for the command with **registerCommandHandler** routine.

3.5.1 findCommand

```
command commandID = sasl.findCommand(string name)
```

Find simulator command by specified **name**. Returns command or **nil** if corresponding command can't be found.

commands.lua, typical command lookup

```
viewOutsideCommand = sasl.findCommand("sim/view/chase")
```

3.5.2 commandBegin

```
sasl.commandBegin(number commandID)
```

Starts command execution. Obtain **commandID** with **findCommand** function.

commands.lua, command execution

```
viewOutsideCommand = sasl.findCommand("sim/view/chase")
sasl.commandBegin(viewOutsideCommand)
```

3.5.3 commandEnd

```
sasl.commandEnd(number commandID)
```

Finishes command execution. Obtain **commandID** with **findCommand** function.

commands.lua, command execution

```
viewOutsideCommand = sasl.findCommand("sim/view/chase")
sasl.commandBegin(viewOutsideCommand)
sasl.commandEnd(viewOutsideCommand)
```

3.5.4 commandOnce

```
sasl.commandOnce(number commandID)
```

Starts and finishes command immediately. Obtain **commandID** with **findCommand** function.

commands.lua, command execution

```
viewOutsideCommand = sasl.findCommand("sim/view/chase")
sasl.commandOnce(viewOutsideCommand)
```

3.5.5 createCommand

```
command commandID = sasl.createCommand(string name, string description)
```

Creates new command, specified by **name** and **description**. Returns command if command was successfully created or **nil** in case of errors. All created command can be seen in corresponding simulator menu along with provided descriptions.

commands.lua, custom command creation

```
testCommand = sasl.createCommand("project/test/test_command", "Description")
```

3.5.6 registerCommandHandler

```
sasl.registerCommandHandler(command commandID, number isBefore, function
    handler)
```

Adds handler to command **commandID**. If **isBefore** equals to 1, **handler** will be called before simulator handles the command. If **isBefore** equals to 0, **handler** will be called after simulator. **handler** is a function with one argument.

```
function handler(number phase)
```

phase can be equal to one of pre-defined constants:

- SASL_COMMAND_BEGIN - command started.
- SASL_COMMAND_CONTINUE - command execution continues.
- SASL_COMMAND_END - command finished.

Command handler must return 0 to stop further command processing or return 1 to allow more handlers to do their job.

commands.lua, specifying commands callback

```
function testCommandHandler(number phase)
    if phase == SASL_COMMAND_BEGIN then
        print("Started!")
    else if phase == SASL_COMMAND_END then
        print("Finished!")
    end
    return 1
end

testCommand = sasl.createCommand("project/test/test_command", "Description")
sasl.registerCommandHandler(testCommand, 0, testCommandHandler)
```

3.5.7 unregisterCommandHandler

```
sasl.unregisterCommandHandler(command commandID, number isBefore)
```

Removes command handler from command **commandID** and **isBefore** pair. Use this function to change behaviour of your command (unregister old callback and register new ones) or to disable command functional (unregister all callbacks).

commands.lua, unregistering command handler

```
sasl.registerCommandHandler(testCommand, 0, testCommandHandler)
...
sasl.unregisterCommandHandler(testCommand, 0)
```

3.6 Menus

SASL provides functions for creating and manipulating simulator menus. There are two basic entities in SASL that allow you to interfere with menus: menu identifiers called **menuID**, and menu items identifiers called **menuItemID**.

All created main menu items can be appended only to simulator **plugins** menu that already exists for that purpose.

Note that you do not have to necessarily clean up menus stuff, when your SASL project is about to be unloaded. This will be performed automatically, so you can use provided clean-up routines (**removeMenuItem**, **clearAllMenuItems**, **destroyMenu**) only if you want to change your menus.

There are two pre-defined constant value of **menuID**:

- PLUGINS_MENU_ID - corresponds to simulator **plugins** menu.
- AIRCRAFT_MENU_ID - corresponds to simulator **aircraft** menu. Only available for **aircraft** projects (XP11).

All menu items has state, that can be defined by following type:

Type: **MenuItemState**
Description: menu item state identifier, can be equal to one of pre-defined constants:

- MENU_NO_CHECK - menu item hasn't check mark. Default state.
- MENU_UNCHECKED - menu item is unchecked now.
- MENU_CHECKED - menu item is checked now.

3.6.1 appendMenuItem

```
menuItemID id = sasl.appendMenuItem(menuID inMenuID, string name, function
    callback)
menuItemID id = sasl.appendMenuItem(menuID inMenuID, string name)
```

Appends new menu item with **name** to the menu with specified **inMenuID**. **callback** is a function without arguments, which will be called, when menu item will be clicked. **callback** argument can be omitted, if the created menu item only must contain other sub-menus. Creating project menus must be started from calling this function and appending menus to the simulator **plugins** menu.

menus.lua, appending new menu item

```
-- Appending new menu item to simulator's plugins menu.
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "TestMenu")
```

3.6.2 appendMenuItemWithCommand (XP11)

```
menuItemID id = sasl.appendMenuItemWithCommand(menuID inMenuID, string name,
     command commandID)
```

Appends new menu item with **name** to the menu with specified **inMenuID**. **commandID** is an identifier of a command, which will be executed after click on corresponding menu item. This function may be used instead of **appendMenuItem** function with providing callback function. Use this function for creating menu items without sub-menus.

3.6.3 removeMenuItem

```
sasl.removeMenuItem(menuID inMenuID, menuItemID inMenuItemID)
```

Removes menu item specified by **inMenuItemID** from menu that corresponds to **inMenuID**.

menus.lua, removing menu item

```
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "TestMenu")
sasl.removeMenuItem(PLUGINS_MENU_ID, testMenuItemID)
```

3.6.4 setMenuItemName

```
sasl.setMenuItemName(menuID inMenuID, menuItemID inMenuItemID, string name)
```

Sets name for menu item, specified by **inMenuItemID** that corresponds to menu with **inMenuID**.

menus.lua, rename menu item

```
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "InitialName")
...
sasl.setMenuItemName(PLUGINS_MENU_ID, testMenuItemID, "ChangedName")
```

3.6.5 setMenuItemState

```
sasl.setMenuItemState(menuID inMenuID, menuItemID inMenuItemID,
    MenuItemState inState)
```

Sets current state of menu item, specified by **inMenuItemID** that corresponds to menu with **inMenuID**.

menus.lua, changing state of menu item

```
sasl.setMenuItemState(PLUGINS_MENU_ID, testMenuItemID, MENU_CHECKED)
```

3.6.6 getMenuItemState

```
MenuItemState state = sasl.getMenuItemState(menuID inMenuID, menuItemID
    inMenuItemID)
```

Gets current state of menu item, specified by **inMenuItemID** that belongs to menu with **inMenuID**.

menus.lua, getting current state of menu item

```
testState = sasl.getMenuItemState(PLUGINS_MENU_ID, testMenuItemID)
```

3.6.7 enableMenuItem

```
sasl.enableMenuItem(menuID inMenuID, menuItemID inMenuItemID, number
    inEnable)
```

Enables or disables menu item, specified by **inMenuItemID** that corresponds to menu **inMenuID**. If **inEnable** is equal to 1, then item will be enabled (active). If **inEnable** is equal to 0, then item will be disabled (greyed out).

menus.lua, disabling menu item

```
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "Name")
...
sasl.enableMenuItem(PLUGINS_MENU_ID, testMenuItemID, 0)
```

3.6.8 createMenu

```
menuID id = sasl.createMenu(string name, menuID parentMenuID, menuItemID
    parentMenuItemID)
```

Creates new child menu with specified parent menu **parentMenuID** in menu item specified by **parentMenuItemID**. Returns identifier of created menu object.

menus.lua, creating menu in menu item

```
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "Project")
testMenuID = sasl.createMenu("Project", PLUGINS_MENU_ID, testMenuItemID)
```

3.6.9 appendMenuSeparator

```
sasl.appendMenuSeparator(menuID inMenuID)
```

Appends menu separator to the menu, specified by **inMenuID**. Separator will be appended to the end of current menu items list.

menus.lua, appending separator to menu

```
sasl.appendMenuSeparator(testMenuID)
```

3.6.10 clearAllMenuItems

```
sasl.clearAllMenuItems(menuID inMenuID)
```

Deletes all menu items from menu that corresponds to **inMenuID**. Use this function if you want to create new list of menu items.

menus.lua, removing all menu items from menu

```
sasl.clearAllMenuItems(PLUGINS_MENU_ID, testMenuID)
```

3.6.11 destroyMenu

```
sasl.destroyMenu(menuID inMenuID)
```

Destroys menu, specified by **inMenuID** argument.

menus.lua, destroying menu

```
testMenuItemID = sasl.appendMenuItem(PLUGINS_MENU_ID, "Project")
testMenuID = sasl.createMenu("Project", PLUGINS_MENU_ID, testMenuItemID)
...
sasl.destroyMenu(testMenuID)
```

3.7 Message Windows

Message windows can be used for simple user informing or basic interaction with user ("Yes or No" dialogs, choices).

3.7.1 messageWindow

```
sasl.messageWindow(number x, number y, number width, number height, string
    title, string message, number buttonsCount, ...)

sasl.messageWindow(number x, number y, number width, number height, string
    title, string message, 0, number lifetime)
sasl.messageWindow(number x, number y, number width, number height, string
    title, string message, 1, string buttonName1, function callback1)
sasl.messageWindow(number x, number y, number width, number height, string
    title, string message, 2, string buttonName1, function callback1, string
     buttonName2, function callback2)
...
```

Creates interactive message window for users. The location of message window is defined by **x**, **y**, **width** and **height** arguments, where **x** and **y** is coordinates of left bottom corner of the window. Window title and the message itself are specified as fifth and sixth argument. **buttonsCount** is a number of dialog buttons for the message window. If **buttonsCount** is equal to 0, then you must specify the lifetime of window as eighth argument (in seconds). In case **buttonsCount** is not equals to 0, then you must provide additional **buttonsCount** pairs of arguments. Every such pair consists from buttons name and callback function. Callbacks for buttons are functions without arguments. Corresponding callback will be called after click on button and then message window will disappear.

messageWindow.lua, showing message windows

```
function testYesCallback()
    print("Yes--pressed")
end

function testNoCallback()
    print("No--pressed")
end

sasl.messageWindow(1000, 600, 400, 250, "MessageTitle", "Message", 2, "YES",
     testYesCallback, "NO", testNoCallback)

sasl.messageWindow(600, 600, 300, 200, "MessageTitle", "Message", 0, 10)
```

3.8 Cameras

Camera state in SASL defined by set of 7 parameters: **x**, **y**, **z**, **pitch**, **yaw**, **roll** and **zoom**. **x**, **y**, **z** is a coordinates of camera in global OpenGL coordinates system. **pitch**, **yaw** and **roll** are rotation factors from a camera facing flat north in degrees. **zoom** defines current zooming factor.

To control the camera you must register a special camera controller callback with **registerCameraControlling** function, take control wit **startCameraControl** function and use **setCamera** function to set current camera position. For leaving camera control you must use **stopCameraControl** function.

Note that you do not necessarily have to unregister your camera controllers, when the SASL project is about to be unloaded. This will be performed automatically.

Current camera state can be defined with following type:

Type: **CameraStatus**
Description: camera state identifier, can be equal to one of pre-defined constants:

- CAMERA_NOT_CONTROLLED - camera not controlled.
- CAMERA_CONTROLLED_UNTIL_VIEW_CHANGE - camera controlled until forced view change.
- CAMERA_CONTROLLED_ALWAYS - camera controlled.

3.8.1 getCamera

```
number x, number y, number z, number pitch, number yaw, number roll, number
    zoom = sasl.getCamera()
```

Gets current camera state.

3.8.2 setCamera

```
sasl.setCamera(number x, number y, number z, number pitch, number yaw,
    number roll, number zoom)
```

Sets current camera state.

3.8.3 registerCameraController

```
number id = sasl.registerCameraController(function callback)
```

Registers new camera controller with provided **callback**. When you take camera control with this controller, **callback** will be called every time when simulator is needed to set camera state. Function returns numeric camera controller identifier that can be used to take camera control.

cameras.lua, registering camera control function

```
planeX = globalPropertyf("sim/flightmodel/position/local_x")
planeY = globalPropertyf("sim/flightmodel/position/local_y")
planeZ = globalPropertyf("sim/flightmodel/position/local_z")

testCameraHeading = 0.0
testCameraPitch = 0.0
testCameraDistance = 200.0
testCameraAdvance = 0.2

function testCameraController()
    testCameraHeading = (testCameraHeading + 0.1) % 360

    if testCameraDistance > 400.0 or testCameraDistance < 45.0 then
        testCameraAdvance = -testCameraAdvance
    end
    testCameraDistance = testCameraDistance + testCameraAdvance

    dx = -testCameraDistance * math.sin(testCameraHeading * 3.1415 / 180.0)
    dz = testCameraDistance * math.cos(testCameraHeading * 3.1415 / 180.0)
    dy = testCameraDistance / 5

    x = get(planeX) + dx
    y = get(planeY) + dy
    z = get(planeZ) + dz

    sasl.setCamera(x, y, z, testCameraPitch, testCameraHeading, -30.0, 1.0)
end

testControllerID = sasl.registerCameraController(testCameraController)
```

3.8.4 unregisterCameraController

```
sasl.unregisterCameraController(number id)
```

Unregisters camera controller function with provided numeric identifier **id**.

cameras.lua, unregistering camera control function

```
sasl.unregisterCameraController(testControllerID)
```

3.8.5 getCurrentCameraStatus

```
CameraStatus status = sasl.getCurrentCameraStatus()
```

Gets current camera status.

3.8.6 startCameraControl

```
sasl.startCameraControl(number id, CameraStatus status)
```

Starts camera control with camera controller, specified by numeric identifier **id** and with provided **status**. Provided **status** can't be equal to CAMERA_NOT_CONTROLLED.

cameras.lua, taking control over camera

```
if (sasl.getCurrentCameraStatus() ~= CAMERA_CONTROLLED_ALWAYS) then
    sasl.startCameraControl(testControllerID,
        CAMERA_CONTROLLED_UNTIL_VIEW_CHANGE)
end
```

3.8.7 stopCameraControl

```
sasl.stopCameraControl()
```

Stops camera controlling.

3.9 Process

3.9.1 startProcessSync

```
boolean status, userdata pOut, number outSize, userdata pErr, number errSize
    , string sOut, string sErr = sasl.startProcessSync(string path, table
    args, string toStdIn, boolean stdOutToString, boolean stdErrToString)
```

Synchronously start process, specified by **path**. If the process needs passed parameters, you can pass them to as table of strings **args**, with each string representing single parameter. If the process need passed data in stdin, pass it to **toStdIn**. By default, output data and error data will be returned as raw pointers and the data sizes, you can set **stdOutToString** as **true** and get data to string **sOut**. Same applies for the error data receiving. If **status** is **true**, the process has finished without errors and you can get output data from **pOut** pointer. **outSize** in the size of output data. Same applies for the error data if returned **status** is **false**.

testProcessWindows.lua, starting process

```
fullPathToCoutHelloWorld = moduleDirectory .. "/Custom␣Module/testExecute/
    coutHelloWorld.exe"
status, pOut, outSize, pErr, errSize, sOut = sasl.startProcessSync(
    fullPathToCoutHelloWorld, "", "", true, false)

if status then
    print(pOut)
    print(outSize)
    print(sOut) -- print "Hello World!"
else
    print(pErr)
    print(errSize)
end
```

3.9.2 startProcessAsync

```
sasl.startProcessAsync(string path, table args, string toStdIn, boolean
    stdOutToString, boolean stdErrToString, function callback)
```

Asynchronously start process, specified by **path**. If the process needs passed parameters, you can pass them to as table of strings **args**, with each string representing single parameter. If the process need passed data in stdin, pass it to **toStdIn**. By default, output data and error data in the **callback** function will be returned as raw pointers and the data sizes, but you can set **stdOutToString** as **true** and get data to string **sOut** in the callback. Same applies for the error data receiving. Last argument is a **callback** function, which will be called upon process finish.

```
function callback(boolean status, number pOut, number outSize, number pErr,
    number errSize, string inOutString, string inErrString)
```

Callback parameters are the same as returned values in **startProcessSync**.

testProcessWindows.lua, starting process

```
function onProcessFinished(inIsOk, inRefToOut, inSizeOut, inRefToError,
    inSizeToError, inOutString, inErrString)
    if inIsOk then
        logInfo("Process finished!")
        logInfo(inRefToOut)
        logInfo(inSizeOut )
    else
        logInfo("Process finished with error!")
        logInfo(inRefToError)
        logInfo(inSizeToError)
        logInfo(inErrString)
    end
end

fullPathToExe = moduleDirectory .. "/Custom Module/testExecute/program.exe"
startProcessAsync(fullPathToExe, "--args", "", true, true, onProcessFinished
    )
```

3.10 Net

3.10.1 downloadFileSync

```
boolean result, string error = sasl.net.downloadFileSync(string url, string
    path)
```

Synchronously downloads file, specified by **url** and writes it to the specified **path**. Function returns only when downloading is done or in case of errors (lost internet connection, etc). Returns **true** on success and returns **false** otherwise. In case of unsuccessful downloading, error message string is returned as second return value.

test.lua, downloading file from server

```
downloadResult, error = sasl.net.downloadFileSync("http://mycoolsite.com/
    myFile.txt", moduleDirectory.."/myFile.txt")

if not downloadResult then
    sasl.logWarning("Downloading:", error)
end
```

3.10.2 downloadFileAsync

```
sasl.net.downloadFileAsync(string url, string path, function callback)
```

Asynchronously downloads file, specified by **url** and writes it to the specified **path**. When downloading ended, callback will be called.

```
callback(string url, string path, boolean isOk, string error)
```

Callback contains **url** and **path** with which **downloadFileAsync** were called, and error message in case of an error.

test.lua, downloading file from server

```
function onFileDownloaded(inUrl, inFilePath, inIsOk, inError)
    if inIsOk then
        logInfo("File downloaded!")
        logInfo(inUrl)
        logInfo(inFilePath)
    else
        logInfo(inUrl)
        logInfo(inFilePath)
        logWarning(inError)
    end
end

sasl.net.downloadFileAsync("http://mycoolsite.com/myFile.txt",
    moduleDirectory.. "/myFile.txt", onFileDownloaded)
```

3.10.3 downloadFileContentsSync

```
boolean result, string contents = sasl.net.downloadFileContentsSync(string
    url)
```

Synchronously downloads contents from file, specified by **url**. Function returns only when downloading is done or in case of errors (lost internet connection, etc). Function returns two values - first one is **result**(equal to **true** on success and to **false** otherwise). If **result** is **true**, second returned value is file contents. Otherwise second value is error message string.

test.lua, downloading file contents from server

```
downloadResult, contents = sasl.net.downloadFileContentsSync("http://
    mycoolsite.com/myFile.txt")

if downloadResult then
    -- ... process data
else
    sasl.logWarning("Downloading:", contents)
end
```

3.10.4 downloadFileContentsAsync

```
sasl.net.downloadFileContentsAsync(string url, function callback)
```

Asynchronously downloads contents from file, specified by **url**. When downloading ended, callback will be called.

```
callback(string url, string contents, boolean isOk, string error)
```

Callback contains **url** with which **downloadFileAsync** were called, **contents** in case of downloading was successful or error message in case of an error.

test.lua, downloading file contents from server

```
function onContentsDownloaded(inUrl, inString, inIsOk, inError)
    if inIsOk then
        logInfo("String downloaded!")
        logInfo(inUrl)
        logInfo(inString)
    else
        logInfo(inUrl)
        logWarning(inError)
    end
end

sasl.net.downloadFileContentsAsync("http://25.io/toau/audio/sample.txt",
    onContentsDownloaded)
```

3.10.5 setDownloadTimeout

```
sasl.net.setDownloadTimeout(TimeoutType type, number time)
sasl.net.setDownloadTimeout(TimeoutType type, number time, number speed)
```

Sets download timeout options for SASL project net functionality. **TimeoutType** can be equal to the following values:

- SASL_TIMEOUT_CONNECTION - sets connection timeout
- SASL_TIMEOUT_OPERATION - sets timeout for whole download operation
- SASL_TIMEOUT_SPEEDLIMIT - sets timeout based on download speed limit

time is value of timeout limit in seconds. In case if timeout will be set based on download speed limit, third argument is required - **speed** (in bytes per second). In this case operation will be aborted if the speed with be slower than **speed** during **time** seconds.

