SCRIPTING API FOR DAVINCI RESOLVE STUDIO

Updated as of 18 July 2023

----------------------------

In this package, you will find a brief introduction to the Scripting API for DaVinci Resolve Studio. Apart from this README.txt file, this package contains folders containing the basic import

modules for scripting access (DaVinciResolve.py) and some representative examples.

From v16.2.0 onwards, the nodeIndex parameters accepted by SetLUT() and SetCDL() are 1-based instead of 0-based, i.e. 1 <= nodeIndex <= total number of nodes.

Overview

--------

As with Blackmagic Design Fusion scripts, user scripts written in Lua and Python programming languages are supported. By default, scripts can be invoked from the Console window in the Fusion page,

or via command line. This permission can be changed in Resolve Preferences, to be only from Console, or to be invoked from the local network. Please be aware of the security implications when

allowing scripting access from outside of the Resolve application.

Prerequisites

-------------

DaVinci Resolve scripting requires one of the following to be installed (for all users):

Lua 5.1

Python 2.7 64-bit

Python >= 3.6 64-bit

Using a script

--------------

DaVinci Resolve needs to be running for a script to be invoked.

For a Resolve script to be executed from an external folder, the script needs to know of the API location.

You may need to set the these environment variables to allow for your Python installation to pick up the appropriate dependencies as shown below:

Mac OS X:

RESOLVE\_SCRIPT\_API="/Library/Application Support/Blackmagic Design/DaVinci Resolve/Developer/Scripting"

RESOLVE\_SCRIPT\_LIB="/Applications/DaVinci Resolve/DaVinci Resolve.app/Contents/Libraries/Fusion/fusionscript.so"

PYTHONPATH="$PYTHONPATH:$RESOLVE\_SCRIPT\_API/Modules/"

Windows:

RESOLVE\_SCRIPT\_API="%PROGRAMDATA%\Blackmagic Design\DaVinci Resolve\Support\Developer\Scripting"

RESOLVE\_SCRIPT\_LIB="C:\Program Files\Blackmagic Design\DaVinci Resolve\fusionscript.dll"

PYTHONPATH="%PYTHONPATH%;%RESOLVE\_SCRIPT\_API%\Modules\"

Linux:

RESOLVE\_SCRIPT\_API="/opt/resolve/Developer/Scripting"

RESOLVE\_SCRIPT\_LIB="/opt/resolve/libs/Fusion/fusionscript.so"

PYTHONPATH="$PYTHONPATH:$RESOLVE\_SCRIPT\_API/Modules/"

(Note: For standard ISO Linux installations, the path above may need to be modified to refer to /home/resolve instead of /opt/resolve)

As with Fusion scripts, Resolve scripts can also be invoked via the menu and the Console.

On startup, DaVinci Resolve scans the subfolders in the directories shown below and enumerates the scripts found in the Workspace application menu under Scripts.

Place your script under Utility to be listed in all pages, under Comp or Tool to be available in the Fusion page or under folders for individual pages (Edit, Color or Deliver). Scripts under Deliver are additionally listed under render jobs.

Placing your script here and invoking it from the menu is the easiest way to use scripts.

Mac OS X:

- All users: /Library/Application Support/Blackmagic Design/DaVinci Resolve/Fusion/Scripts

- Specific user: /Users/<UserName>/Library/Application Support/Blackmagic Design/DaVinci Resolve/Fusion/Scripts

Windows:

- All users: %PROGRAMDATA%\Blackmagic Design\DaVinci Resolve\Fusion\Scripts

- Specific user: %APPDATA%\Roaming\Blackmagic Design\DaVinci Resolve\Support\Fusion\Scripts

Linux:

- All users: /opt/resolve/Fusion/Scripts (or /home/resolve/Fusion/Scripts/ depending on installation)

- Specific user: $HOME/.local/share/DaVinciResolve/Fusion/Scripts

The interactive Console window allows for an easy way to execute simple scripting commands, to query or modify properties, and to test scripts. The console accepts commands in Python 2.7, Python 3.6

and Lua and evaluates and executes them immediately. For more information on how to use the Console, please refer to the DaVinci Resolve User Manual.

This example Python script creates a simple project:

#!/usr/bin/env python

import DaVinciResolveScript as dvr\_script

resolve = dvr\_script.scriptapp("Resolve")

fusion = resolve.Fusion()

projectManager = resolve.GetProjectManager()

projectManager.CreateProject("Hello World")

The resolve object is the fundamental starting point for scripting via Resolve. As a native object, it can be inspected for further scriptable properties - using table iteration and "getmetatable"

in Lua and dir, help etc in Python (among other methods). A notable scriptable object above is fusion - it allows access to all existing Fusion scripting functionality.

Running DaVinci Resolve in headless mode

----------------------------------------

DaVinci Resolve can be launched in a headless mode without the user interface using the -nogui command line option. When DaVinci Resolve is launched using this option, the user interface is disabled.

However, the various scripting APIs will continue to work as expected.

Basic Resolve API

-----------------

Some commonly used API functions are described below (\*). As with the resolve object, each object is inspectable for properties and functions.

Resolve

Fusion() --> Fusion # Returns the Fusion object. Starting point for Fusion scripts.

GetMediaStorage() --> MediaStorage # Returns the media storage object to query and act on media locations.

GetProjectManager() --> ProjectManager # Returns the project manager object for currently open database.

OpenPage(pageName) --> Bool # Switches to indicated page in DaVinci Resolve. Input can be one of ("media", "cut", "edit", "fusion", "color", "fairlight", "deliver").

GetCurrentPage() --> String # Returns the page currently displayed in the main window. Returned value can be one of ("media", "cut", "edit", "fusion", "color", "fairlight", "deliver", None).

GetProductName() --> string # Returns product name.

GetVersion() --> [version fields] # Returns list of product version fields in [major, minor, patch, build, suffix] format.

GetVersionString() --> string # Returns product version in "major.minor.patch[suffix].build" format.

LoadLayoutPreset(presetName) --> Bool # Loads UI layout from saved preset named 'presetName'.

UpdateLayoutPreset(presetName) --> Bool # Overwrites preset named 'presetName' with current UI layout.

ExportLayoutPreset(presetName, presetFilePath) --> Bool # Exports preset named 'presetName' to path 'presetFilePath'.

DeleteLayoutPreset(presetName) --> Bool # Deletes preset named 'presetName'.

SaveLayoutPreset(presetName) --> Bool # Saves current UI layout as a preset named 'presetName'.

ImportLayoutPreset(presetFilePath, presetName) --> Bool # Imports preset from path 'presetFilePath'. The optional argument 'presetName' specifies how the preset shall be named. If not specified, the preset is named based on the filename.

Quit() --> None # Quits the Resolve App.

ImportRenderPreset(presetPath) --> Bool # Import a preset from presetPath (string) and set it as current preset for rendering.

ExportRenderPreset(presetName, exportPath) --> Bool # Export a preset to a given path (string) if presetName(string) exists.

ImportBurnInPreset(presetPath) --> Bool # Import a data burn in preset from a given presetPath (string)

ExportBurnInPreset(presetName, exportPath) --> Bool # Export a data burn in preset to a given path (string) if presetName (string) exists.

ProjectManager

ArchiveProject(projectName,

filePath,

isArchiveSrcMedia=True,

isArchiveRenderCache=True,

isArchiveProxyMedia=False) --> Bool # Archives project to provided file path with the configuration as provided by the optional arguments

CreateProject(projectName) --> Project # Creates and returns a project if projectName (string) is unique, and None if it is not.

DeleteProject(projectName) --> Bool # Delete project in the current folder if not currently loaded

LoadProject(projectName) --> Project # Loads and returns the project with name = projectName (string) if there is a match found, and None if there is no matching Project.

GetCurrentProject() --> Project # Returns the currently loaded Resolve project.

SaveProject() --> Bool # Saves the currently loaded project with its own name. Returns True if successful.

CloseProject(project) --> Bool # Closes the specified project without saving.

CreateFolder(folderName) --> Bool # Creates a folder if folderName (string) is unique.

DeleteFolder(folderName) --> Bool # Deletes the specified folder if it exists. Returns True in case of success.

GetProjectListInCurrentFolder() --> [project names...] # Returns a list of project names in current folder.

GetFolderListInCurrentFolder() --> [folder names...] # Returns a list of folder names in current folder.

GotoRootFolder() --> Bool # Opens root folder in database.

GotoParentFolder() --> Bool # Opens parent folder of current folder in database if current folder has parent.

GetCurrentFolder() --> string # Returns the current folder name.

OpenFolder(folderName) --> Bool # Opens folder under given name.

ImportProject(filePath, projectName=None) --> Bool # Imports a project from the file path provided with given project name, if any. Returns True if successful.

ExportProject(projectName, filePath, withStillsAndLUTs=True) --> Bool # Exports project to provided file path, including stills and LUTs if withStillsAndLUTs is True (enabled by default). Returns True in case of success.

RestoreProject(filePath, projectName=None) --> Bool # Restores a project from the file path provided with given project name, if any. Returns True if successful.

GetCurrentDatabase() --> {dbInfo} # Returns a dictionary (with keys 'DbType', 'DbName' and optional 'IpAddress') corresponding to the current database connection

GetDatabaseList() --> [{dbInfo}] # Returns a list of dictionary items (with keys 'DbType', 'DbName' and optional 'IpAddress') corresponding to all the databases added to Resolve

SetCurrentDatabase({dbInfo}) --> Bool # Switches current database connection to the database specified by the keys below, and closes any open project.

# 'DbType': 'Disk' or 'PostgreSQL' (string)

# 'DbName': database name (string)

# 'IpAddress': IP address of the PostgreSQL server (string, optional key - defaults to '127.0.0.1')

Project

GetMediaPool() --> MediaPool # Returns the Media Pool object.

