Summary

This document is a reference manual for X-Plane® SASL plugin.

Trademarks

Microsoft and Windows are registered trademarks of Microsoft Corporation.
Mac OS is a registered trademark of Apple, Inc.
Linux is a registered trademark of Linus Torvalds.
Ubuntu is a registered trademark of Canonical Ltd.
X-Plane is a registered trademark of Laminar Research.

All other trademarks mentioned in this document are the property of their respective owners.
Contents

1 Introduction And Basics ........................... 1
   1.1 Plugin ........................................ 1
   1.2 Project ....................................... 2
   1.3 Configuration ................................ 4
      1.3.1 [project] section .......................... 5
      1.3.2 [sceneryProject] section .................. 7
      1.3.3 [widget] section ......................... 8
   1.4 Entities ...................................... 9

2 Components ..................................... 10
   2.1 Default variables (fields) ................. 11
   2.2 Common properties ......................... 11
   2.3 Common callbacks ............................ 14
      2.3.1 General callbacks ..................... 15
      2.3.2 Mouse Callbacks ....................... 17
      2.3.3 Keyboard Callbacks ..................... 19
   2.4 Aircraft Global Parameters ............... 25

3 Functional .................................... 27
   3.1 Components .................................. 27
      3.1.1 subpanel ................................ 27
      3.1.2 contextWindow (XP11) .................. 28
         3.1.2.1 ContextWindow object (XP11) .... 31
            3.1.2.1.1 setSizeLimits .................. 31
            3.1.2.1.2 getSizeLimits .................. 31
            3.1.2.1.3 setIsVisible .................. 32
            3.1.2.1.4 isVisible ..................... 32
            3.1.2.1.5 setProportional ................. 32
            3.1.2.1.6 setMovable ..................... 32
            3.1.2.1.7 setResizable .................... 33
            3.1.2.1.8 setTitle ........................ 33
            3.1.2.1.9 setGravity ...................... 33
            3.1.2.1.10 setPosition .................... 34
            3.1.2.1.11 getPosition .................... 34
            3.1.2.1.12 setMode ........................ 34
            3.1.2.1.13 isPoppedOut .................. 35
            3.1.2.1.14 isInVR ......................... 35
      3.1.3 updateAll ................................ 36
3.1.4 drawAll .................................................. 36
3.1.5 drawAll3D .............................................. 36
3.1.6 drawAllObjects ....................................... 37
3.1.7 Search paths ......................................... 37
  3.1.7.1 addSearchPath .................................. 37
  3.1.7.2 addSearchResourcesPath ...................... 37
3.1.8 Logging .............................................. 38
  3.1.8.1 logInfo ......................................... 38
  3.1.8.2 print ........................................... 38
  3.1.8.3 logWarning ..................................... 38
  3.1.8.4 logError ....................................... 38
  3.1.8.5 logDebug ....................................... 38
3.2 Properties ............................................. 40
  3.2.1 Components Properties ........................... 40
    3.2.1.1 defineProperty ................................ 40
  3.2.2 Simulator Properties ............................... 40
    3.2.2.1 globalProperty ................................ 40
    3.2.2.2 createGlobalProperty ......................... 41
    3.2.2.3 createFunctionalProperty ................. 42
    3.2.2.4 size .......................................... 43
  3.2.3 Get and Set ....................................... 44
    3.2.3.1 get ........................................... 44
    3.2.3.2 set ........................................... 44
3.3 Options ................................................ 46
  3.3.1 Performance ....................................... 46
    3.3.1.1 setAircraftPanelRendering .................. 46
    3.3.1.2 set3DRendering ................................ 46
    3.3.1.3 setInteractivity ................................ 46
  3.3.2 Rendering Customization .......................... 47
    3.3.2.1 setRenderingMode2D .......................... 47
    3.3.2.2 setPanelRenderingMode ....................... 47
  3.3.3 Debugging ......................................... 48
    3.3.3.1 setLuaErrorsHandling ....................... 48
    3.3.3.2 setLuaStackTraceLimit ..................... 48
3.4 Windows (XP11) ...................................... 50
  3.4.1 getMonitorsIDsGlobal ............................. 50
  3.4.2 getMonitorsIDsOS ................................ 50
  3.4.3 getMonitorBoundsGlobal ......................... 50
  3.4.4 getMonitorBoundsOS ................................ 51
  3.4.5 getScreenBoundsGlobal .......................... 51
3.5 Commands .............................................. 53
  3.5.1 findCommand ...................................... 53
  3.5.2 commandBegin ..................................... 53
  3.5.3 commandEnd ....................................... 53
  3.5.4 commandOnce ...................................... 54
  3.5.5 createCommand .................................... 54
  3.5.6 registerCommandHandler ......................... 54
  3.5.7 unregisterCommandHandler ...................... 55
3.6 Menus
3.6.1 appendMenuItem
3.6.2 appendMenuItemWithCommand (XP11)
3.6.3 removeMenuItem
3.6.4 setMenuItemName
3.6.5 setMenuItemState
3.6.6 getMenuItemState
3.6.7 enableMenuItem
3.6.8 createMenu
3.6.9 appendMenuSeparator
3.6.10 clearAllMenuItems
3.6.11 destroyMenu

3.7 Message Windows
3.7.1 messageWindow

3.8 Cameras
3.8.1 getCamera
3.8.2 setCamera
3.8.3 registerCameraController
3.8.4 unregisterCameraController
3.8.5 getCurrentCameraStatus
3.8.6 startCameraControl
3.8.7 stopCameraControl

3.9 Net
3.9.1 downloadFileSync
3.9.2 downloadFileContentsSync

3.10 Timers
3.10.1 createTimer
3.10.2 deleteTimer
3.10.3 startTimer
3.10.4 pauseTimer
3.10.5 resumeTimer
3.10.6 stopTimer
3.10.7 getElapsedSeconds
3.10.8 getElapsedMicroseconds
3.10.9 getCurrentCycle

