Avolites Titan: Macros

By: Sebastian Beutel, March 2016. Kindly cross-read and corrected by Gregory Haynes.

Overview

Macros are a way to automatically call specific functions or set specific values. Macros can be called from workspace buttons, or can be triggered from specific actions (like: from starting a cue in a cue list or – as of Titan v10 – starting a set list track).

Simple macros are just a protocol of button presses and can be recorded inside Titan. While one can do amazing things with this, there are limitations, like: it's always the button press which is recorded, almost regardless which state this button has.

Factory macros are predefined macros from the macro library which is installed with the software and updated with any personality update. These macros make use of a specific programming language, while the technique – essentially XML documents – is mighty but easy to read and write. Since these macros connect almost directly to the Titan engine, they are a very effective way to automate specific tasks and processes.

It is possible – and encouraged – to write such macros yourself. This document is the attempt to reveal some parts of the programming structure. It is, however, only based on the existing macros, on information publicly available e.g. in the Avolites forum, and on own experiments.

This document assumes you are familiar with the current Avolites Titan software, and have some basic knowledge about scripting languages, and writing some very basic code (like: HTML, JavaScript, or the like). You are greatly encouraged to read the existing macros files and use them for first own experiments.

This document tries to build the bridge to this kind of programming even for someone who is not so fluent in writing any kind of code or knows about programming paradigms. That's why some descriptions – e.g. for XML or the object model - are extremely simplified. Please know that there is plenty of literature about any such theme. The scope of this document, however, is only a quick introduction into Avolites.Titan.Macros ☺

Basics and Fundamentals

Macro files are XML files. XML stands for eXtensible Markup Language, and refers to the contents of the files. Essentially it's a plain-text file with a long list of commands which are structured in a certain way. The relevant parts of this structure are described below.

Writing XML files

It is recommended to use a proper text editor. Please do not use Microsoft Word or a similar 'text processor': it is hard, if not impossible, to get just plain text – in the correct formatting – out of this. While, in theory, Windows' own notepad would do what you need, you'll soon learn that this is less than basic. Of course you could instead use big programming suites like Eclipse – this would be far too big if you asked me, you'd need to spend months to learn this tool only to write a couple of lines.

What I prefer is something like Notepad++ which is officially available for free, supports UTF-8, comes with syntax highlighting, fold/unfold sections and many more useful tools – and is both: mighty and light-weight. Optionally you can add an XML tools plugin which might come in handy, e.g. for syntax checking. Of course it

cannot check whether the macro does what you intended it to do. But it checks on typos, invalid characters, forgotten closing quotes/braces and such things.

For other tasks you might additionally consider installing 7-zip, a free, light-weight, universal archiver/ decompressor which uncompresses more than you might think (e.g. the downloaded FixtureLibrary ☺).

The macro files must be encoded in UTF-8, with BOM (Byte Order Mark). You might think of 'encoding' as the translation between bits and bytes on one side, and letters and symbols on the other. If you want to know more: google it, or simply open an existing macro file and edit it, instead of creating a new one.

Line-endings are Windows-style (\r\n).

Deploying macro files

Macro files need to be in a designated folder in order for Titan to find it:

On consoles, the factory library is hidden in `D:\TitanData\Macros`. This is overwritten/updated with every personality update you might install. Hence, putting your macros there is not the best idea. However this folder is not visible by default, and this is by purpose. Just don't temper with it.

On consoles, from v10 on, there is a place for user macros which is not overwritten with personality updates: `D:\Macros`. Put your macros there.

As for Titan PC Suite (Titan One, Simulator, Titan Mobile), the folder for macros is `C:\Program Files\Avolites\Titan\Macros`. You will soon learn that Windows prevents you from editing files directly there unless you start the editor with Administrator privileges or disable UAC. To my mind the best way is to have a working copy outside this folder which you can edit and then copy into the required folder. However, as of Titan v10, there is also a separate folder for user macros which is easily accessible: `%userprofile%\documents\Titan\Macros` which translates to 'My Documents\Titan\Macros'.

There may – and most likely will – be multiple macro files in this folder which are all scanned upon program start. I have no idea what happens if you have the same macro ID multiple times in one or more files. Just prevent it.

Macro files are scanned when Titan starts. Hence, when you apply changes or create a new macro or a new file, you need to restart the software for the changes to take effect.