Pass SASL_TIMEOUT_VALUE_DEFAULT as **time** to reset the timeout to the default value.

3.11 Timers

3.11.1 createTimer

```
timerID id = sasl.createTimer()
```

Creates new simple timer object based on simulator clock and returns its identifier **id**.

timers.lua, new timer object creation

```
testTimerID = sasl.createTimer()
```

3.11.2 createPerformanceTimer

```
timerID id = sasl.createPerformanceTimer()
```

Creates new high resolution performance timer object and returns its identifier **id**. This timer can be used to measure time within callbacks.

timers.lua, new timer object creation

```
testTimerID = sasl.createPerformanceTimer()
```

3.11.3 deleteTimer

```
sasl.deleteTimer(timerID id)
```

Deletes timer object by specific timer **id**. Note that you do not necessarily need to delete your timers when your project is about to be unloaded. This will be performed automatically by SASL.

timers.lua, deleting timer

```
testTimerID = sasl.createTimer()
...
sasl.deleteTimer(testTimerID)
```

3.11.4 startTimer

```
sasl.startTimer(timerID id)
```

Starts timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, starting timer

```
testTimerID = sasl.createTimer()
sasl.startTimer(testTimerID)
```

3.11.5 pauseTimer

```
sasl.pauseTimer(timerID id)
```

Pauses timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, set timer to pause

```
testTimerID = sasl.createTimer()
sasl.startTimer(testTimerID)
...
sasl.pauseTimer(testTimerID)
```

3.11.6 resumeTimer

```
sasl.resumeTimer(timerID id)
```

Resumes previously paused timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, set timer to pause

```
sasl.pauseTimer(testTimerID)
...
sasl.resumeTimer(testTimerID)
```

3.11.7 stopTimer

```
sasl.stopTimer(timerID id)
```

Stops timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, stopping timer

```
sasl.startTimer(testTimerID)
...
sasl.stopTimer(testTimerID)
```

3.11.8 resetTimer

```
sasl.resetTimer(timerID id)
```

Resets timer to its initial state.

3.11.9 getElapsedSeconds

```
number seconds = sasl.getElapsedSeconds(timerID id)
```

Returns elapsed time in seconds for timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, getting current time

```
sasl.startTimer(testTimerID)
...
time = sasl.getElapsedSeconds(testTimerID)
```

3.11.10 getElapsedMicroseconds

```
number seconds = sasl.getElapsedMicroseconds(timerID id)
```

Returns elapsed time in microseconds for timer, specified by **id**. Obtain timer identifier with **createTimer** function.

timers.lua, getting current time

```
sasl.startTimer(testTimerID)
...
time = sasl.getElapsedMicroseconds(testTimerID)
```

3.11.11 getCurrentCycle

```
number currentCycle = sasl.getCurrentCycle()
```

Returns overall count of performed updating cycles in simulator.

3.12 Interplugin Communications

Plugins in simulator system can be identified by value of following type:

Type: **PluginID**
Description: plugin identifier, can be equal to special pre-defined constant:

- NO_PLUGIN_ID - means that ID not corresponds to any available plugin.

3.12.1 Interplugin Utilities

3.12.1.1 scheduleProjectReboot

```
sasl.scheduleProjectReboot()
```

Schedules manual SASL project reboot in the next update cycle. This utility function can be used if you need to reboot SASL project asynchronously from Lua code.

main.lua, requesting reboot

```
-- Reboot SASL on "Start New Flight" XP event
function onAirportLoaded(flightNumber)
    if flightNumber > 1 then
        sasl.scheduleProjectReboot()
    end
end
```

3.12.1.2 getMyPluginID

```
PluginID id = sasl.getMyPluginID()
```

Returns identifier of the SASL project plugin in simulator plugins system.

3.12.1.3 getMyPluginPath

```
string path = sasl.getMyPluginPath()
```

Returns full path to SASL project plugin.

3.12.1.4 getXPlanePath

```
string path = sasl.getXPlanePath()
```

Returns full path to simulator folder.

3.12.1.5 getProjectPath

```
string path = sasl.getProjectPath()
```

Returns full path to the SASL project folder (for every project type and location).

3.12.1.6 getProjectName

```
string name = sasl.getProjectName()
```

Returns name of SASL project.

3.12.1.7 getAircraftPath

```
string path = sasl.getAircraftPath()
```

Returns full path to the currently loaded aircraft. Function must be called only after aircraft is loaded.

3.12.1.8 getAircraft

```
string filename = sasl.getAircraft()
```

Returns filename of the currently loaded aircraft. Function must be called only after aircraft is loaded.

3.12.1.9 countPlugins

```
number count = sasl.countPlugins()
```

Returns total number of currently loaded plugins in simulator system.

3.12.1.10 getNthPlugin

```
PluginID id = sasl.getNthPlugin(number index)
```

Returns the identifier of the plugin, represented by **index** in simulator plugins system. **index** is 0-based from 0 to **countPlugins()**-1. Plugins identifiers may be returned in any arbitrary order. In case there is no plugin with such **index**, a special value NO_PLUGIN_ID will be returned.

3.12.1.11 findPluginByPath

```
PluginID id = sasl.findPluginByPath(string path)
```

Returns the identifier of the plugin, which located in specified **path**. A special value NO_PLUGIN_ID may be returned.

3.12.1.12 findPluginBySignature

```
PluginID id = sasl.findPluginBySignature(string signature)
```

Returns the identifier of the plugin with specified **signature**. A special value NO_PLUGIN_ID may be returned.

3.12.1.13 getPluginInfo

```
string name, string path, string signature, string description =
    sasl.getPluginInfo(PluginID id)
```

Returns the set of information about plugin, specified by **id**. The information contains plugin name, path to plugin, its signature and description. If there is no such **id** in the plugin system, then returned info will contain empty strings.

3.12.1.14 isPluginEnabled

```
number enabled = sasl.isPluginEnabled(PluginID id)
```

Returns the state (enabled or disabled) identifier of plugin with specified **id**. Returns 1 if plugin is enabled and 0 if plugin is disabled.

3.12.1.15 enablePlugin

```
number success = sasl.enablePlugin(PluginID id)
```

Enables plugin, specified by **id**. The function returns 1 in case of successful enabling or 0 if the plugin refused to enable.

3.12.1.16 disablePlugin

```
sasl.disablePlugin(PluginID id)
```

Disables plugin, specified by **id**.

3.12.1.17 reloadPlugins

```
sasl.reloadPlugins()
```

Reloads all plugins in simulator system.

3.12.2 Messages

SASL projects are able to send/receive messages to/from any other plugin, loaded in simulator system. It is also possible to send data with the message. To receive messages you must register message handlers. Note that you do not need to necessarily unregister your handlers when SASL project is about to be unloaded. This will be performed automatically on project unloading stage.

Type: **MessageDataType**
Description: type identifier for interplugin messaging, can be equal to one of pre-defined constants:

- TYPE_UNKNOWN
- TYPE_INT_ARRAY
- TYPE_FLOAT_ARRAY
- TYPE_STRING

3.12.2.1 registerMessageHandler

```
sasl.registerMessageHandler(number messageID, MessageDataType type, function
    callback)
```

Registers new message handler for message with unique specified **messageID**. Handler receives data that corresponds to data type identifier **type**. **callback** will be called every time, when you receive corresponding message. **callback** function might have following signatures (depending on specified data type):

```
function callback(PluginID id, number messageID, string data),
function callback(PluginID id, number messageID, table data),
function callback(PluginID id, number messageID).
```

data argument can be omitted if the data is not supplied with message, which has been sent to your project plugin.

Warning: be careful in case of handling messages with some data, such messaging is works only in case of messaging between two SASL plugins. If you want to exchange some data between SASL plugin and other plugin, other plugin should use technique described in **Appendix B**.

messages.lua, registering message handler

```
function testMessageCallback(id, messageID)
    print("Message!")
end

-- Register simple message handler
sasl.registerMessageHandler(someMessageID, TYPE_UNKNOWN, testMessageCallback
    )
```

3.12.2.2 unregisterMessageHandler

```
sasl.unregisterMessageHandler(number messageID)
```

Unregisters message handler for message with unique specified **messageID**.

messages.lua, unregistering message handler

```
sasl.registerMessageHandler(someMessageID, TYPE_UNKNOWN, testMessageCallback
    )
...
sasl.unregisterMessageHandler(someMessageID)
```

3.12.2.3 sendMessageToPlugin

```
sasl.sendMessageToPlugin(PluginID id, number messageID, MessageDataType type
    , string data)
sasl.sendMessageToPlugin(PluginID id, number messageID, MessageDataType type
    , table data)
sasl.sendMessageToPlugin(PluginID id, number messageID, MessageDataType type
    )
```

Sends message with unique identifier **messageID** to the plugin with identifier **id**. If **id** is equal to NO_PLUGIN_ID, then the message will be sent to all enabled plugins. Last argument **data** can be omitted in case if **type** equals to TYPE_UNKNOWN.

messages.lua, sending messages

```
sasl.sendMessageToPlugin(NO_PLUGIN_ID, 20222, TYPE_STRING, "Hello!")
sasl.sendMessageToPlugin(somePluginID, 20223, TYPE_UNKNOWN")
```

3.13 Auxiliary Mouse Input System

SASL provides a number of functions for implementing advanced mouse input handling. Functions from this section must be used only in case if standard mouse handling with components can't help. Advanced mouse input system represents a simple finite state automaton and you are able to query current system state parameters and set these parameters.

Developer can use custom cursor to show depending on current cursor location or other factor. Cursor bitmaps located in special texture **cursors.png** inside **components** folder. Developer can use the default one or replace it with other texture. Requirements for this texture - size, size of one cursor and name. Texture size must be **512x512**, and one cursor size is **64x64**. Current custom cursor can be identified via **CursorID** numeric identifier. Below you will find correspondence scheme between texture part and **CursorID**:

id:11	id:12	id:13	id:14	id:15	id:16	id:17	id:18
id:21	id:22	id:23	id:24	id:25	id:26	id:27	id:28
id:31	id:32	id:33	id:34	id:35	id:36	id:37	id:38
id:41	id:42	id:43	id:44	id:45	id:46	id:47	id:48
id:51	id:52	id:53	id:54	id:55	id:56	id:57	id:58
id:61	id:62	id:63	id:64	id:65	id:66	id:67	id:68
id:71	id:72	id:73	id:74	id:75	id:76	id:77	id:78
id:81	id:82	id:83	id:84	id:85	id:86	id:87	id:88

Figure 3.1: Cursors ID scheme

There are a number of events that may be caught for three mouse buttons (left, right and middle):

- Mouse button down event
- Mouse button up event
- Mouse button hold event
- Mouse button double-click event
- Mouse button drag event
- Mouse wheel rotation events

Type: **MouseButton**
Description: mouse button identifier, can be equal to one of pre-defined constants:

- MB_LEFT
- MB_RIGHT
- MB_MIDDLE

3.13.1 setAuxiliaryClickSystem [DEPRECATED]

```
sasl.setAuxiliaryClickSystem(boolean isActive)
```

Enables auxiliary click system if **isActive** is **true**, and disables auxiliary click system if **isActive** is **false**.

Deprecation note: auxiliary click system is permanently enabled.

3.13.2 setCSDClickInterval

```
sasl.setCSDClickInterval(number interval)
```

Sets interval in seconds for double-click events. If the two clicks will happen in less than a time **interval**, double-click event will be generated.

mouseEvents.lua, setting interval for double-click

```
sasl.setCSDClickInterval(0.4)
```

3.13.3 getCSDClickInterval

```
number interval = sasl.getCSDClickInterval()
```

Returns interval in seconds for double-click events.

3.13.4 setCSMode

```
sasl.setCSMode(number mode)
```

Sets current auxiliary click system **mode**, represented by number. Currently there are only two modes. If **mode** is equal to 0, click events will not be generated and standard simulator cursor will be drawn. If **mode** is equal to 1, all click events will be generated and custom cursor will be drawn.

3.13.5 getCSMode

```
number mode = sasl.getCSMode()
```

Returns current auxiliary click system **mode**.

3.13.6 setCSShowCursor

```
sasl.setCSShowCursor(number cursorID)
```

Enables custom cursor showing and sets the cursor that corresponds to **cursorID**. If **cursorID** is equal to 0, custom cursor will be not shown. Custom cursors that can be drawn defined in special texture **cursors.png**, which located inside **components** folder.

mouseEvents.lua, show custom cursor with ID = 15

```
sasl.setCSMode(1)
sasl.setCSShowCursor(15)
```

3.13.7 getCSShowCursor

```
number cursorID = sasl.getCSShowCursor()
```

Returns current selected cursor identifier **cursorID**.

3.13.8 setCSWheelInteractionDelay

```
sasl.setCSWheelInteractionDelay(number delay)
```

Sets delay in seconds between last mouse zoom event and first possible mouse wheel custom event handling.

3.13.9 getCSWheelInteractionDelay

```
number delay = sasl.getCSWheelInteractionDelay()
```

Returns delay in seconds between last mouse zoom event and first possible mouse wheel custom event handling.

3.13.10 setCSPassWheelEventFlag

```
sasl.setCSPassWheelEventFlag(number flag)
```

Sets special flag for auxiliary click system to ignore mouse wheel events one update cycle. Pass 1 to set the flag.

3.13.11 setCSCursorScale [DEPRECATED]

```
sasl.setCSCursorScale(number scale)
```

Sets the scale of the drawn custom cursor. **scale** equals to 1.0 means standard cursor size, **scale** equals to 2.0 means cursor that two times larger and etc.

Deprecation note: does nothing in the current implementation.

3.13.12 getCSClickDown

```
number state = sasl.getCSClickDown(MouseButton buttonID)
```

Gets current mouse down state for mouse button, specified by **buttonID**. The function returns current button state identifier. If **state** is equal to 0, button is not down. If **state** is equal to 1, then button is down.

mouseEvents.lua, querying current right mouse button state

```
if (sasl.getCSClickDown(MB_RIGHT) == 1) then
    showContextMenu()
end
```

3.13.13 getCSClickUp

```
number state = sasl.getCSClickUp(MouseButton buttonID)
```

Gets current mouse up state for mouse button, specified by **buttonID**. The function returns current button state identifier. If **state** is equal to 0, button is not up. If **state** is equal to 1, then button is up.

3.13.14 getCSClickHold

```
number state = sasl.getCSClickHold(MouseButton buttonID)
```

Gets current mouse hold state for mouse button, specified by **buttonID**. The function returns current button state identifier. If **state** is equal to 0, button is not in hold state. If **state** is equal to 1, then button is in hold state.

3.13.15 getCSDoubleClick

```
number state = sasl.getCSDoubleClick(MouseButton buttonID)
```

Gets current mouse double-click state for mouse button, specified by **buttonID**. The function returns current button state identifier. If **state** is equal to 0, button was double-clicked. If **state** is equal to 1, then button was not double-clicked.

3.13.16 getCSWheelClicks

```
number wheelClicks = sasl.getCSWheelClicks()
```

Gets the number of performed wheel clicks. Negative number means that mouse wheel was rotated in down direction, and positive number means that mouse wheel was rotated in up direction.

3.13.17 getCSMouseXPos

```
number xCoordinate = sasl.getCSMouseXPos()
```

Gets current abscissa coordinate of mouse pointer in simulator window.

3.13.18 getCSMouseYPos

```
number xCoordinate = sasl.getCSMouseYPos()
```

Gets current ordinate coordinate of mouse pointer in simulator window.

3.13.19 getCSDragDirection

```
number direction = sasl.getCSDragDirection()
```

Gets dragging direction. Drag event generated when the mouse button was clicked and holds with changing cursor position. **direction** is value between 0 and 360.

3.13.20 getCSDragValue

```
number value = sasl.getCSDragValue()
```

Gets dragging value. Drag event generated when the mouse button was clicked and holds with changing cursor position. **value** is a distance between the point where mouse button was clicked and current cursor position.

3.13.21 getCSCursorOnInterface

```
number state = sasl.getCSCursorOnInterface()
```

Determines if cursor is currently on SASL window context. If **state** is equal to 0, then cursor is located on some other window. If **state** is equal to 1, the cursor is on SASL window context. Basically, you need to react on some mouse input events only if **state** is equal to 1. In other case, such events must be handled by other plugins or by simulator

itself.

mouseEvents.lua, check currect cursor context

```
sasl.setAuxiliaryClickSystem(true)
...
if (sasl.getCSCursorOnInterface() == 1 and sasl.getCSMouseDown(MB_MIDDLE))
    then
    reactSomehow()
end
```

3.13.22 getCSMouseIsOnPanel

```
boolean onPanel = sasl.getCSMouseIsOnPanel()
```

Determines if mouse cursor is currently on 3D panel. Works only in case if interactivity option is enabled for project.

3.13.23 getCSPanelMousePos

```
number x, number y = sasl.getCSPanelMousePos()
```

Returns 3D panel mouse position in texture coordinates $(0.0 - 1.0)$ in case if mouse is currently on 3D panel and returns **nil** otherwise. Works only in case if interactivity option is enabled for project.

3.14 Auxiliary Keyboard Input System

Generally, keyboard input handling in SASL must be performed with components callbacks **onKeyDown** and **onKeyUp**. Only focused components receive keyboard input in this case. But you can also perform global keyboard input handling with functions, described in this section. This can be done via registering global key callback with **registerGlobalKeyHandler** function. Registered callbacks will be called independently of components focusing.

Type: **KeyEventType**
Description: type of keyboard event, can be equal to one of pre-defined constants:

- KB_DOWN_EVENT
- KB_UP_EVENT
- KB_HOLD_EVENT

3.14.1 registerGlobalKeyHandler

```
number id = sasl.registerGlobalKeyHandler(function inCallback)
```

Registers keyboard input callback. **inCallback** is a function with 6 arguments. Returns unique numeric identifier of registered callback.

```
function inCallback(AsciiKeyCode char, VirtualKeyCode key, number shiftDown,
     number ctrlDown, number altOptDown, KeyEventType event)
```

This function will be called when the keyboard event is processed. The argument **char** contains the pressed character ASCII code if the button has corresponding code. The argument **key** contains virtual key code of the pressed button.

shiftDown, **ctrlDown**, **altOptDown** - additional arguments which defines if special buttons is pressed or not. They can be equal to 0 (up) or 1 (down).

event identifies the current event type.

Callback must return **true** if the key was processed and event shouldn't be passed to other handlers.

keyCallbackExample.lua, key input handling

```
function keyHandler(charCode, virtualKeyCode, shiftDown, ctrlDown,
    altOptDown, event)
    ...

end

sasl.registerGlobalKeyHandler(keyHandler)
```

3.14.2 unregisterGlobalKeyHandler

```
sasl.unregisterGlobalKeyHandler(number id)
```

Unregisters keyboard input callback, specified with numeric identifier **id**.