GetTimelineCount() --> int # Returns the number of timelines currently present in the project.

GetTimelineByIndex(idx) --> Timeline # Returns timeline at the given index, 1 <= idx <= project.GetTimelineCount()

GetCurrentTimeline() --> Timeline # Returns the currently loaded timeline.

SetCurrentTimeline(timeline) --> Bool # Sets given timeline as current timeline for the project. Returns True if successful.

GetGallery() --> Gallery # Returns the Gallery object.

GetName() --> string # Returns project name.

SetName(projectName) --> Bool # Sets project name if given projectName (string) is unique.

GetPresetList() --> [presets...] # Returns a list of presets and their information.

SetPreset(presetName) --> Bool # Sets preset by given presetName (string) into project.

AddRenderJob() --> string # Adds a render job based on current render settings to the render queue. Returns a unique job id (string) for the new render job.

DeleteRenderJob(jobId) --> Bool # Deletes render job for input job id (string).

DeleteAllRenderJobs() --> Bool # Deletes all render jobs in the queue.

GetRenderJobList() --> [render jobs...] # Returns a list of render jobs and their information.

GetRenderPresetList() --> [presets...] # Returns a list of render presets and their information.

StartRendering(jobId1, jobId2, ...) --> Bool # Starts rendering jobs indicated by the input job ids.

StartRendering([jobIds...], isInteractiveMode=False) --> Bool # Starts rendering jobs indicated by the input job ids.

# The optional "isInteractiveMode", when set, enables error feedback in the UI during rendering.

StartRendering(isInteractiveMode=False) --> Bool # Starts rendering all queued render jobs.

# The optional "isInteractiveMode", when set, enables error feedback in the UI during rendering.

StopRendering() --> None # Stops any current render processes.

IsRenderingInProgress() --> Bool # Returns True if rendering is in progress.

LoadRenderPreset(presetName) --> Bool # Sets a preset as current preset for rendering if presetName (string) exists.

SaveAsNewRenderPreset(presetName) --> Bool # Creates new render preset by given name if presetName(string) is unique.

SetRenderSettings({settings}) --> Bool # Sets given settings for rendering. Settings is a dict, with support for the keys:

# Refer to "Looking up render settings" section for information for supported settings

GetRenderJobStatus(jobId) --> {status info} # Returns a dict with job status and completion percentage of the job by given jobId (string).

GetSetting(settingName) --> string # Returns value of project setting (indicated by settingName, string). Check the section below for more information.

SetSetting(settingName, settingValue) --> Bool # Sets the project setting (indicated by settingName, string) to the value (settingValue, string). Check the section below for more information.

GetRenderFormats() --> {render formats..} # Returns a dict (format -> file extension) of available render formats.

GetRenderCodecs(renderFormat) --> {render codecs...} # Returns a dict (codec description -> codec name) of available codecs for given render format (string).

GetCurrentRenderFormatAndCodec() --> {format, codec} # Returns a dict with currently selected format 'format' and render codec 'codec'.

SetCurrentRenderFormatAndCodec(format, codec) --> Bool # Sets given render format (string) and render codec (string) as options for rendering.

GetCurrentRenderMode() --> int # Returns the render mode: 0 - Individual clips, 1 - Single clip.

SetCurrentRenderMode(renderMode) --> Bool # Sets the render mode. Specify renderMode = 0 for Individual clips, 1 for Single clip.

GetRenderResolutions(format, codec) --> [{Resolution}] # Returns list of resolutions applicable for the given render format (string) and render codec (string). Returns full list of resolutions if no argument is provided. Each element in the list is a dictionary with 2 keys "Width" and "Height".

RefreshLUTList() --> Bool # Refreshes LUT List

GetUniqueId() --> string # Returns a unique ID for the project item

InsertAudioToCurrentTrackAtPlayhead(mediaPath, --> Bool # Inserts the media specified by mediaPath (string) with startOffsetInSamples (int) and durationInSamples (int) at the playhead on a selected track on the Fairlight page. Returns True if successful, otherwise False.

startOffsetInSamples, durationInSamples)

LoadBurnInPreset(presetName) --> Bool # Loads user defined data burn in preset for project when supplied presetName (string). Returns true if successful.

ExportCurrentFrameAsStill(filePath) --> Bool # Exports current frame as still to supplied filePath. filePath must end in valid export file format. Returns True if succssful, False otherwise.

MediaStorage

GetMountedVolumeList() --> [paths...] # Returns list of folder paths corresponding to mounted volumes displayed in Resolve’s Media Storage.

GetSubFolderList(folderPath) --> [paths...] # Returns list of folder paths in the given absolute folder path.

GetFileList(folderPath) --> [paths...] # Returns list of media and file listings in the given absolute folder path. Note that media listings may be logically consolidated entries.

RevealInStorage(path) --> Bool # Expands and displays given file/folder path in Resolve’s Media Storage.

AddItemListToMediaPool(item1, item2, ...) --> [clips...] # Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is one or more file/folder paths. Returns a list of the MediaPoolItems created.

AddItemListToMediaPool([items...]) --> [clips...] # Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is an array of file/folder paths. Returns a list of the MediaPoolItems created.

AddItemListToMediaPool([{itemInfo}, ...]) --> [clips...] # Adds list of itemInfos specified as dict of "media", "startFrame" (int), "endFrame" (int) from Media Storage into current Media Pool folder. Returns a list of the MediaPoolItems created.

AddClipMattesToMediaPool(MediaPoolItem, [paths], stereoEye) --> Bool # Adds specified media files as mattes for the specified MediaPoolItem. StereoEye is an optional argument for specifying which eye to add the matte to for stereo clips ("left" or "right"). Returns True if successful.

AddTimelineMattesToMediaPool([paths]) --> [MediaPoolItems] # Adds specified media files as timeline mattes in current media pool folder. Returns a list of created MediaPoolItems.

MediaPool

GetRootFolder() --> Folder # Returns root Folder of Media Pool

AddSubFolder(folder, name) --> Folder # Adds new subfolder under specified Folder object with the given name.

RefreshFolders() --> Bool # Updates the folders in collaboration mode

CreateEmptyTimeline(name) --> Timeline # Adds new timeline with given name.

AppendToTimeline(clip1, clip2, ...) --> [TimelineItem] # Appends specified MediaPoolItem objects in the current timeline. Returns the list of appended timelineItems.

AppendToTimeline([clips]) --> [TimelineItem] # Appends specified MediaPoolItem objects in the current timeline. Returns the list of appended timelineItems.

AppendToTimeline([{clipInfo}, ...]) --> [TimelineItem] # Appends list of clipInfos specified as dict of "mediaPoolItem", "startFrame" (int), "endFrame" (int), (optional) "mediaType" (int; 1 - Video only, 2 - Audio only), "trackIndex" (int) and "recordFrame" (int). Returns the list of appended timelineItems.

CreateTimelineFromClips(name, clip1, clip2,...) --> Timeline # Creates new timeline with specified name, and appends the specified MediaPoolItem objects.

CreateTimelineFromClips(name, [clips]) --> Timeline # Creates new timeline with specified name, and appends the specified MediaPoolItem objects.

CreateTimelineFromClips(name, [{clipInfo}]) --> Timeline # Creates new timeline with specified name, appending the list of clipInfos specified as a dict of "mediaPoolItem", "startFrame" (int), "endFrame" (int), "recordFrame" (int).

ImportTimelineFromFile(filePath, {importOptions}) --> Timeline # Creates timeline based on parameters within given file (AAF/EDL/XML/FCPXML/DRT/ADL) and optional importOptions dict, with support for the keys:

# "timelineName": string, specifies the name of the timeline to be created. Not valid for DRT import

# "importSourceClips": Bool, specifies whether source clips should be imported, True by default. Not valid for DRT import

# "sourceClipsPath": string, specifies a filesystem path to search for source clips if the media is inaccessible in their original path and if "importSourceClips" is True

# "sourceClipsFolders": List of Media Pool folder objects to search for source clips if the media is not present in current folder and if "importSourceClips" is False. Not valid for DRT import

# "interlaceProcessing": Bool, specifies whether to enable interlace processing on the imported timeline being created. valid only for AAF import

DeleteTimelines([timeline]) --> Bool # Deletes specified timelines in the media pool.

GetCurrentFolder() --> Folder # Returns currently selected Folder.

SetCurrentFolder(Folder) --> Bool # Sets current folder by given Folder.

DeleteClips([clips]) --> Bool # Deletes specified clips or timeline mattes in the media pool

ImportFolderFromFile(filePath, sourceClipsPath="") --> Bool # Returns true if import from given DRB filePath is successful, false otherwise

# sourceClipsPath is a string that specifies a filesystem path to search for source clips if the media is inaccessible in their original path, empty by default

DeleteFolders([subfolders]) --> Bool # Deletes specified subfolders in the media pool

MoveClips([clips], targetFolder) --> Bool # Moves specified clips to target folder.

MoveFolders([folders], targetFolder) --> Bool # Moves specified folders to target folder.

GetClipMatteList(MediaPoolItem) --> [paths] # Get mattes for specified MediaPoolItem, as a list of paths to the matte files.

GetTimelineMatteList(Folder) --> [MediaPoolItems] # Get mattes in specified Folder, as list of MediaPoolItems.

DeleteClipMattes(MediaPoolItem, [paths]) --> Bool # Delete mattes based on their file paths, for specified MediaPoolItem. Returns True on success.

RelinkClips([MediaPoolItem], folderPath) --> Bool # Update the folder location of specified media pool clips with the specified folder path.

UnlinkClips([MediaPoolItem]) --> Bool # Unlink specified media pool clips.