Structure of macro files

Macro files are XML – and XML is a strictly structured markup language. Hence, you are required to obey some rules, like: every tag, once opened, needs to be closed; never cross-nest tags; obey the grammar. If your file doesn't work, just scan it for any commas, colons, periods and other symbols which are not where they are expected. There is a vast variety of documentation available on any aspect of XML – start reading if you want to know.

The outer frame

Any file must start with

```
<?xml version="1.0" encoding="utf-8"?>
```

This is called the XML declaration, and is required for most software to correctly recognize the contents.

The next line must – at least – be

```
<avolites.macros>
```

This is the first tag, and needs to be exactly this. It makes sure all the macros are correctly scanned by Titan.

In some files you will find this tag in a longer version:

```
<avolites.macros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"↵
       xsi:noNamespaceSchemaLocation="Avolites.Menus.xsd">
```

This is the tag with a so-called namespace and schema definition. I'd say it doesn't hurt if you make a habit of always using the longer version.

I said any tag which is open needs to be closed again – so let's have a look at the very bottom of our file. There you'll find

```
</avolites.macros>
```

This is exactly the closing version of our first tag – closing tags start with '</'.

So, the outer frame of our macro file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<avolites.macros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"↵
       xsi:noNamespaceSchemaLocation="Avolites.Menus.xsd">
<!-- Macros only applicable to PC Suite -->
<!-- 13-06-2021  Created by Mickey Mouse -->

</avolites.macros>
```

The lines starting with ,<!-- ' and ending with ,-->' are comments. They are not interpreted by the software and are there just for your convenience. Make use of it.

You could (almost) write all code in one long line, which obviously isn't exactly readable. You can put each element, and even each part of an element (like a property definition) on a new line, like so:

```
<macro id="Avolites.Macros.CloseAllWindows"
       name="Close All Windows">
```

However, strings (the text within quotes) must not contain line breaks.

Indenting of lines is optional, but greatly improves readability.

If a line break occurs only in this document (to fit the page) but does not occur in the original code then this is denoted with a ↵ symbol, and the next line is indented. You need to remove both, the ↵ and the indention, when copying this code into your own macros.

Between the opening <avolites.macros> and the closing </avolites.macros> we can put our macros: one, some, or many.

Macros – the inner structure

A macro may look like this>

```
<macro id="Avolites.Macros.CloseAllWindows" name="Close All Windows">
  <description>Closes all the workspace windows.</description>
  <sequence>
    <step>Windows.Stack.CloseAll("mainWindowStack")</step>
  </sequence>
</macro>
```

There is the `<macro...>` opening tag and – at the end - the `</macro>` closing tag. It has the two properties 'id' and 'name' which are defined in the opening tag. Note the grammar of these definitions:

```
property="value"
```

The value needs to be written in double quotes – which is the reason why you usually cannot have double quotes inside a property's value. (There are ways to achieve this – a.k.a. escaping – but there is no need for us to make things complicated…)

Without knowing all details, the id must be a unique identifier, while the name is what is printed on the button, like this:

From other experience I'd say:

- The id must consist of alphanumeric characters and the . (point), any other characters including whitespaces are not allowed. Also, I'd avoid any non-ASCII (non-standard) letters, like äüö. Why not using a similar pattern like in the existing macros, e.g. "Avolites.UserMacros.MyFirstMacro". This way it's easy to make sure the ID is unique. If it isn't unique, only one macro with that ID will be available in Titan, and it's hard to predict which one. So, let the ID be unique.
- The name is a little less strict as it may contain whitespaces. However, make sure it is not too long, as this would mess with a neat display. Personally, I'd also avoid non-ASCII characters here, albeit I assume they would work.

Inside the `<macro...> … </macro>`, the next element is

```
<description>Closes all the workspace windows.</description>
```

This is a different grammar, because ‚Closes all the workspace windows.' is not a property, but the contents of the `<description>…</description>` element. As the name suggests, this is to contain a short description of what the macro does. Currently this isn't displayed within Titan. But I could imagine that in a future version Titan could show this as a little tooltip for each macro. So, why not giving a short description?

Next is an optional element: `<active binding="… "/>`. This is supposed to highlight the macro button if set to true (e.g. by giving it the current value of a variable, or an object's property). However, I found it used only in one factory macro…

(example: `<active binding="UserSettings.Setting.List.SnapMode"/>` in Snap on/off).

Next element is