3.11 Interplugin Communications
3.11.1 Interplugin Utilities
3.11.1.1 getMyPluginID
3.11.1.2 getMyPluginPath
3.11.1.3 getXPlanePath
3.11.1.4 getProjectPath
3.11.1.5 getProjectName
3.11.1.6 getAircraftPath
3.11.1.7 getAircraft
3.11.1.8 countPlugins
3.11.1.9 getNthPlugin
3.11.1.10 findPluginByPath
3.11.1.11 findPluginBySignature
3.11.1.12 `getPluginInfo` ........................................ 70
3.11.1.13 `isPluginEnabled` ...................................... 70
3.11.1.14 `enablePlugin` ......................................... 70
3.11.1.15 `disablePlugin` ........................................ 70
3.11.1.16 `reloadPlugins` ........................................ 70
3.11.2 Messages .................................................... 70
   3.11.2.1 `registerMessageHandler` ............................. 71
   3.11.2.2 `unregisterMessageHandler` .......................... 72
   3.11.2.3 `sendMessageToPlugin` ................................ 72
3.12 Auxiliary Mouse Input System ................................ 73
   3.12.1 `setAuxiliaryClickSystem` ............................... 74
   3.12.2 `setCSDClickInterval` ................................... 74
   3.12.3 `setCSMode` ............................................. 74
   3.12.4 `setCSShowCursor` ....................................... 74
   3.12.5 `setCSCursorScale` ..................................... 75
   3.12.6 `getCSClickDown` ....................................... 75
   3.12.7 `getCSClickUp` .......................................... 75
   3.12.8 `getCSClickHold` ....................................... 76
   3.12.9 `getCSDoubleClick` ..................................... 76
   3.12.10 `getCSWheelClicks` .................................... 76
   3.12.11 `getCSMouseXPos` ..................................... 76
   3.12.12 `getCSMouseYPos` ..................................... 76
   3.12.13 `getCSDragDirection` .................................. 77
   3.12.14 `getCSDragValue` ..................................... 77
   3.12.15 `getCSCursorOnInterface` .............................. 77
3.13 Auxiliary Keyboard Input System ............................. 78
   3.13.1 `registerGlobalKeyHandler` ............................. 78
   3.13.2 `unregisterGlobalKeyHandler` ......................... 79
   3.13.3 Hot Keys Management .................................... 79
       3.13.3.1 `registerHotKey` .................................. 79
       3.13.3.2 `unregisterHotKey` ................................. 79
       3.13.3.3 `setHotKeyCombination` ............................ 80
3.14 Utilities ....................................................... 81
   3.14.1 Operating system ....................................... 81
       3.14.1.1 `getOS` ............................................ 81
       3.14.1.2 `getXPVersion` .................................... 81
       3.14.1.3 `listFiles` ........................................ 81
       3.14.1.4 `setClipboardText` ................................ 81
       3.14.1.5 `getClipboardText` ................................. 82
   3.14.2 Mathematics ............................................. 82
       3.14.2.1 `newInterpolator` .................................. 82
       3.14.2.2 `deleteInterpolator` ................................. 82
       3.14.2.3 `interpolate` (1-dimensional) ....................... 83
       3.14.2.4 `interpolate` (general) ............................. 83
       3.14.2.5 `selfInterpolator` ................................ 83
       3.14.2.6 `isInTheRect` ...................................... 84
3.14.3 Files And Scripts ......................................... 84
   3.14.3.1 `isFileExists` ....................................... 84
3.14.3.2 extractFileName ........................................... 84
3.14.3.3 openFile .................................................. 84
3.14.3.4 findFile .................................................. 84
3.14.3.5 include .................................................. 84
3.14.4 Miscellaneous ............................................. 85
3.14.4.1 toboolean .............................................. 85
3.15 Logging ......................................................... 86
3.15.1 logInfo ....................................................... 86
3.15.2 logWarning ................................................ 86
3.15.3 logError .................................................... 86
3.15.4 logDebug ................................................... 86
3.15.5 logTrace .................................................... 87
3.15.6 getLogLevel ................................................. 87
3.15.7 setLogLevel ................................................ 87
3.16 Basic Navigation ............................................. 88
3.16.1 Navigational Aids ....................................... 89
3.16.1.1 getFirstNavAid ....................................... 89
3.16.1.2 getNextNavAid ....................................... 89
3.16.1.3 findFirstNavAidOfType .............................. 89
3.16.1.4 findLastNavAidOfType ............................... 89
3.16.1.5 findNavAid ............................................ 89
3.16.1.6 getNavAidInfo ....................................... 90
3.16.2 FMS ........................................................ 90
3.16.2.1 countFMSEntries ................................... 90
3.16.2.2 getDisplayedFMSEntry ............................. 90
3.16.2.3 getDestinationFMSEntry .......................... 90
3.16.2.4 setDisplayedFMSEntry ............................ 90
3.16.2.5 setDestinationFMSEntry ......................... 91
3.16.2.6 getFMSEntryInfo .................................. 91
3.16.2.7 setFMSEntryInfo .................................. 91
3.16.2.8 setFMSEntryLatLon ................................. 91
3.16.2.9 clearFMSEntry ....................................... 91
3.16.3 GPS ........................................................ 91
3.16.3.1 getGPSDestinationType ......................... 91
3.16.3.2 getGPSDestination ................................. 92
3.17 Scenery ......................................................... 93
3.17.1 Resources Management .................................. 93
3.17.1.1 loadObject .......................................... 93
3.17.1.2 loadObjectAsync (XP11) ......................... 93
3.17.1.3 unloadObject ....................................... 93
3.17.2 Scenery Functions ....................................... 94
3.17.2.1 drawObject .......................................... 94
3.17.2.2 reloadScenery ...................................... 95
3.17.2.3 worldToLocal ...................................... 95
3.17.2.4 localToWorld ....................................... 95
3.17.2.5 modelToLocal ....................................... 95
3.17.2.6 localToModel ....................................... 95
3.17.2.7 probeTerrain ....................................... 95
3.17.3 Magnetic Variation (XP11) ........................................ 96
  3.17.3.1 getMagneticVariation ........................................ 96
  3.17.3.2 degMagneticToDegTrue ...................................... 96
  3.17.3.3 degTrueToDegMagnetic ...................................... 96
3.17.4 Instancing (XP11) .............................................. 96
  3.17.4.1 createInstance ................................................ 97
  3.17.4.2 destroyInstance ............................................. 97
  3.17.4.3 setInstancePosition .......................................... 97
3.18 Graphics .......................................................... 99
  3.18.1 Resources Management ......................................... 99
    3.18.1.1 loadImage .................................................. 99
    3.18.1.2 loadVectorImage .......................................... 99
    3.18.1.3 loadImageFromMemory .................................... 100
    3.18.1.4 unloadImage .............................................. 100
    3.18.1.5 loadBitmapFont .......................................... 101
    3.18.1.6 unloadBitmapFont ........................................ 101
    3.18.1.7 loadFont ................................................. 101
    3.18.1.8 unloadFont ................................................ 101
    3.18.1.9 loadShader ................................................ 102
  3.18.2 Color .......................................................... 102
3.18.3 2D Graphics ..................................................... 102
  3.18.3.1 Primitives ................................................... 102
    3.18.3.1.1 drawLine .............................................. 102
    3.18.3.1.2 drawWideLine ........................................ 103
    3.18.3.1.3 drawPolyLine .......................................... 103
    3.18.3.1.4 drawWidePolyLine .................................... 103
    3.18.3.1.5 drawTriangle .......................................... 103
    3.18.3.1.6 drawRectangle ......................................... 104
    3.18.3.1.7 drawFrame .............................................. 104
    3.18.3.1.8 setLinePattern ......................................... 104
    3.18.3.1.9 drawLinePattern ....................................... 104
    3.18.3.1.10 drawPolyLinePattern ................................ 105
    3.18.3.1.11 drawBezierLineQ ..................................... 105
    3.18.3.1.12 drawWideBezierLineQ ................................. 105
    3.18.3.1.13 drawBezierLineQAadaptive .......................... 105
    3.18.3.1.14 drawWideBezierLineQAadaptive ...................... 106
    3.18.3.1.15 drawBezierLineC ..................................... 106
    3.18.3.1.16 drawWideBezierLineC ................................ 106
    3.18.3.1.17 drawBezierLineCAdaptive ............................ 106
    3.18.3.1.18 drawWideBezierLineCAdaptive ....................... 106
    3.18.3.1.19 drawCircle ........................................... 107
    3.18.3.1.20 drawArc .............................................. 107
    3.18.3.1.21 drawConvexPolygon .................................. 107
  3.18.3.2 Textures ..................................................... 108
    3.18.3.2.1 drawTexture ........................................... 108
    3.18.3.2.2 drawRotatedTexture ................................... 108
    3.18.3.2.3 drawRotatedTextureCenter ............................ 108
    3.18.3.2.4 drawTexturePart ....................................... 109
3.18.3.2.5 drawRotatedTexturePart
3.18.3.2.6 drawRotatedTexturePartCenter
3.18.3.2.7 drawTextureCoords
3.18.3.2.8 drawTextureWithRotatedCoords
3.18.3.2.9 getTextureSize
3.18.3.2.10 getTextureSourceSize
3.18.3.2.11 setTextureWrapping
3.18.3.2.12 importTexture
3.18.3.2.13 recreateTexture
3.18.3.2.14 setRenderTarget
3.18.3.2.15 restoreRenderTarget
3.18.3.2.16 createRenderTarget
3.18.3.2.17 getTargetTextureData
3.18.3.2.18 createTextureDataStorage
3.18.3.2.19 deleteTextureDataStorage
3.18.3.2.20 getTextureDataPointer
3.18.3.2.21 getRawTextureData
3.18.3.2.22 setRawTextureData

3.18.3.3 Text
3.18.3.3.1 drawBitmapText
3.18.3.3.2 drawRotatedBitmapText
3.18.3.3.3 drawText
3.18.3.3.4 drawRotatedText
3.18.3.3.5 setFontOutlineThickness
3.18.3.3.6 setFontOutlineColor
3.18.3.3.7 measureBitmapText
3.18.3.3.8 measureText
3.18.3.3.9 setRenderTextPixelAligned

3.18.3.4 Shaders
3.18.3.4.1 createShaderProgram
3.18.3.4.2 deleteShaderProgram
3.18.3.4.3 linkShaderProgram
3.18.3.4.4 setShaderUniform
3.18.3.4.5 useShaderProgram
3.18.3.4.6 stopShaderProgram

3.18.3.5 Blending
3.18.3.5.1 setBlendFunction
3.18.3.5.2 setBlendFunctionSeparate
3.18.3.5.3 setBlendEquation
3.18.3.5.4 setBlendEquationSeparate
3.18.3.5.5 setBlendColor
3.18.3.5.6 resetBlending