ImportMedia([items...]) --> [MediaPoolItems] # Imports specified file/folder paths into current Media Pool folder. Input is an array of file/folder paths. Returns a list of the MediaPoolItems created.

ImportMedia([{clipInfo}]) --> [MediaPoolItems] # Imports file path(s) into current Media Pool folder as specified in list of clipInfo dict. Returns a list of the MediaPoolItems created.

# Each clipInfo gets imported as one MediaPoolItem unless 'Show Individual Frames' is turned on.

# Example: ImportMedia([{"FilePath":"file\_%03d.dpx", "StartIndex":1, "EndIndex":100}]) would import clip "file\_[001-100].dpx".

ExportMetadata(fileName, [clips]) --> Bool # Exports metadata of specified clips to 'fileName' in CSV format.

# If no clips are specified, all clips from media pool will be used.

GetUniqueId() --> string # Returns a unique ID for the media pool

Folder

GetClipList() --> [clips...] # Returns a list of clips (items) within the folder.

GetName() --> string # Returns the media folder name.

GetSubFolderList() --> [folders...] # Returns a list of subfolders in the folder.

GetIsFolderStale() --> bool # Returns true if folder is stale in collaboration mode, false otherwise

GetUniqueId() --> string # Returns a unique ID for the media pool folder

Export(filePath) --> bool # Returns true if export of DRB folder to filePath is successful, false otherwise

MediaPoolItem

GetName() --> string # Returns the clip name.

GetMetadata(metadataType=None) --> string|dict # Returns the metadata value for the key 'metadataType'.

# If no argument is specified, a dict of all set metadata properties is returned.

SetMetadata(metadataType, metadataValue) --> Bool # Sets the given metadata to metadataValue (string). Returns True if successful.

SetMetadata({metadata}) --> Bool # Sets the item metadata with specified 'metadata' dict. Returns True if successful.

GetMediaId() --> string # Returns the unique ID for the MediaPoolItem.

AddMarker(frameId, color, name, note, duration, --> Bool # Creates a new marker at given frameId position and with given marker information. 'customData' is optional and helps to attach user specific data to the marker.

customData)

GetMarkers() --> {markers...} # Returns a dict (frameId -> {information}) of all markers and dicts with their information.

# Example of output format: {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1', 'customData': ''}, ...}

# In the above example - there is one 'Green' marker at offset 96 (position of the marker)

GetMarkerByCustomData(customData) --> {markers...} # Returns marker {information} for the first matching marker with specified customData.

UpdateMarkerCustomData(frameId, customData) --> Bool # Updates customData (string) for the marker at given frameId position. CustomData is not exposed via UI and is useful for scripting developer to attach any user specific data to markers.

GetMarkerCustomData(frameId) --> string # Returns customData string for the marker at given frameId position.

DeleteMarkersByColor(color) --> Bool # Delete all markers of the specified color from the media pool item. "All" as argument deletes all color markers.

DeleteMarkerAtFrame(frameNum) --> Bool # Delete marker at frame number from the media pool item.

DeleteMarkerByCustomData(customData) --> Bool # Delete first matching marker with specified customData.

AddFlag(color) --> Bool # Adds a flag with given color (string).

GetFlagList() --> [colors...] # Returns a list of flag colors assigned to the item.

ClearFlags(color) --> Bool # Clears the flag of the given color if one exists. An "All" argument is supported and clears all flags.

GetClipColor() --> string # Returns the item color as a string.

SetClipColor(colorName) --> Bool # Sets the item color based on the colorName (string).

ClearClipColor() --> Bool # Clears the item color.

GetClipProperty(propertyName=None) --> string|dict # Returns the property value for the key 'propertyName'.

# If no argument is specified, a dict of all clip properties is returned. Check the section below for more information.

SetClipProperty(propertyName, propertyValue) --> Bool # Sets the given property to propertyValue (string). Check the section below for more information.

LinkProxyMedia(proxyMediaFilePath) --> Bool # Links proxy media located at path specified by arg 'proxyMediaFilePath' with the current clip. 'proxyMediaFilePath' should be absolute clip path.

UnlinkProxyMedia() --> Bool # Unlinks any proxy media associated with clip.

ReplaceClip(filePath) --> Bool # Replaces the underlying asset and metadata of MediaPoolItem with the specified absolute clip path.

GetUniqueId() --> string # Returns a unique ID for the media pool item

TranscribeAudio() --> Bool # Transcribes audio of the MediaPoolItem. Returns True if successful; False otherwise

ClearTranscription() --> Bool # Clears audio transcription of the MediaPoolItem. Returns True if successful; False otherwise.

Timeline

GetName() --> string # Returns the timeline name.

SetName(timelineName) --> Bool # Sets the timeline name if timelineName (string) is unique. Returns True if successful.

GetStartFrame() --> int # Returns the frame number at the start of timeline.

GetEndFrame() --> int # Returns the frame number at the end of timeline.

SetStartTimecode(timecode) --> Bool # Set the start timecode of the timeline to the string 'timecode'. Returns true when the change is successful, false otherwise.

GetStartTimecode() --> string # Returns the start timecode for the timeline.

GetTrackCount(trackType) --> int # Returns the number of tracks for the given track type ("audio", "video" or "subtitle").

AddTrack(trackType, optionalSubTrackType) --> Bool # Adds track of trackType ("video", "subtitle", "audio"). Second argument optionalSubTrackType is required for "audio"

# optionalSubTrackType can be one of {"mono", "stereo", "5.1", "5.1film", "7.1", "7.1film", "adaptive1", ... , "adaptive24"}

DeleteTrack(trackType, trackIndex) --> Bool # Deletes track of trackType ("video", "subtitle", "audio") and given trackIndex. 1 <= trackIndex <= GetTrackCount(trackType).

SetTrackEnable(trackType, trackIndex, Bool) --> Bool # Enables/Disables track with given trackType and trackIndex

# trackType is one of {"audio", "video", "subtitle"}

# 1 <= trackIndex <= GetTrackCount(trackType).

GetIsTrackEnabled(trackType, trackIndex) --> Bool # Returns True if track with given trackType and trackIndex is enabled and False otherwise.

# trackType is one of {"audio", "video", "subtitle"}

# 1 <= trackIndex <= GetTrackCount(trackType).

SetTrackLock(trackType, trackIndex, Bool) --> Bool # Locks/Unlocks track with given trackType and trackIndex

# trackType is one of {"audio", "video", "subtitle"}

# 1 <= trackIndex <= GetTrackCount(trackType).

GetIsTrackLocked(trackType, trackIndex) --> Bool # Returns True if track with given trackType and trackIndex is locked and False otherwise.

# trackType is one of {"audio", "video", "subtitle"}

# 1 <= trackIndex <= GetTrackCount(trackType).

DeleteClips([timelineItems], Bool) --> Bool # Deletes specified TimelineItems from the timeline, performing ripple delete if the second argument is True. Second argument is optional (The default for this is False)

SetClipsLinked([timelineItems], Bool) --> Bool # Links or unlinks the specified TimelineItems depending on second argument.

GetItemListInTrack(trackType, index) --> [items...] # Returns a list of timeline items on that track (based on trackType and index). 1 <= index <= GetTrackCount(trackType).

AddMarker(frameId, color, name, note, duration, --> Bool # Creates a new marker at given frameId position and with given marker information. 'customData' is optional and helps to attach user specific data to the marker.

customData)

GetMarkers() --> {markers...} # Returns a dict (frameId -> {information}) of all markers and dicts with their information.

# Example: a value of {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1', 'customData': ''}, ...} indicates a single green marker at timeline offset 96

GetMarkerByCustomData(customData) --> {markers...} # Returns marker {information} for the first matching marker with specified customData.

UpdateMarkerCustomData(frameId, customData) --> Bool # Updates customData (string) for the marker at given frameId position. CustomData is not exposed via UI and is useful for scripting developer to attach any user specific data to markers.

GetMarkerCustomData(frameId) --> string # Returns customData string for the marker at given frameId position.

DeleteMarkersByColor(color) --> Bool # Deletes all timeline markers of the specified color. An "All" argument is supported and deletes all timeline markers.

DeleteMarkerAtFrame(frameNum) --> Bool # Deletes the timeline marker at the given frame number.

DeleteMarkerByCustomData(customData) --> Bool # Delete first matching marker with specified customData.

ApplyGradeFromDRX(path, gradeMode, item1, item2, ...)--> Bool # Loads a still from given file path (string) and applies grade to Timeline Items with gradeMode (int): 0 - "No keyframes", 1 - "Source Timecode aligned", 2 - "Start Frames aligned".

ApplyGradeFromDRX(path, gradeMode, [items]) --> Bool # Loads a still from given file path (string) and applies grade to Timeline Items with gradeMode (int): 0 - "No keyframes", 1 - "Source Timecode aligned", 2 - "Start Frames aligned".

GetCurrentTimecode() --> string # Returns a string timecode representation for the current playhead position, while on Cut, Edit, Color, Fairlight and Deliver pages.

SetCurrentTimecode(timecode) --> Bool # Sets current playhead position from input timecode for Cut, Edit, Color, Fairlight and Deliver pages.

GetCurrentVideoItem() --> item # Returns the current video timeline item.

GetCurrentClipThumbnailImage() --> {thumbnailData} # Returns a dict (keys "width", "height", "format" and "data") with data containing raw thumbnail image data (RGB 8-bit image data encoded in base64 format) for current media in the Color Page.

# An example of how to retrieve and interpret thumbnails is provided in 6\_get\_current\_media\_thumbnail.py in the Examples folder.

GetTrackName(trackType, trackIndex) --> string # Returns the track name for track indicated by trackType ("audio", "video" or "subtitle") and index. 1 <= trackIndex <= GetTrackCount(trackType).