3.14.3 Hot Keys Management

Use functional provided below to manage Hot Keys for your SASL project. Basically, you just need to register new Hot Key via **registerHotKey** function and associate this Hot Key with specific action (callback). Then user will have the ability to select preferred combination for this action.

You can also redefine key combination by calling **setHotKeyCombination** function.

3.14.3.1 registerHotKey

```
number id = sasl.registerHotKey(VirtualKeyCode key, number shiftDown, number
    ctrlDown, number altOptDown, string description, function callback)
```

Registers new Hot Key ID. Four first parameters defines key combination. **description** provides description for user about this Hot Key combination. Last argument is a **callback**, which will be called when the Hot Key is triggered by user. **callback** is a function without arguments and return value. Returns numeric identifier **id** of created Hot Key.

main.lua, registering Hot Key

```
function someAction()
    print("Hello!")
end

local test = sasl.registerHotKey(SASL_VK_A, 0, 1, 1, "Testing", someAction)
```

3.14.3.2 unregisterHotKey

```
sasl.unregisterHotKey(number id)
```

Unregisters Hot Key, specified by numeric identifier **id**. You don't necessarily need to manually unregister your Hot Keys when your SASL project is about to be unloaded. This will be performed automatically.

3.14.3.3 setHotKeyCombination

```
number id = sasl.setHotKeyCombination(number id, VirtualKeyCode key, number
    shiftDown, number ctrlDown, number altOptDown)
```

Sets key combination for the Hot Key. First parameter is a Hot Key numeric identifier, and four other parameters defines key combination. You can obtain **id** by calling **registerHotKey** function.

main.lua, setting Hot Key combination

```
myHotKey = sasl.registerHotKey(...)
...
sasl.setHotKeyCombination(myHotKey, SASL_VK_ENTER, 1, 1, 0)
```

3.15 Utilities

3.15.1 Operating system

3.15.1.1 getOS

```
string osName = sasl.getOS()
```

Returns operating system identifier. The **osName** values that may be returned: "Windows", "Linux", "Mac".

3.15.1.2 getXPVersion

```
number version = sasl.getXPVersion()
```

Returns numeric identifier of current X-Plane version.

3.15.1.3 listFiles

```
table contents = sasl.listFiles(string path)
```

Returns specific array-like table containing data about directories and files, which located in the specified **path**. Each table entry is a table with two fields: field **name** contains name of file or directory. Field **type** contains string "directory" or "file" depending on the entry type. In case of errors during querying directory contents - returns empty **table**.

main.lua, print information about main X-Plane directory contents

```
local contents = sasl.listFiles(sasl.getXPlanePath())
if #contents > 0 then
    for i = 1, #contents do
        local currentName = contents[i].name
        print(currentName, contents[i].type)
    end
end
```

3.15.1.4 setClipboardText

```
sasl.setClipboardText(string text)
```

Sets text, specified in **text** into OS clipboard.
Warning: this function might not be available for **Linux**.

3.15.1.5 getClipboardText

```
string text = sasl.getClipboardText()
```

Gets current text from OS clipboard.
Warning: this function might not be available for **Linux**.

3.15.1.6 getEnvVariable

```
boolean result, string value = sasl.getEnvVariable(string env)
```

Returns the **value** of environment variable **env** for current process. **result** will be **true** if variable exists and **false** otherwise.

3.15.1.7 setEnvVariable

```
boolean result = sasl.setEnvVariable(string env, string value)
```

Sets the **value** of environment variable **env** for current process. Variable will be created, if it doesn't exist. **result** will be **true** if variable is succesfully set/created and **false** otherwise.

3.15.1.8 unsetEnvVariable

```
boolean result = sasl.unsetEnvVariable(string env)
```

Perform unsetting of environment variable **env** for current process. **result** will be **true** if variable is succesfully unset and **false** otherwise.

3.15.2 Mathematics

3.15.2.1 createLinearInterpolator

```
number createLinearInterpolator(table x, table f)
number createLinearInterpolator(table x, table y, table(matrix) f)
number createLinearInterpolator(table x, table y, table z, table(matrix) f)
```

Creates a stepwise linear interpolator object from given arguments **x**, **y**, **z** and function values **f**.

Creation example of 2d interpolator

```
number handle2d = createLinearInterpolator( { x1, x2 }, { y1, y2 }, { {
    f_x1_y1, f_x1_y2},{f_x2_y1, f_x2_y2} })
```

Returns interpolator object numeric handle or **nil** in case of errors.

3.15.2.2 deleteLinearInterpolator

```
deleteLinearInterpolator(number handle)
```

Deletes the interpolator object by given handle.

Deletion example of interpolator

```
deleteLinearInterpolator(handle)
```

3.15.2.3 interpolateLinear

```
number interpolateLinear(number handle, number x)
number interpolateLinear(number handle, table args)
number interpolateLinear(number handle, boolean extrapolate, number x)
number interpolateLinear(number handle, boolean extrapolate, table args)
```

Interpolates given point using interpolation object specified by handle. Returns interpolated value in given point. If **extrapolate = false**, will not extrapolate function. **extrapolate = true** by default.

Interpolation

```
number result1d = interpolateLinear(handle1d, x)
number result1d = interpolateLinear(handle1d, { x })
number result2d = interpolateLinear(handle2d, { x, y })
number result3d = interpolateLinear(handle3d, { x, y, z })
```

3.15.2.4 newInterpolator [DEPRECATED]

```
number handle = newInterpolator(table g1, table g2, ... , table gn, table(
    matrix) T)
```

Creates a stepwise linear interpolator object from given grids **g1**, **g2**, ... **gn**, which are arrays of variable length, and result N-dimensional matrix **T**. This result matrix is used to interpolate points in N-dimensional space, represented by the grids. Returns interpolator object numeric handle or **nil** in case of errors.

interpolation.lua, creating simple 1-dimensional interpolator

```
-- X values = {5, 20} and Y(X) values = {0.025, 0.1}
testInterp = newInterpolator({5,20},{{0.025,0.1}})
```

interpolation.lua, creating simple 2-dimensional interpolator

```
-- X and Y grid
local xGrid = { 140, 200, 260 }
local yGrid = { 5, 15, 20, 25 }

-- Z(X, Y) table
local ZFunc = { { 3.6, 1.3, 1.2, 1.7}, { 2.8, 1.2, 1.4, 1.7}, { 2.0, 1.0, 1
    .2, 1.6} }

testInterp = newInterpolator(xGrid, yGrid, { ZFunc })
```

3.15.2.5 deleteInterpolator [DEPRECATED]

```
deleteInterpolator(number handle)
```

Deletes interpolator object, represented by numeric **handle** identifier. Note that you don't necessarily need to manually delete your interpolators when your SASL project is about to be unloaded. This will be performed automatically.

interpolation.lua, deleting interpolator object

```
testInterp = newInterpolator({5,20},{{0.025,0.1}})
...
deleteInterpolator(testInterp)
```

3.15.2.6 interpolate (1-dimensional) [DEPRECATED]

```
number y = interpolate(number x, number handle)
```

Interpolates given value **x** and using given interpolator object with numeric identifier **handle**. Returns result value. Works only for one-dimensional interpolators.

interpolation.lua, interpolation example

```
testInterp = newInterpolator({5,20},{{0.025,0.1}})

-- Prints result - 0.05
print(interpolate(10, testInterp)
```

3.15.2.7 interpolate (general) [DEPRECATED]

```
number result = interpolate(table x, number handle, boolean isClosed)
```

Interpolates given N-dimensional point **x** represented by table, and a given interpolator object with numeric identifier **handle**. If **isClosed** is **true**, interpolation will be cut at the edges. If **isClosed** is **false**, the result value will be extrapolated. This parameter pass can be omitted. By default, interpolation is not closed. Returns result value.

interpolation.lua, 2-dimensional interpolation

```
-- X and Y grid
local xGrid = { 140, 200, 260 }
local yGrid = { 5, 15, 20, 25 }

-- Z(X, Y) table
local ZFunc = { { 3.6, 1.3, 1.2, 1.7}, { 2.8, 1.2, 1.4, 1.7}, { 2.0, 1.0, 1
    .2, 1.6} }

testInterp = newInterpolator(xGrid, yGrid, { ZFunc })
...
result = interpolate({175.5, 17.7}, testInterp, true)
```

3.15.2.8 selfInterpolator [DEPRECATED]

```
InterpolatorObject handle = selfInterpolator(table g1, table g2, ... , table
    gn, table(matrix) T)
```

Acts like **newInterpolator**, but returns Lua table with **interpolate** field (function). **interpolate** function in returned table acts like global **interpolate** function, but doesn't take **handle** argument.

interpolation.lua, self-interpolator example

```
testInterp = selfInterpolator({5,20},{{0.025,0.1}})
result = testInterp.interpolate(10)
```

3.15.2.9 isInRect

```
boolean result = isInRect(table rect, number x, number y)
```

Determines if the point, specified by **x**, **y** coordinates lies inside **rect** rectangle. **rect** is a table: { **x, y, width, height** }.

3.15.2.10 newStereographicProjection

```
StereographicProjection p = newStereographicProjection(number lat, number
    lon, number nmRange, number cartesianRange)
```

Creates new stereographic projection object. **lat** and **lon** specifies geographic projection center. **nmRange** and **cartesianRange** specifies the scale to be applied for projection values. Returns **StereographicProjection** object with methods to use the projection.

```
proj = newStereographicProjection(34.456, 145.2987, 80, 600)
```

3.15.2.11 StereographicProjection object

3.15.2.11.1 setProjectionCenter

```
StereographicProjection:setProjectionCenter(number lat, number lon)
```

Sets geographic projection center for stereographic projection object.

```
proj = newStereographicProjection(...)
proj:setProjectionCenter(35.6098, 154.2902)
```

3.15.2.11.2 setScale

```
StereographicProjection:setScale(number nmRange, number cartesianRange)
```

Sets scale values for stereographic projection object.

```
proj = newStereographicProjection(...)
proj:setProjectionScale(40, 600)
```

3.15.2.11.3 LLtoXY

```
x, y = StereographicProjection:LLtoXY(number lat, number lon)
```

Converts geographic position to position in Cartesian space, using stereographic projection parameters.

```
proj = newStereographicProjection(...)
local x, y = proj:LLtoXY(45.2356, 123.4589)
```

3.15.2.11.4 XYtoLL

```
x, y = StereographicProjection:XYtoLL(number x, number Y)
```

Converts position in Cartesian space to geographic position, using stereographic projection parameters.

```
proj = newStereographicProjection(...)
local lat, lon = proj:XYtoLL(50.0, 0.0)
```

3.15.3 Files And Scripts

3.15.3.1 isFileExists

```
boolean result = isFileExists(string pathToFile)
```

Returns **true** if the file, specified by full path **pathToFile** exists and returns **false** otherwise.

3.15.3.2 extractFileName

```
string path = extractFileName(string pathToFile)
```

Removes extension from specified full **pathToFile** and returns the result.

3.15.3.3 openFile

```
function chunk = openFile(string fileName)
```

Loads chunk of Lua code in Lua function, specified by **fileName** and returns it as a function that may be executed. File will be searched like components, according to current list of search paths. Use **addSearchPath** function to add new search paths. Returns **nil** in case of failure.

3.15.3.4 findResourceFile

```
string fullPath = findResourceFile(string fileName)
```

Searches the file, specified by name **fileName**. File will be searched according to current list of search resources paths. Use **addSearchResourcesPath** function to add new search resources path. Returns full path to file or **nil**, if the specified file is not found.

3.15.3.5 include

```
include(string fileName)
```

Executes Lua script in context of current component. File will be searched like components, according to current list of search paths. Use **addSearchPath** function to add new search paths. Use **include** function to make your project structured.

main.lua, including scripts

```
include("basic.lua")
include("variables.lua")
...
function update()
    ...
end
...
```

3.15.3.6 request

```
local module = request(string fileName, boolean forceInstance)
```

Loads and executes Lua module in a manner similar to standard Lua **require** function, but with a few differences. First of all, **request** will only load Lua modules (not C), and will use SASL project search paths for lookup instead of standard virtual paths model for **require**. Module will be searched like components, according to current list of search paths. Use **addSearchPath** function to add new search paths. Use **request** function to load specific modules and to make your project structured. Optional **forceInstance** parameter can be used to specify that you want to create a separate instance of loaded module instead of using the cache if the module was already loaded, default value is **false**.

Warning: don't use **request** function to load 3rd-party modules from **3rd-modules** folder or to load standard Lua/LuaJIT modules. Use **require** function for that.

main.lua, requesting scripts

```
local myModule = request("warningSystem.lua")
myModule.addWarning(...)
myModule.run()
```

3.15.3.7 readConfig

```
table t = sasl.readConfig(string pathToFile, string format)
```

Reads file specified by **pathToFile** using data **format** specification. Returns corresponding Lua table or **nil** in case of fail. Supports **.ini**, **.xml**, **.info**, **.json** formats. ini or

INI, xml or XML, etc. - both options are correct.

configReadTest.lua, config read test

```
t = sasl.readConfig(INIpath, "ini") -- reading from ini
t = sasl.readConfig(XMLpath, "xml") -- reading from xml
t = sasl.readConfig(INFOpath, "info") -- reading from info
t = sasl.readConfig(JSONpath, "json") -- reading from json
```

3.15.3.8 writeConfig

```
boolean result = sasl.writeConfig(string pathToFile, string format, table t)
```

Converts table **t** to the data in specified **format** and writes data to file **pathToFile**. Returns **true** if data was successfully written and **false** otherwise. Supports **.ini**, **.xml**, **.info**, **.json** extensions. ini or INI, xml or XML, etc. - both options are correct.

configWriteTest.lua, write config test

```
t = {
    foo = "bar",
    widget = {
        name = "simpleName"
    }
}
sasl.writeConfig(INFOpath, "info", t)
sasl.writeConfig(INIpath, "ini", t)
sasl.writeConfig(JSONpath, "JSON", t)
sasl.writeConfig(XMLpath, "XML", t)
```

3.15.4 Miscellaneous

3.15.4.1 toboolean

```
any result = toboolean(any value)
```

Converts **value** to **false** if **value** equals to 0.

3.16 Logging

Type: **LogLevelID**
Description: identifier of logging level, can be equal to one of pre-defined constants:

- LOG_DEFAULT - all log messages will be shown.
- LOG_TRACE - all log messages will be shown.
- LOG_DEBUG - all log messages will be shown.
- LOG_INFO - all log messages will be shown, except "debug" level.
- LOG_WARN - only "warn"-level and "error"-level messages will be shown.
- LOG_ERROR - only "error"-level messages will be shown.

3.16.1 logInfo

```
sasl.logInfo(...)
```

Writes data into simulator log and SASL log with level "info".

3.16.2 print

```
sasl.print(...)
```

Writes data into simulator log and SASL log with level "info".

3.16.3 logWarning

```
sasl.logWarning(...)
```

Writes data into simulator log and SASL log with level "warning".

3.16.4 logError

```
sasl.logError(...)
```

Writes data into simulator log and SASL log with level "error".

3.16.5 logDebug

```
sasl.logDebug(...)
```

Writes data into simulator log and SASL log with level "debug".

3.16.6 logTrace

```
sasl.logTrace(...)
```

Same as **logDebug**.

3.16.7 log

```
sasl.log(number level, boolean isFatal, string header, string message)
sasl.log(number level, boolean isFatal, string header, string message,
    number stackTraceDepth)
sasl.log(number level, boolean isFatal, string header, string message,
    number stackTraceDepth, Color color)
```

Outputs **message** with **header** into log files using specified parameters. This function provides a way to specify every parameter, which may be used to define log entry. **level** is the log entry level (LOG_INFO, LOG_WARN etc) and **isFatal** parameter may be used to force Lua VM stop after message output. Optionally **stackTraceDepth** may be specified for current log action (0 is default value). **color** will be used in SASL Developer Widget to display corresponding log entry. Every specified parameter will override global logger configuration (log level, default colors, stack trace depth etc).

component.lua, using customized log output

```
function update()
    ...
    if someCondition then
        sasl.log(LOG_INFO, false, "TEST", "Message", 0, { 0.0, 1.0, 0.0, 1.0
            })
    end
end
```

3.16.8 getLogLevel

```
LogLevelID id = sasl.getLogLevel()
```

Returns current logging level. Default value is LOG_DEFAULT, so every log message will be shown.

3.16.9 setLogLevel

```
sasl.setLogLevel(LogLevelID id)
```

Sets current log verbosity level.

main.lua, setup logging for release build

```
sasl.setLogLevel(LOG_INFO)
...
-- Will be dumped
sasl.logInfo("Test", "+", 123)
sasl.logInfo("Info")
sasl.logWarning("Warning")
sasl.logError("Error")
...
-- Will be omitted
sasl.logDebug("Debug!")
```

3.17 Basic Navigation

SASL provides access to simulator navigation API. Simulator supports a number of different navigation points.

Type: **NavAidType**
Description: identifier of navigation point type, can be equal to one of pre-defined constants:

- NAV_UNKNOWN - unknown navigation point type.
- NAV_AIRPORT - airfield or helipad.
- NAV_NDB - non-directional beacon.
- NAV_VOR - VOR site.
- NAV_ILS - Instrument Landing System.
- NAV_LOCALIZER - localizer part of ILS.
- NAV_GLIDESLOPE - glide slope part of ILS.
- NAV_OUTERMARKER - outer marker.
- NAV_MIDDLEMARKER - middle marker.
- NAV_INNERMARKER - inner marker.
- NAV_FIX - intersection.
- NAV_DME - distance measuring equipment.

Type: **NavAidID**
Description: numeric identifier of navigation point, can be equal to special pre-defined constant:

- NAV_NOT_FOUND - navigation point was not found. Returned by all functions when nothing to iterate.

3.17.1 Navigational Aids

3.17.1.1 getFirstNavAid

```
NavAidID id = sasl.getFirstNavAid()
```

Returns identifier of first entry in the navigation database. Use **getNextNavAid** function with this identifier to iterate all navigation points.

3.17.1.2 getNextNavAid

```
NavAidID nextID = sasl.getNextNavAid(NavAidID id)
```

Returns identifier of the navigation point which next to the point with **id** identifier. Returns NAV_NOT_FOUND if no entry left in database.

3.17.1.3 findFirstNavAidOfType

```
NavAidID id = sasl.findFirstNavAidOfType(NavAidType type)
```

Returns identifier of first navigation point of specified **type** in database. Returns NAV_NOT_FOUND if there is no navigation points of that type in database.

3.17.1.4 findLastNavAidOfType

```
NavAidID id = sasl.findLastNavAidOfType(NavAidType type)
```

Returns identifier of last navigation point of specified **type** in database. Returns NAV_NOT_FOUND if there is no navigation points of that type in database.

3.17.1.5 findNavAid

```
NavAidID id = sasl.findNavAid(string fragmentName, string fragmentID, number
    latitude, number longitude, number frequency, NavAidType type)
```

Search in database for navigation points. Argument **type** must be a sum of navigation point types for lookup. Other arguments may be equal to **nil** if not needed. If **latitude** and **longitude** is not **nil**, function returns identifier of the nearest navigation point of specified **type**, otherwise it returns last found navigation point. If **frequency** is not equal to **nil**, then any navigation points considered must match this frequency. Note that this will

screen out radio beacons that do not have frequency data published (like inner markers), but not fixes and airports. If **fragmentName** is not equal to **nil**, only navigation points which contain the **fragmentName** in their name will be returned. If **fragmentID** is not equal to **nil**, only navigation points which contain the **fragmentID** in their IDs will be returned.