```
<sequence>
  ...
</sequence>
```

And inside this – its contents – is

```
<step>Windows.Stack.CloseAll("mainWindowStack")</step>
```

It appears that this is where the magic happens: some program code is called – this is what the big next section is for.

When you look through the factory files you find entries like this:

```
<step>
  <menuLink id="Expert.Copy"/>
</step>
```

Here, the step contains another element. As mentioned before, the line breaks are optional – the same thing could be written like this:

```
<step><menuLink id="Expert.Copy"/></step>
```

Also, there are macros like this:

```
<sequence>
   <step>Command.RunCommand(Command.CommandLineText)</step>
   <step>Programmer.Editor.Fixtures.RemainderDim()</step>
   <step>Command.StartNewCommand()</step>
</sequence>
```

I'd say it's safe to assume that a sequence may contain one or more steps, which are executed consecutively when called.

Steps can be paused by setting their property pause:

```
<step pause="0.01">Chases.IncrementWheelParameter(1, -0.1)</step>
```

Finally, there are step definitions like this:

```
<step condition="Math.IsEnumEqual(UserSettings.TempoUnits, 'BPM')">↵
          Chases.IncrementWheelParameter(0, -0.02)</step>
```

So, a step may depend from a condition and is only executed when the condition is true. The condition is written in the same way a 'normal' property is written (see above "<macro id="… "). The condition itself is again some fancy programming code which we'll have a look at later on. This code is, similar to a macro's id and name, written in double quotes: " … ".

Summary XML structure

It's time to sum up our first macro grammar lesson:

- A macro file always starts with the XML declaration
- Comments can be everywhere, as long as they are on a line itself. They are not interpreted, and only for the programmer, to hold notes.
- The root element of all macros is the <avolites.macros> element which may contain any number of macros.
- The Macro element should have the properties id (which should be unique) and name (for the button), and may contain a description and a sequence.
- The description element may contain a plain string, to briefly describe what the macro does.
- There is an optional element <active binding /> which is yet to be explained.
- The sequence element holds one or more step elements which are called/executed consecutively.
- The step elements hold the real program magic: usually some fancy programming code. Steps might be conditional in which case the condition is a property of the step and written as such. Another optional property is pause which delays the step by the given time.
- Step elements might, instead of some programming code, contain a menulink element. This is a so-called empty element (opened and closed in one tag, ending with />), with id as required property, and stack and behaviour as optional properties.
- Property values are defined with double quotes.

The program instruction syntax

You will already have seen that almost all instructions are written in a special way, e.g. `Handles.Presets.ChangePage(0), Windows.Stack.MinMaxWindow("mainWindowStack")` or `ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Normal")`. This style is derived from object oriented programming structures. If you are familiar to JavaScript or more complicated languages then this is no news for you. Here comes an extremely brief rundown of the essentials which we need here.

- Everything is done in a strict hierarchic structure.
- There is always at least a root object. Usually this is not denoted. However, we always need to declare which object we are dealing with.
- Any object may have child objects.
- Any object may have properties (which can be set and changed)
- Any object may have functions (which usually take arguments) (The purists like to call this 'methods', not 'functions'. But we are not in an o.o.p. lesson here…)

Hence, the individual parts are separated with a . (point) while functions expect their arguments in braces (). Names of objects, properties and functions are always alphanumeric, must not contain any whitespace and usually are written in CamelCase, for better readability.

Some examples:

```
Object.ChildObject
Object.ChildObject.Property
Object.ChildObject.Function(Arguments)
```

Now we can read our examples clearly:

`Handles.Presets.ChangePage(0)`:

- `Handles` is the main object
- `Presets` is one of its Child objects – you can identify it as `Handles.Presets`
- `ChangePage()` is a function of `Handles.Presets` and takes a number as argument (remember that in most programming contexts the first item in a row is numbered 0). When numbers are given as argument then no quotes are required.

From the names we are safe to assume that this macro changes the preset handles to page 1.

`Windows.Stack.MinMaxWindow("mainWindowStack")`:

- `Windows` is the main object
- `Stack` is a child object of `Windows`
- `MinMaxWindow()` is a function of `Windows.Stack`. It takes a string as argument – that's why the double quotes. `"mainWindowStack"` is the identifier of the stack holding our windows. There is also another macro with something like this:
  `Windows.Stack.SizePositionWindow("mainWindowStack",false)`
  Here the function takes two arguments: `"mainWindowStack"` and `false`. Arguments are separated with commas, `false` and `true` are constants which do not require quotes.

`ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Normal")`:

- `ActionScript` is the main object (which, by the way, appears to be very special, since it seems to be able to manipulate other objects)

- `SetProperty` is a child object of `ActionScript`
- `Enum` is a function of `ActionScript.SetProperty` and takes two strings as arguments

Well, after reading the real examples with ActionScript you might better understand this as: SetProperty is a method (changing a given property of some kind to some value) with Enum, Float, Boolean (and probably some more) being sub-functions only for particular variable formats. (Enum is a set of fixed values, Float might hold numbers, and Boolean is just true or false.)

Also, we need to distinguish between variables (essentially object properties) and their values, like in this example:

```
ActionScript.SetProperty.Boolean("Timecode.Enabled", !Timecode.Enabled)
```

- We now know that this is a function of the object `ActionScript.SetProperty` which sets another property to true or false (hence the `Boolean`). This function expects two arguments: the name of the property to be set, and the designated value. Both arguments are written inside the braces.
- The first argument is the string identifying the property. And since it is a string, it needs to be written in quotes: `"Timecode.Enabled"`
- The second argument is either true or false, and is here taken from a previously set variable/property which just happens to be the same which is being set here. But: we want to invert it. Inversion is commonly written with a ! (exclamation mark). However, it is not the string name of the property, but its contents/value. And that's why we don't have quotes around the second argument: `!Timecode.Enabled`.

This command could be explained as: set the variable `Timecode.Enabled` to true if it was false before and vice-versa. Or simply: toggle it.

Available Menus with <menuLink>

Similar to the described instruction syntax, the menus are organized in a hierarchical style. Here are the menu ids currently used in the factory macros:

`Expert.WindowMenu` = Open Workspace Window

Example:

```
<step>
  <menuLink id="Expert.WindowMenu" />
</step>
```

Expert.LocateMenu	(Opens the locate fixture menu)
Expert.ClearMenu	(Opens the clear fixture menu.)
Expert.Copy	(Opens the copy/move menu). Need to set handles in copy or move mode in 2nd step.
Expert.Shapes	(Opens the shape generator menu.)
PixelMapper.Root	(Effects Menu)
Legend.Set.SelectHandle	(Set Legend)
Expert.UserSettings	
Groups.SelectLayoutGroup	(Edit Group Layout)

Expert.Group.Edit

Expert.Program.SelectIf

Expert.SteppedPlayback.RecordStep (Record Step)

Available windows with <menuLink>

Windows are called/opened almost as like as menus are called.

`Windows.Playbacks` = the Playbacks Window

Example: `<menuLink id="Windows.Playbacks" stack="mainWindowStack" behaviour="PushOrRaise" />`

However, calling windows this way requires the stack and behaviour arguments set in order to bring the window to the front – this is exactly the same thing with the following calls:

Windows.Fixtures

Windows.Groups

Windows.Colours

Windows.Positions

Windows.Beams

Windows.Effects

Windows.Hud

Windows.ChannelGrid

Windows.Macros

Windows.ShapeLibrary

Windows.Attributes

Windows.Visualiser

Windows.Compatibility.FixturesAndPlaybacks

Windows.Compatibility.GroupsAndPalettes

Windows.Timecode

Windows.Dmx

Windows.ShowLibrary

Windows.EffectEditor

Windows.PixelMapPreview

Windows.ActivePlaybacks

Windows.PatchView

Windows.SetListView

Windows.EventLogViewer

Windows.StaticPlaybacks

Windows.MobileWing

Windows.Audio

Windows.SelectWindow

Windows.ShapeView

Windows.CueView

Windows.PixelEffectView

Commands inside steps

Windows.Stack.MinMaxWindow("mainWindowStack") = Min/Max Window

Windows.Stack.SizePositionWindow("mainWindowStack",false) = Size/Position Window

Menu.Stack.UpOneLevel("mainWindowStack") = Close Window

Windows.Stack.CloseAll("mainWindowStack") = Close All Windows

Windows.Stack.CycleActiveWindowRegion("mainWindowStack") = Move Screen

Menu.Stack.LowerToBottom("mainWindowStack", Global.ActiveWindowId) = Next Window