SetTrackName(trackType, trackIndex, name) --> Bool # Sets the track name (string) for track indicated by trackType ("audio", "video" or "subtitle") and index. 1 <= trackIndex <= GetTrackCount(trackType).

DuplicateTimeline(timelineName) --> timeline # Duplicates the timeline and returns the created timeline, with the (optional) timelineName, on success.

CreateCompoundClip([timelineItems], {clipInfo}) --> timelineItem # Creates a compound clip of input timeline items with an optional clipInfo map: {"startTimecode" : "00:00:00:00", "name" : "Compound Clip 1"}. It returns the created timeline item.

CreateFusionClip([timelineItems]) --> timelineItem # Creates a Fusion clip of input timeline items. It returns the created timeline item.

ImportIntoTimeline(filePath, {importOptions}) --> Bool # Imports timeline items from an AAF file and optional importOptions dict into the timeline, with support for the keys:

# "autoImportSourceClipsIntoMediaPool": Bool, specifies if source clips should be imported into media pool, True by default

# "ignoreFileExtensionsWhenMatching": Bool, specifies if file extensions should be ignored when matching, False by default

# "linkToSourceCameraFiles": Bool, specifies if link to source camera files should be enabled, False by default

# "useSizingInfo": Bool, specifies if sizing information should be used, False by default

# "importMultiChannelAudioTracksAsLinkedGroups": Bool, specifies if multi-channel audio tracks should be imported as linked groups, False by default

# "insertAdditionalTracks": Bool, specifies if additional tracks should be inserted, True by default

# "insertWithOffset": string, specifies insert with offset value in timecode format - defaults to "00:00:00:00", applicable if "insertAdditionalTracks" is False

# "sourceClipsPath": string, specifies a filesystem path to search for source clips if the media is inaccessible in their original path and if "ignoreFileExtensionsWhenMatching" is True

# "sourceClipsFolders": string, list of Media Pool folder objects to search for source clips if the media is not present in current folder

Export(fileName, exportType, exportSubtype) --> Bool # Exports timeline to 'fileName' as per input exportType & exportSubtype format.

# Refer to section "Looking up timeline export properties" for information on the parameters.

GetSetting(settingName) --> string # Returns value of timeline setting (indicated by settingName : string). Check the section below for more information.

SetSetting(settingName, settingValue) --> Bool # Sets timeline setting (indicated by settingName : string) to the value (settingValue : string). Check the section below for more information.

InsertGeneratorIntoTimeline(generatorName) --> TimelineItem # Inserts a generator (indicated by generatorName : string) into the timeline.

InsertFusionGeneratorIntoTimeline(generatorName) --> TimelineItem # Inserts a Fusion generator (indicated by generatorName : string) into the timeline.

InsertFusionCompositionIntoTimeline() --> TimelineItem # Inserts a Fusion composition into the timeline.

InsertOFXGeneratorIntoTimeline(generatorName) --> TimelineItem # Inserts an OFX generator (indicated by generatorName : string) into the timeline.

InsertTitleIntoTimeline(titleName) --> TimelineItem # Inserts a title (indicated by titleName : string) into the timeline.

InsertFusionTitleIntoTimeline(titleName) --> TimelineItem # Inserts a Fusion title (indicated by titleName : string) into the timeline.

GrabStill() --> galleryStill # Grabs still from the current video clip. Returns a GalleryStill object.

GrabAllStills(stillFrameSource) --> [galleryStill] # Grabs stills from all the clips of the timeline at 'stillFrameSource' (1 - First frame, 2 - Middle frame). Returns the list of GalleryStill objects.

GetUniqueId() --> string # Returns a unique ID for the timeline

CreateSubtitlesFromAudio() --> Bool # Creates subtitles from audio for the timeline. Returns True on success, False otherwise.

DetectSceneCuts() --> Bool # Detects and makes scene cuts along the timeline. Returns True if successful, False otherwise.

TimelineItem

GetName() --> string # Returns the item name.

GetDuration() --> int # Returns the item duration.

GetEnd() --> int # Returns the end frame position on the timeline.

GetFusionCompCount() --> int # Returns number of Fusion compositions associated with the timeline item.

GetFusionCompByIndex(compIndex) --> fusionComp # Returns the Fusion composition object based on given index. 1 <= compIndex <= timelineItem.GetFusionCompCount()

GetFusionCompNameList() --> [names...] # Returns a list of Fusion composition names associated with the timeline item.

GetFusionCompByName(compName) --> fusionComp # Returns the Fusion composition object based on given name.

GetLeftOffset() --> int # Returns the maximum extension by frame for clip from left side.

GetRightOffset() --> int # Returns the maximum extension by frame for clip from right side.

GetStart() --> int # Returns the start frame position on the timeline.

SetProperty(propertyKey, propertyValue) --> Bool # Sets the value of property "propertyKey" to value "propertyValue"

# Refer to "Looking up Timeline item properties" for more information

GetProperty(propertyKey) --> int/[key:value] # returns the value of the specified key

# if no key is specified, the method returns a dictionary(python) or table(lua) for all supported keys

AddMarker(frameId, color, name, note, duration, --> Bool # Creates a new marker at given frameId position and with given marker information. 'customData' is optional and helps to attach user specific data to the marker.

customData)

GetMarkers() --> {markers...} # Returns a dict (frameId -> {information}) of all markers and dicts with their information.

# Example: a value of {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1', 'customData': ''}, ...} indicates a single green marker at clip offset 96

GetMarkerByCustomData(customData) --> {markers...} # Returns marker {information} for the first matching marker with specified customData.

UpdateMarkerCustomData(frameId, customData) --> Bool # Updates customData (string) for the marker at given frameId position. CustomData is not exposed via UI and is useful for scripting developer to attach any user specific data to markers.

GetMarkerCustomData(frameId) --> string # Returns customData string for the marker at given frameId position.

DeleteMarkersByColor(color) --> Bool # Delete all markers of the specified color from the timeline item. "All" as argument deletes all color markers.

DeleteMarkerAtFrame(frameNum) --> Bool # Delete marker at frame number from the timeline item.

DeleteMarkerByCustomData(customData) --> Bool # Delete first matching marker with specified customData.

AddFlag(color) --> Bool # Adds a flag with given color (string).

GetFlagList() --> [colors...] # Returns a list of flag colors assigned to the item.

ClearFlags(color) --> Bool # Clear flags of the specified color. An "All" argument is supported to clear all flags.

GetClipColor() --> string # Returns the item color as a string.

SetClipColor(colorName) --> Bool # Sets the item color based on the colorName (string).

ClearClipColor() --> Bool # Clears the item color.

AddFusionComp() --> fusionComp # Adds a new Fusion composition associated with the timeline item.

ImportFusionComp(path) --> fusionComp # Imports a Fusion composition from given file path by creating and adding a new composition for the item.

ExportFusionComp(path, compIndex) --> Bool # Exports the Fusion composition based on given index to the path provided.

DeleteFusionCompByName(compName) --> Bool # Deletes the named Fusion composition.

LoadFusionCompByName(compName) --> fusionComp # Loads the named Fusion composition as the active composition.

RenameFusionCompByName(oldName, newName) --> Bool # Renames the Fusion composition identified by oldName.

AddVersion(versionName, versionType) --> Bool # Adds a new color version for a video clip based on versionType (0 - local, 1 - remote).

GetCurrentVersion() --> {versionName...} # Returns the current version of the video clip. The returned value will have the keys versionName and versionType(0 - local, 1 - remote).

DeleteVersionByName(versionName, versionType) --> Bool # Deletes a color version by name and versionType (0 - local, 1 - remote).

LoadVersionByName(versionName, versionType) --> Bool # Loads a named color version as the active version. versionType: 0 - local, 1 - remote.

RenameVersionByName(oldName, newName, versionType)--> Bool # Renames the color version identified by oldName and versionType (0 - local, 1 - remote).

GetVersionNameList(versionType) --> [names...] # Returns a list of all color versions for the given versionType (0 - local, 1 - remote).

GetMediaPoolItem() --> MediaPoolItem # Returns the media pool item corresponding to the timeline item if one exists.

GetStereoConvergenceValues() --> {keyframes...} # Returns a dict (offset -> value) of keyframe offsets and respective convergence values.

GetStereoLeftFloatingWindowParams() --> {keyframes...} # For the LEFT eye -> returns a dict (offset -> dict) of keyframe offsets and respective floating window params. Value at particular offset includes the left, right, top and bottom floating window values.

GetStereoRightFloatingWindowParams() --> {keyframes...} # For the RIGHT eye -> returns a dict (offset -> dict) of keyframe offsets and respective floating window params. Value at particular offset includes the left, right, top and bottom floating window values.

GetNumNodes() --> int # Returns the number of nodes in the current graph for the timeline item

ApplyArriCdlLut() --> Bool # Applies ARRI CDL and LUT. Returns True if successful, False otherwise.

SetLUT(nodeIndex, lutPath) --> Bool # Sets LUT on the node mapping the node index provided, 1 <= nodeIndex <= total number of nodes.

# The lutPath can be an absolute path, or a relative path (based off custom LUT paths or the master LUT path).

# The operation is successful for valid lut paths that Resolve has already discovered (see Project.RefreshLUTList).

GetLUT(nodeIndex) --> String # Gets relative LUT path based on the node index provided, 1 <= nodeIndex <= total number of nodes.

SetCDL([CDL map]) --> Bool # Keys of map are: "NodeIndex", "Slope", "Offset", "Power", "Saturation", where 1 <= NodeIndex <= total number of nodes.