3.17.1.6 getNavAidInfo

```
NavAidType type, number latitude, number longitude, number height, number
    frequency, number heading, string id, string name, boolean
    isInsideLoadedDSFs = sasl.getNavAidInfo(NavAidID id)
```

Returns all available information about navigation point, represented by identifier **id**. Last boolean return value **isInsideLoadedDSFs** is **true** if this navigation point is lies inside the local area of currently loaded DSF's, and **false** otherwise. All frequencies except NDB are multiplied by 100.

navigation.lua, get nearest airport information

```
NavAidID testID = sasl.findNavAid(nil, nil, acfLatitude, acfLongitude, nil,
    NAV_AIRPORT)
type, arptLat, arptLon, height, freg, heading, id, name, inCurDSF =
    sasl.getNavAidInfo(testID)
```

3.17.2 FMS

3.17.2.1 countFMSEntries

```
number count = sasl.countFMSEntries()
```

Returns number of entries in FMS.

3.17.2.2 getDisplayedFMSEntry

```
number index = sasl.getDisplayedFMSEntry()
```

Returns index of entry, displayed on FMS.

3.17.2.3 getDestinationFMSEntry

```
number index = sasl.getDestinationFMSEntry()
```

Returns index of entry aircraft flying to.

3.17.2.4 setDisplayedFMSEntry

```
sasl.setDisplayedFMSEntry(number index)
```

Sets displayed FMS entry with specified **index**.

3.17.2.5 setDestinationFMSEntry

```
sasl.setDestinationFMSEntry(number index)
```

Changes which entry the FMS is flying the aircraft toward. This entry is specified by **index**.

3.17.2.6 getFMSEntryInfo

```
NavAidType type, string name, NavAidID id, number altitude, number latitude,
    number longitude = sasl.getFMSEntryInfo(number index)
```

Returns information about FMS entry. For latitude/longitude entry **id** equals to NAV_NOT_FOUND.

3.17.2.7 setFMSEntryInfo

```
sasl.setFMSEntryInfo(number index, NavAidID id, number altitude)
```

Changes entry in FMS at specified **index** to navigation point which corresponds to **id** argument. This routine can be used only for airports, fixes, VOR's and NDB's.

3.17.2.8 setFMSEntryLatLon

```
sasl.setFMSEntryLatLon(number index, number latitude, number longitude,
    number altitude)
```

Changes the entry in the FMS to a latitude/longitude entry with the given coordinates.

3.17.2.9 clearFMSEntry

```
sasl.clearFMSEntry(number index)
```

Clears the given entry, specified by **index**, potentially shortening the flight plan.

3.17.3 GPS

3.17.3.1 getGPSDestinationType

```
NavAidType type = sasl.getGPSDestinationType()
```

Returns the type of currently selected GPS destination, one of fix, airport, VOR or NDB.

3.17.3.2 getGPSDestination

```
NavAidID id = sasl.getGPSDestination()
```

Returns identifier of the navigation point, which is current GPS destination.

3.18 Scenery

3.18.1 Resources Management

3.18.1.1 loadObject

```
number id = sasl.loadObject(string fileName)
```

Loads simulator object from file, specified by **fileName**. It looks for files using the current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new path for searching. Function returns numeric identifier of loaded object or **nil** in case of errors.

objects.lua, loading objects

```
carTestObj = sasl.loadObject("misc_objects/car.obj");
busTestObj = sasl.loadObject("misc_objects/bus.obj");
```

3.18.1.2 loadObjectAsync (XP11)

```
number id = sasl.loadObjectAsync(string fileName, function callback)
```

Asynchronously loads simulator object from file, specified by **fileName**. It looks for files using the current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new path for searching. **callback** function will be called when object will be actually loaded and object **id** will be ready for use. **callback** is a function with one argument, loaded object identifier will be passed to callback when loading is done. You also may get an error during asynchronous load, function won't be called in this case. Function returns numeric identifier of the object or **nil** in case of errors.

Important: Do not use result object **id** for object drawing or instancing until you got confirmation callback after loading is done.

objects.lua, loading objecs asynchronously

```
function carCallback(id)
    print("Loaded!")
end

carTestObj = sasl.loadObjectAsync("misc_objects/car.obj", carCallback);
```

3.18.1.3 unloadObject

```
sasl.unloadObject(number id)
```

Unloads object to decrease memory usage. Object specified by numeric identifier **id**. Note that you do not necessarily need to unload your objects when SASL project is about to be unloaded. This will be performed automatically. Use this routine only to free some memory on the fly.

objects.lua, loading objects

```
carTestObj = sasl.loadObject("misc_objects/car.obj");
...
sasl.unloadObject(carTestObj)
```

3.18.2 Scenery Functions

Type: **TerrainProbeResult**
Description: identifier of terrain probing result, can be equal to one of pre-defined constants:

- PROBE_HIT_TERRAIN - terrain found at specified location.
- PROBE_ERROR - internal error.
- PROBE_MISSED - terrain not found at specified location.

3.18.2.1 saveXpSituation

```
sasl.saveXpSituation(string path)
```

Saves XP situation data file using provided **path**, relative to root X-Plane folder.

3.18.2.2 loadXpSituation

```
sasl.loadXpSituation(string path)
```

Loads XP situation data file from provided **path**, relative to root X-Plane folder.

3.18.2.3 placeUserAtAirport

```
sasl.placeUserAtAirport(string id)
```

Changes the user aircraft location and places it at the airport specified by **id**.

3.18.2.4 placeUserAtLocation

```
sasl.placeUserAtLocation(number lat, number lon, number elevation, number
    heading, number speed)
```

Changes the user aircraft location and places it at the location specified by provided latitude, longitude and elevation. Additionally, **heading** (true heading, in degrees) and **speed** (in MPS) should be specified.

3.18.2.5 drawObject

```
sasl.drawObject(number id, number x, number y, number z, number pitch,
    number heading, number roll, number lighting, number earthRelative)
```

Draws simulator object, specified by numeric identifier **id**. To obtain object identifier, use **loadObject** routine. **x**, **y** and **z** are local OpenGL object coordinates. **pitch**, **heading** and **roll** are object orientation in degrees. If **lighting** is equal to 1, the night version of object will be drawn with night-only lights lit up. If **lighting** is equal to 0, the daytime version of object will be drawn.

earthRelative parameter controls the coordinate system. If this is 1, the rotations you specify are applied to the object after its coordinate system is transformed from local to earth-relative coordinates – that is, an object with no rotations will point toward true north and the Y axis will be up against gravity. If this is 0, the object is drawn with your rotations from local coordinates – that is, an object with no rotations is drawn pointing down the $-Z$ axis and the Y axis of the object matches the local coordinate Y axis.

Important: It is recommended to use Instancing interface for objects managing and drawing in XP11 (you can find corresponding subsection below).

3.18.2.6 reloadScenery

```
sasl.reloadScenery()
```

Reloads simulator scenery files.

3.18.2.7 worldToLocal

```
number x, number y, number z = sasl.worldToLocal(number latitude, number
    longitude, number altitude)
```

Converts simulator world coordinates to local OpenGL coordinates. Returns local coordinates as 3 values: **x**, **y** and **z**.

3.18.2.8 localToWorld

```
number latitude, number longitude, number altitude = sasl.localToWorld(
    number x, number y, number z)
```

Converts simulator local OpenGL coordinates to world coordinates. Returns world coordinates as 3 values: **latitude**, **longitude** and **altitude**.

3.18.2.9 modelToLocal

```
number x, number y, number z = sasl.modelToLocal(number u, number v, number
    w)
```

Converts aircraft model OpenGL coordinates (with origin in aircraft center and aircraft orientation) to local OpenGL coordinates. Returns local coordinates as 3 values: **x**, **y** and **z**.

3.18.2.10 localToModel

```
number u, number v, number w = sasl.localToModel(number x, number y, number
    z)
```

Converts local OpenGL coordinates to the aircraft OpenGL model coordinates (with origin in aircraft center and aircraft orientation). Returns model coordinates as 3 values: **u**, **v** and **w**.

3.18.2.11 probeTerrain

```
TerrainProbeResult result, number locationX, number locationY, number
    locationZ, number normalX, number normalY, number normalZ, number
    velocityX, number velocityY, number velocityZ, number isWet =
    sasl.probeTerrain(number x, number y, number z)
```

Locates physical scenery terrain mesh. Pass location of interest in local OpenGL coordinates as **x**, **y** and **z**. Returns probe result and terrain parameters.

If **result** is equal to PROBE_HIT_TERRAIN then additional return values is filled. **locationX**, **locationY**, **locationZ** is location of terrain point hit in local coordinates. **normalX**, **normalY**, **normalZ** is normal vector of terrain found. **velocityX**, **velocityY**, **velocityZ** is velocity vector of terrain found. **isWet** equals to 1 if water is found, and equals to 0 otherwise.

3.18.3 Magnetic Variation (XP11)

3.18.3.1 getMagneticVariation

```
number magVar = sasl.getMagneticVariation(number latitude, number longitude)
```

Returns magnetic variation (declination) corresponding to the geographic location, identified by **latitude** and **longitude**.

3.18.3.2 degMagneticToDegTrue

```
number headingTrue = sasl.degMagneticToDegTrue(number heading)
```

Converts a **heading** in degrees relative to magnetic north at the user's current location into a value relative to true north.

3.18.3.3 degTrueToDegMagnetic

```
number headingMagnetic = sasl.degTrueToDegMagnetic(number heading)
```

Converts a **heading** in degrees relative to true north into a value relative to magnetic north at the user's current location.

3.18.4 Instancing (XP11)

Instancing interface is an alternative way to manage and draw simulator objects (**OBJ** files). This functional is should replace previous legacy approach to draw objects via **drawObjects** components callback and using **drawObject** routine. The main difference is that with instancing interface you don't need to draw your object every frame in draw callback - in this approach you may create instance of your object, configure it initially and set the object position and parameters (dataref driven) when it's needed.

Using instancing interface instead of an old way will increase performance related to managing simulator objects and will make your code work even after simulator migration to the next-gen graphics API (Vulkan/Metal). It is recommended to start using Instancing as soon as possible.

3.18.4.1 createInstance

```
number instanceId = sasl.createInstance(number objectId, table datarefs)
```

Creates a new instance of the object based on the **objectId** object identifier. **datarefs** is a table of strings, where every string represents a simulator float dataref identifier. Use **loadObject** or **loadObjectAsync** function to obtain **objectId** from simulator OBJ file.

instancingEx.lua, creation of instance

```
carTestObj = sasl.loadObject("misc_objects/car.obj");
carTestDatarefs = {
    "myProject/car/first",
    "myProject/car/second",
    "myProject/car/third"
}

testInstance = sasl.createInstance(carTestObj, carTestDatarefs)
```

3.18.4.2 destroyInstance

```
sasl.destroyInstance(number instanceId)
```

Destroys an instance of the object with **instanceId** identifier. Note that you don't necessarily need to destroy your objects instances when your project is about to be unloaded - this will be performed automatically. Destroy your instances only if you need to delete them on the fly, free some memory and keep project working.

instancingEx.lua, instance destruction

```
sasl.destroyInstance(testInstance)
```

3.18.4.3 setInstancePosition

```
sasl.setInstancePosition(number instanceId, number x, number y, number z,
    number pitch, number heading, number roll, table data)
```

Sets position and dataref values for the object instance, specified by numeric identifier **instanceId**. To obtain instance identifier, use **createInstance** function. **x**, **y** and **z** are local OpenGL object coordinates. **pitch**, **heading** and **roll** are object orientation in degrees. **data** is a table containing float values for datarefs, that drives the object. The values order in the **data** table and the table size should correspond to the order of float datarefs string table used to create the instance.

Use this function to update object appearance in simulator, when it's needed.

3.19 Graphics

Type: **ShaderTypeID**
Description: identifier of shader type, can be equal to one of pre-defined constants:

- SHADER_TYPE_VERTEX - vertex shader type.
- SHADER_TYPE_FRAGMENT - fragment shader type.
- SHADER_TYPE_GEOMETRY - geometry shader type.

Type: **FontHinterPreferenceID**
Description: identifier of font hinter, can be equal to one of pre-defined constants:

- FONT_HINTER_AUTO - prefer automatic font hinting algorithm implemented in font loaded library (default option).
- FONT_HINTER_NATIVE - prefer hinting algorithm specified in font file.
- FONT_HINTER_DISABLED - disable font hinting.

3.19.1 Resources Management

3.19.1.1 loadImage

```
number id = sasl.gl.loadImage(string fileName)
number id = sasl.gl.loadImage(string fileName, number width, number height)
number id = sasl.gl.loadImage(string fileName, number x, number y, number
    width, number height)
```

Loads image into memory. Returns texture numeric handle **id** or **nil** if image is not found. Image will be searched according to current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

Version with **width** and **height** specified loads only central part of image.

Version with **x**, **y**, **width** and **height** specified loads only part of image defined with these parameters. **x** and **y** defines left bottom corner of image part.

Loading different parts of the same image file won't create additional texture targets and won't consume more VRAM, it will reuse original cached texture.

images.lua, loading images

```
-- Load whole image
testImage1 = sasl.gl.loadImage("images/EGTSign.png")

-- Load part of the image
testImage2 = sasl.gl.loadImage("images/EGTSign.png", 0, 0, 450, 300)
```

3.19.1.2 loadVectorImage

```
number id = sasl.gl.loadVectorImage(string fileName, number rasterWidth,
    number rasterHeight)
number id = sasl.gl.loadVectorImage(string fileName, number rasterWidth,
    number rasterHeight, number width, number height)
number id = sasl.gl.loadVectorImage(string fileName, number rasterWidth,
    number rasterHeight, number x, number y, number width, number height)
```

Loads SVG image into memory and render it to texture of specified size (**rasterWidth** x **rasterHeight**). Returns numeric texture handle **id** or **nil** if image is not found. Image will be searched according to current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

Version with **width** and **height** specified loads only central part of image.

Version with **x**, **y**, **width** and **height** specified loads only part of image defined with these parameters. **x** and **y** defines left bottom corner of image part.

Loading different parts of the same image file with same **rasterWidth** and **rasterHeight** won't create additional texture targets and won't consume more VRAM, it will reuse original cached texture.

Following SVG 1.1 features supported:

1. Path data - parsing, simplification
2. Transform list - parsing, simplification
3. Color, including ICC color - parsing
4. Lengths - highly configurable handling of SVG lengths, full support of all SVG length units
5. Basic shapes - optional conversion to path
6. Style attribute - parsing, taking in account differences in parsing presentation attributes and style properties
7. Automatic marker positions calculation
8. Viewport and viewbox handling

vectorimages.lua, loading SVG images

```
-- Load whole SVG image
testImage1 = sasl.gl.loadVectorImage("images/background.svg", 512, 512)

-- Load part of the SVG image
testImage2 = sasl.gl.loadVectorImage("images/background.svg", 512, 512, 0,
    0, 450, 300)
```

3.19.1.3 loadImageFromMemory (loadTextureFromMemory)

```
number id = sasl.gl.loadImageFromMemory(string data)
number id = sasl.gl.loadImageFromMemory(string data, number width, number
    height)
number id = sasl.gl.loadImageFromMemory(string data, number x, number y,
    number width, number height)
```

Loads image from **data** string. You can safely free data after calling this function. Returns texture numeric handle **id** or **nil** if texture was not loaded.

Version with **width** and **height** specified loads only central part of image.

Version with **x**, **y**, **width** and **height** specified loads only part of image defined with these parameters. **x** and **y** defines left bottom corner of image part.

3.19.1.4 unloadImage (unloadTexture)

```
sasl.gl.unloadImage(number id)
```

Unloads texture image from memory. Image specified by **id**. Because of cache, texture image can still remain in memory after calling this function, if this texture uses shared resource. Note that you do not have to necessarily unload your images when your SASL project is about to be unloaded. This will be performed automatically.

images.lua, unloading images (first case)

```
testImage1 = sasl.gl.loadImage("images/EGTSign.png")
...
-- Memory is unloaded
sasl.gl.unloadImage(testImage1)
```

images.lua, unloading images (second case)

```
testImage1 = sasl.gl.loadImage("images/EGTSign.png")
testImage2 = sasl.gl.loadImage("images/EGTSign.png", 0, 20, 400, 440)
...
-- Memory is not fully freed, because same texture memory is used for
    testImage2
sasl.gl.unloadImage(testImage1)
```

3.19.1.5 loadBitmapFont

```
number id = sasl.gl.loadBitmapFont(string fileName)
```

Loads font in bitmap texture format (SASL-2.x style). Returns font numeric handle **id** or **nil**, if the font texture is not found. Font will be searched according to the current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

3.19.1.6 unloadBitmapFont

```
sasl.gl.unloadBitmapFont(number id)
```

Unloads bitmap font, specified by **id**. Use this function only to unload bitmap fonts. Note that you do not necessarily need to unload you bitmap fonts when SASL project is about to be unloaded. This will be performed automatically.

3.19.1.7 loadFont

```
number id = sasl.gl.loadFont(string fileName)
number id = sasl.gl.loadFont(string fileName, number textureId)
number id = sasl.gl.loadFont(string fileName, number textureId, number x,
    number y, number width, number height)
```

Loads font in common format (TrueType fonts, OpenType fonts and etc). Returns font numeric handle **id** or **nil**, if the font is not found. Font will be searched according to the current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

By default, every loaded font uses special texture to store generated glyphs. This texture will be expanded on the fly, if needed. Separate texture is used for every font file in file system. Optional parameters may be used to customize texture storage for the specified instance of font. **textureId** parameter specifies texture storage for generated glyphs and optional last 4 parameters define texture region to use. Such configuration can be applied to optimize drawing and reduce changes of graphics state. It's a common practice to pack many graphics assets in one texture and this way any font can also use shared texture. Minimum storage size is $128 * 128$. In case of custom texture storage, it won't be expanded when there will be not enough space for a new glyph, instead there will be a corresponding notification from graphics system.

fonts.lua, standard usage

```
local myFontId = sasl.gl.loadFont("fonts/myFont.ttf")

function draw()
    sasl.gl.drawTextI(myFontId, 0, 0, "TEST", TEXT_ALIGN_LEFT, {1, 1, 1, 1})
end
```

fonts.lua, loading font with custom texture storage

```
local myImageId = sasl.gl.loadImage("images/myImage.png")
local myFontId = sasl.gl.loadFont("fonts/myFont.ttf", true, myImageId, 512,
    512, 256, 256)

function draw()
    sasl.gl.drawTextI(myFontId, 0, 0, "TEST", TEXT_ALIGN_LEFT, {1, 1, 1, 1})
end
```

3.19.1.8 loadFontHinted

```
number id = sasl.gl.loadFontHinted(string fileName, FontHinterPreferenceId
    hinter)
number id = sasl.gl.loadFontHinted(string fileName, FontHinterPreferenceId
    hinter, number textureId)
number id = sasl.gl.loadFontHinted(string fileName, FontHinterPreferenceId
    hinter, number textureId, number x, number y, number width, number
    height)
```

Works the same as **loadFont**, but allows to specify hinter preference for loaded font. Hinting algorithm may slightly affect how font glyphs are drawn, because hinting adjusts font glyphs rendering depending on pixel grid and font size in order to produce more accurate visual representation of the glyphs. By default, automatic hinting is used for every font loaded with **loadFont** function. However, in specific cases native font hinter algorithm may be preferred over automatic one.

fonts.lua, loading font using native font hinter

```
local myFontId = sasl.gl.loadFontHinted("fonts/myFont.ttf",
    FONT_HINTER_NATIVE)
```

3.19.1.9 unloadFont

```
sasl.gl.unloadFont(number id)
```

Unloads font, specified by **id**. Use this function to unload only fonts in common formats. Note that you do not necessarily need to unload you fonts when SASL project is about to be unloaded. This will be performed automatically.