```
<step condition="!ActionScript.Test.String.IsNullOrEmpty(Global.ActiveWindowId)">↵
       Menu.Stack.LowerToBottom("mainWindowStack", Global.ActiveWindowId)</step>
```

Programmer.Editor.Fixtures.LocateSelectedFixtures(false) = Locate Fixture

Programmer.Editor.Clear(Attribute.Mask.Clear.Value, Programmer.Editor.Fixtures.Clear.Presets)
= Clear Fixture

Fixtures.Macros.FireMacro("Lamp On")

Fixtures.Macros.FireMacro("Lamp Off")

Fixtures.Macros.FireMacro("Reset")

History.Undo()

History.Redo()

Expert.Profiles.SelectProgramProfile = Select Key Profile

Expert.Profiles.Select.Edit.Current = Edit Key Profile

Playbacks.ReleaseAllPlaybacks(Expert.ReleasePlayback.FadeTime,
Expert.ReleasePlayback.UseMaster) = Release All Playbacks

Masters.ResetAllMasters()

Group.ResetAllMasters()

Command.RunCommand(Command.CommandLineText)

Programmer.Editor.Fixtures.RemainderDim()

Command.StartNewCommand()

Programmer.Editor.Fixtures.SetDimmerLevel(100)

Handles.Presets.ChangePage(0) (also with (1), (2), (3))

Handles.StaticPlaybacks.ChangePage(0) (also with (1), (2), (3))

Handles.StaticPlaybacks.PreviousPage()

Handles.StaticPlaybacks.NextPage()

CueLists.CutNextCueToLive(CueLists.ConnectedHandle)

CueLists.SnapBack(CueLists.ConnectedHandle)

Menu.InjectInput("OnSelect", "Snap", "NoGroup", 0) (together with <active binding>)

CueLists.Review(CueLists.ConnectedHandle)

CueLists.GoBack(CueLists.ConnectedHandle)

Menu.InjectInput("OnButtonDown", "Go", "NoGroup", 0) (together with OnButtonUp)

Menu.InjectInput("OnButtonUp", "Go", "NoGroup", 0)

Menu.InjectInput("OnButtonDown", "Stop", "NoGroup", 0) (together with OnButtonUp)

Menu.InjectInput("OnButtonUp", "Stop", "NoGroup", 0)

Chases.ConnectedChaseTap()

Chases.Direction.Backwards() (together with NextStep(…))

Chases.NextStep(Chases.ConnectedHandle)

Chases.Direction.Forwards() (together with NextStep(…))

Chases.Direction.Random()

Chases.Direction.Bounce()

Chases.IncrementWheelParameter(1, -0.4) (XFade 0%)

Chases.IncrementWheelParameter(1, 0.4) (XFade 100%)

Chases.IncrementWheelParameter(1, -0.3) (XFade 25%; first set to 100%)

Chases.IncrementWheelParameter(1, -0.2) (XFade 50%; first set to 100%)

Chases.IncrementWheelParameter(1, -0.1) (XFade 75%; first set to 100%)

Chases.IncrementWheelParameter(0, 0.02) (Speed +5 BPM)

Chases.IncrementWheelParameter(0, 0.01) (Speed +0.1 s)

Chases.IncrementWheelParameter(0, -0.02) (Speed -5 BPM)

Chases.IncrementWheelParameter(0, -0.01) (Speed -0.1 s)

Chases.IncrementWheelParameter(0, -14.4) (Speed 0 BPM)

Chases.IncrementWheelParameter(0, 0.24) (Speed 60 BPM after setting to 0 BPM)

Playbacks.Editor.CueSelection.SelectCueByNumber(Playbacks.Editor.SelectedPlayback, CueLists.LiveCueNumber)

Playbacks.Editor.CueSelection.SelectCueByNumber(Playbacks.Editor.SelectedPlayback, CueLists.NextCueNumber)

Editor.Shapes.SetSelectedViewShapes(CueLists.LiveCueHandle)

Editor.Shapes.SetSelectedPlaybackCueHandle(CueLists.LiveCueHandle)

Editor.Shapes.SetSelectedViewShapes(CueLists.NextCueHandle)

Editor.Shapes.SetSelectedPlaybackCueHandle(CueLists.NextCueHandle)

Editor.PixelMapper.SetSelectedViewPixelEffectHandles(CueLists.LiveCueHandle)

Editor.PixelMapper.SetSelectedPlaybackCueHandle(CueLists.LiveCueHandle)

Editor.PixelMapper.SetSelectedViewPixelEffectHandles(CueLists.NextCueHandle)

Editor.PixelMapper.SetSelectedPlaybackCueHandle(CueLists.NextCueHandle)

Playbacks.Editor.CueSelection.StepSelection(Playbacks.Editor.SelectedPlayback, 1)

Playbacks.Editor.CueSelection.StepSelection(Playbacks.Editor.SelectedPlayback, -1)

Command.RunCommand("PATTERN 1.1") (Odd. Followed by
ActionScript.SetProperty.Boolean("Selection.Context.Global.RepeatPattern", true)

Command.RunCommand("PATTERN 0.1.1") (Even. Followed by
ActionScript.SetProperty.Boolean("Selection.Context.Global.RepeatPattern", true)

Command.RunCommand("PATTERN 1 IN 3") (Pattern 1 in 3. Followed by
ActionScript.SetProperty.Boolean("Selection.Context.Global.RepeatPattern", true)

(up to pattern 1 in 10)

Selection.Context.Global.SetRandomFixtureOrder() (randomize Selection)

Selection.Context.Global.SelectAllCells()

Selection.Context.Global.InvertPattern()

Selection.Context.Global.ClearPatternSelect()

Programmer.Editor.Selection.SelectProgrammer()

Timecode.SetEnabled(Timecode.Enabled) (after setting Timecode.Enabled true or false by ActionScript)

Playbacks.FirePlaybackAtLevel("Location=Playbacks,1",Math.AbsoluteAdjust(1),true)
(Fire First Playback)

Playbacks.FirePlaybackAtLevel("Location=Playbacks,1,1",Math.AbsoluteAdjust(1),true)
(Fire First Playback Page 1)

Playbacks.FirePlaybackAtLevel("cueHandleUN=1",Math.AbsoluteAdjust(1),true)
(Fire Playback 1)

Playbacks.KillPlayback("Location=Playbacks,1") (Kill First Playback)

Playbacks.KillPlayback("Location=Playbacks,1,1") (Kill First Playback Page 1)

Playbacks.KillPlayback("cueHandleUN=1") (Kill Playback 1)

Playbacks.ReleasePlayback("Location=Playbacks,1",0,true) (Release First Playback)

Playbacks.ReleasePlayback("Location=Playbacks,1,1",0,true) (Release First Playback Page 1)