3.18.3.6 Clipping
3.18.3.6.1 setClipArea
3.18.3.6.2 resetClipArea

3.18.3.7 Masking
3.18.3.7.1 drawMaskStart
3.18.3.7.2 drawUnderMask
3.18.3.8 Transformations ................................................. 122
  3.18.3.8.1 saveGraphicsContext ................................. 122
  3.18.3.8.2 restoreGraphicsContext .............................. 122
  3.18.3.8.3 setTranslateTransform ............................... 122
  3.18.3.8.4 setRotateTransform ................................. 123
  3.18.3.8.5 setScaleTransform .................................... 123
3.18.3.9 Rendering Stages ............................................. 123
  3.18.3.9.1 isLitStage ........................................... 123
  3.18.3.9.2 isNonLitStage ....................................... 123
  3.18.3.9.3 isPanelBeforeStage .................................. 123
  3.18.3.9.4 isPanelAfterStage ................................... 123
3.18.4 3D Graphics ...................................................... 124
  3.18.4.1 Primitives ................................................ 124
    3.18.4.1.1 drawLine3D ....................................... 124
    3.18.4.1.2 drawTriangle3D .................................... 124
    3.18.4.1.3 drawCircle3D ...................................... 124
    3.18.4.1.4 drawAngle3D ...................................... 124
    3.18.4.1.5 drawStandingCone3D ............................... 124
  3.18.4.2 Transformations .......................................... 125
    3.18.4.2.1 saveGraphicsContext3D ........................... 125
    3.18.4.2.2 restoreGraphicsContext3D ........................ 125
    3.18.4.2.3 setTranslateTransform3D ......................... 125
    3.18.4.2.4 setRotateTransformX3D ........................... 125
    3.18.4.2.5 setRotateTransformY3D ........................... 125
    3.18.4.2.6 setRotateTransformZ3D ........................... 126
    3.18.4.2.7 setRotateTransform3D ............................. 126
    3.18.4.2.8 setScaleTransform3D .............................. 126

3.19 Cursors .......................................................... 127

3.20 Sound ............................................................. 128
  3.20.1 Resources Management .................................... 128
    3.20.1.1 loadSample ......................................... 128
    3.20.1.2 unloadSample ....................................... 128
  3.20.2 Sound management .......................................... 128
  3.20.3 Sound playback ............................................. 130
    3.20.3.1 playSample .......................................... 130
    3.20.3.2 stopSample .......................................... 130
    3.20.3.3 pauseSample ......................................... 130
    3.20.3.4 rewindSample ........................................ 130
    3.20.3.5 isSamplePlaying ...................................... 131
    3.20.3.6 getSamplePlayingRemaining .......................... 131
  3.20.4 Sound settings .............................................. 131
    3.20.4.1 setSampleGain ........................................ 131
    3.20.4.2 setMasterGain ....................................... 131
    3.20.4.3 setMinimumSampleGain ............................... 131
    3.20.4.4 setMaximumSampleGain ............................... 131
    3.20.4.5 setSamplePitch ...................................... 132
    3.20.4.6 setSampleOffset ..................................... 132
Chapter 1

Introduction And Basics

SASL - is a scripting plugin 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) global plugins, plugins for their aircraft and sceneries. SASL can basically do everything the native X-Plane® SDK can do, and even more. SASL has many preprogrammed features for convenient add-ons developing process: drawing text and graphics primitives (both 2D and 3D), drawing textures, simple shaders support, timers, commands, advanced keyboard and mouse input handling, clipboard support, cameras handling, interpolations helpers and many other built-in features.

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

SASL provides a big amount of different functionality and supports following platforms:

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

1.1 Plugin

SASL plugin must be placed in plugins folder as an aircraft plugin, scenery plugin or global plugin (standard plugin installation for X-Plane®).

SASL plugin distribution is just an engine that runs SASL projects inside simulator environment. The basic SASL plugin distribution is shown on following figure:

Folders 64 and liblinux contains plugin binaries and additional libraries. data folder contains all SASL engine stuff:

- Standard SASL components inside components folder.
- SASL engine scripts inside init folder.
empty output folder. Logs and other output will be dumped there while plugin is running in simulator.

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. Projects in this location classified as Outer Project. For example, with using this terminology, all aircraft SASL plugins in previous major SASL versions were Outer Projects. Developer is free to choose the project place and corresponding project class with only one limitation: Global SASL projects must be Inner Projects because of X-Plane® global plugins structure. More information about Global project and other project types will follow in next chapter.

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.

1.2 Project

SASL project can be of three types:

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

Every SASL project must have pre-defined structure.

Figure 1.2 shows an example of how SASL project files tree might look like. The core of the project is the main.lua root script and the Custom Module folder that contains other project’s scripts in general. But to validate the project 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, so it’s vital to properly configure your project. Project type (aircraft, scenery, 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 chapter.

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 lookup additional Lua modules (LuaSocket, etc).

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:
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:

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

[sceneryProject]
centerLatitude=0.0000000
centerLongitude=0.0000000
maxElevation=0.0
radius=0.0
```
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 required section. Other sections are optional and depends on Project section values.

id is an integer number, greater than 0. Currently not used for any purpose. 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 following values:

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

startDisabled is an integer number, which identifies 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 widget for the project. Can be equal to following values:

• 0 - Widget Off
• 1 - Widget On

SASL Widget - is a special widget, which designed to make add-on developing process more convenient. Generally, widget must be enabled only in development process and must be disabled in release version of project for better performance. More information about SASL 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 center of scenery project.

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

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 zone, where scenery project will be active. If the user aircraft is outside to this zone, scenery project will become inactive. Activating and deactivating processes will be performed automatically, based on current user aircraft location.
1.3.3 [widget] section

This section in configuration.ini file is used for configuring SASL Widget visual options in case if SASL Widget enabled. Section must be specified in this case, it 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 (in TTF or other acceptable format) must be located inside configuration/widgetResources folder. There is 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.

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

### 1.4 Entities

SASL gives the access to most manageable entities in X-Plane® (datarefs, commands, cameras, menus, internal communication, objects, gauges, windows, sound, etc.).

In terms of graphics, SASL interacts with:

- Aircraft panel
- 2D pop-up panels and windows
- 3D World

Note that aircraft panels cannot be accessed from scenery project.
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** (common ones or defined by the 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 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.

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

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

```lua
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**

```lua
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**

```lua
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.

```lua
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.

```lua
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.
**Warning**: it is not allowed to create FrameBuffer Objects in subcomponents of components with their own FrameBuffer Objects.

---

**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.

```lua
someComponent.lua, setting clipping
```

```lua
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

```lua
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.

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

```lua
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.

**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.

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

**Callback: onModuleDone()**

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

someComponent.lua, saving state for future use

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

**Callback: onAirportLoaded()**

**Description:** called when the user’s plane is positioned at new airport.

**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.

```
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, x, y, MouseButton, parentX, 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.

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

**Callback**: `onMouseUp(component, x, y, MouseButton, parentX, 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, x, y, MouseButton, parentX, 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, x, y, MouseButton, parentX, 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.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
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

```lua
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**

```lua
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 proportionalToXWindow 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.

```lua
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, customDecore, decoration, noResize, noMove, fbo, clip, clipSize, command, description, resizeCallback. 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.

**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 header depiction and events (when **customDecore = true**). You can specify only those callbacks or values which you want to override.

```plaintext
decoration = {
    headerHeight = ... -- Default header height is 25px.
    draw = function(u, 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**.

**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**.

**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.

**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 3 arguments: higher level component, new width and height of resized window.
The `components` field is a table of context window child components.

**main.lua, creation of context window**

```lua
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

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

Applies provided size limits for the context window.

```lua
myWindow = contextWindow {
    ...  
}
myWindow:setSizeLimits(50, 50, 650, 600)
```

#### 3.1.2.1.2 getSizeLimits

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

Returns current size limits for the context window.

```lua
myWindow = contextWindow {
    ...  
}
print(myWindow:getSizeLimits())
```
3.1.2.1.3  setIsVisible

ContextWindow:isSetVisible ( boolean isVisible )

Changes context window visibility state.

```plaintext
myWindow = contextWindow {
    ...
}
myWindow:isSetVisible ( false )
```

3.1.2.1.4  isVisible

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

Returns context window visibility state.

myWindow = contextWindow {
    ...
}
print ( myWindow:isVisible () )
```

3.1.2.1.5  setProportional

```plaintext
ContextWindow:setProportional ( boolean isProportional )

Enables/disables proportional mode for context window contents.

myWindow = contextWindow {
    ...
}
myWindow:setProportional ( true )
```

3.1.2.1.6  setMovable

```plaintext
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.
```
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

```lua
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.