3.19.1.10 loadShader

```
sasl.gl.loadShader(number id, string fileName, ShaderTypeID type)
```

Adds new shader to shader program, specified by numeric identifier **id**. Shaders will be searched according to the current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

Note that there is no corresponding unload-function, because this function is just adds shader sources to already created shader program object, specified by **id**. And thus, unloading can be performed only for whole shader program with **deleteShaderProgram** function.

3.19.2 Color

Type: **Color**
Description: represents color which can be used to draw graphics primitives and other entities. Color in SASL can be represented by following:

- Table with four values (RGBA) in range $[0.0, 1.0]$.
- Table with three values (RGB) in range $[0.0, 1.0]$. Alpha is default.
- Four consecutive arguments (RGBA) in range $[0.0, 1.0]$.
- Three consecutive arguments (RGB) in range $[0.0, 1.0]$. Alpha is default.
- nil. Default color with default alpha. Corresponds to omitted **Color** argument in SASL graphics functions.

Default color is $\{1.0, 1.0, 1.0\}$ and default alpha is 1.0 in case of non-texture primitives drawing. In case of texture drawing functions, default color is background color and default alpha is 1.0.

3.19.3 2D Graphics

3.19.3.1 Primitives

Type: **ShapeExtrusionMode**
Description: identifier of shape extrusion mode, can be equal to one of the following pre-defined values:

- SHAPE_EXTRUDE_INNER
- SHAPE_EXTRUDE_OUTER
- SHAPE_EXTRUDE_CENTER

3.19.3.1.1 drawLine

```
sasl.gl.drawLine(number x1, number y1, number x2, number y2, Color color)
```

Draws line between **x1**, **y1** point and **x2**, **y2** point with specified **color**.

testComponent.lua, draw two red lines on panel of plane

```
local clRed = {1.0, 0.0, 0.0, 1.0}

function draw()
    sasl.gl.drawLine(0, 0, 100, 100, clRed)
    sasl.gl.drawLine(100, 100, 200, 0, clRed)
end
```

3.19.3.1.2 drawWideLine

```
sasl.gl.drawWideLine(number x1, number y1, number x2, number y2, number
    thickness, Color color)
```

Draws wide line between **x1**, **y1** point and **x2**, **y2** point with specified **color** and with specified **thickness**.

3.19.3.1.3 drawPolyLine

```
sasl.gl.drawPolyLine(table points, Color color)
```

Draws poly-line between specified **points** with specified **color**. **points** is a table with size **pointsNumber** $* 2$, contains coordinates of points.

testComponent.lua, drawing simple poly-line

```
local clRed = {1.0, 0.0, 0.0, 1.0}

function draw()
    sasl.gl.drawPolyLine({0, 0, 100, 100, 100, 150, 150, 150}, clRed)
end
```

3.19.3.1.4 drawWidePolyLine

```
sasl.gl.drawWidePolyLine(table points, number thickness, Color color)
```

Acts like **drawPolyLine** function, but takes line **thickness** parameter into account.

3.19.3.1.5 drawTriangle

```
sasl.gl.drawTriangle(number x1, number y1, number x2, number y2, number x3,
    number y3, Color color)
```

Draws filled triangle by given points **x1**, **y1**, **x2**, **y2** and **x3**, **y3** with specified **color**.

3.19.3.1.6 drawRectangle

```
sasl.gl.drawRectangle(number x, number y, number width, number height, Color
     color)
```

Draws filled rectangle, specified by **x**, **y**, **width** and **height** with specified **color**.

3.19.3.1.7 drawFrame

```
sasl.gl.drawFrame(number x, number y, number width, number height, Color
    color)
```

Draws frame, specified by **x**, **y**, **width** and **height** with specified **color**.

3.19.3.1.8 setLinePattern

```
sasl.gl.setLinePattern(table pattern)
```

Sets current line pattern, which will be used with **drawLinePattern** function. **pattern** argument is a table values which defines pattern. A positive values means visible part, negative values means non-visible part.

testComponent.lua, simple dashed lines

```
function draw()
    sasl.gl.setLinePattern({5.0, -5.0})
end
```

testComponent.lua, non-standard patterns

```
function draw()
    sasl.gl.setLinePattern({5.0, -3.0, 1.0})
end
```

3.19.3.1.9 drawLinePattern

```
sasl.gl.drawLinePattern(number x1, number y1, number x2, number y2, boolean
    savePatternState, Color color)
```

Draws line between **x1**, **y1** and **x2**, **y2** points using pattern and specified **color**. Use **setLinePattern** to set current line pattern. If **savePatternState** is **true**, then current pattern state will be saved across function calling. Set this parameter to **true** for drawing custom geometry shapes with defined pattern. Use this function for drawing dashed lines, dotted lines, etc.

testComponent.lua, drawing patterned lines

```
local clGreen = {0.0, 1.0, 0.0}

function draw()
    sasl.gl.setLinePattern({5.0, -5.0})
    sasl.gl.drawLinePattern(0, 0, 100, 100, true, clGreen)
    sasl.gl.drawLinePattern(100, 100, 200, 0, true, clGreen)
end
```

3.19.3.1.10 drawPolyLinePattern

```
sasl.gl.drawPolyLinePattern(table points, Color color)
```

Acts like **drawLinePattern** function, but draws poly-line with current selected pattern. **points** is a table with size **pointsNumber** $* 2$, contains coordinates of points.

testComponent.lua, drawing patterned poly-lines

```
local clGreen = {0.0, 1.0, 0.0}

function draw()
    sasl.gl.setLinePattern({5.0, -5.0})
    sasl.gl.drawPolyLinePattern({0, 0, 100, 100, 100, 150}, clGreen)
end
```

3.19.3.1.11 drawBezierLineQ

```
sasl.gl.drawBezierLineQ(number x1, number y1, number x2, number y2, number
    x3, number y3, number parts, Color color)
```

Draws curved quadratic Bezier line, specified by anchor points **x1**, **y1** and **x3**, **y3**, and control point **x2**, **y2**. **parts** parameter defines how many flat lines will be used to draw Bezier line. The last parameter is **color**.

3.19.3.1.12 drawWideBezierLineQ

```
sasl.gl.drawWideBezierLineQ(number x1, number y1, number x2, number y2,
    number x3, number y3, number parts, float thickness, Color color)
```

Works as **drawBezierLineQ** function, but draws Bezier line with specific **thickness**.

3.19.3.1.13 drawBezierLineQAdaptive

```
sasl.gl.drawBezierLineQAdaptive(number x1, number y1, number x2, number y2,
    number x3, number y3, Color color)
```

Draws curved quadratic Bezier line, specified by anchor points **x1**, **y1** and **x3**, **y3**, and control point **x2**, **y2**. The last parameter is **color**. This function draws Bezier line with adaptive technique and should be used in general case. Use this function for better performance with smooth result.

3.19.3.1.14 drawWideBezierLineQAdaptive

```
sasl.gl.drawWideBezierLineQAdaptive(number x1, number y1, number x2, number
    y2, number x3, number y3, float thickness, Color color)
```

Works as **drawBezierLineQAdaptive** function, but draws Bezier line with specific **thickness**.

3.19.3.1.15 drawBezierLineC

```
sasl.gl.drawBezierLineC(number x1, number y1, number x2, number y2, number
    x3, number y3, number x4, number y4, number parts, Color color)
```

Draws curved cubic Bezier line, specified by anchor points **x1**, **y1** and **x4**, **y4**, and control points **x2**, **y2** and **x3**, **y3**. **parts** parameter defines how many flat lines will be used to draw Bezier line. The last parameter is **color**.

3.19.3.1.16 drawWideBezierLineC

```
sasl.gl.drawWideBezierLineC(number x1, number y1, number x2, number y2,
    number x3, number y3, number x4, number y4, number parts, float
    thickness, Color color)
```

Works as **drawBezierLineC** function, but draws Bezier line with specific **thickness**.

3.19.3.1.17 drawBezierLineCAdaptive

```
sasl.gl.drawBezierLineCAdaptive(number x1, number y1, number x2, number y2,
    number x3, number y3, number x4, number y4, Color color)
```

Draws curved cubic Bezier line, specified by anchor points **x1**, **y1** and **x4**, **y4**, and control points **x2**, **y2** and **x3**, **y3**. The last parameter is **color**. This function draws Bezier line with adaptive technique and should be used in general case. Use this function for better performance with smooth result.

3.19.3.1.18 drawWideBezierLineCAdaptive

```
sasl.gl.drawWideBezierLineCAdaptive(number x1, number y1, number x2, number
    y2, number x3, number y3, number x4, number y4, float thickness, Color
    color)
```

Works as **drawBezierLineCAdaptive** function, but draws Bezier line with specific **thickness**.

3.19.3.1.19 drawCircle

```
sasl.gl.drawCircle(number x, number y, number radius, boolean isFilled,
    Color color)
```

Draws circle with center in **x**, **y** coordinates and specified **radius**. If **isFilled** is **true**, circle will be filled. The last parameter is **color**.

3.19.3.1.20 drawArc

```
sasl.gl.drawArc(number x, number y, number radiusInner, number radiusOuter,
    number startAngle, number arcAngle, Color color)
```

Draws arc with center in **x**, **y**. **radiusInner** and **radiusOuter** defines arc form, arc will be drawn between these distances. **startAngle** is the angle, where the arc should start. 0 means full right direction and the arc will be drawn anti-clockwise. All angles must be specified in degrees. The last parameter is **color**.

3.19.3.1.21 drawArcLine

```
sasl.gl.drawArcLine(number x, number y, number radius, number startAngle,
    number arcAngle, Color color)
```

Draws arc with center in **x**, **y** and of specified **radius**. **startAngle** is the angle, where the arc should start. 0 means full right direction and the arc will be drawn anti-clockwise. All angles must be specified in degrees. The last parameter is **color**.

3.19.3.1.22 drawConvexPolygon

```
sasl.gl.drawConvexPolygon(table points, boolean isFilled, number thickness,
    Color color)
```

Draws custom convex shape defined with **points** table, contains coordinates of shape points. Size of **points** table is $pointsNumber * 2$. Parameter **isFilled** controls the shape type (filled or not). **thickness** can be used to draw polygon with wide lines in case if **isFilled** is **false**. Thickness applying will be performed (extruded) from shape centroid. The last parameter is **color**.

testComponent.lua, drawing custom convex shapes

```
local clGreen = {0.0, 1.0, 0.0}

function draw()
    sasl.gl.drawConvexPolygon({0, 0, 100, 100, 100, 150, 0, 200}, false, 5,
        clGreen)
end
```

3.19.3.1.23 drawConvexPolygonMC

```
sasl.gl.drawConvexPolygonMC(table points, table colors)
```

Draws custom convex shape defined with **points** table and **colors** table. Size of **points** table is $pointsNumber * 2$, and size of **colors** table is $pointsNumber * 4$.

testComponent.lua, drawing custom convex shapes

```
function draw()
    sasl.gl.drawConvexPolygonMC({ 110, 10, 150, 70, 200, 50, 275, 5 }, { 1,
        0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0.5, 0.8, 0.3, 1})
end
```

3.19.3.1.24 setPolygonExtrudeMode

```
sasl.gl.setPolygonExtrudeMode(ShapeExtrusionMode mode)
```

Sets shape extrusion **mode** for drawing polygons. Default value is SHAPE_EXTRUDE_OUTER.

3.19.3.1.25 setWideLineExtrudeMode

```
sasl.gl.setWideLineExtrudeMode(ShapeExtrusionMode mode)
```

Sets shape extrusion **mode** for wide lines drawing. Default value is SHAPE_EXTRUDE_CENTER.

3.19.3.1.26 setInternalLineWidth

```
sasl.gl.setInternalLineWidth(number width)
```

Wrapper around **glLineWidth** OpenGL function. Sets **width** for lines-based shapes during drawing. Default value is 1.0.

3.19.3.1.27 setInternalLineStipple

```
sasl.gl.setInternalLineStipple(boolean enabled)
sasl.gl.setInternalLineStipple(boolean enabled, number factor, number
    pattern)
```

Wrapper around **glLineStipple** OpenGL function. Enables and disables stipple effect based on provided arguments.

test.lua, drawing dashed arc

```
function draw()
    sasl.gl.setInternalLineWidth(2.0)
    sasl.gl.setInternalLineStipple(true, 8, 0xAAAA)
    sasl.gl.drawArcLine(240, 295, 90, 0.0, 90.0, 1, 0, 0, 1)
    sasl.gl.drawArcLine(240, 295, 100, 0.0, 90.0, 1, 0, 0, 1)
    sasl.gl.setInternalLineWidth(1.0)
    sasl.gl.setInternalLineStipple(false)
end
```

3.19.3.1.28 saveInternalLineState

```
sasl.gl.saveInternalLineState()
```

Saves to the settings stack current OpenGL internal line settings like **width**, **pattern** and **stipple** to be able **restoreInternalLineState()** in future.

saving and restoring settings

```
...
sasl.gl.saveInternalLineState()
...
sasl.gl.setInternalLineWidth(5)
sasl.gl.setInternalLineStipple(true, 2, 0xA0AA)
sasl.gl.drawLine(6, 550, 600, 550, 0, 1, 0, 1)
...
sasl.gl.restoreInternalLineState()
...
```

3.19.3.1.29 restoreInternalLineState

```
sasl.gl.restoreInternalLineState()
```

Restores from the settings stack OpenGL internal line settings like **width**, **pattern** and **stipple**.

saving and restoring settings

```
...
sasl.gl.saveInternalLineState()
...
sasl.gl.setInternalLineWidth(5)
sasl.gl.setInternalLineStipple(true, 2, 0xA0AA)
sasl.gl.drawLine(6, 550, 600, 550, 0, 1, 0, 1)
...
sasl.gl.restoreInternalLineState()
...
```

3.19.3.2 Textures

Type: **TextureWrappingMode**
Description: identifier of texture wrapping mode, can be equal to one of the following pre-defined values:

- TEXTURE_CLAMP - texture will be clamped. Default mode.
- TEXTURE_REPEAT - texture will be repeated.
- TEXTURE_MIRROR_CLAMP - texture will be mirrored and clamped.
- TEXTURE_MIRROR_REPEAT - texture will be mirrored and repeated.

3.19.3.2.1 drawTexture

```
sasl.gl.drawTexture(number id, number x, number y, number width, number
    height, Color color)
```

Draws texture specified by numeric texture handle **id** at position, specified by coordinates **x**, **y**. Texture will be drawn with specified **width** and **height**. The last parameter is **color**.

testComponent.lua, drawing textures

```
testImage1 = sasl.gl.loadImage("images/loadsheet_back.png", 0, 0, 400, 300)
testImage2 = sasl.gl.loadImage("images/loadsheet_back.png", 580, 10, 100,
    100)

function draw()
    sasl.gl.drawTexture(testImage1, 0, 0, 400, 300)
    sasl.gl.drawTexture(testImage2, 400, 0, 100, 100, {0.8, 0.8, 0.8, 1.0})
end
```

3.19.3.2.2 drawRotatedTexture

```
sasl.gl.drawRotatedTexture(number id, number angle, number x, number y,
    number width, number height, Color color)
```

Draws texture specified by numeric texture handle **id** at position, specified by coordinates **x**, **y**, rotated around texture center by **angle** in degrees. The last parameter is **color**.

3.19.3.2.3 drawRotatedTextureCenter

```
sasl.gl.drawRotatedTextureCenter(number id, number angle, number rx, number
    ry, number x, number y, number width, number height, Color color)
```

Draws texture specified by numeric texture handle **id** at position, specified by coordinates **x**, **y**, rotated around **rx**, **ry** point by **angle** in degrees. The last parameter is **color**.

3.19.3.2.4 drawTexturePart

```
sasl.gl.drawTexturePart(number id, number x, number y, number width, number
    height, number tx, number ty, number twidth, number theight, Color color
    )
```

Draws texture like **drawTexture** function, but only the part of specified texture will be drawn. Use **tx**, **ty**, **twidth** and **theight** arguments to specify the part of texture, which will be drawn. The last parameter is **color**.

testComponent.lua, drawing texture part (half)

```
testImage1 = sasl.gl.loadImage("images/loadsheet.png")

function draw()
    sasl.gl.drawTexturePart(testImage1, 0, 0, 400, 300, 0, 0, 200, 100, {1.0
        , 1.0, 1.0, 1.0})
end
```

3.19.3.2.5 drawRotatedTexturePart

```
sasl.gl.drawRotatedTexturePart(number id, number angle, number x, number y,
    number width, number height, number tx, number ty, number twidth, number
     theight, Color color)
```

Draws texture like **drawRotatedTexture** function, but only the part of specified texture will be drawn. Use **tx**, **ty**, **twidth** and **theight** arguments to specify the part of texture, which will be drawn. The last parameter is **color**.

3.19.3.2.6 drawRotatedTexturePartCenter

```
sasl.gl.drawRotatedTexturePartCenter(number id, number angle, number rx,
    number ry, number x, number y, number width, number height, number tx,
    number ty, number twidth, number theight, Color color)
```

Draws texture like **drawRotatedTextureCenter** function, but only the part of specified texture will be drawn. Use **tx**, **ty**, **twidth** and **theight** arguments to specify the part of texture, which will be drawn. The last parameter is **color**.

3.19.3.2.7 drawTextureCoords

```
sasl.gl.drawTextureCoords(number id, number x1, number y1, number x2, number
     y2, number x3, number y3, number x4, number y4, Color color)
```

Draws texture specified by numeric texture handle **id** using four points to specify drawn shape. Use **x1**, **y1**, **x2**, **y2**, **x3**, **y3**, **x4**, **y4** coordinates to specify this shape. The last argument is **color**.

3.19.3.2.8 drawTextureWithRotatedCoords

```
sasl.gl.drawTextureWithRotatedCoords(number id, number angle, number x,
    number y, number width, number height, number tx, number ty, number
    twidth, number theight)
```

Draws texture specified by numeric texture handle **id** at position, specified by **x**, **y**, **width**, **height**. Use **tx**, **ty**, **twidth**, **theight** arguments to specify the texture part, which texture coordinates will be rotated. The last argument is **color**.

3.19.3.2.9 getTextureSize

```
number width, number height = sasl.gl.getTextureSize(number id)
```

Returns **width** and **height** of texture, specified by numeric texture handle **id**.

3.19.3.2.10 getTextureSourceSize

```
number width, number height = sasl.gl.getTextureSourceSize(number id)
```

Same as **getTextureSize**.

3.19.3.2.11 setTextureWrapping

```
sasl.gl.setTextureWrapping(number id, TextureWrappingMode mode)
```

Sets current texture wrapping **mode**. Texture is specified by numeric identifier **id**. Wrapping modes defines how texture will be drawn in case if texture coordinates will end up out of $[0.0, 1.0]$ range.

3.19.3.2.12 importTexture

```
number id = sasl.gl.importTexture(number inSpecID)
```

Imports specific texture with identifier **inSpecID** in SASL textures system and returns numeric texture handle of imported texture - **id**. Use this function to get access to textures, owned by other plugins or by simulator itself. The following constant values can be used to access special simulator textures:

- GENERAL_INTERFACE_TEX

- AIRCRAFT_PAINT_TEX
- AIRCRAFT_LITE_MAP_TEX

testComponent.lua, importing texture

```
testInterfaceTexture = sasl.gl.importTexture(GENERAL_INTERFACE_TEX)
```

testComponent.lua, importing shared texture

```
sharedTextureDataref = globalPropertyi("some_plugin/shared_texture_id")
...
testTexture = sasl.gl.importTexture(get(sharedTextureDataref))
```

3.19.3.2.13 exportTexture

```
number openglTexId = sasl.gl.exportTexture(number inSpecID)
```

Exports specific texture with identifier **inSpecID** from SASL textures system and returns underlying graphics API texture handler - **openglTexId**. Use this function to share textures owned by SASL project with other plugins or libraries.