# Example python code - SetCDL({"NodeIndex" : "1", "Slope" : "0.5 0.4 0.2", "Offset" : "0.4 0.3 0.2", "Power" : "0.6 0.7 0.8", "Saturation" : "0.65"})

AddTake(mediaPoolItem, startFrame, endFrame) --> Bool # Adds mediaPoolItem as a new take. Initializes a take selector for the timeline item if needed. By default, the full clip extents is added. startFrame (int) and endFrame (int) are optional arguments used to specify the extents.

GetSelectedTakeIndex() --> int # Returns the index of the currently selected take, or 0 if the clip is not a take selector.

GetTakesCount() --> int # Returns the number of takes in take selector, or 0 if the clip is not a take selector.

GetTakeByIndex(idx) --> {takeInfo...} # Returns a dict (keys "startFrame", "endFrame" and "mediaPoolItem") with take info for specified index.

DeleteTakeByIndex(idx) --> Bool # Deletes a take by index, 1 <= idx <= number of takes.

SelectTakeByIndex(idx) --> Bool # Selects a take by index, 1 <= idx <= number of takes.

FinalizeTake() --> Bool # Finalizes take selection.

CopyGrades([tgtTimelineItems]) --> Bool # Copies the current grade to all the items in tgtTimelineItems list. Returns True on success and False if any error occurred.

SetClipEnabled(Bool) --> Bool # Sets clip enabled based on argument.

GetClipEnabled() --> Bool # Gets clip enabled status.

UpdateSidecar() --> Bool # Updates sidecar file for BRAW clips or RMD file for R3D clips.

GetUniqueId() --> string # Returns a unique ID for the timeline item

LoadBurnInPreset(presetName) --> Bool # Loads user defined data burn in preset for clip when supplied presetName (string). Returns true if successful.

GetNodeLabel(nodeIndex) --> string # Returns the label of the node at nodeIndex.

CreateMagicMask(mode) --> Bool # Returns True if magic mask was created successfully, False otherwise. mode can "F" (forward), "B" (backward), or "BI" (bidirection)

RegenerateMagicMask() --> Bool # Returns True if magic mask was regenerated successfully, False otherwise.

Stabilize() --> Bool # Returns True if stabilization was successful, False otherwise

SmartReframe() --> Bool # Performs Smart Reframe. Returns True if successful, False otherwise.

Gallery

GetAlbumName(galleryStillAlbum) --> string # Returns the name of the GalleryStillAlbum object 'galleryStillAlbum'.

SetAlbumName(galleryStillAlbum, albumName) --> Bool # Sets the name of the GalleryStillAlbum object 'galleryStillAlbum' to 'albumName'.

GetCurrentStillAlbum() --> galleryStillAlbum # Returns current album as a GalleryStillAlbum object.

SetCurrentStillAlbum(galleryStillAlbum) --> Bool # Sets current album to GalleryStillAlbum object 'galleryStillAlbum'.

GetGalleryStillAlbums() --> [galleryStillAlbum] # Returns the gallery albums as a list of GalleryStillAlbum objects.

GalleryStillAlbum

GetStills() --> [galleryStill] # Returns the list of GalleryStill objects in the album.

GetLabel(galleryStill) --> string # Returns the label of the galleryStill.

SetLabel(galleryStill, label) --> Bool # Sets the new 'label' to GalleryStill object 'galleryStill'.

ImportStills([filePaths]) --> Bool # Imports GalleryStill from each filePath in [filePaths] list. True if at least one still is imported successfully. False otherwise.

ExportStills([galleryStill], folderPath, filePrefix, format) --> Bool # Exports list of GalleryStill objects '[galleryStill]' to directory 'folderPath', with filename prefix 'filePrefix', using file format 'format' (supported formats: dpx, cin, tif, jpg, png, ppm, bmp, xpm, drx).

DeleteStills([galleryStill]) --> Bool # Deletes specified list of GalleryStill objects '[galleryStill]'.

GalleryStill # This class does not provide any API functions but the object type is used by functions in other classes.

List and Dict Data Structures

-----------------------------

Beside primitive data types, Resolve's Python API mainly uses list and dict data structures. Lists are denoted by [ ... ] and dicts are denoted by { ... } above.

As Lua does not support list and dict data structures, the Lua API implements "list" as a table with indices, e.g. { [1] = listValue1, [2] = listValue2, ... }.

Similarly the Lua API implements "dict" as a table with the dictionary key as first element, e.g. { [dictKey1] = dictValue1, [dictKey2] = dictValue2, ... }.

Looking up Project and Clip properties

--------------------------------------

This section covers additional notes for the functions "Project:GetSetting", "Project:SetSetting", "Timeline:GetSetting", "Timeline:SetSetting", "MediaPoolItem:GetClipProperty" and

"MediaPoolItem:SetClipProperty". These functions are used to get and set properties otherwise available to the user through the Project Settings and the Clip Attributes dialogs.

The functions follow a key-value pair format, where each property is identified by a key (the settingName or propertyName parameter) and possesses a value (typically a text value). Keys and values are

designed to be easily correlated with parameter names and values in the Resolve UI. Explicitly enumerated values for some parameters are listed below.

Some properties may be read only - these include intrinsic clip properties like date created or sample rate, and properties that can be disabled in specific application contexts (e.g. custom colorspaces

in an ACES workflow, or output sizing parameters when behavior is set to match timeline)

Getting values:

Invoke "Project:GetSetting", "Timeline:GetSetting" or "MediaPoolItem:GetClipProperty" with the appropriate property key. To get a snapshot of all queryable properties (keys and values), you can call

"Project:GetSetting", "Timeline:GetSetting" or "MediaPoolItem:GetClipProperty" without parameters (or with a NoneType or a blank property key). Using specific keys to query individual properties will

be faster. Note that getting a property using an invalid key will return a trivial result.

Setting values:

Invoke "Project:SetSetting", "Timeline:SetSetting" or "MediaPoolItem:SetClipProperty" with the appropriate property key and a valid value. When setting a parameter, please check the return value to

ensure the success of the operation. You can troubleshoot the validity of keys and values by setting the desired result from the UI and checking property snapshots before and after the change.

The following Project properties have specifically enumerated values:

"superScale" - the property value is an enumerated integer between 0 and 4 with these meanings: 0=Auto, 1=no scaling, and 2, 3 and 4 represent the Super Scale multipliers 2x, 3x and 4x.

for super scale multiplier '2x Enhanced', exactly 4 arguments must be passed as outlined below. If less than 4 arguments are passed, it will default to 2x.

Affects:

• x = Project:GetSetting('superScale') and Project:SetSetting('superScale', x)

• for '2x Enhanced' --> Project:SetSetting('superScale', 2, sharpnessValue, noiseReductionValue), where sharpnessValue is a float in the range [0.0, 1.0] and noiseReductionValue is a float in the range [0.0, 1.0]

"timelineFrameRate" - the property value is one of the frame rates available to the user in project settings under "Timeline frame rate" option. Drop Frame can be configured for supported frame rates

by appending the frame rate with "DF", e.g. "29.97 DF" will enable drop frame and "29.97" will disable drop frame

Affects:

• x = Project:GetSetting('timelineFrameRate') and Project:SetSetting('timelineFrameRate', x)

The following Clip properties have specifically enumerated values:

"Super Scale" - the property value is an enumerated integer between 1 and 4 with these meanings: 1=no scaling, and 2, 3 and 4 represent the Super Scale multipliers 2x, 3x and 4x.

for super scale multiplier '2x Enhanced', exactly 4 arguments must be passed as outlined below. If less than 4 arguments are passed, it will default to 2x.

Affects:

• x = MediaPoolItem:GetClipProperty('Super Scale') and MediaPoolItem:SetClipProperty('Super Scale', x)

• for '2x Enhanced' --> MediaPoolItem:SetClipProperty('Super Scale', 2, sharpnessValue, noiseReductionValue), where sharpnessValue is a float in the range [0.0, 1.0] and noiseReductionValue is a float in the range [0.0, 1.0]

Looking up Render Settings

--------------------------

This section covers the supported settings for the method SetRenderSettings({settings})

The parameter setting is a dictionary containing the following keys:

- "SelectAllFrames": Bool (when set True, the settings MarkIn and MarkOut are ignored)

- "MarkIn": int

- "MarkOut": int

- "TargetDir": string

- "CustomName": string

- "UniqueFilenameStyle": 0 - Prefix, 1 - Suffix.

- "ExportVideo": Bool

- "ExportAudio": Bool

- "FormatWidth": int

- "FormatHeight": int

- "FrameRate": float (examples: 23.976, 24)

- "PixelAspectRatio": string (for SD resolution: "16\_9" or "4\_3") (other resolutions: "square" or "cinemascope")

- "VideoQuality" possible values for current codec (if applicable):

- 0 (int) - will set quality to automatic

- [1 -> MAX] (int) - will set input bit rate

- ["Least", "Low", "Medium", "High", "Best"] (String) - will set input quality level

- "AudioCodec": string (example: "aac")

- "AudioBitDepth": int

- "AudioSampleRate": int

- "ColorSpaceTag" : string (example: "Same as Project", "AstroDesign")

- "GammaTag" : string (example: "Same as Project", "ACEScct")

- "ExportAlpha": Bool

- "EncodingProfile": string (example: "Main10"). Can only be set for H.264 and H.265.

- "MultiPassEncode": Bool. Can only be set for H.264.

- "AlphaMode": 0 - Premultiplied, 1 - Straight. Can only be set if "ExportAlpha" is true.

- "NetworkOptimization": Bool. Only supported by QuickTime and MP4 formats.