Playbacks.ReleasePlayback("cueHandleUN=1",0,true) (Release Playback 1)

Playbacks.ReleasePlayback(this,0,true) (Release Me)

CueLists.SetNextCue(this,1.0) (Goto My Cue 1)
CueLists.Play(this) (Goto My Cue 1)

Dmx.SetMergePriority(0) (sACN Priority 0)

Dmx.SetMergePriority(50) (sACN Priority 50) also with 75, 100, 125, …

UserMacros.SetCurrentMacroFromUserNumber(1) (see Gregory's example)

UserMacros.Export(UserMacros.CurrentMacroId) (see Gregory's example)

Panel.Midi.NoteOn(0,1,127) (Note 1 On, see Olie's example)

Panel.Midi.NoteOn(0,2,127) (Note 2 On)

Panel.Midi.NoteOff(0,1,127) (see Olie's example)

Handles.MobileWingAPlaybacks.NextPage() (found by S. Beutel)

Handles.MobileWingAPlaybacks.PreviousPage() (found by S. Beutel)

Handles.MobileWingAPlaybacks.ChangePage(0) (found by S. Beutel)

Handles.MobileWingAExecutor.NextPage() (found by S. Beutel)

Handles.MobileWingAExecutor.PreviousPage() (found by S. Beutel)

Handles.MobileWingAExecutor.ChangePage(0) (found by S. Beutel)

Handles.Macros.ChangePage(0) (found by S. Beutel)

Handles.Macros.PreviousPage() (found by S. Beutel)
(as of v9, decrements as expected – but you can go down to page 0 which is weird)

Handles.Macros.NextPage() (found by S. Beutel)
(as of v9, doesn't increment, but always goes to page 1. Olie: There is a max pages which is set to 1 and next page will always wrap around…)

Actionscript

ActionScript lets you evaluate and set certain properties of objects.

ActionScript.SetProperty.Enum("Handles.OperationMode","copy")

ActionScript.SetProperty.Enum("Handles.OperationMode","move")

ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Tiny")

ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Small")

ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Normal")

ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Large")

ActionScript.SetProperty.Enum("UserSettings.Setting.HandleButtonSize","Huge")

ActionScript.SetProperty.Float("Palette.MasterFadeTime",10)

ActionScript.SetProperty.Float("Palette.MasterFadeTime",5)

ActionScript.SetProperty.Float("Palette.MasterFadeTime",3)

ActionScript.SetProperty.Float("Palette.MasterFadeTime",2)

ActionScript.SetProperty.Float("Palette.MasterFadeTime",0.5)

ActionScript.SetProperty.Float("Palette.MasterFadeTime",0)

ActionScript.SetProperty.Float("Palette.MasterOverlap",100)

ActionScript.SetProperty.Float("Palette.MasterOverlap",50)

ActionScript.SetProperty.Float("Palette.MasterOverlap",0)

ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback",CueLists.ConnectedHandle)

ActionScript.SetProperty.Boolean("Selection.Context.Global.RepeatPattern", true)

ActionScript.SetProperty.Boolean("Timecode.Enabled", true)

ActionScript.SetProperty.Boolean("Timecode.Enabled", false)

ActionScript.SetProperty.Boolean("Timecode.Enabled", !Timecode.Enabled)

ActionScript.SetProperty("Handles.SourceHandle", Playbacks.Editor.SelectedPlayback) (see Examples)

ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", Chases.ConnectedHandle) (see Examples)

ActionScript.SetProperty.Float("Playbacks.Editor.Times.ChaseSpeed", ↵
Playbacks.Editor.Times.ChaseSpeed / 2)

ActionScript.SetProperty.Float("Playbacks.Editor.Times.ChaseSpeed", 0.0)

ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", Handles.SourceHandle) (see examples)

Conditions

These conditions are currently used for conditional steps:

Math.IsEnumEqual(UserSettings.TempoUnits, 'BPM')

Math.IsEnumEqual(UserSettings.TempoUnits, 'Seconds')

!ActionScript.Test.String.IsNullOrEmpty(Global.ActiveWindowId)

Math.IsLessThan(Playbacks.Editor.Times.ChaseSpeed, 1.0) (see examples)

Math.IsEqual(Playbacks.Editor.Times.ChaseSpeed, 0.0) (see examples)

Math

Playbacks.Editor.Times.ChaseSpeed * 2 (see examples)

Playbacks.Editor.Times.ChaseSpeed / 2 (see examples)

Math.Min("Playbacks.Editor.Times.ChaseSpeed", Playbacks.Editor.Times.ChaseSpeed * 2, 3600.0)

(Gregory: Since the Min function is older it takes three parameters, the first being a string with the name of the property where the result should be stored. http://forum.avolites.com/viewtopic.php?f=20&t=3744#p16031)

Gregory:

"…some simple mathematical operations are possible e.g. addition ("+"), subtraction ("-"), multiplication ("*") and division ("/") along with logical operators NOT ("!"), AND ("&" – note that the ampersand has to be escaped) and OR ("||"). There are also the standard relational operators as well."

User Macros – button actions

Macros which are recorded directly inside Titan are somewhat different since they record the button presses instead of the logic behind.

Menu.Stack.PushOrReloadMenu("Primary", "Expert.Chases.AppendStep")

Menu.InjectInput("OnButtonUp","FaderlessPlaybackSelect.0","StaticPlaybacks",0)

Menu.InjectInput("OnButtonUp","Softkey.2","NoGroup",2)

Menu.InjectInput("OnButtonDown","NextFixture.0","NoGroup",0)

Menu.InjectInput("OnValueChanged","Dmx","NoGroup",48)

(this is recording DMX input into a user macro. Functionality has been removed since v8)

Examples

From the library:

```
<macro name="Open Live Cue" id="Avolites.Macros.OpenLiveCue">
    <description>Shows the live cue of the connected cue list in the Cue View.</description>
    <sequence>
      <step>ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback",↵
            CueLists.ConnectedHandle)</step>
      <step>Playbacks.Editor.CueSelection.SelectCueByNumber(↵
            Playbacks.Editor.SelectedPlayback, CueLists.LiveCueNumber)</step>
      <step>
        <menuLink id="Windows.CueView" stack="mainWindowStack" behaviour="PushOrRaise"/>
      </step>
    </sequence>
  </macro>
```

By Gregory, http://forum.avolites.com/viewtopic.php?f=20&t=3744#p15848

"If you are interested this is what I came up for doubling and halving chase BPM.

The problem with this code is that it requires the chase to be set as the selected playback in the editor (used for the Edit Times menu). This means that the Playback View which could be showing a cue list switches to show the chase instead, to get around this I have stored the previous selected playback and restore that afterwards but this does cause the display to flicker."

```
<macro name="Chase Speed Double" id="Avolites.Macros.ChaseSpeedDouble">
  <description>Double the speed of the currently connected chase.</description>
  <sequence>
    <step>ActionScript.SetProperty("Handles.SourceHandle",↵
          Playbacks.Editor.SelectedPlayback)</step>
    <step>ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", ↵
          Chases.ConnectedHandle)</step>
    <step>Math.Min("Playbacks.Editor.Times.ChaseSpeed", ↵
          Playbacks.Editor.Times.ChaseSpeed * 2, 3600.0)</step>
    <step condition="Math.IsEqual(Playbacks.Editor.Times.ChaseSpeed, 0.0)">↵
          ActionScript.SetProperty.Float("Playbacks.Editor.Times.ChaseSpeed", 1.0)</step>
    <step>ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", ↵
          Handles.SourceHandle)</step>
  </sequence>
</macro>