3.19.3.2.14 recreateTexture

```
sasl.gl.recreateTexture(number id, number width, number height, boolean
    saveContents)
```

Recreates texture specified by numeric identifier **id** with new **width** and **height**. If **saveContents** is **true**, then the function will try to copy texture contents in recreated texture.

3.19.3.2.15 setRenderTarget

```
sasl.gl.setRenderTarget(number id)
sasl.gl.setRenderTarget(number id, boolean isNeedClear)
sasl.gl.setRenderTarget(number id, boolean isNeedClear, number inAALevel)
```

Start rendering into texture, specified by numeric handle **id**. If **isNeedClear** is **true**, the texture contents will be cleared before rendering. If **isNeedClear** is **false**, texture contents will be unmodified before rendering. **isNeedClear** is an optional parameter and by default equals to **true**. Always call **restoreRenderTarget** function after you are done rendering into texture.

Additionally, anti-aliasing level can be specified for render target - **inAALevel**. Default value is 1, which means AA disabled. Accepted variant are 1, 2, 4, 8, 16, 32. Higher AA

levels may not be supported on specific machine, and in this case automatic fallback will be performed to maximum supported AA level. When this parameter is omitted, AA level specified with **createRenderTarget** may be used.

It is recommended to draw into texture targets in **update** callback for performance reasons.

3.19.3.2.16 clearRenderTarget

```
sasl.gl.clearRenderTarget(number x, number y, number width, number height)
```

Clears rectangular area of current render target (can be default render target or custom one), specified by **x**, **y**, **width**, **height**.

3.19.3.2.17 restoreRenderTarget

```
sasl.gl.restoreRenderTarget()
```

Finishes rendering to texture and continue rendering to default render target (panel of popup window).

3.19.3.2.18 createRenderTarget

```
number id = sasl.gl.createRenderTarget(number width, number height)
number id = sasl.gl.createRenderTarget(number width, number height, number
    aaLevel)
number id = sasl.gl.createRenderTarget(number width, number height, number
    aaLevel, RenderTargetAttachment attachment)
```

Generates new RGBA texture that can be used as render target and returns its numeric handle **id**. Use **width** and **height** arguments to specify render target size. Note that such generated textures will be automatically cleaned up when SASL project is about to be unloaded.

Additionally, anti-aliasing level can be specified for render target - **inAALevel**. Default value is 1, which means AA disabled. Accepted variant are 1, 2, 4, 8, 16, 32. Higher AA levels may not be supported on specific machine, and in this case automatic fallback will be performed to maximum supported AA level.

attachment parameter may be used to configure attachment buffers for created render target. Available options are RENDER_TARGET_C (only color attachment) and RENDER_TARGET_CDS (color, depth, stencil). Default value is RENDER_TARGET_CDS.

- RENDER_TARGET_C – render target will use only color buffer and won't have masking functional available.
- RENDER_TARGET_CDS – render target will use color, depth and stencil buffers (default option).

3.19.3.2.19 destroyRenderTarget

```
sasl.gl.destroyRenderTarget(number id)
```

Destroys render target specified by **id**, previously created with **createRenderTarget** function. It's not necessary to destroy your render targets when SASL project is about to be unloaded - this will be performed automatically. Use this function only for destroying render targets during project working time.

3.19.3.2.20 createTexture

```
sasl.gl.createTexture(number width, number height)
```

Creates new RGBA texture and returns its numeric handle **id**. Use **width** and **height** arguments to specify texture size. Note that created textures will be automatically cleaned up when SASL project is about to be unloaded. Use **unloadTexture**, **unloadImage** functions to destroy texture manually.

3.19.3.2.21 getTargetTextureData

```
sasl.gl.getTargetTextureData(number id, number x, number y, number width,
    number height)
```

Copies current data from render target to texture, specified by numeric identifier **id**. **x**, **y**, **width** and **height** specifies the position of rectangle, which will be copied from target to your texture. Use this function to get current contents of 2D window (in popups drawing callbacks) and panel (in panel drawing callbacks).

3.19.3.2.22 createTextureDataStorage

```
number id = sasl.gl.createTextureDataStorage(number width, number height)
```

Creates new texture data storage object and returns its numeric identifier **id**. **width** and **height** specifies size of RGBA texture, that can be stored inside this storage. Use this function when you need to interact with raw texture data (change texture contents in manual way, for example) inside your SASL project.

3.19.3.2.23 deleteTextureDataStorage

```
sasl.gl.deleteTextureDataStorage(number id)
```

Deletes texture data storage object, specified by numeric handle **id**. Use this function to unload memory, when your storage is not needed anymore. Note that you do not have to necessarily delete all your storages when SASL project is about to be unloaded. This will be performed automatically.

3.19.3.2.24 getTextureDataPointer

```
userdata data = sasl.gl.getTextureDataPointer(number id)
```

Returns raw texture data, stored in storage with specified numeric identifier **id**. Use this data to modify storage contents.

3.19.3.2.25 getRawTextureData

```
userdata data = sasl.gl.getRawTextureData(number texID, number storageID)
```

Fills storage, specified by numeric handle **storageID**, by texture raw data and returns this data. Texture is specified by numeric handle **texID**. Use this pointer to modify data in storage.

3.19.3.2.26 setRawTextureData

```
sasl.gl.setRawTextureData(number texID, number storageID)
```

Updates texture, specified by numeric handle **texID**, with current storage data. Storage is specified by numeric identifier **storageID**. Use this function to change actual texture contents.

3.19.3.2.27 imageFromTexture

```
sasl.gl.imageFromTexture(string filename, number texID)
```

Saves texture, specified by numeric handle **texID**, in file **filename**. Supported files extensions/formats are: **jpg** or **jpeg**, **bmp**, **tga**, **png**.

3.19.3.3 Text

Type: **TextAlignment**
Description: identifier of text alignment for drawing routines, can be equal to one of the following pre-defined values:

- TEXT_ALIGN_LEFT
- TEXT_ALIGN_RIGHT
- TEXT_ALIGN_CENTER
- TEXT_ALIGN_TOP
- TEXT_ALIGN_BOTTOM

3.19.3.3.1 drawBitmapText

```
sasl.gl.drawBitmapText(number id, number x, number y, string text,
    TextAlignment alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using bitmap font, specified by numeric identifier **id**. Use **TextAlignment** argument to set text alignment. The last argument is **color**. Use **loadBitmapFont** routine to get bitmap font handle.

testComponent.lua, drawing bitmap text on panel

```
testFont = sasl.gl.loadBitmapFont("fonts/calibri.fnt")

function draw()
    sasl.gl.drawBitmapText(testFont, 20, 50, TEXT_ALIGN_LEFT, "Hello_SASL")
end
```

3.19.3.3.2 drawRotatedBitmapText

```
sasl.gl.drawRotatedBitmapText(number id, number cx, number cy, number angle,
    number x, number y, string text, TextAlignment alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using bitmap font, specified by numeric identifier **id**. Text will be drawn rotated around **cx**, **cy** point on specified **angle**. Use **TextAlignment** argument to set text alignment. The last argument is **color**. Use **loadBitmapFont** routine to get bitmap font handle.

3.19.3.3.3 measureBitmapText

```
number width = sasl.gl.measureBitmapText(number id, string text)
```

Measures **text** string using bitmap font, specified by numeric identifier **id**. Returns the **width** of text bounding box in pixels.

3.19.3.3.4 measureBitmapTextGlyphs

```
number width = sasl.gl.measureBitmapTextGlyphs(number id, string text)
```

Measures **text** string using bitmap font, specified by numeric identifier **id**. Returns the **width** of used text glyphs in pixels.

3.19.3.3.5 setFontOutlineThickness

```
sasl.gl.setFontOutlineThickness(number id, number outlineThickness)
```

Sets outline thickness for font instance, specified by numeric font instance handle **id**. By default outline thickness of loaded font is set to 0.0 (no visible outline). Note that for visible outline you must also set outline color for this font with **setFontOutlineColor** routine.

3.19.3.3.6 setFontOutlineColor

```
sasl.gl.setFontOutlineColor(number id, Color color)
```

Sets outline **color** for font instance, specified by **id**.

testComponent.lua, setting up font outline

```
clRed = {1.0, 0.0, 0.0, 1.0}

testFont = sasl.gl.loadFont("fonts/calibri.ttf")
sasl.gl.setFontOutlineThickness(testFont, 2)
sasl.gl.setFontOutlineColor(testFont, {0.0, 1.0, 0.0})

function draw()
    sasl.gl.drawText(testFont, 20, 50, "Hello_SASL_1", 30, false, false,
        TEXT_ALIGN_LEFT, clRed)
    sasl.gl.drawText(testFont, 20, 80, "Hello_SASL_2", 30, false, true,
        TEXT_ALIGN_CENTER, clRed)
end
```

3.19.3.3.7 setFontRenderMode

```
sasl.gl.setFontRenderMode(number id, FontRenderMode mode)
sasl.gl.setFontRenderMode(number id, FontRenderMode mode, number value)
```

Type: **FontRenderMode**
Description: identifier of font instance render mode (only for modern text drawing functional), can be equal to one of the following pre-defined values:

- TEXT_RENDER_DEFAULT
- TEXT_RENDER_FORCED_MONO

Sets render **mode** for font instance, specified by **id**. For specific modes, additional parameter can be passed - **value**. For example, for mode TEXT_RENDER_FORCED_MONO - fixed glyphs mono spacing to the font size ratio should be passed as **value**.

3.19.3.3.8 setFontSize

```
sasl.gl.setFontSize(number id, number size)
```

Sets **size** for font instance, specified by **id**. Specified size will be used in instance-specific text routines - **drawTextI**, **drawRotatedTextI**, **measureTextI**, etc. Default font size is 12.

3.19.3.3.9 setFontDirection

```
sasl.gl.setFontDirection(number id, FontDirectionMode mode)
```

Type: **FontDirectionMode**
Description: identifier of font instance drawing direction mode (only for modern text drawing functional), can be equal to one of the following pre-defined values:

- TEXT_DIRECTION_HORIZONTAL
- TEXT_DIRECTION_VERTICAL

Sets **direction** for font instance, specified by **id**. Specified direction will be used for this font, horizontal id default.

3.19.3.3.10 setFontBold

```
sasl.gl.setFontBold(number id, boolean isBold)
```

Enables/disables bold mode for font instance, specified by **id**. Specified parameter will be used in instance-specific text routines - **drawTextI**, **drawRotatedTextI**, **measureTextI**, etc. Default value is **false**.

3.19.3.3.11 setFontItalic

```
sasl.gl.setFontItalic(number id, boolean isItalic)
```

Enables/disables italic mode for font instance, specified by **id**. Specified parameter will be used in instance-specific text routines - **drawTextI**, **drawRotatedTextI**, **measureTextI**, etc. Default value is **false**.

3.19.3.3.12 setFontBckMode

```
sasl.gl.setFontBckMode(number id, TextBckMode mode)
```

Type: **TextBckMode**
Description: identifier of text background rendering mode, can be equal to one of the following pre-defined values:

- TEXT_BCK_NONE - default mode, no background
- TEXT_BCK_STANDARD - background based on glyphs positions
- TEXT_BCK_RECTANGLE - rectangular background

Sets text background rendering mode for font instance, specified by **id**.

3.19.3.3.13 setFontBckColor

```
sasl.gl.setFontBckColor(number id, Color color)
```

Sets background **color** for font instance, specified by **id**.

3.19.3.3.14 setFontBckPadding

```
sasl.gl.setFontBckPadding(number id, number left)
sasl.gl.setFontBckPadding(number id, number left, number top)
sasl.gl.setFontBckPadding(number id, number left, number top, number right)
sasl.gl.setFontBckPadding(number id, number left, number top, number right,
    number bottom)
```

Sets text background padding values for font instance, specified by **id**. **left**, **top**, **right** and **bottom** are corresponding padding values. If **top** or **right** value is not specified, it will be equal to the **left**. If **bottom** value is not specified, it will be equal to the **top**.

3.19.3.3.15 setFontGlyphSpacingFactor

```
sasl.gl.setFontGlyphSpacingFactor(number id, number factor)
```

Sets glyph spacing **factor** for font instance, specified by **id**. Default value is 1.0.

3.19.3.3.16 setFontUnicode

```
sasl.gl.setFontUnicode(number id, boolean unicode)
```

Enables/disables Unicode glyphs support for the font instance (**id**). By default Unicode support is enabled.

3.19.3.3.17 saveFontState

```
sasl.gl.saveFontState(number id)
```

Saves current font instance configuration on stack. Use **sasl.gl.restoreFontState** function to restore saved state. Maximum size of stack is 16.

3.19.3.3.18 restoreFontState

```
sasl.gl.restoreFontState(number id)
```

Restores previously saved font instance configuration.

3.19.3.3.19 setRenderTextPixelAligned

```
sasl.gl.setRenderTextPixelAligned(boolean enabled)
```

Enables or disables special mode for text rendering. If this mode is enabled, all text drawing will be pixel aligned - this will avoid text distortion in case of resulting coordinates with fractional parts. By default this mode is disabled.

3.19.3.3.20 drawText

```
sasl.gl.drawText(number id, number x, number y, string text, number size,
    boolean isBold, boolean isItalic, TextAlignment alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using font instance, specified by numeric identifier **id**. **size** parameter defines font size. Text will be drawn with specified **alignment**. Drawn text may be bold or italic with setting corresponding **isBold** and **isItalic** parameters set to true. The last argument is text **color**.

testComponent.lua, drawing text on panel

```
testFont = sasl.gl.loadFont("fonts/calibri.ttf")
clRed = {1.0, 0.0, 0.0, 1.0}

function draw()
    sasl.gl.drawText(testFont, 20, 50, "Hello_SASL_1", 30, false, false,
        TEXT_ALIGN_LEFT, clRed)
    sasl.gl.drawText(testFont, 20, 80, "Hello_SASL_2", 30, false, true,
        TEXT_ALIGN_CENTER, clRed)
end
```

3.19.3.3.21 drawTextI

```
sasl.gl.drawTextI(number id, number x, number y, string text, TextAlignment
    alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using font instance, specified by numeric identifier **id**. Text will be drawn with specified **alignment**. The last argument is text **color**.

testComponent.lua, drawing text on panel

```
testFont = sasl.gl.loadFont("fonts/calibri.ttf")
sasl.gl.setFontSize(testFont, 30)
clRed = {1.0, 0.0, 0.0, 1.0}

function draw()
    sasl.gl.drawText(testFont, 20, 50, "Hello_SASL_1", TEXT_ALIGN_LEFT,
        clRed)
    sasl.gl.drawText(testFont, 20, 80, "Hello_SASL_2", TEXT_ALIGN_CENTER,
        clRed)
end
```

3.19.3.3.22 drawRotatedText

```
sasl.gl.drawRotatedText(number id, number x, number y, number cx, number cy,
     number angle, string text, number size, boolean isBold, boolean
    isItalic, TextAlignment alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using font instance, specified by numeric identifier **id**. Text will be drawn rotated around **cx**, **cy** point on specified **angle**. **size** parameter defines font size. Text will be drawn with specified **alignment**. Drawn text may be bold or italic with setting corresponding **isBold** and **isItalic** parameters set to true. The last argument is text **color**.

3.19.3.3.23 drawRotatedTextI

```
sasl.gl.drawRotatedTextI(number id, number x, number y, number cx, number cy
    , number angle, string text, TextAlignment alignment, Color color)
```

Draws **text** string at specified position **x**, **y** using font instance, specified by numeric identifier **id**. Text will be drawn rotated around **cx**, **cy** point on specified **angle**. Text will be drawn with specified **alignment**. The last argument is text **color**.

3.19.3.3.24 measureText

```
number width, number height = sasl.gl.measureText(number id, string text,
    number size, boolean isBold, boolean isItalic)
```

Measures **text** string using font instance, specified by numeric identifier **id**. **size** parameter defines font size. Use **isBold** and **isItalic** parameters to set text options. Returns the **width** and **height** of text bounding box in pixels.

3.19.3.3.25 measureTextGlyphs

```
number width = sasl.gl.measureTextGlyphs(number id, string text, number size
    , boolean isBold)
```

Measures **text** string using font instance, specified by numeric identifier **id**. **size** parameter defines font size. Use **isBold** and **isItalic** parameters to set text options. Returns the **width** of used text glyphs in pixels.

3.19.3.3.26 measureTextI

```
number width, number height = sasl.gl.measureTextI(number id, string text)
```

Measures **text** string using font instance, specified by numeric identifier **id**. Returns the **width** and **height** of text bounding box in pixels.

3.19.3.3.27 measureTextGlyphsI

```
number width = sasl.gl.measureTextGlyphsI(number id, string text)
```

Measures **text** string using font instance, specified by numeric identifier **id**. Returns the **width** of used text glyphs in pixels.

3.19.3.3.28 drawFontTexture

```
sasl.gl.drawFontTexture(number id, number x, number y)
```

Draws texture used as glyphs storage for font with specified **id** at position **x**, **y**. This is a utility function, which may be helpful in case of custom glyphs storages usage.

3.19.3.3.29 getFontInfo

```
string info = sasl.gl.getFontInfo(number id)
```

Returns information about font instance with specified **id**. This is a utility function, which may be helpful in case of custom glyphs storages usage.

3.19.3.4 Shaders

SASL provides functions for creating simple shader programs and using them for advanced rendering. You need to create empty shader program with **createShaderProgram** routine, add shaders to this program with **loadShader** function, link shader with **linkShaderProgram** to get working program and use it whenever you need in drawing process inside **draw** callback.

Type: **ShaderUniformType**
Description: identifier of shader uniform type, can be equal to one of the following pre-defined values:

- TYPE_INT - int.
- TYPE_FLOAT - float.
- TYPE_INT_ARRAY - int array.
- TYPE_FLOAT_ARRAY - float array.
- TYPE_SAMPLER - texture.

3.19.3.4.1 createShaderProgram

```
number id = sasl.gl.createShaderProgram()
```

Creates new empty shader program and returns its numeric identifier **id**. Use **loadShader** function to add shaders to program.

3.19.3.4.2 deleteShaderProgram

```
sasl.gl.deleteShaderProgram(number id)
```

Deletes shader program, specified by numeric identifier **id**. Note that you do not have to necessarily delete your shader programs, when your SASL project is about to be unloaded. This will be performed automatically.

3.19.3.4.3 linkShaderProgram

```
sasl.gl.linkShaderProgram(number id)
```

Links shader program, specified by numeric identifier **id**. You must call this function after you are loaded all shaders into program to get working shader program.

3.19.3.4.4 setShaderUniform

```
sasl.gl.setShaderUniform(number shaderID, string name, TYPE_INT, number data
    )
sasl.gl.setShaderUniform(number shaderID, string name, TYPE_FLOAT, number
    data)
sasl.gl.setShaderUniform(number shaderID, string name, TYPE_INT_ARRAY, table
     data)
sasl.gl.setShaderUniform(number shaderID, string name, TYPE_FLOAT_ARRAY,
    table data)
sasl.gl.setShaderUniform(number shaderID, string name, TYPE_SAMPLER, number
    textureID, number textureUnit)
```

Sets shader uniform variables with different types. **name** is a name of uniform variable. Shader specified by numeric identifier **shaderID**. Last function version, which accepts TYPE_SAMPLER parameter also takes additional parameter **textureUnit**. Basically, if you need to use a few sampler simultaneously in your shader, they must to be set in different texture units.

Required uniforms variables values most commonly is set right after **useShaderProgram** function to set up shader in current draw cycle.

testComponent.lua, setting uniforms