Looking up timeline export properties

-------------------------------------

This section covers the parameters for the argument Export(fileName, exportType, exportSubtype).

exportType can be one of the following constants:

- resolve.EXPORT\_AAF

- resolve.EXPORT\_DRT

- resolve.EXPORT\_EDL

- resolve.EXPORT\_FCP\_7\_XML

- resolve.EXPORT\_FCPXML\_1\_8

- resolve.EXPORT\_FCPXML\_1\_9

- resolve.EXPORT\_FCPXML\_1\_10

- resolve.EXPORT\_HDR\_10\_PROFILE\_A

- resolve.EXPORT\_HDR\_10\_PROFILE\_B

- resolve.EXPORT\_TEXT\_CSV

- resolve.EXPORT\_TEXT\_TAB

- resolve.EXPORT\_DOLBY\_VISION\_VER\_2\_9

- resolve.EXPORT\_DOLBY\_VISION\_VER\_4\_0

- resolve.EXPORT\_DOLBY\_VISION\_VER\_5\_1

- resolve.EXPORT\_OTIO

exportSubtype can be one of the following enums:

- resolve.EXPORT\_NONE

- resolve.EXPORT\_AAF\_NEW

- resolve.EXPORT\_AAF\_EXISTING

- resolve.EXPORT\_CDL

- resolve.EXPORT\_SDL

- resolve.EXPORT\_MISSING\_CLIPS

Please note that exportSubType is a required parameter for resolve.EXPORT\_AAF and resolve.EXPORT\_EDL. For rest of the exportType, exportSubtype is ignored.

When exportType is resolve.EXPORT\_AAF, valid exportSubtype values are resolve.EXPORT\_AAF\_NEW and resolve.EXPORT\_AAF\_EXISTING.

When exportType is resolve.EXPORT\_EDL, valid exportSubtype values are resolve.EXPORT\_CDL, resolve.EXPORT\_SDL, resolve.EXPORT\_MISSING\_CLIPS and resolve.EXPORT\_NONE.

Note: Replace 'resolve.' when using the constants above, if a different Resolve class instance name is used.

Unsupported exportType types

---------------------------------

Starting with DaVinci Resolve 18.1, the following export types are not supported:

- resolve.EXPORT\_FCPXML\_1\_3

- resolve.EXPORT\_FCPXML\_1\_4

- resolve.EXPORT\_FCPXML\_1\_5

- resolve.EXPORT\_FCPXML\_1\_6

- resolve.EXPORT\_FCPXML\_1\_7

Looking up Timeline item properties

-----------------------------------

This section covers additional notes for the function "TimelineItem:SetProperty" and "TimelineItem:GetProperty". These functions are used to get and set properties mentioned.

The supported keys with their accepted values are:

"Pan" : floating point values from -4.0\*width to 4.0\*width

"Tilt" : floating point values from -4.0\*height to 4.0\*height

"ZoomX" : floating point values from 0.0 to 100.0

"ZoomY" : floating point values from 0.0 to 100.0

"ZoomGang" : a boolean value

"RotationAngle" : floating point values from -360.0 to 360.0

"AnchorPointX" : floating point values from -4.0\*width to 4.0\*width

"AnchorPointY" : floating point values from -4.0\*height to 4.0\*height

"Pitch" : floating point values from -1.5 to 1.5

"Yaw" : floating point values from -1.5 to 1.5

"FlipX" : boolean value for flipping horizontally

"FlipY" : boolean value for flipping vertically

"CropLeft" : floating point values from 0.0 to width

"CropRight" : floating point values from 0.0 to width

"CropTop" : floating point values from 0.0 to height

"CropBottom" : floating point values from 0.0 to height

"CropSoftness" : floating point values from -100.0 to 100.0

"CropRetain" : boolean value for "Retain Image Position" checkbox

"DynamicZoomEase" : A value from the following constants

- DYNAMIC\_ZOOM\_EASE\_LINEAR = 0

- DYNAMIC\_ZOOM\_EASE\_IN

- DYNAMIC\_ZOOM\_EASE\_OUT

- DYNAMIC\_ZOOM\_EASE\_IN\_AND\_OUT

"CompositeMode" : A value from the following constants

- COMPOSITE\_NORMAL = 0

- COMPOSITE\_ADD

- COMPOSITE\_SUBTRACT

- COMPOSITE\_DIFF

- COMPOSITE\_MULTIPLY

- COMPOSITE\_SCREEN

- COMPOSITE\_OVERLAY

- COMPOSITE\_HARDLIGHT

- COMPOSITE\_SOFTLIGHT

- COMPOSITE\_DARKEN

- COMPOSITE\_LIGHTEN

- COMPOSITE\_COLOR\_DODGE

- COMPOSITE\_COLOR\_BURN

- COMPOSITE\_EXCLUSION

- COMPOSITE\_HUE

- COMPOSITE\_SATURATE

- COMPOSITE\_COLORIZE

- COMPOSITE\_LUMA\_MASK

- COMPOSITE\_DIVIDE

- COMPOSITE\_LINEAR\_DODGE

- COMPOSITE\_LINEAR\_BURN

- COMPOSITE\_LINEAR\_LIGHT

- COMPOSITE\_VIVID\_LIGHT

- COMPOSITE\_PIN\_LIGHT

- COMPOSITE\_HARD\_MIX

- COMPOSITE\_LIGHTER\_COLOR

- COMPOSITE\_DARKER\_COLOR

- COMPOSITE\_FOREGROUND

- COMPOSITE\_ALPHA

- COMPOSITE\_INVERTED\_ALPHA

- COMPOSITE\_LUM

- COMPOSITE\_INVERTED\_LUM

"Opacity" : floating point value from 0.0 to 100.0

"Distortion" : floating point value from -1.0 to 1.0

"RetimeProcess" : A value from the following constants

- RETIME\_USE\_PROJECT = 0

- RETIME\_NEAREST

- RETIME\_FRAME\_BLEND

- RETIME\_OPTICAL\_FLOW

"MotionEstimation" : A value from the following constants

- MOTION\_EST\_USE\_PROJECT = 0

- MOTION\_EST\_STANDARD\_FASTER

- MOTION\_EST\_STANDARD\_BETTER

- MOTION\_EST\_ENHANCED\_FASTER

- MOTION\_EST\_ENHANCED\_BETTER

- MOTION\_EST\_SPEED\_WRAP

"Scaling" : A value from the following constants

- SCALE\_USE\_PROJECT = 0

- SCALE\_CROP

- SCALE\_FIT

- SCALE\_FILL

- SCALE\_STRETCH

"ResizeFilter" : A value from the following constants

- RESIZE\_FILTER\_USE\_PROJECT = 0

- RESIZE\_FILTER\_SHARPER

- RESIZE\_FILTER\_SMOOTHER

- RESIZE\_FILTER\_BICUBIC

- RESIZE\_FILTER\_BILINEAR

- RESIZE\_FILTER\_BESSEL

- RESIZE\_FILTER\_BOX

- RESIZE\_FILTER\_CATMULL\_ROM

- RESIZE\_FILTER\_CUBIC

- RESIZE\_FILTER\_GAUSSIAN

- RESIZE\_FILTER\_LANCZOS

- RESIZE\_FILTER\_MITCHELL

- RESIZE\_FILTER\_NEAREST\_NEIGHBOR

- RESIZE\_FILTER\_QUADRATIC

- RESIZE\_FILTER\_SINC

- RESIZE\_FILTER\_LINEAR

Values beyond the range will be clipped

width and height are same as the UI max limits

The arguments can be passed as a key and value pair or they can be grouped together into a dictionary (for python) or table (for lua) and passed

as a single argument.

Getting the values for the keys that uses constants will return the number which is in the constant

Deprecated Resolve API Functions

--------------------------------

The following API functions are deprecated.

ProjectManager

GetProjectsInCurrentFolder() --> {project names...} # Returns a dict of project names in current folder.

GetFoldersInCurrentFolder() --> {folder names...} # Returns a dict of folder names in current folder.

Project

GetPresets() --> {presets...} # Returns a dict of presets and their information.

GetRenderJobs() --> {render jobs...} # Returns a dict of render jobs and their information.

GetRenderPresets() --> {presets...} # Returns a dict of render presets and their information.

MediaStorage

GetMountedVolumes() --> {paths...} # Returns a dict of folder paths corresponding to mounted volumes displayed in Resolve’s Media Storage.

GetSubFolders(folderPath) --> {paths...} # Returns a dict of folder paths in the given absolute folder path.

GetFiles(folderPath) --> {paths...} # Returns a dict of media and file listings in the given absolute folder path. Note that media listings may be logically consolidated entries.

AddItemsToMediaPool(item1, item2, ...) --> {clips...} # Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is one or more file/folder paths. Returns a dict of the MediaPoolItems created.

AddItemsToMediaPool([items...]) --> {clips...} # Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is an array of file/folder paths. Returns a dict of the MediaPoolItems created.

Folder

GetClips() --> {clips...} # Returns a dict of clips (items) within the folder.

GetSubFolders() --> {folders...} # Returns a dict of subfolders in the folder.

MediaPoolItem

GetFlags() --> {colors...} # Returns a dict of flag colors assigned to the item.

Timeline

GetItemsInTrack(trackType, index) --> {items...} # Returns a dict of Timeline items on the video or audio track (based on trackType) at specified

TimelineItem

GetFusionCompNames() --> {names...} # Returns a dict of Fusion composition names associated with the timeline item.

GetFlags() --> {colors...} # Returns a dict of flag colors assigned to the item.

GetVersionNames(versionType) --> {names...} # Returns a dict of version names by provided versionType: 0 - local, 1 - remote.

Unsupported Resolve API Functions

---------------------------------

The following API (functions and parameters) are no longer supported. Use job IDs instead of indices.

Project

StartRendering(index1, index2, ...) --> Bool # Please use unique job ids (string) instead of indices.

StartRendering([idxs...]) --> Bool # Please use unique job ids (string) instead of indices.