```lua
myWindow = contextWindow {
...
}

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

3.1.2.1.11 getPosition

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

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

```lua
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

```lua
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 want 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 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

3.1.2.1.14 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

3.1.3 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 addSearchResourcesPath

addSearchResourcesPath(string path)

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

3.1.8.1 logInfo

logInfo(string info)

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

3.1.8.2 print

print(string info)

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

3.1.8.3 logWarning

logWarning(string warning)

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

3.1.8.4 logError

logError(string error)

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

3.1.8.5 logDebug
logDebug(string debug)

Writes debug string 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

```lua
defineProperty(name, 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.

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

3.2.2 Simulator Properties

3.2.2.1 globalProperty

```lua
property p = globalProperty(name)  
property p = globalPropertyf(name)  
property p = globalPropertyd(name)  
property p = globalPropertyS(name)  
property p = globalPropertyia(name)  
property p = globalPropertyfa(name)
```

```lua
property p = globalProperty(name, boolean suppCastsWarn)  
property p = globalPropertyf(name, boolean suppCastsWarn)  
property p = globalPropertyd(name, boolean suppCastsWarn)  
property p = globalPropertyS(name, boolean suppCastsWarn)  
property p = globalPropertyia(name, boolean suppCastsWarn)  
property p = globalPropertyfa(name, boolean suppCastsWarn)
```
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 (int, float, double, string(data), int array, float array) with corresponding name. suppCastsWarn is a boolean optional argument that changes function verbosity. If its 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

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

Creates new simulator property of type (int, float, double, string(data), int array, float array) with corresponding name and assign default value to it. There is optional boolean parameters isNotPublished and isShared. If isNotPublished is true, then such
property will be not accessible in DataRefEditor plugin. If \texttt{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. By default \texttt{isNotPublished} and \texttt{isShared} is false.

Array variants of function (\texttt{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.

\texttt{someComponent.lua, creating our project’s datarefs}

\begin{verbatim}
panelBr = 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))
\end{verbatim}

3.2.2.3 \texttt{createFunctionalProperty}

\begin{verbatim}
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)
property p = createFunctionalPropertyia(string name, function getter, function setter, boolean isNotPublished)
property p = createFunctionalPropertyfa(string name, function getter, function setter, boolean isNotPublished)
\end{verbatim}

Creates new functional simulator property of type (\texttt{int, float, double, string or data, int array, float array}) with corresponding \texttt{name}. There is optional boolean parameter \texttt{isNotPublished}. If \texttt{isNotPublished} is true, then such property will be not accessible in DataRefEditor plugin. By default \texttt{isNotPublished} is false.
For **int**, **float**, **double** and **string** properties: **getter** callback is a function without arguments that returns value of property type. **setter** function takes one argument of property type.

```lua
function getterFunction()
    return something
end

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

For **int array** and **float array** properties: **getter** callback is a function which takes 2 arguments and returns value of property type. **setter** function takes 2 arguments.

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

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

Returns reference to created functional property.

**someComponent.lua, functional property creation**

```lua
-- 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

```lua
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

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

### 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

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

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

someComponent.lua, getting components position

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

```lua
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

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

```lua
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

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

#### 3.2.3.2 set

```lua
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 Performance

While it’s not mandatory, using this functional is good for the 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.

3.3.1.1 setAircraftPanelRendering

```lua
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.

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

3.3.1.2 set3DRendering

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

Enables/disables 3D rendering for SASL project. `isOn` defines whether the rendering should be enabled or disabled.

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

3.3.1.3 setInteractivity

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

Enables/disables interactive abilities for SASL project (mouse and keyboard callbacks). `isOn` defines whether the interactivity should be enabled or disabled.

```lua
-- Disabling interactivity
sasl.options.setInteractivity(false)
```
3.3.2 Rendering Customization

3.3.2.1 setRenderingMode2D

```python
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

```python
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 } }
}
```

3.3.2.2 setPanelRenderingMode

```python
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

```lua
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 Debugging

#### 3.3.3.1 `setLuaErrorsHandling`

```lua
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 occurs. 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

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

#### 3.3.3.2 `setLuaStackTraceLimit`

48
| 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)
### 3.4 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

```python
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

```python
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

```python
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**

```lua
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

```lua
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**

```lua
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

```lua
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

```lua
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

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

3.5.2 commandBegin

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

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

`commands.lua`, command execution

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

3.5.3 commandEnd

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

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

`commands.lua`, command execution

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

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

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

**commands.lua, command execution**

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

3.5.5 createCommand

```lua
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**

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

3.5.6 registerCommandHandler

```lua
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.

```lua
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:

<table>
<thead>
<tr>
<th>Type: MenuItemState</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: menu item state identifier, can be equal to one of pre-defined constants:</td>
</tr>
</tbody>
</table>

- 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

```lua
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
Appends new menu item to simulator’s plugins menu.

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

### 3.6.2 appendMenuItemWithCommand (XP11)

```lua
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

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

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

**menus.lua, removing menu item**

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

### 3.6.4 setMenuItemName

```lua
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**

```lua
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
3.6.9 appendMenuSeparator

```lua
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**

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

3.6.10 clearAllMenuItems

```lua
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**

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

3.6.11 destroyMenu

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

Destroys menu, specified by `inMenuID` argument.

**menus.lua, destroying menu**

```lua
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`

```lua
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)  
```

Create 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 equal to 0, then you must provide additional `buttonsCount` pairs of arguments. Every such pair consists from button 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

```lua
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 with 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:

<table>
<thead>
<tr>
<th>Type: CameraStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: camera state identifier, can be equal to one of pre-defined constants:</td>
</tr>
</tbody>
</table>

- 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

define 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

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

Gets current camera status.

3.8.6  startCameraControl

```java
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.

```lua
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

```java
sasl.stopCameraControl()
```

Stops camera controlling.
3.9 Net

3.9.1 downloadFileSync

```lua
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**

```lua
downloadResult, error = sasl.net.downloadFileSync("http://mycoolsite.com/myFile.txt", moduleDirectory.."/myFile.txt")
if not downloadResult then
    sasl.logWarning("Downloading:", error)
end
```

3.9.2 downloadFileContentsSync

```lua
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**

```lua
downloadResult, contents = sasl.net.downloadFileContentsSync("http://mycoolsite.com/myFile.txt")
if downloadResult then
    -- ... process data
else
    sasl.logWarning("Downloading:", contents)
end
3.10 Timers

3.10.1 createTimer

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

Creates new timer object and returns its identifier `id`.

*timers.lua, new timer object creation*

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

3.10.2 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)
```

3.10.3 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)
```

3.10.4 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)
```

### 3.10.5 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)
```

### 3.10.6 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)
```

### 3.10.7 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)
```
3.10.8  getElapsedTimeMicroseconds

```lua
number seconds = sasl.getElapsedTimeMicroseconds(timerID id)
```

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

`timers.lua`, getting current time

```lua
sasl.startTimer(testTimerID)
...
time = sasl.getElapsedTimeMicroseconds(testTimerID)
```

3.10.9  getCurrentCycle

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

Returns overall count of performed updating cycles in simulator.
3.11 Interplugin Communications

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

<table>
<thead>
<tr>
<th>Type</th>
<th>PluginID</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>plugin identifier, can be equal to special pre-defined constant:</td>
</tr>
<tr>
<td>• NO_PLUGIN_ID - means that ID not corresponds to any available plugin.</td>
<td></td>
</tr>
</tbody>
</table>

3.11.1 Interplugin Utilities

3.11.1.1 getMyPluginID

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

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

3.11.1.2 getMyPluginPath

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

Returns full path to SASL project plugin.

3.11.1.3 getXPlanePath

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

Returns full path to simulator folder.

3.11.1.4 getProjectPath

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

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

3.11.1.5 getProjectName

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

Returns name of SASL project.
3.11.1.6  getAircraftPath

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

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

3.11.1.7  getAircraft

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

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

3.11.1.8  countPlugins

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

Returns total number of currently loaded plugins in simulator system.

3.11.1.9  getNthPlugin

```java
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.

3.11.1.10  findPluginByPath

```java
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.

3.11.1.11  findPluginBySignature

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

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

```plaintext
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.

3.11.13 isPluginEnabled

```plaintext
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.