```
dotted = loadImage("dotted.png")

testShaderParameter = 0.0
testUniformTable1 = {0.0, 0.3, 1.0, 1.0, 0.1, 1.0, 2.0, 12.33, 0.0, 0.0, 1.0
    , 0.33, 1.0, 5.3, 1.0, 0.33}
testUniformTable2 = {1.0, 12.33, 1.0, 0.0, 0.0, 1.0, 0.33, 1.0, 1.0}

sasl.gl.setShaderUniform(testShaderProgram, "testParameter", TYPE_FLOAT,
    testShaderParameter)
sasl.gl.setShaderUniform(testShaderProgram, "testTextureWidth", TYPE_FLOAT,
    16)
```

```
sasl.gl.setShaderUniform(testShaderProgram, "testTextureHeight", TYPE_FLOAT,
     16)
sasl.gl.setShaderUniform(testShaderProgram, "testTexture", TYPE_SAMPLER,
    dotted, 0)
sasl.gl.setShaderUniform(testShaderProgram, "testArray1", TYPE_FLOAT_ARRAY,
    testUniformTable1)
sasl.gl.setShaderUniform(testShaderProgram, "testArray2", TYPE_FLOAT_ARRAY,
    testUniformTable2)
```

3.19.3.4.5 useShaderProgram

```
sasl.gl.useShaderProgram(number id)
```

Start using shader program, specified by numeric identifier **id**, for rendering. You must call **stopShaderProgram** function after you are done rendering with current shader program.

testComponent.lua, using shader in draw process

```
testShaderProgram = sasl.gl.createShaderProgram()
sasl.gl.loadShader(testShaderProgram, "shaders/dotted.vert",
    SHADER_TYPE_VERTEX)
sasl.gl.loadShader(testShaderProgram, "shaders/dotted.frag",
    SHADER_TYPE_FRAGMENT)
sasl.gl.linkShaderProgram(testShaderProgram)

function draw()
    sasl.gl.useShaderProgram(testShaderProgram)
    -- Setting uniform variables if needed
    ...
    -- Rendering
    ...
    sasl.gl.stopShaderProgram()
end
```

3.19.3.4.6 stopShaderProgram

```
sasl.gl.stopShaderProgram()
```

Stops shader program usage.

3.19.3.5 Blending

Type: **BlendFunctionID**
Description: identifier of blending function, can be equal to one of the following pre-defined values:

- BLEND_SOURCE_COLOR

- BLEND_ONE_MINUS_SOURCE_COLOR
- BLEND_SOURCE_ALPHA
- BLEND_ONE_MINUS_SOURCE_ALPHA
- BLEND_DESTINATION_ALPHA
- BLEND_ONE_MINUS_DESTINATION_ALPHA
- BLEND_DESTINATION_COLOR
- BLEND_ONE_MINUS_DESTINATION_COLOR
- BLEND_SOURCE_ALPHA_SATURATE
- BLEND_CONSTANT_COLOR
- BLEND_ONE_MINUS_CONSTANT_COLOR
- BLEND_CONSTANT_ALPHA
- BLEND_ONE_MINUS_CONSTANT_ALPHA

Type: **BlendEquationID**
Description: identifier of blending equation, can be equal to one of the following pre-defined values:

- BLEND_EQUATION_ADD
- BLEND_EQUATION_MIN
- BLEND_EQUATION_MAX
- BLEND_EQUATION_SUBTRACT
- BLEND_EQUATION_REVERSE_SUBTRACT

Refer to official OpenGL documentation to get the meaning of every selectable option, both for available functions values and equation values. You can change current blending options in your **draw** callbacks.

Every blending state is defined by 2 values of blending function (for source and destination color) and blending equation. Blending state may also be defined by 5 values in case of using separate blending functions for alpha components of source and destination color.

Default blending equation value is BLEND_EQUATION_ADD and default blending functions for source and destination is specific to version of simulator. Use **resetBlending** function to reset blending options to defaults.

3.19.3.5.1 setBlendFunction

```
sasl.gl.setBlendFunction(BlendFunctionID sourceBlend, BlendFunctionID
    destBlend)
```

Sets current blending functions for source and destination - **sourceBlend** and **destBlend**.

3.19.3.5.2 setBlendFunctionSeparate

```
sasl.gl.setBlendFunctionSeparate(BlendFunctionID sourceBlendRGB,
    BlendFunctionID destBlendRGB, BlendFunctionID sourceBlendAlpha,
    BlendFunctionID destBlendAlpha)
```

Sets current blending functions separately for RGB components (**sourceBlendRGB** and **destBlendRGB**) and for alpha component (**sourceBlendAlpha** and **destBlendAlpha**).

3.19.3.5.3 setBlendEquation

```
sasl.gl.setBlendEquation(BlendEquationID id)
```

Sets current blending equation, specified by identifier **id**.

3.19.3.5.4 setBlendEquationSeparate

```
sasl.gl.setBlendEquationSeparate(BlendEquationID equationIDRGB,
    BlendEquationID equationIDAlpha)
```

Sets current blending equations separately for RGB components and for alpha component - **equationIDRGB** and **equationIDAlpha**.

3.19.3.5.5 setBlendColor

```
sasl.gl.setBlendColor(Color color)
```

Sets current blend **color**. This color will be used in case of using one of the following **BlendFunctionIDs**:

- BLEND_CONSTANT_COLOR
- BLEND_ONE_MINUS_CONSTANT_COLOR

- BLEND_CONSTANT_ALPHA
- BLEND_ONE_MINUS_CONSTANT_ALPHA

3.19.3.5.6 resetBlending

```
sasl.gl.resetBlending()
```

Sets blending options to defaults. Use this function when you want to restore default blending options after you are done with drawing.

3.19.3.6 Clipping

3.19.3.6.1 setClipArea

```
sasl.gl.setClipArea(number x, number y, number width, number height)
```

Setup current clipping area by rectangle position, defined by **x**, **y**, **width** and **height**. Clip areas may be nested, but for every **setClipArea** function call, there must be corresponding **resetClipArea** function call. Use this function with caution - components hierarchy drawing may be broken in case of forgotten **resetClipArea** call.

testComponent.lua, setting clip area

```
function draw()
    sasl.gl.setClipArea(0, 0, 200, 200)
    -- Drawing
    ...
    sasl.gl.resetClipArea()
end
```

3.19.3.6.2 resetClipArea

```
sasl.gl.resetClipArea()
```

Resets current clip area. The previous clip area will be restored if there is any clip area left on the stack.

3.19.3.7 Masking

Next functions provides ability to draw with custom masks shape inside your components or into render target. To use this functional inside draw callbacks of your components, **fbo** property must be set to **true**.

3.19.3.7.1 drawMaskStart

```
sasl.gl.drawMaskStart()
```

Enables masking for current mask level and prepares drawing context to draw mask shape. Call this function before drawing mask shape with available drawing primitives (geometry shapes or alpha textures). During this stage of drawing color values may be omitted.

3.19.3.7.2 drawUnderMask

```
sasl.gl.drawUnderMask()
sasl.gl.drawUnderMask(boolean invertMaskLogic)
```

Prepares drawing context to draw under mask. Call this function before drawing under mask. Masking must be enabled. Call this function after you are done drawing mask shape. Also there's optional argument **invertMaskLogic**, which may be used to invert masking logic during this stage of drawing. By default **invertMaskLogic** is **false**.

3.19.3.7.3 drawMaskEnd

```
sasl.gl.drawMaskEnd()
```

Restores drawing context and previous drawing state, and disables masking.

3.19.3.8 Transformations

SASL has a number of specific routines to draw rotated textures and texture parts, but if you need to draw transformed geometry, text and other 2D stuff, you can use next functions to interact with current transformation matrix.

Use these routines with caution. You must always restore the transformation matrix to initial state after you are done drawing in component.

3.19.3.8.1 saveGraphicsContext

```
sasl.gl.saveGraphicsContext()
```

Saves current transformation matrix on the stack. Always call **restoreGraphicsContext** to restore previous transformation matrix.

3.19.3.8.2 restoreGraphicsContext

```
sasl.gl.restoreGraphicsContext()
```

Restores previous transformation matrix.

3.19.3.8.3 setTranslateTransform

```
sasl.gl.setTranslateTransform(number x, number y)
```

Multiplies current transformation matrix on translation matrix, specified by translation coordinates **x**, **y**.

3.19.3.8.4 setRotateTransform

```
sasl.gl.setRotateTransform(number angle)
```

Multiplies current transformation matrix on rotation matrix to rotate current context on **angle** degrees.

3.19.3.8.5 setScaleTransform

```
sasl.gl.setScaleTransform(number scaleX, number scaleY)
```

Multiplies current transformation matrix on scaling matrix, specified by scaling factors **scaleX**, **scaleY**.

3.19.3.9 Rendering Stages

In case of advanced 2D rendering selected (**Options** section), you need to know current rendering stage inside current **draw** call. Use functions from this section to determine current rendering stages.

3.19.3.9.1 isLitStage

```
boolean isLit = sasl.gl.isLitStage()
```

Returns **true** in case if SASL now draws in lit stage, and returns **false** otherwise.

3.19.3.9.2 isNonLitStage

```
boolean isNonLit = sasl.gl.isNonLitStage()
```

Returns **true** in case if SASL now draws in non-lit stage, and returns **false** otherwise.

3.19.3.9.3 isPanelBeforeStage

```
boolean isBeforePanel = sasl.gl.isPanelBeforeStage()
```

Returns **true** in case if SASL now draws before X-Plane, and returns **false** otherwise.

3.19.3.9.4 isPanelAfterStage

```
boolean isAfterPanel = sasl.gl.isPanelAfterStage()
```

Returns **true** in case if SASL now draws after X-Plane, and returns **false** otherwise.

3.19.3.10 Telemetry

3.19.3.10.1 startGraphicsTelemetry

```
sasl.gl.startGraphicsTelemetry()
```

Starts telemetry recording.

3.19.3.10.2 stopGraphicsTelemetry

```
number vertices, number batches = sasl.gl.stopGraphicsTelemetry()
```

Stops telemetry recording and returns amount of vertices and batches since start of record.

3.19.4 3D Graphics [DEPRECATED]

3.19.4.1 Primitives

3.19.4.1.1 drawLine3D [DEPRECATED]

```
sasl.gl.drawLine3D(number x1, number y1, number z1, number x2, number y2,
    number z2, Color color)
```

Draws 3D line between **x1**, **y1**, **z1** and **x2**, **y2**, **z2** points with specified **color** in OpenGL local 3D coordinates.

3.19.4.1.2 drawTriangle3D [DEPRECATED]

```
sasl.gl.drawTriangle3D(number x1, number y1, number z1, number x2, number y2
    , number z2, number x3, number y3, number z3, Color color)
```

Draws 3D triangle, specified by **x1**, **y1**, **z1** and **x2**, **y2**, **z2** and **x3**, **y3**, **z3** points with specified **color** in OpenGL local 3D coordinates.

3.19.4.1.3 drawCircle3D [DEPRECATED]

```
sasl.gl.drawCircle3D(number x, number y, number z, number radius, number
    pitch, number yaw, boolean isFilled, Color color)
```

Draws 3D circle, specified by **x**, **y**, **z** center and **radius** with specified **color** in OpenGL local 3D coordinates. Use **pitch** and **yaw** arguments to orient circle in 3D space. If **isFilled** is **true**, then circle will be filled.

3.19.4.1.4 drawAngle3D [DEPRECATED]

```
sasl.gl.drawAngle3D(number x, number y, number z, number angle, number
    length, number rays, number pitch, number yaw, Color color)
```

Draws 3D angle, centered at **x**, **y**, **z** and angular width **angle**, with specified **length**, made out of **rays** count. Use **pitch** and **yaw** arguments to orient angle in 3D space. The last argument is **color**.

3.19.4.1.5 drawStandingCone3D [DEPRECATED]

```
sasl.gl.drawStandingCone3D(number x, number y, number z, number radius,
    number height, Color color)
```

Draws standing 3D cone at **x**, **y**, **z** with **radius** and **height**. The last argument is **color**.

3.19.4.2 Transformations

Use functions from this section to interact with current 3D transformation matrix. Use them when you need to draw transformed 3D entities.

Use these routines with caution. You must always restore the transformation matrix to initial state after you are done drawing.

3.19.4.2.1 saveGraphicsContext3D [DEPRECATED]

```
sasl.gl.saveGraphicsState3D()
```

Saves current transformation matrix on the stack. Always call **restoreGraphicsContext3D** to restore previous transformation matrix state.

3.19.4.2.2 restoreGraphicsContext3D [DEPRECATED]

```
sasl.gl.restoreGraphicsContext3D()
```

Restores previous transformation matrix state.

3.19.4.2.3 setTranslateTransform3D [DEPRECATED]

```
sasl.gl.setTranslateTransform3D(number x, number y, number z)
```

Multiplies current transformation matrix on translation matrix, specified by translation coordinates **x**, **y**, **z**.

3.19.4.2.4 setRotateTransformX3D [DEPRECATED]

```
sasl.gl.setRotateTransformX3D(number angle)
```

Multiplies current transformation matrix on rotation matrix around X axis, specified by **angle** in degrees.

3.19.4.2.5 setRotateTransformY3D [DEPRECATED]

```
sasl.gl.setRotateTransformY3D(number angle)
```

Multiplies current transformation matrix on rotation matrix around Y axis, specified by **angle** in degrees.

3.19.4.2.6 setRotateTransformZ3D [DEPRECATED]

```
sasl.gl.setRotateTransformZ3D(number angle)
```

Multiplies current transformation matrix on rotation matrix around Z axis, specified by **angle** in degrees.

3.19.4.2.7 setRotateTransform3D [DEPRECATED]

```
sasl.gl.setRotateTransform3D(number angle, number x, number y, number z)
```

Multiplies current transformation matrix on rotation matrix around vector, specified by **x**, **y** and **z** coordinates. **angle** is rotation angle in degrees.

3.19.4.2.8 setScaleTransform3D [DEPRECATED]

```
sasl.gl.setScaleTransform3D(number x, number y, number z)
```

Multiplies current transformation matrix on scaling matrix, specified by scaling factors **x**, **y** and **z**.

3.20 Cursors

Every drawable 2D component (from panel or window layer) may have special component, which associates specific cursor with that component. Typical representation of cursors setup:

```
components = {
    someComponent {
        position = { 0, 0, 200, 100 },
        ...,
        cursor = {
            x = -8,
            y = -8,
            width = 16,
            height = 16,
            shape = sasl.gl.loadImage("myFancyCursor.png"),
            hideOSCursor = true
        }
    }
}
```

cursor - is a special component, which defines cursor shape and offsets for parent component (for **someComponent** component, in example).

shape - texture identifier for cursor image.

x - horizontal offset of left bottom cursor image corner.

y - vertical offset of left bottom cursor image corner.

width - width of cursor image.

height - height of cursor image.

hideOSCursor - boolean parameter, which defines how cursor will be drawn: on top of default cursor image, or instead of it (like in example).

3.21 Sound

3.21.1 Resources Management

3.21.1.1 loadSample

```
number id = sasl.al.loadSample(string fileName)
number id = sasl.al.loadSample(string fileName, boolean isNeedTimer)
number id = sasl.al.loadSample(string fileName, boolean isNeedTimer, boolean
    isNeedReversed)
```

Loads wave sample (**.wav** format) into memory from file, specified by **fileName**. Returns sample numeric handle **id**. Sample will be searched according to current **searchResourcesPath** list. Use **addSearchResourcesPath** function to add new search resources path.

Version with **isNeedTimer** parameter allows you to associate special timer object with this sample (this is needed for **getSamplePlayingRemaining** function). By default timer object is not created.

Version with **isNeedReversed** parameter allows you to reverse sample buffer.

Warning: use only mono samples for 3D sounds, or 3D sound will not work for this sample.

sound.lua, loading samples

```
testSound1 = sasl.al.loadSample("mySamples/EngineINN.wav")
testSound2 = sasl.al.loadSample("mySamples/EngineOUT.wav")
```

3.21.1.2 unloadSample

```
sasl.al.unloadSample(number id)
```

Unloads sample, specified by numeric identifier **id**. Note that you don't necessarily need to unload your samples when SASL project is unloads. This will be performed automatically. Use this function only when you want to unload sample and free some memory (and/or audio context occupancy) on the fly.

sound.lua, unloading samples

```
testSound1 = sasl.al.loadSample("mySamples/EngineINN.wav")
...
sasl.al.unloadSample(testSound1)
```

3.21.2 Sound management

Typical sound management in SASL consist from three parts:

- Loading samples from files
- Setting up options for samples (common ones, which applies to all samples, and specific ones)
- Calling playback functions

After loading samples with **loadSample** function, a number of specific options can be set for each loaded sample:

- Position
- Direction
- Velocity
- Cone
- Gain
- Pitch
- Playback offset
- Attachment point
- Environment

Every sample also has a number of advanced options, which defines sound computational and mixing parameters:

- Minimum Gain
- Maximum Gain
- Maximum Distance
- Rolloff factor
- Reference Distance

Listener options automatically controlled by SASL - listener position and orientation attached to camera position. Every SASL project has sound system origin and orientation. Positions, directions, cones and velocities of samples must be set relatively to this origin. Sound system origin and orientation depends on project type.

If SASL project is configured as **Aircraft plugin** - sound system origin located at aircraft CG and orientation configured to match aircraft orientation in 3D space. If SASL project is configured as **Global plugin** or **Scenery plugin** - sound system origin located in local OpenGL origin and oriented respectively.

Every sample has environment option (e.g. where sound will be active):

Type: **SoundEnvironment**
Description: identifier of sound environment, can be equal to one of pre-defined constants:

- SOUND_INTERNAL - corresponds to aircraft internal camera views.
- SOUND_EXTERNAL - corresponds to external camera views.
- SOUND_EVERYWHERE - corresponds to any camera view.

3.21.3 Sound playback

3.21.3.1 playSample

```
sasl.al.playSample(number id)
sasl.al.playSample(number id, boolean isLooping)
```

Starts playing sample with specified **id**. Use **loadSample** routine to obtain sample identifier. By default, sample will be played once. You can use optional parameter **isLooping** to make sample looped.

3.21.3.2 stopSample

```
sasl.al.stopSample(number id)
```

Stops playing sample with specified numeric **id**.

3.21.3.3 pauseSample

```
sasl.al.pauseSample(number id)
```

Pauses playing sample with specified numeric **id**. Sample will be played from the pause point after next **playSample** function call.

3.21.3.4 rewindSample

```
sasl.al.rewindSample(number id)
```

Sets sample playback position to zero.

3.21.3.5 isSamplePlaying

```
boolean isPlaying = sasl.al.isSamplePlaying(number id)
```

Returns **true** if sample, specified by **id** is playing right now and **false** otherwise.

3.21.3.6 getSamplePlayingRemaining

```
number time = sasl.al.getSamplePlayingRemaining(number id)
```

Returns remaining time of playing for sample, specified by numeric identifier **id**, if the sample is playing right now. Returns 0 otherwise. Note that this function will return proper values only if you load your samples with **needCreateTimer** argument set to true.

3.21.4 Sound settings

3.21.4.1 setSampleGain

```
sasl.al.setSampleGain(number id, number gain)
```

Sets gain of sample, specified by numeric identifier **id**. Argument **gain** must be in [0..1000] range.

3.21.4.2 setMasterGain