<macro name="Chase Speed Half" id="Avolites.Macros.ChaseSpeedHalf">
  <description>Halves the speed of the currently connected chase.</description>
  <sequence>
    <step>ActionScript.SetProperty("Handles.SourceHandle", ↵
          Playbacks.Editor.SelectedPlayback)</step>
    <step>ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", ↵
          Chases.ConnectedHandle)</step>
    <step>ActionScript.SetProperty.Float("Playbacks.Editor.Times.ChaseSpeed",↵
          Playbacks.Editor.Times.ChaseSpeed / 2)</step>
    <step condition="Math.IsLessThan(Playbacks.Editor.Times.ChaseSpeed, 1.0)">↵
          ActionScript.SetProperty.Float("Playbacks.Editor.Times.ChaseSpeed", 0.0)</step>
    <step>ActionScript.SetProperty("Playbacks.Editor.SelectedPlayback", ↵
          Handles.SourceHandle)</step>
  </sequence>
</macro>
```

By Gregory, http://forum.avolites.com/viewtopic.php?f=20&t=4361#p15847

"I have written a macro which should allow you to export a macro from a show file to an XML file… It will export the macro with user number 1."

```
<macro id="Avolites.Macros.ExportMacro" name="Export Macro Number 1">
  <description>Exports the macro with user number 1.</description>
  <sequence>
    <step>UserMacros.SetCurrentMacroFromUserNumber(1)</step>
    <step>UserMacros.Export(UserMacros.CurrentMacroId)</step>
  </sequence>
</macro>
```

S. Beutel: "I can confirm this macro does exactly what it is supposed to do. It exports the macro #1 into the file %userprofile%\documents\Titan\Macros\Macro1.xml" – this translates to 'My Documents\Titan\Macros'.

Note that this macro works on consoles only from v10 on: it writes the macro to D:\Macros which was not possible on consoles until version 9.

By Olie, http://forum.avolites.com/viewtopic.php?f=20&t=4342

"Here are macros to send Note On and Off commands from the MIDI out port on the console. Notes 1 to 20 are supported."

```
<macro name="MIDI Note 1 On" id="Avolites.Macros.MidiNote1On">
  <sequence>
    <step>Panel.Midi.NoteOn(0,1,127)</step>
  </sequence>
</macro>

<macro name="MIDI Note 2 On" id="Avolites.Macros.MidiNote2On">
  <sequence>
    <step>Panel.Midi.NoteOn(0,2,127)</step>
  </sequence>
</macro>
...
<macro name="MIDI Note 1 Off" id="Avolites.Macros.MidiNote1Off">
  <sequence>
    <step>Panel.Midi.NoteOff(0,1,127)</step>
  </sequence>
</macro>

<macro name="MIDI Note 2 Off" id="Avolites.Macros.MidiNote2Off">
  <sequence>
    <step>Panel.Midi.NoteOff(0,2,127)</step>
  </sequence>
</macro>
```

(Originally all macros had the same ID so that only one macro would show up in the macro library. This is corrected here – you will get the idea…)