3.11.14 enablePlugin

```plaintext
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.

3.11.15 disablePlugin

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

Disables plugin, specified by `id`.

3.11.16 reloadPlugins

```plaintext
sasl.reloadPlugins()
```

Reloads all plugins in simulator system.

3.11.2 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:

- TYPEUNKNOWN  
- TYPE_INT_ARRAY  
- TYPE_FLOAT_ARRAY  
- TYPE_STRING

### 3.11.2.1 registerMessageHandler

```lua
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):

```lua
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**

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

-- Register simple message handler
sasl.registerMessageHandler(someMessageID, TYPEUNKNOWN, testMessageCallback)
```
3.11.2.2 unregisterMessageHandler

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

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

Example:
```
messages.lua, unregistering message handler
sasl.registerMessageHandler(someMessageID, TYPE_UNKNOWN, testMessageCallback)
...
sasl.unregisterMessageHandler(someMessageID)
```

3.11.2.3 sendMessageToPlugin

```lua
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.

Example:
```
messages.lua, sending messages
sasl.sendMessageToPlugin(NO_PLUGIN_ID, 20222, TYPE_STRING, "Hello!")
sasl.sendMessageToPlugin(somePluginID, 20223, TYPE_UNDEFINED)
```
3.12 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`:

```
<table>
<thead>
<tr>
<th>id:11</th>
<th>id:12</th>
<th>id:13</th>
<th>id:14</th>
<th>id:15</th>
<th>id:16</th>
<th>id:17</th>
<th>id:18</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:31</td>
<td>id:32</td>
<td>id:33</td>
<td>id:34</td>
<td>id:35</td>
<td>id:36</td>
<td>id:37</td>
<td>id:38</td>
</tr>
<tr>
<td>id:41</td>
<td>id:42</td>
<td>id:43</td>
<td>id:44</td>
<td>id:45</td>
<td>id:46</td>
<td>id:47</td>
<td>id:48</td>
</tr>
<tr>
<td>id:51</td>
<td>id:52</td>
<td>id:53</td>
<td>id:54</td>
<td>id:55</td>
<td>id:56</td>
<td>id:57</td>
<td>id:58</td>
</tr>
<tr>
<td>id:61</td>
<td>id:62</td>
<td>id:63</td>
<td>id:64</td>
<td>id:65</td>
<td>id:66</td>
<td>id:67</td>
<td>id:68</td>
</tr>
<tr>
<td>id:71</td>
<td>id:72</td>
<td>id:73</td>
<td>id:74</td>
<td>id:75</td>
<td>id:76</td>
<td>id:77</td>
<td>id:78</td>
</tr>
<tr>
<td>id:81</td>
<td>id:82</td>
<td>id:83</td>
<td>id:84</td>
<td>id:85</td>
<td>id:86</td>
<td>id:87</td>
<td>id:88</td>
</tr>
</tbody>
</table>
```

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

### 3.12.1 setAuxiliaryClickSystem

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

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

### 3.12.2 setCSDClickInterval

```lua
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**

```lua
sasl.setAuxiliaryClickSystem(true)
sasl.setCSDClickInterval(0.4)
```

### 3.12.3 setCSMode

```lua
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.

### 3.12.4 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

```lua
sasl.setAuxiliaryClickSystem(true)
sasl.setCSMode(1)
sasl.setCSShowCursor(15)
```

3.12.5 setCSCursorScale

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.

3.12.6 getCSClickDown

```lua
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

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

3.12.7 getCSClickUp

```lua
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.
3.12.8 getCSClickHold

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

Gets current mouse hold state for mouse button, specified by `MouseButton 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.

3.12.9 getCSDoubleClick

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

Gets current mouse double-click state for mouse button, specified by `MouseButton 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.

3.12.10 getCSWheelClicks

```c
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.

3.12.11 getCSMouseXPos

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

Gets current abscissa coordinate of mouse pointer in simulator window.

3.12.12 getCSMouseYPos

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

Gets current ordinate coordinate of mouse pointer in simulator window.
3.12.13  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.

3.12.14  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.

3.12.15  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 current cursor context

```
sasl.setAuxiliaryClickSystem(true)
...
if (sasl.getCSCursorOnInterface() == 1 and sasl.getCSMouseDown(MB_MIDDLE)) then
  reactSomehow()
end
```
3.13 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.

<table>
<thead>
<tr>
<th>Type: KeyEventType</th>
<th>Description: type of keyboard event, can be equal to one of pre-defined constants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• KB_DOWN_EVENT</td>
<td></td>
</tr>
<tr>
<td>• KB_UP_EVENT</td>
<td></td>
</tr>
<tr>
<td>• KB_HOLD_EVENT</td>
<td></td>
</tr>
</tbody>
</table>

3.13.1 registerGlobalKeyHandler

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

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

```plaintext
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

```plaintext
function keyHandler(charCode, virtualKeyCode, shiftDown, ctrlDown, altOptDown, event)
...
end
sasl.registerGlobalKeyHandler(keyHandler)
```
3.13.2 unregisterGlobalKeyHandler

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

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

3.13.3 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.

3.13.3.1 registerHotKey

```python
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**

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

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

3.13.3.2 unregisterHotKey

```python
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.

3.13.3.3 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)
3.14 Utilities

3.14.1 Operating system

3.14.1.1 getOS

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

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

3.14.1.2 getXPVersion

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

Returns numeric identifier of current X-Plane version.

3.14.1.3 listFiles

```lua
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.

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

```lua
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
```

3.14.1.4 setClipboardText

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

Sets text, specified in `text` into OS clipboard.

**Warning:** this function isn’t available for Linux.
3.14.1.5  getClipboardText

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

Gets current text from OS clipboard.

**Warning:** this function isn’t available for Linux.

3.14.2  Mathematics

3.14.2.1  newInterpolator

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

Creates a stepwise linear interpolator object from given grids \(g_1, g_2, g_n\), 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.

**interpolation.lua**, creating simple 1-dimensional interpolator

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

**interpolation.lua**, creating simple 2-dimensional interpolator

```lua
-- 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 \} \}
local testInterp = newInterpolator(xGrid, yGrid, { ZFunc })
```

3.14.2.2  deleteInterpolator

```lua
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

```lua
local testInterp = newInterpolator({5,20},{{0.025,0.1}})
...
deleteInterpolator(testInterp)
```
3.14.2.3  interpolate (1-dimensional)

```lua
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**

```lua
testInterp = newInterpolator({5 ,20} ,{{0 .025 ,0 .1 }})
-- Prints result - 0.05
print(interpolate(10 , testInterp ))
```

3.14.2.4  interpolate (general)

```lua
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**

```lua
-- 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)
```

3.14.2.5  selfInterpolator

```lua
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**

```lua
testInterp = selfInterpolator({5 ,20} ,{{0 .025 ,0 .1 }})
result = testInterp.interpolate(10)
```
3.14.2.6 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: \(\{ x, y, width, height \}\).

3.14.3 Files And Scripts

3.14.3.1 isFileExists

boolean result = isFileExists(string pathToFile)

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

3.14.3.2 extractFileName

string path = extractFileName(string pathToFile)

Extracts the name of file from specified full \(pathToFile\) and returns it.

3.14.3.3 openFile

function chunk = openFile(string pathToFile)

Loads chunk of Lua code in Lua function, specified by \(pathToFile\) 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.

3.14.3.4 findFile

string fullPath = findFile(string pathToFile)

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.

3.14.3.5 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 code structured.

main.lua, including scripts

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

3.14.4 Miscellaneous

3.14.4.1 toboolean

boolean result = toboolean(number value)

Converts value to true or false value.
3.15 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.

3.15.1 logInfo

```c
sasl.logInfo(string str)
```

Writes string `str` into simulator log and SASL log with level "info".

3.15.2 logWarning

```c
sasl.logWarning(string str)
```

Writes string `str` into simulator log and SASL log with level "warning".

3.15.3 logError

```c
sasl.logError(string str)
```

Writes string `str` into simulator log and SASL log with level "error".

3.15.4 logDebug

```c
sasl.logDebug(string str)
```

Writes string `str` into simulator log and SASL log with level "debug".
3.15.5 logTrace

\[
sasl.logTrace(string\ str)
\]
Same as logDebug.

3.15.6 getLogLevel

\[
LogLevelID\ id = sasl.getLogLevel()
\]
Returns current logging level. Default value is LOG_DEFAULT, so every log message will be shown.