```
sasl.al.setMasterGain(number gain)
```

Adjusts gain of all samples in SASL sound system. Argument **gain** must be in [0..1000] range.

3.21.4.3 setSampleMinimumGain

```
sasl.al.setSampleMinimumGain(number id, number minGain)
```

Sets minimum gain of sample, specified by numeric identifier **id**. Argument **minGain** must be in [0..1000] range.

3.21.4.4 setSampleMaximumGain

```
sasl.al.setSampleMaximumGain(number id, number maxGain)
```

Sets maximum gain of sample, specified by numeric identifier **id**. Argument **maxGain** must be in $[0..1000]$ range.

3.21.4.5 setSamplePitch

```
sasl.al.setSamplePitch(number id, number pitch)
```

Sets pitch (frequency) of sample, specified by numeric identifier **id**. Argument **pitch** must be in $[0..any]$ range. Default pitch is 1000. Each reduction by 50 percent equals a pitch shift of -12 semitones (one octave reduction). Each doubling equals a pitch shift of 12 semitones (one octave increase).

3.21.4.6 setSampleOffset

```
sasl.al.setSampleOffset(number id, number offset)
```

Sets current playback point position for sample, specified with numeric identifier **id**. **offset** is specified in seconds from beginning sample point and must not exceed sample duration.

3.21.4.7 getSampleOffset

```
number offset = sasl.al.getSampleOffset(number id)
```

Returns current playback point position for sample, specified with numeric identifier **id**. **offset** is returned in seconds from beginning sample point.

3.21.4.8 getSampleDuration

```
number duration = sasl.al.getSampleDuration(number id)
```

Returns total duration of the sample (in seconds), specified with numeric identifier **id**.

3.21.4.9 setSamplePosition

```
sasl.al.setSamplePosition(number id, number x, number y, number z)
```

Sets 3D position of sample, specified by numeric identifier **id**. **x**, **y** and **z** specifies sample position. Default sample position is $\{0, 0, 0\}$. Sound can be used as non-positional (attached to camera) with use of **setSampleRelative** function.

3.21.4.10 getSamplePosition

```
number x, number y, number z = sasl.al.getSamplePosition(number id)
```

Returns 3D position of sample, specified by numeric identifier **id**.

3.21.4.11 setSampleDirection

```
sasl.al.setSampleDirection(number id, number x, number y, number z)
```

Sets direction vector of sample, specified by numeric identifier **id**. Vector is specified by **x**, **y** and **z**. Default sample direction is $\{0, 0, 0\}$ (non directional sound).

3.21.4.12 getSampleDirection

```
number x, number y, number z = sasl.al.getSampleDirection(number id)
```

Returns direction vector of sample, specified by numeric identifier **id**.

3.21.4.13 setSampleVelocity

```
sasl.al.setSampleVelocity(number id, number x, number y, number z)
```

Sets spatial velocity vector for sample, specified by numeric identifier **id**. Vector is specified by **x**, **y** and **z**. Default sample velocity is $\{0, 0, 0\}$.

3.21.4.14 getSampleVelocity

```
number x, number y, number z = sasl.al.getSampleVelocity(number id)
```

Returns spatial velocity vector for sample, specified by numeric identifier **id**.

3.21.4.15 setSampleCone

```
sasl.al.setSampleCone(number id, number outerGain, number innerAngle, number
    outerAngle)
```

Sets sound cone parameters for sample, specified by numeric identifier **id**. Each sample has three cone parameters. Sound cones applies only for directional sounds.

outerGain - the factor with which the sample gain is multiplied to determine the effective gain outside the cone. Must be in the range $[0..1000]$. **innerAngle** - inside angle of the sound cone in degrees. Default value is 360. **outerAngle** - outer angle of the sound cone in degrees. Default is 360.

When both inner and outer angle equals to 360 then the zone for angle depended attenuation is zero.

3.21.4.16 getSampleCone

```
number outerGain, number innerAngle, number outerAngle =
    sasl.al.getSampleCone(number id)
```

Returns sound cone parameters for sample, specified by numeric identifier **id**.

3.21.4.17 setSampleEnv

```
sasl.al.setSampleEnv(number id, SoundEnvironment env)
```

Sets sound environment option for sample, specified by numeric identifier **id**. Sound environment is specified by **env**. Default sound environment is **SOUND_EVERYWHERE**.

3.21.4.18 getSampleEnv

```
SoundEnvironment env = sasl.al.getSampleEnv(number id)
```

Returns sound environment value for sample, specified by numeric identifier **id**.

3.21.4.19 setSampleRelative

```
sasl.al.setSampleRelative(number id, number isRelative)
```

Sets attachment point for sample, specified by numeric identifier **id**. If **isRelative** parameter is equal to 1 - sample will be attached to camera (non-positional). If **isRelative** parameter is equal to 0 - sample will be attached to default attachment point for current SASL project type.

3.21.4.20 getSampleRelative

```
number isRelative = sasl.al.getSampleRelative(number id)
```

Returns attachment point identifier for sample, specified by numeric identifier **id**.

3.21.4.21 setSampleMaxDistance

```
sasl.al.setSampleMaxDistance(number id, number distance)
```

Sets maximum **distance**, at which sample can be heard. Sample is specified by numeric identifier **id**.

3.21.4.22 setSampleRolloffFactor

```
sasl.al.setSampleRolloffFactor(number id, number factor)
```

Sets computational rolloff **factor** for sample, specified by **id**. Argument **factor** must be in range $[0..any]$. Default rolloff factor for samples is 1.0.

3.21.4.23 setSampleRefDistance

```
sasl.al.setSampleRefDistance(number id, number distance)
```

Sets reference **distance** for sample, specified by **id**. Reference distance is the distance at which listener experience corresponding **gain** (set with **setSampleGain** function). Argument **distance** must be in range $[0..any]$. Default reference distance is 1.0.

Appendix A

SASL Developer Widget

SASL developer widget is designed to be a useful tool during development process. SASL Developer Widget must be enabled through the corresponding option in project configuration file. Don't forget to disable it for release versions!

> **Important:** SASL Widget will be automatically disabled in commercial versions of SASL.

If widget is enabled, you will find corresponding menu entry in X-Plane plugins menu - "Project Name (SASL)". There you can show and hide developer widget, as well as reset default widget position.. Widget configuration is saved between SASL plugin loadings (such as positions, widget window mode, visibility, selected tab, many on-off options etc.)

SASL Developer Widget will notify you in case if update for SASL is available.

A.1 Tabs

Each tab has a number of buttons and check-boxes to configure tab operations.

Telemetry tab can be used for viewing current performance and resources usage data during SASL project processing.

Console tab can be used for viewing console output inside simulator environment.

Project Tree tab can be used for viewing your project tree in Lua environment. You can also set values here on the fly.

DataRefs tab can be used for viewing and manipulating datarefs values (used by your project and all simulator datarefs).

Commands tab can be used viewing and calling commands (both used by your project and all simulator commands).

About will inform you about current SASL version and provide basic description.

A.2 Buttons and Check-Boxes

Close button will close SASL developer widget.

Reboot button will reload your SASL project. For reloading you can also use special command with following identifier - "sasl/reload/project_name".

Running check-box can enable and disable SASL project processing (should be useful in case of error messages spamming). You can also use corresponding commands with following identifiers - "sasl/start/project_name", "sasl/stop/project_name")

Pop-out (Pop-in) button will change SASL widget mode - OS window mode, or in-sim floating mode.

To the left of the **Pop-out (Pop-in)** button is the font size selection field.

For **DataRefs** and **Commands** tabs you can see the search field where you can enter both a single query and a more accurate search using the / character as a separator. If there are several options suitable for your request, the widget will show all of them.

In the **Commands** tab, you can select a command and call it using the button **Execute Command** at the bottom right or using the **Enter** button.

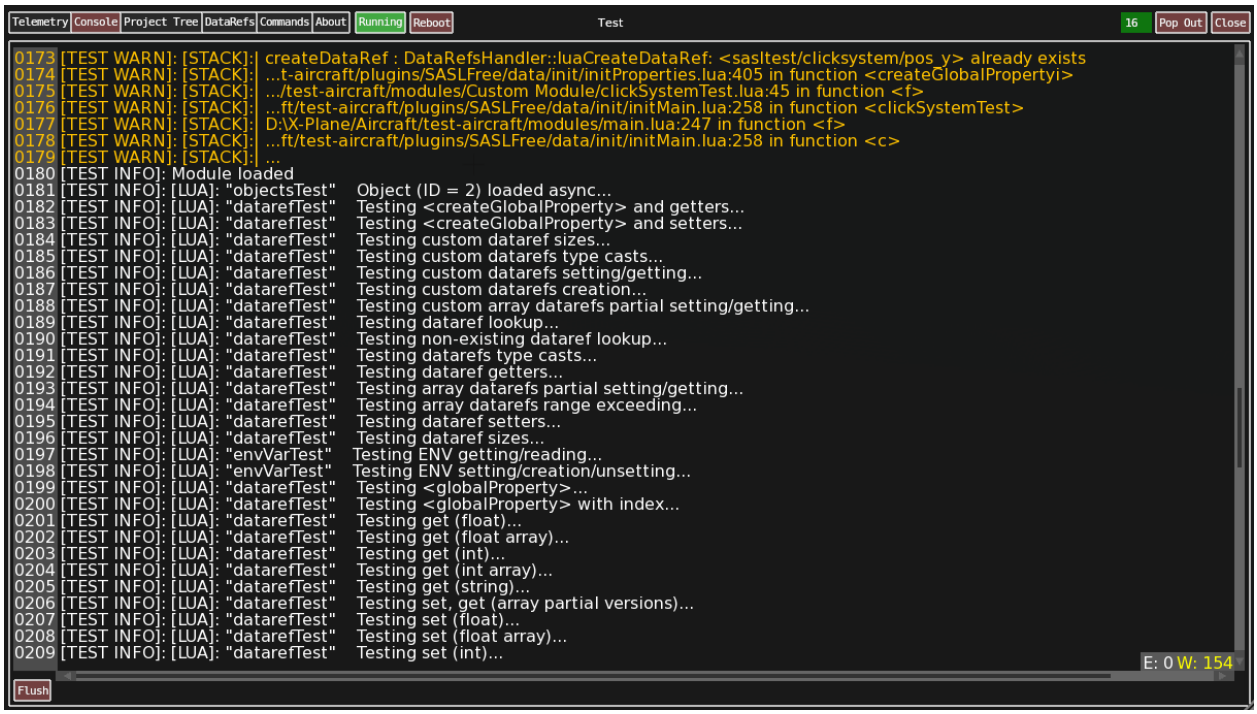

Figure A.1: Console Tab

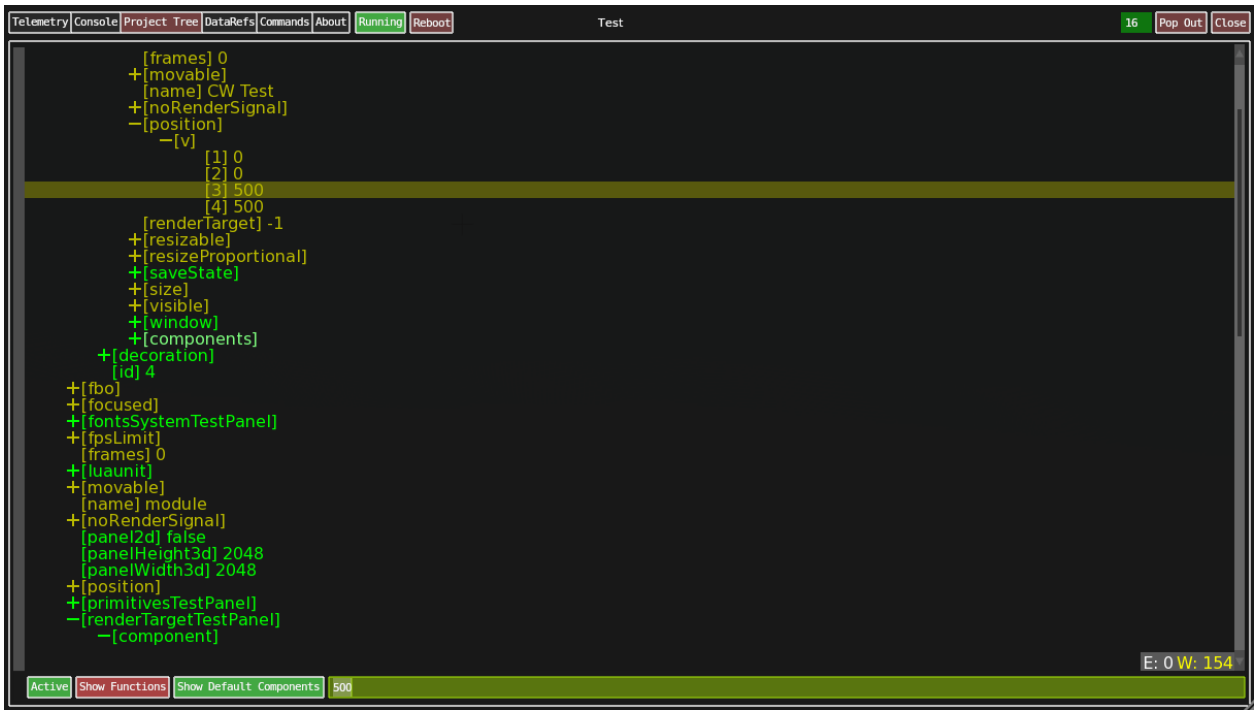

Figure A.2: Project Tree Tab

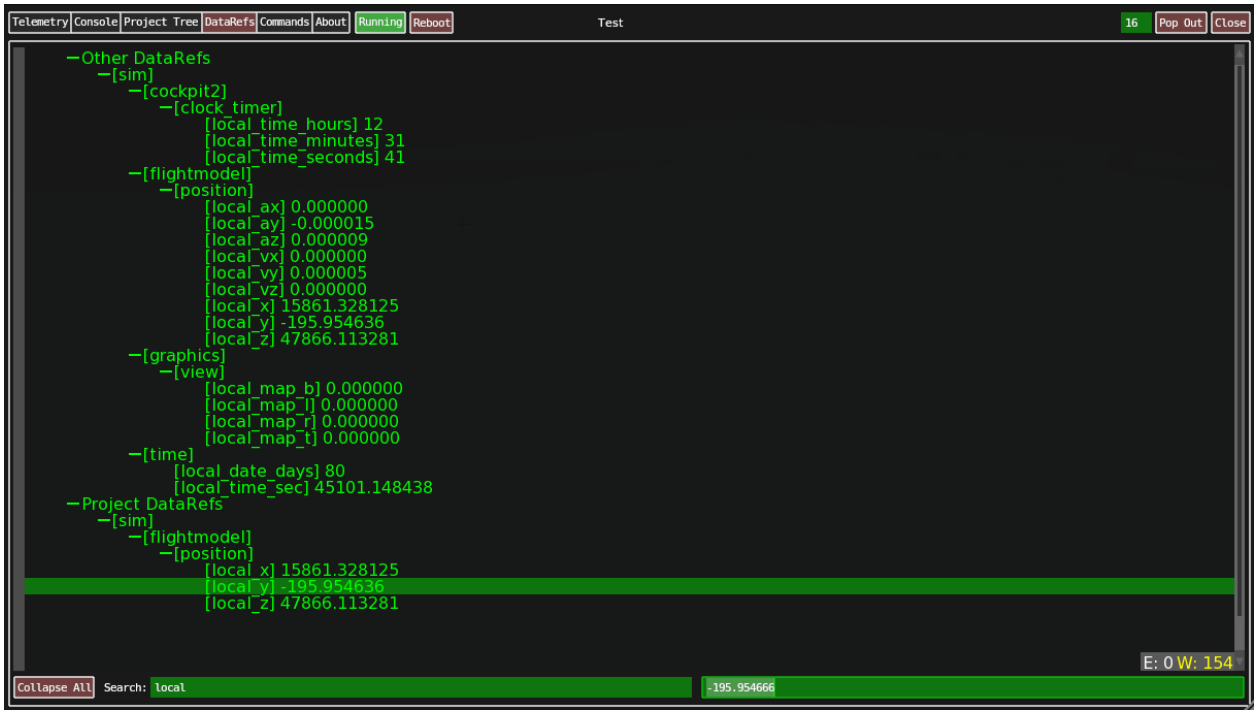

Figure A.3: DataRefs Tab

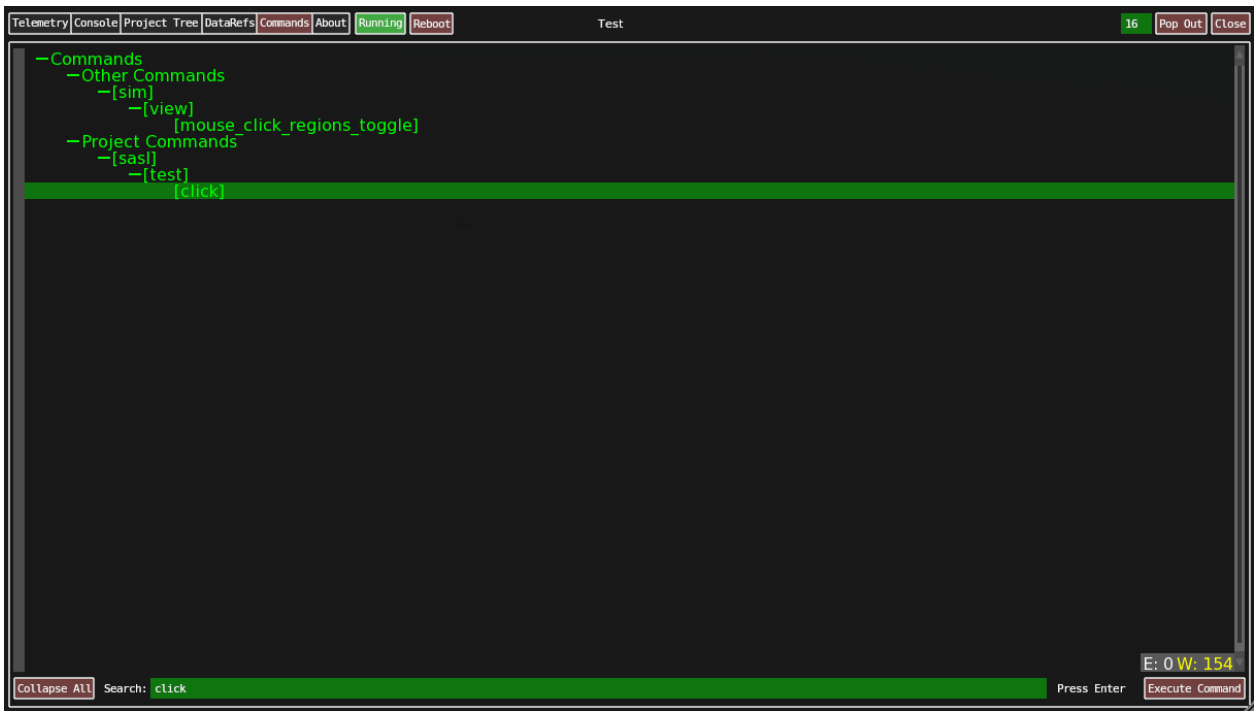

Figure A.4: Commands Tab

Appendix B

Interplugin communications

If custom X-Plane plugin wants to send or receive some data that supports SASL plugin, the pointers to the following C/C++ structures must be used with X-Plane interplugin messaging API:

```
struct IntArrayData {
    size_t mSize;
    int* mData;
};

struct FloatArrayData {
    size_t mSize;
    float* mData;
};

struct StringData {
    size_t mSize;
    char* mData;
};
```

In all above cases **mSize** member must be set to the size of array or string (number of elements, not size in bytes).