DeleteRenderJobByIndex(idx) --> Bool # Please use unique job ids (string) instead of indices.

GetRenderJobStatus(idx) --> {status info} # Please use unique job ids (string) instead of indices.

GetSetting and SetSetting --> {} # settingName videoMonitorUseRec601For422SDI is now replaced with videoMonitorUseMatrixOverrideFor422SDI and videoMonitorMatrixOverrideFor422SDI.

# settingName perfProxyMediaOn is now replaced with perfProxyMediaMode which takes values 0 - disabled, 1 - when available, 2 - when source not available.

WORKFLOW INTEGRATIONS

Updated as of 25 August, 2020

---------------------------

In this package, you will find a brief introduction to the Workflow Integration Plugins support for DaVinci Resolve Studio. Apart from this README.txt file, this package contains following folders:

Examples: containing some representative sample plugin, and a sample script.

Scripts: containing some sample workflow scripts to interact with Resolve.

Overview

--------

DaVinci Resolve Studio now supports Workflow Integration Plugins to be loaded and communicate with Resolve. Resolve can run one or more Workflow Integration Plugins at the same time.

Users can write their own Workflow Integration Plugin (an Electron app) which could be loaded into DaVinci Resolve Studio. To interact with Resolve, Resolve's JavaScript APIs can be used from the plugin.

Alternatively, a Python or Lua script can be invoked, with the option of a user interface built with Resolve's built-in Qt-based UIManager, or with an external GUI manager. See the "Sample Workflow Integration Script" section below for details.

Sample Workflow Integration Plugin

----------------------------------

A sample Workflow Integration Plugin is available in the "Examples/SamplePlugin" directory. In order for Resolve to register this plugin, this directory needs to be copied to 'Workflow Integration Plugins' root directory (mentioned in below section).

Once a plugin is registered, plugin can be loaded from UI sub-menu under 'Workspace->Workflow Integrations'. This will load the plugin and show the plugin HTML page in a separate window.

Sample plugin helps to understand how a plugin should be structured and how it works with Resolve. Please refer to the directory/file structure, manifest file info, plugin loading, JavaScript API usage examples, etc.

This sample plugin and scripts demonstrates few basic scriptable JavaScript API usages to interact with Resolve.

Loading Workflow Integration Plugin

-----------------------------------

On startup, DaVinci Resolve Studio scans the Workflow Integration Plugins root directory and enumerates all plugin modules. For each valid plugin module, it creates a UI sub-menu entry under 'Workspace->Workflow Integrations' menu.

DaVinci Resolve Studio reads the basic details of the plugin from its manifest.xml file during load time. Once plugin is loaded, user can click on the 'Workflow Integrations' sub-menu to load the corresponding plugin.

Workflow Integration Plugin directory structure

-----------------------------------------------

com.<company>.<plugin\_name>/

package.js

main.js

index.html

manifest.xml

node\_modules/

<Node.js modules>

js/

<supporting js files>

css/

<css files containing styling info>

img/

<image files>

Workflow Integration Plugins root directory

-------------------------------------------

User should place their Workflow Integration Plugin under the following directory:

Mac OS X:

"/Library/Application Support/Blackmagic Design/DaVinci Resolve/Workflow Integration Plugins/"

Windows:

"%PROGRAMDATA%\Blackmagic Design\DaVinci Resolve\Support\Workflow Integration Plugins\"

Supported platforms

-------------------

Plugins: Windows, Mac OS X (not supported on Linux currently)

Scripts: Windows, Mac OS X, Linux

Using scriptable JavaScript API

-------------------------------

Scriptable JavaScript API execution happens under HTML environment like any typical website. Once HTML page is loaded it can execute scriptable JavaScript API as needed (like clicking on a button, etc.)

This example JavaScript snippet creates a simple project in DaVinci Resolve Studio:

const WorkflowIntegration = require('./WorkflowIntegration.node');

isInitialized = WorkflowIntegration.Initialize('com.blackmagicdesign.resolve.sampleplugin');

if (isInitialized) {

resolve = WorkflowIntegration.GetResolve();

resolve.GetProjectManager().CreateProject("Hello World");

}

The resolve object is the fundamental starting point for scripting via Resolve. As a native object, it can be inspected for further scriptable properties and functions in JavaScript.

WorkflowIntegration module API

-------------------------------

To interact with Resolve you need to use 'WorkflowIntegration.node' Node.js module file in your plugin app. Below are the WorkflowIntegration (module) JavaScript API functions to communicate with Resolve.

WorkflowIntegration

Initialize(<pluginId>) --> Bool # Returns true if initialization is successful, false otherwise. <pluginId> is the unique plugin id string configured in the manifest.xml file.

GetResolve() --> Resolve # Returns Resolve object.

RegisterCallback(callbackName, callbackFunc) --> Bool # Returns true if input callback name/function is registered successfully, false otherwise.

# 'callbackName' should be a valid supported callback string name (refer to the below section 'Supported callbacks').

# 'callbackFunc' should be a valid JavaScript function without any arguments.

DeregisterCallback(callbackName) --> Bool # Returns true if input callback name is deregistered successfully, false otherwise.

CleanUp() --> Bool # Returns true if cleanup is successful, false otherwise. This should be called during plugin app quit.

SetAPITimeout(valueInSecs) --> Bool # By default, apis dont timeout. In order to enable timeout, set a non-zero positive integer value in the arg 'valueInSecs'.

# Setting it to 0 will disable timeout. This function will return true if the timeout is set/reset successfully.

Supported callbacks

-------------------

'RenderStart'

'RenderStop'

Please note that there is no console based support for JavaScript API.

Sample Workflow Integration Script

----------------------------------

A sample Workflow Integration Python script is also available in the "Examples" directory. In order for Resolve to register this script, it needs to be copied to the 'Workflow Integration Plugins' root directory (mentioned in the above section).

Once a script is registered, it can be also loaded from the 'Workspace' menu, under 'Workflow Integrations'. This will invoke the script and show the sample UIManager window.

Workflow Integration scripts work similarly to other scripts in Resolve, and use the same scripting API. This example script provides a basic introduction into creating a popup Workflow application using a UIManager window, with simple layout of text fields and buttons, and event handlers to dispatch functions for integration with the user's facility. Alternatively, third-party UI managers such PyQt may be used instead, or no GUI at all.

When launched by Resolve, plugin scripts are automatically provided with 'resolve' and 'project' variables for immediate and easy access to Resolve's scripting API. Additional third-party modules may be imported for access to asset-management systems as desired.

UIManager Introduction

----------------------

There are two main objects needed to manage a window, the UIManager that handles layout, and the UIDispatcher that manages interaction events, accessed as follows:

ui = fusion.UIManager()

dispatcher = bmd.UIDispatcher(ui)

Windows are created with the the UIDispatcher, passing a dictionary of attributes like ID and Text, with GUI elements in nested layouts all created with the UIManager.

UIDispatcher Functions

--------------------

The UIDispatcher object has a few important functions to manage processing of events. The most important are:

AddWindow(props, children): Accepts a dictionary of properties and a list of children, returns a Window object

AddDialog(props, children): Accepts a dictionary of properties and a list of children, returns a Dialog object

int RunLoop(): Call when your window is ready to receive user clicks and other events

ExitLoop(int): Terminates the event processing, and returns any supplied exit code from RunLoop()

Common usage is to create your window and set up any event handlers, including a Close handler for the window that calls ExitLoop(), then Show() your window and call RunLoop() to wait for user interaction:

ui = fusion.UIManager

dispatcher = bmd.UIDispatcher(ui)

win = dispatcher.AddWindow({ 'ID': 'myWindow' }, [ ui.Label({ 'Text': 'Hello World!' }) ])

def OnClose(ev):

dispatcher.ExitLoop()

win.On.myWindow.Close = OnClose

win.Show()

dispatcher.RunLoop()

AddWindow() will also accept a single child without needing a list, or a single dictionary containing both proprties and child elements, for ease of use.

As well as constructing new child elements and layouts, the UIManager also offers a few useful functions:

FindWindow(ID): Returns an element with matching ID

FindWindows(ID): Returns a list of all elements with matching ID

QueueEvent(element, event, info): Calls the element's event handler for 'event', passing it the dictionary 'info'

UIManager Elements

------------------

The element's ID is used to find, manage, and dispatch events for that element. GUI elements also support a set of common attributes including Enabled, Hidden, Visible, Font, WindowTitle, BackgroundColor, Geometry, ToolTip, StatusTip, StyleSheet, WindowOpacity, MinimumSize, MaximumSize, and FixedSize. Some other common GUI elements and their main attributes include:

Label: Text, Alignment, FrameStyle, WordWrap, Indent, Margin

Button: Text, Down, Checkable, Checked, Icon, IconSize, Flat

CheckBox: Text, Down, Checkable, Checked, Tristate, CheckState

ComboBox: ItemText, Editable, CurrentIndex, CurrentText, Count

SpinBox: Value, Minimum, Maximum, SingleStep, Prefix, Suffix, Alignment, ReadOnly, Wrapping

Slider: Value, Minimum, Maximum, SingleStep, PageStep, Orientation, Tracking, SliderPosition

LineEdit: Text, PlaceholderText, Font, MaxLength, ReadOnly, Modified, ClearButtonEnabled

TextEdit: Text, PlaceholderText, HTML, Font, Alignment, ReadOnly, TextColor, TextBackgroundColor, TabStopWidth, Lexer, LexerColors

ColorPicker: Text, Color, Tracking, DoAlpha

Font: Family, StyleName, PointSize, PixelSize, Bold, Italic, Underline, Overline, StrikeOut, Kerning, Weight, Stretch, MonoSpaced