3.15.7 setLogLevel

\[
sasl.setLogLevel(LogLevelID\ id)
\]
Sets current log verbosity level.

*main.lua, setup logging for release build*

```lua
sasl.setLogLevel(LOG_INFO)
...
-- Will be dumped
sasl.logInfo("Info.")
sasl.logWarning("Warning!")
sasl.logError("Error!")
...
-- Will be omitted
sasl.logDebug("Debug!")
```
3.16 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_UNUNKNOWN - 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_MIDDELMARKER - 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.
3.16.1 Navigational Aids

3.16.1.1 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.

3.16.1.2 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.

3.16.1.3 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.

3.16.1.4 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.

3.16.1.5 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.
3.16.1.6  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 \texttt{id}. Last boolean return value \texttt{isInsideLoadedDSFs} is \texttt{true} if this navigation point is lies inside the local area of currently loaded DSFs, and \texttt{false} otherwise. All frequencies except NDB are multiplied by 100.

\texttt{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)
```

3.16.2  FMS

3.16.2.1  countFMSEntries

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

Returns number of entries in FMS.

3.16.2.2  getDisplayedFMSEntry

```
number index = sasl.getDisplayedFMSEntry()
```

Returns index of entry, displayed on FMS.

3.16.2.3  getDestinationFMSEntry

```
number index = sasl.getDestinationFMSEntry()
```

Returns index of entry aircraft flying to.

3.16.2.4  setDisplayedFMSEntry

```
sasl.setDisplayedFMSEntry(number index)
```

Sets displayed FMS entry with specified \texttt{index}.
3.16.2.5 setDestinationFMSEntry

```
sasl.setDestinationFMSEntry(number index)
```
Changes which entry the FMS is flying the aircraft toward. This entry is specified by `index`.

3.16.2.6 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`.

3.16.2.7 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, VORs and NDBs.

3.16.2.8 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.

3.16.2.9 clearFMSEntry

```
sasl.clearFMSEntry(number index)
```
Clears the given entry, specified by `index`, potentially shortening the flight plan.

3.16.3 GPS

3.16.3.1 getGPSDestinationType

```
NavAidType type = sasl.getGPSDestinationType()
```
Returns the type of currently selected GPS destination, one of fix, airport, VOR or NDB.
3.16.3.2 getGPSDestination

```java
NavAidID id = sasl.getGPSDestination();
```

Returns identifier of the navigation point, which is current GPS destination.
3.17 Scenery

3.17.1 Resources Management

3.17.1.1 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**

```lua
carTestObj = sasl.loadObject("misc_objects/car.obj");
busTestObj = sasl.loadObject("misc_objects/bus.obj");
```

3.17.1.2 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 objects asynchronously**

```lua
function carCallback(id)
    print("Loaded!")
end

carTestObj = sasl.loadObjectAsync("misc_objects/car.obj", carCallback);
```

3.17.1.3 unloadObject

```
sasl.unloadObject(number id)
```

Unloads object to decrease memory usage. Object specified by numeric identifier \texttt{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.

\begin{verbatim}
objects.lua, loading objects

\texttt{carTestObj = sasl.loadObject("misc_objects/cor.obj");
...
\texttt{sasl.unloadObject(carTestObj)}
\end{verbatim}

### 3.17.2 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.

#### 3.17.2.1 drawObject

\begin{verbatim}
\texttt{sasl\_drawObject(number id, number x, number y, number z, number pitch, number heading, number roll, number lighting, number earthRelative)}
\end{verbatim}

Draws simulator object, specified by numeric identifier \texttt{id}. To obtain object identifier, use \texttt{loadObject} routine. \texttt{x}, \texttt{y} and \texttt{z} are local OpenGL object coordinates. \texttt{pitch}, \texttt{heading} and \texttt{roll} are object orientation in degrees. If \texttt{lighting} is equal to 1, the night version of object will be drawn with night-only lights lit up. If \texttt{lighting} is equal to 0, the daytime version of object will be drawn.

\texttt{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).
3.17.2.2 reloadScenery

```javascript
sasl.reloadScenery()
```

Reloads simulator scenery files.

3.17.2.3 worldToLocal

```javascript
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.

3.17.2.4 localToWorld

```javascript
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.

3.17.2.5 modelToLocal

```javascript
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.

3.17.2.6 localToModel

```javascript
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.

3.17.2.7 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 \( \text{result} \) is equal to PROBE_HIT_TERRAIN then additional return values is filled. \( \text{locationX} \), \( \text{locationY} \), \( \text{locationZ} \) is location of terrain point hit in local coordinates. \( \text{normalX} \), \( \text{normalY} \), \( \text{normalZ} \) is normal vector of terrain found. \( \text{velocityX} \), \( \text{velocityY} \), \( \text{velocityZ} \) is velocity vector of terrain found. \( \text{isWet} \) equals to 1 if water is found, and equals to 0 otherwise.

### 3.17.3 Magnetic Variation (XP11)

#### 3.17.3.1 getMagneticVariation

```c
number magVar = sasl.getMagneticVariation(number latitude, number longitude)
```

Returns magnetic variation (declination) corresponding to the geographic location, identified by \( \text{latitude} \) and \( \text{longitude} \).

#### 3.17.3.2 degMagneticToDegTrue

```c
number headingTrue = sasl.degMagneticToDegTrue(number heading)
```

Converts a \( \text{heading} \) in degrees relative to magnetic north at the user’s current location into a value relative to true north.

#### 3.17.3.3 degTrueToDegMagnetic

```c
number headingMagnetic = sasl.degTrueToDegMagnetic(number heading)
```

Converts a \( \text{heading} \) in degrees relative to true north into a value relative to magnetic north at the user’s current location.

### 3.17.4 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 \texttt{drawObjects} components callback and using \texttt{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.

3.17.4.1 createInstance

```lua
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

```lua
carTestObj = sasl.loadObject("misc_objects/car.obj");
carTestDatarefs = {
    "myProject/car/first",
    "myProject/car/second",
    "myProject/car/third"
}
testInstance = sasl.createInstance(carTestObj, carTestDatarefs)
```

3.17.4.2 destroyInstance

```lua
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

```lua
sasl.destroyInstance(testInstance)
```

3.17.4.3 setInstancePosition

```lua
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.
3.18 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.

3.18.1 Resources Management

3.18.1.1 loadImage

```lua
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.

images.lua, loading images

```lua
-- 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)
```

3.18.1.2 loadVectorImage

```lua
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 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 SVG images

```lua
-- 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)
```

3.18.1.3 loadImageFromMemory

```lua
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.

3.18.1.4 unloadImage

```lua
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.

--- Unloading images

```lua
testImage1 = sasl.gl.loadImage("images/EGTSign.png")
...
-- Memory is unloaded
sasl.gl.unloadImage(testImage1)
```
images.lua, unloading images (second case)

```lua
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)
```

3.18.1.5 loadBitmapFont

```lua
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.

3.18.1.6 unloadBitmapFont

```lua
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.

3.18.1.7 loadFont

```lua
number id = sasl.gl.loadFont(string fileName)
```

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.

3.18.1.8 unloadFont

```lua
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.
3.18.1.9 loadShader

```c
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.

3.18.2 Color

<table>
<thead>
<tr>
<th>Type:</th>
<th>Color</th>
</tr>
</thead>
</table>

**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.

3.18.3 2D Graphics

3.18.3.1 Primitives

3.18.3.1.1 drawLine

```c
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

3.18.3.1.2 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.

3.18.3.1.3 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

3.18.3.1.4 drawWidePolyLine

sasl.gl.drawWidePolyLine(table points, number thickness, Color color)

Acts like drawPolyLine function, but takes line thickness parameter into account.

3.18.3.1.5 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.
3.18.3.1.6 drawRectangle

```
sasl.gl.drawRectangle(number x, number y, number width, number height, Color color)
```

Draws filled rectangle, specified by \( x \), \( y \), \( \text{width} \) and \( \text{height} \) with specified \( \text{color} \).

3.18.3.1.7 drawFrame

```
sasl.gl.drawFrame(number x, number y, number width, number height, Color color)
```

Draws frame, specified by \( x \), \( y \), \( \text{width} \) and \( \text{height} \) with specified \( \text{color} \).

3.18.3.1.8 setLinePattern

```
sasl.gl.setLinePattern(table pattern)
```

Sets current line pattern, which will be used with \( \text{drawLinePattern} \) function. \( \text{pattern} \) 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
```

3.18.3.1.9 drawLinePattern

```
sasl.gl.drawLinePattern(number x1, number y1, number x2, number y2, boolean savePatternState, Color color)
```

Draws line between \( x_1 \), \( y_1 \) and \( x_2 \), \( y_2 \) points using pattern and specified \( \text{color} \). Use \( \text{setLinePattern} \) to set current line pattern. If \( \text{savePatternState} \) is \( \text{true} \), then current pattern state will be saved across function calling. Set this parameter to \( \text{true} \) 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

3.18.3.1.10 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

3.18.3.1.11 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.

3.18.3.1.12 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.

3.18.3.1.13 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 $x_1$, $y_1$ and $x_3$, $y_3$, and control point $x_2$, $y_2$. 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.

3.18.3.1.14 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.

3.18.3.1.15 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 $x_1$, $y_1$ and $x_4$, $y_4$, and control points $x_2$, $y_2$ and $x_3$, $y_3$. parts parameter defines how many flat lines will be used to draw Bezier line. The last parameter is color.

3.18.3.1.16 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.

3.18.3.1.17 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 $x_1$, $y_1$ and $x_4$, $y_4$, and control points $x_2$, $y_2$ and $x_3$, $y_3$. 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.

3.18.3.1.18 drawWideBezierLineCAdaptive

```
sasl.gl.drawWideBezierLineCAdaptive(number x1, number y1, number x2, number y2, number x3, number y3, number x4, number y4, Color color)
```

106
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.

3.18.3.1.19 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 \( \text{isFilled} \) is true, circle will be filled. The last parameter is \( \text{color} \).

3.18.3.1.20 drawArc

sasl.gl.drawArc(number x, number y, number radiusInner, number radiusOuter, number startAngle, number arcAngle, Color color)

Draws arc with center in \( x, y \). \( \text{radiusInner} \) and \( \text{radiusOuter} \) defines arc form, arc will be drawn between these distances. \( \text{startAngle} \) 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 \( \text{color} \).

3.18.3.1.21 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 \( \text{pointsNumber} \times 2 \). 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 \( \text{color} \).

`testComponent.lua`, drawing custom convex shapes

```lua
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
```

3.18.3.2 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.

3.18.3.2.1 drawTexture

```lua
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

```lua
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
```

3.18.3.2.2 drawRotatedTexture

```lua
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`.

3.18.3.2.3 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.

3.18.3.2.4 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

3.18.3.2.5 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.

3.18.3.2.6 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.
3.18.3.2.7 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`.

3.18.3.2.8 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`.

3.18.3.2.9 getTextureSize

```
number width, number height = sasl.gl.getTextureSize(number id)
```

Returns `width` and `height` of texture, specified by numeric texture handle `id`. Note that this function does not return the size of source image, but size of loaded texture, which is power of 2 in most cases. Use `getTextureSourceSize` function to get actual size of image, which was used to load texture.

3.18.3.2.10 getTextureSourceSize

```
number width, number height = sasl.gl.getTextureSourceSize(number id)
```

Returns `width` and `height` of original image, which was used to create texture, specified by numeric texture handle `id`.

3.18.3.2.11 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 are out of range \([0.0, 1.0]\).
3.18.3.2.12 importTexture

```lua
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**

```lua
testInterfaceTexture = sasl.gl.importTexture(GENERAL_INTERFACE_TEX)
```

**testComponent.lua, importing shared texture**

```lua
sharedTextureDataref = globalPropertyi("some_plugin/shared_texture_id")
...
testTexture = sasl.gl.importTexture(get(sharedTextureDataref))
```

3.18.3.2.13 recreateTexture

```lua
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.

3.18.3.2.14 setRenderTarget

```lua
sasl.gl.setRenderTarget(number id)
sasl.gl.setRenderTarget(number id, boolean isNeedClear)
```

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 `false`. Always call `restoreRenderTarget` function after you are done rendering into texture.

3.18.3.2.15 restoreRenderTarget
3.18.3.2.16 createRenderTarget

```plaintext
number id = sasl.gl.createRenderTarget(number width, number height)
```

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 on when SASL project is about to be
unloaded.

3.18.3.2.17 getTargetTextureData

```plaintext
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).

3.18.3.2.18 createTextureDataStorage

```plaintext
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.

3.18.3.2.19 deleteTextureDataStorage

```plaintext
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.
3.18.3.2.20  getTextureDataPointer

```c
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.

3.18.3.2.21  getRawTextureData

```c
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.

3.18.3.2.22  setRawTextureData

```c
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.

3.18.3.3  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

3.18.3.3.1  drawBitmapText

```c
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

```lua
testFont = sasl.gl.loadBitmapFont("fonts/calibri.fnt")
function draw()
  sasl.gl.drawBitmapText(testFont, 20, 50, TEXT_ALIGN_LEFT, "Hello_SASL")
end
```

### 3.18.3.3.2 drawRotatedBitmapText

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.

### 3.18.3.3.3 drawText

Draws text string at specified position \( x, y \) using font, 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

```lua
testFont = sasl.gl.loadFont("fonts/calibri.ttf")
c1Red = {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, c1Red)
  sasl.gl.drawText(testFont, 20, 80, "Hello_SASL_2", 30, false, true, TEXT_ALIGN_CENTER, c1Red)
end
```

### 3.18.3.3.4 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, 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.

3.18.3.3.5 setFontOutlineThickness

sasl.gl.setFontOutlineThickness(number id, number outlineThickness)

Sets outline thickness for font, specified by numeric font 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.

3.18.3.3.6 setFontOutlineColor

sasl.gl.setFontOutlineColor(number id, Color color)

Sets outline color for font, 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

3.18.3.3.7 measureBitmapText

number length = sasl.gl.measureBitmapText(number id, string text)

Measures text string using bitmap font, specified by numeric identifier id. Returns the horizontal length of string in pixels.
3.18.3.3.8 measureText

```java
number length = sasl.gl.measureText(number id, string text, number size,
   boolean isBold, boolean isItalic)
```

Measures text string using font, specified by numeric identifier id. size parameter defines font size. Use isBold and isItalic parameters to set text options. Returns the horizontal length of string in pixels.

3.18.3.3.9 setRenderTextPixelAligned

```java
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.

3.18.3.4 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.

3.18.3.4.1 createShaderProgram

```java
number id = sasl.gl.createShaderProgram()
```

Creates new empty shader program and returns its numeric identifier id. Use loadShader function to add shaders to program.
### 3.18.3.4.2 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.

### 3.18.3.4.3 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.

### 3.18.3.4.4 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, 0.0, 1.0, 0.0, 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)```
3.18.3.4.5 \texttt{useShaderProgram}

\begin{verbatim}
sasl.gl.useShaderProgram(number id)

Start using shader program, specified by numeric identifier \texttt{id}, for rendering. You must call \texttt{stopShaderProgram} function after you are done rendering with current shader program.
\end{verbatim}

\texttt{testComponent.lua, using shader in draw process}

\begin{verbatim}
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
    ...
end
\end{verbatim}

3.18.3.4.6 \texttt{stopShaderProgram}

\begin{verbatim}
sasl.gl.stopShaderProgram()

Stops shader program usage.
\end{verbatim}

3.18.3.5 \texttt{Blending}

\textbf{Type:} BlendFunctionID

\textbf{Description:} identifier of blending function, can be equal to one of the following pre-defined values:

- \texttt{BLEND\_SOURCE\_COLOR}
- \texttt{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_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.
3.18.3.5.1 setBlendFunction

```c
sasl.gl.setBlendFunction(BlendFunctionID sourceBlend, BlendFunctionID destBlend)
```

Sets current blending functions for source and destination - `sourceBlend` and `destBlend`.

3.18.3.5.2 setBlendFunctionSeparate

```c
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`).

3.18.3.5.3 setBlendEquation

```c
sasl.gl.setBlendEquation(BlendEquationID id)
```

Sets current blending equation, specified by identifier `id`.

3.18.3.5.4 setBlendEquationSeparate

```c
sasl.gl.setBlendEquationSeparate(BlendEquationID equationIDRGB, BlendEquationID equationIDAlpha)
```

Sets current blending equations separately for RGB components and for alpha component - `equationIDRGB` and `equationIDAlpha`.

3.18.3.5.5 setBlendColor

```c
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
3.18.3.5.6  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.