Icon: File

TabBar: CurrentIndex, TabsClosable, Expanding, AutoHide, Movable, DrawBase, UsesScrollButtons, DocumentMode, ChangeCurrentOnDrag

Tree: ColumnCount, SortingEnabled, ItemsExpandable, ExpandsOnDoubleClick, AutoExpandDelay, HeaderHidden, IconSize, RootIsDecorated,

Animated, AllColumnsShowFocus, WordWrap, TreePosition, SelectionBehavior, SelectionMode, UniformRowHeights, Indentation,

VerticalScrollMode, HorizontalScrollMode, AutoScroll, AutoScrollMargin, TabKeyNavigation, AlternatingRowColors,

FrameStyle, LineWidth, MidLineWidth, FrameRect, FrameShape, FrameShadow

TreeItem: Selected, Hidden, Expanded, Disabled, FirstColumnSpanned, Flags, ChildIndicatorPolicy

Some elements also have property arrays, indexed by item or column (zero-based), e.g. newItem.Text[2] = 'Third column text'

Combo: ItemText[]

TabBar: TabText[], TabToolTip[], TabWhatsThis[], TabTextColor[]

Tree: ColumnWidth[]

Treeitem: Text[], StatusTip[], ToolTip[], WhatsThis[], SizeHint[], TextAlignment[], CheckState[], BackgroundColor[], TextColor[], Icon[], Font[]

Some elements like Label and Button will automatically recognise and render basic HTML in their Text attributes, and TextEdit is capable of displaying and returning HTML too. Element attributes can be specified when creating the element, or can be read or changed later:

win.Find('myButton').Text = "Processing..."

Most elements have functions that can be called from them as well:

Show()

Hide()

Raise()

Lower()

Close() Returns boolean

Find(ID) Returns child element with matching ID

GetChildren() Returns list

AddChild(element)

RemoveChild(element)

SetParent(element)

Move(point)

Resize(size)

Size() Returns size

Pos() Returns position

HasFocus() Returns boolean

SetFocus(reason) Accepts string "MouseFocusReason", "TabFocusReason", "ActiveWindowFocusReason", "OtherFocusreason", etc

FocusWidget() Returns element

IsActiveWindow() Returns boolean

SetTabOrder(element)

Update()

Repaint()

SetPaletteColor(r,g,b)

QueueEvent(name, info) Accepts event name string and dictionary of event attributes

GetItems() Returns dictionary of all child elements

Some elements have extra functions of their own:

Label: SetSelection(int, int), bool HasSelection(), string SelectedText(), int SelectionStart()

Button: Click(), Toggle(), AnimateClick()

CheckBox: Click(), Toggle(), AnimateClick()

ComboBox: AddItem(string), InsertItem(string), AddItems(list), InsertItems(int, list), InsertSeparator(int), RemoveItem(int), Clear(),

SetEditText(string), ClearEditText(), Count(), ShowPopup(), HidePopup()

SpinBox: SetRange(int, int), StepBy(int), StepUp(), StepDown(), SelectAll(), Clear()

Slider: SetRange(int, int), TriggerAction(string)

LineEdit: SetSelection(int, int), bool HasSelectedText(), string SelectedText(), int SelectionStart(), SelectAll(), Clear(), Cut(), Copy(), Paste(),

Undo(), Redo(), Deselect(), Insert(string), Backspace(), Del(), Home(bool), End(bool), int CursorPositionAt(point)

TextEdit: InsertPlainText(string), InsertHTML(string), Append(string), SelectAll(), Clear(), Cut(), Copy(), Paste(), Undo(), Redo(),

ScrollToAnchor(string), ZoomIn(int), ZoomOut(int), EnsureCursorVisible(), MoveCursor(moveOperation, moveMode), bool CanPaste(),

string AnchorAt(point), bool Find(string, findFlags)

TabBar: int AddTab(strubg), int InsertTab(string), int Count(), RemoveTab(int), MoveTab(int, int)

Tree: AddTopLevelItem(item), InsertTopLevelItem(item), SetHeaderLabel(string), int CurrentColumn(), int SortColumn(),

int TopLevelItemCount(), item CurrentItem(), item TopLevelItem(int), item TakeTopLevelItem(int), item InvisibleRootItem(),

item HeaderItem(), int IndexOfTopLevelItem(item), item ItemAbove(item), item ItemBelow(item), item ItemAt(point),

Clear(), rect VisualItemRect(item), SetHeaderLabels(list), SetHeaderItem(item), InsertTopLevelItems(list), AddTopLevelItems(list),

list SelectedItems(), list FindItems(string, flags), SortItems(int, order), ScrollToItem(item), ResetIndentation(),

SortByColumn(int, order), int FrameWidth()

TreeItem: AddChild(item), InsertChild(item), RemoveChild(iitem), SortChildren(int, order), InsertChildren(int, list), AddChildren(list),

int IndexOfChild(item), item Clone(), tree TreeWidget(), item Parent(), item Child(int), item TakeChild(int),

int ChildCount(), int ColumnCount()

Window: Show(), Hide(), RecalcLayout()

Dialog: Exec(), IsRunning(), Done(), RecalcLayout()

Elements can be accessed by the window's FindWindow(id) function, or by assigning them to a variable for later usage, which is more efficient. The GetItems() function will return a dictionary of all child elements for ease of access.

UIManager Layout

----------------

Additionally, elements can be nested to define layout, using the HGroup and VGroup elements. As with Window and other elements, tou can pass a single dictionary or list with all properties and children, or separate them into a dict of properties and list of children, for convenience:

winLayout = ui.VGroup([

ui.Label({ 'Text': "A 2x2 grid of buttons", 'Weight': 1 }),

ui.HGroup({ 'Weight': 5 }, [

ui.Button({ 'ID': "myButton1", 'Text': "Go" }),

ui.Button({ 'ID': "myButton2", 'Text': "Stop" }),

]),

ui.VGap(2),

ui.HGroup({ 'Weight': 5 }, [

ui.Button({ 'ID': "myButtonA", 'Text': "Begin" }),

ui.Button({ 'ID': "myButtonB", 'Text': "End" }),

]),

]),

win = dispatcher.AddWindow({ 'ID': "myWindow" }, winLayout)

HGap and VGap elements can included for finer spacing control. Note also the Weight attribute, which can be applied to most elements to control how they adjust their relative sizes. A Weight of 0 will use the element's minimum size.

Event Handlers

--------------

Window objects will call user-defined event handler functions in response to various interaction events. Event handlers are managed using a window member called 'On'. This has sub-members for each GUI element with an ID, and those have members for each available event. To set up an event handler, define a function for it, then assign the function to the window's On.ID.Event member as follows:

def OnClose(ev):

dispatcher.ExitLoop()

win.On.myWindow.Close = OnClose

Alternatively, if your object's ID is stored in a string variable called 'buttonID', you could use:

win.On[buttonID].Clicked = OnButtonClicked

Many objects have specific events that can be handled:

Button: Clicked, Toggled, Pressed, Released

CheckBox: Clicked, Toggled, Pressed, Released

ComboBox: CurrentIndexChanged, CurrentTextChanged, TextEdited, EditTextChanged, EditingFinished, ReturnPressed, Activated

SpinBox: ValueChanged, EditingFinished

Slider: ValueChanged, SliderMoved, ActionTriggered, SliderPressed, SliderReleased, RangeChanged

LineEdit: TextChanged, TextEdited, EditingFinished, ReturnPressed, SelectionChanged, CursorPositionChanged

TextEdit: TextChanged, SelectionChanged, CursorPositionChanged

ColorPicker: ColorChanged

TabBar: CurrentChanged, CloseRequested, TabMoved, TabBarClicked, TabBarDoubleClicked

Tree: CurrentItemChanged, ItemClicked, ItemPressed, ItemActivated, ItemDoubleClicked, ItemChanged, ItemEntered,

ItemExpanded, ItemCollapsed, CurrentItemChanged, ItemSelectionChanged

Window: Close, Show, Hide, Resize, MousePress, MouseRelease, MouseDoubleClick, MouseMove, Wheel, KeyPress, KeyRelease,

FocusIn, FocusOut, ContextMenu, Enter, Leave

Event handler functions are called with a dictionary of related attributes such as who, what, when, sender, and modifiers. Common events and some additional attributes they receive include:

MousePress: Pos, GlobalPos, Button, Buttons

MouseRelease: Pos, GlobalPos, Button, Buttons

MouseDoubleClick: Pos, GlobalPos, Button, Buttons

MouseMove: Pos, GlobalPos, Button, Buttons

Wheel: Pos, GlobalPos, Buttons, Delta, PixelDelta, AngleDelta, Orientiation, Phase

KeyPress: Key, Text, IsAutoRepeat, Count

KeyRelease: Key, Text, IsAutoRepeat, Count

ContextMenu: Pos, GlobalPos

Move: Pos, OldPos

FocusIn: Reason

FocusOut: Reason

Event handlers can be enabled or disabled for a given element by turning them on or off in the Events attribute:

ui.Slider({ 'ID': 'mySlider', 'Events': { 'SliderMoved': true } })

Some common events like Clicked or Close are enabled by default.

Basic Resolve API

-----------------

Please refer to the [Basic Resolve API] section in '../Developer/Scripting/README.txt' file for the list of the functions that Resolve offers for scripted control. For plugin scripts, the 'resolve' and 'project' variables are automatically set up for you, and may be used to access any part of Resolve's API.

Further Information

-------------------

This document provides a basic introduction only, and does not list all available UIManager elements and attributes. As UIManager is based on Qt, you can refer to the Qt documentation at https://doc.qt.io/qt-5/qwidget.html for more information on element types and their attributes. There are also many third-party examples and discussions available on user forums for DaVinci Resolve and Fusion Studio.