3.18.3.6  Clipping

3.18.3.6.1  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
```

3.18.3.6.2  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.

3.18.3.7  Masking

Next functions provides ability to draw with custom masks shape inside your components. To use this functional inside draw callbacks of your components, `fbo` property must be set to `true`.

3.18.3.7.1  drawMaskStart

```
sasl.gl.drawMaskStart()
```
Enables masking 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.
3.18.3.7.2 drawUnderMask

```javascript
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`.

3.18.3.7.3 drawMaskEnd

```javascript
sasl.gl.drawMaskEnd()
```

Restores drawing context and disables masking.

3.18.3.8 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.

3.18.3.8.1 saveGraphicsContext

```javascript
sasl.gl.saveGraphicsContext()
```

Saves current transformation matrix on the stack. Always call `restoreGraphicsContext` to restore previous transformation matrix.

3.18.3.8.2 restoreGraphicsContext

```javascript
sasl.gl.restoreGraphicsContext()
```

Restores previous transformation matrix.

3.18.3.8.3 setTranslateTransform

```javascript
sasl.gl.setTranslateTransform(number x, number y)
```

Multiplies current transformation matrix on translation matrix, specified by translation coordinates `x, y`.
3.18.3.4 setRotateTransform

```javascript
sasl.gl.setRotateTransform(number angle)
```

Multiplies current transformation matrix on rotation matrix to rotate current context on `angle` degrees.

3.18.3.5 setScaleTransform

```javascript
sasl.gl.setScaleTransform(number scaleX, number scaleY)
```

Multiplies current transformation matrix on scaling matrix, specified by scaling factors `scaleX`, `scaleY`.

3.18.3.9 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.

3.18.3.9.1 isLitStage

```javascript
boolean isLit = sasl.gl.isLitStage()
```

Returns `true` in case if SASL now draws in lit stage, and returns `false` otherwise.

3.18.3.9.2 isNonLitStage

```javascript
boolean isNonLit = sasl.gl.isNonLitStage()
```

Returns `true` in case if SASL now draws in non-lit stage, and returns `false` otherwise.

3.18.3.9.3 isPanelBeforeStage

```javascript
boolean isBeforePanel = sasl.gl.isPanelBeforeStage()
```

Returns `true` in case if SASL now draws before X-Plane, and returns `false` otherwise.

3.18.3.9.4 isPanelAfterStage
boolean isAfterPanel = sasl.gl.isPanelAfterStage()

Returns true in case if SASL now draws after X-Plane, and returns false otherwise.

3.18.4 3D Graphics

3.18.4.1 Primitives

3.18.4.1.1 drawLine3D

sasl.gl.drawLine3D(number x1, number y1, number z1, number x2, number y2, number z2, Color color)

Draws 3D line between \(x_1, y_1, z_1\) and \(x_2, y_2, z_2\) points with specified color in OpenGL local 3D coordinates.

3.18.4.1.2 drawTriangle3D

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 \(x_1, y_1, z_1\) and \(x_2, y_2, z_2\) and \(x_3, y_3, z_3\) points with specified color in OpenGL local 3D coordinates.

3.18.4.1.3 drawCircle3D

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.

3.18.4.1.4 drawAngle3D

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.

3.18.4.1.5 drawStandingCone3D
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.

3.18.4.2 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.

3.18.4.2.1 saveGraphicsContext3D

sasl.gl.saveGraphicsState3D()

Saves current transformation matrix on the stack. Always call restoreGraphicsContext3D to restore previous transformation matrix state.

3.18.4.2.2 restoreGraphicsContext3D

sasl.gl.restoreGraphicsContext3D()

Restores previous transformation matrix state.

3.18.4.2.3 setTranslateTransform3D

sasl.gl.setTranslateTransform3D(number x, number y, number z)

Multiplies current transformation matrix on translation matrix, specified by translation coordinates $x$, $y$, $z$.

3.18.4.2.4 setRotateTransformX3D

sasl.gl.setRotateTransformX3D(number angle)

Multiplies current transformation matrix on rotation matrix around X axis, specified by angle in degrees.

3.18.4.2.5 setRotateTransformY3D
sasl.gl.setRotateTransformY3D(number angle)

Multiplies current transformation matrix on rotation matrix around Y axis, specified by angle in degrees.

3.18.4.2.6 setRotateTransformZ3D

sasl.gl.setRotateTransformZ3D(number angle)

Multiplies current transformation matrix on rotation matrix around Z axis, specified by angle in degrees.

3.18.4.2.7 setRotateTransform3D

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.

3.18.4.2.8 setScaleTransform3D

sasl.gl.setScaleTransform3D(number x, number y, number z)

Multiplies current transformation matrix on scaling matrix, specified by scaling factors x, y and z.
3.19 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).
3.20  Sound

3.20.1  Resources Management

3.20.1.1  loadSample

```lua
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.

```lua
testSound1 = sasl.al.loadSample("mySamples/EngineINN.wav")
testSound2 = sasl.al.loadSample("mySamples/EngineOUT.wav")
```

3.20.1.2  unloadSample

```lua
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.

```lua
testSound1 = sasl.al.loadSample("mySamples/EngineINN.wav")
...
sasl.al.unloadSample(testSound1)
```

3.20.2  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.

3.20.3 Sound playback
3.20.3.1 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.

3.20.3.2 stopSample

```
sasl.al.stopSample(number id)
```

Stops playing sample with specified numeric id.

3.20.3.3 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.

3.20.3.4 rewindSample

```
sasl.al.rewindSample(number id)
```

Sets sample playback position to zero.
3.20.3.5 isSamplePlaying

```java
boolean isPlaying = sasl.al.isSamplePlaying(number id)
```

Returns true if sample, specified by id is playing right now and false otherwise.

3.20.3.6 getSamplePlayingRemaining

```java
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.

3.20.4 Sound settings

3.20.4.1 setSampleGain

```java
sasl.al.setSampleGain(number id, number gain)
```

Sets gain of sample, specified by numeric identifier id. Argument gain must be in [0..1000] range.

3.20.4.2 setMasterGain

```java
sasl.al.setMasterGain(number gain)
```

Adjusts gain of all samples in SASL sound system. Argument gain must be in [0..1000] range.

3.20.4.3 setMinimumSampleGain

```java
sasl.al.setMinimumSampleGain(number id, number minGain)
```

Sets minimum gain of sample, specified by numeric identifier id. Argument minGain must be in [0..1000] range.

3.20.4.4 setMaximumSampleGain

```java
sasl.al.setMaximumSampleGain(number id, number maxGain)
```

Sets maximum gain of sample, specified by numeric identifier id. Argument maxGain must be in [0..1000] range.
3.20.4.5 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).

3.20.4.6 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.

3.20.4.7 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.

3.20.4.8 getSampleDuration

```
number duration = sasl.al.getSampleDuration(number id)
```

Returns total duration of the sample (in seconds), specified with numeric identifier `id`.

3.20.4.9 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.

3.20.4.10 getSamplePosition
number x, number y, number z = sasl.al.getSamplePosition(number id)

Returns 3D position of sample, specified by numeric identifier id.

3.20.4.11 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).

3.20.4.12 getSampleDirection

number x, number y, number z = sasl.al.getSampleDirection(number id)

Returns direction vector of sample, specified by numeric identifier id.

3.20.4.13 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\}.

3.20.4.14 getSampleVelocity

number x, number y, number z = sasl.al.getSampleVelocity(number id)

Returns spatial velocity vector for sample, specified by numeric identifier id.

3.20.4.15 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.

3.20.4.16 getSampleCone

```c
number outerGain, number innerAngle, number outerAngle = sasl.al.getSampleCone(number id)
```

Returns sound cone parameters for sample, specified by numeric identifier `id`.

3.20.4.17 setSampleEnv

```c
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`.

3.20.4.18 getSampleEnv

```c
SoundEnvironment env = sasl.al.getSampleEnv(number id)
```

Returns sound environment value for sample, specified by numeric identifier `id`.

3.20.4.19 setSampleRelative

```c
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.

3.20.4.20 getSampleRelative

```c
number isRelative = sasl.al.getSampleRelative(number id)
```

Returns attachment point identifier for sample, specified by numeric identifier `id`.

3.20.4.21 setSampleMaxDistance
3.20.4.22  setSampleRolloffFactor

```c
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.

3.20.4.23  setSampleRefDistance

```c
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.

![Figure A.1: Console Tab](image)
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:

```c
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).