Recorded vs. coded macros

For quite many years/versions Avolites Titan allows for recording macros one the fly. This involves the <Macro> button, and records all the button presses you do, in exactly this order. This sequence can then be played back by just recalling the recorded macro. More information on this is available in the manual at page 67. There is the additional benefit of calling the macro with a number: the macro will then be executed the number of times you called it with, something like <5> [Macro X] will execute Macro X 5 times.

While this is of great help when quickly creating some shortcuts which you frequently need when programming, there are some caveats. You cannot edit a recorded macro, you do not even see what's recorded in the macro, and: the macro stores only button presses, regardless which menu, function or state this button is currently linked to.

In turn, coded macros do not call button presses, but call functions or operate on menus and values. Thus, the outcome is much more predictable. However, it requires some coding skills, and you always need to know the required functions.

Limitations

While macros already make a powerful tool, there is still lots of headroom. To spare you some time, here are some hints on things with are currently (Titan v10.1) **not possible** with macros:

- more user input (like parameters)
- loops (while, for, each, until)
- breakpoints (run until a certain point, wait for a trigger, then proceed)
- console output (would be helpful for debugging)

and most importantly

- **update macros while Titan is running**
- **edit macros within Titan**

However, not only this wiki, but also Titan is under constant development: let's see what the future might bring.

The Fix+1 example

In order to illustrate power, caveats, possibilities and weaknesses of the various kinds of macros, her comes an example: we want to create an on/off chaser on a bigger number of fixtures/lamps: imagine you had 24 par cans, patched as fixtures 1~24, and wanted one can lit, then the next, then the third… A - German - video to illustrate this is available at https://www.youtube.com/watch?v=smxprAfTNkI

manually recording

When programming this manually you would do like this:

1. <Record> <Record> {Select playback} // starts recording a chaser on a particular playback
2. <1> <@> <@> // selects fixture 1 and puts it at 100%
3. {Select playback} // records this step
4. <Clear> // clears programmer
5. <2> <@> <@> // selects fixture 2 and puts it at 100%
6. {Select playback} // records this step
7. <Clear> // clears programmer
8. .... // repeat with fixture 3..24
9. <Exit> // finish recording chaser

You will soon learn that this is prone to errors, pressing wrong buttons etc.

advanced manually recording

There are some little helper functions available: <Fix+1> advances through the selected fixtures, [Append Step] appends a step regardless where the chaser is currently recorded, and instead of <Clear> - which also deletes the fixture selection - you can set the fixture to 0 with <@> <0> <Enter>. You would do like this:

1. {Select group of fixtures}
2. <Record> <Record> {Select playback} // starts recording a chaser on a particular playback
3. <Fix+1> // selects next fixture
4. <@> <@> // puts fixture at 100%
5. [Append Step] // appends this as new chase step
6. <@> <0> <Enter> // puts fixture at 0%
7. {repeat steps 3~6} // until fixture 24 is reached
8. <Clear> // clears programmer
9. <Exit> // finish recording chaser

recording a macro

When there is something repeated then it makes sense to put this into a macro - in this case: steps 3~6. However, since the current state of the console might change, it is essential that the macro is recorded in the correct context. In this case: when recording a chase. You would do like this:

1. {Select group of fixtures}
2. <Record> <Record> {Select playback} // starts recording a chaser on a particular playback
3. <Macro> [Record] {Select button to store macro on} // starts recording the macro, macro LED is flashing
4. <Fix+1> // selects next fixture
5. <@> <@> // puts fixture at 100%

6. [Append Step] // appends this as new chase step
7. <@> <0> <Enter> // puts fixture at 0%
8. <Macro> // finish recording this macro - macro LED stops flashing
9. {macro button where macro is stored} // until fixture 24 is reached, and/or
10. <20> {macro button where macro is stored} // repeats macro 20 times
11. <Clear> // clears programmer
12. <Exit> // finish recording chaser

You will soon like this - in particular as you can re-use the macro. Next time you simply do

1. {Select group of fixtures}
2. <Record> <Record> {Select playback} // starts recording a chaser on a particular playback
3. <24> {Recall previously stored macro} // repeats macro 24 times
4. <Clear> // clears programmer
5. <Exit> // finish recording chaser

This way, programming a chase even of some dozen fixtures is merely a question of seconds.

Investigating the recorded macro

At some point you'll stumble across the Export Macro and you are curious enough to export the above chaser macro (give this the usernumber 1, retrieve and execute the export macro, and open the exported macro). The result will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<avolites.macros>
  <!-- Automatically exported from Unsaved Show - Titan Mobile 10.0.32.2
(SB-LENOVO) on 20/03/2016 11:24:49. -->
  <macro id="UserMacro.Macro1">
    <name>Macro1</name>
    <sequence>
      <step pause="0.001">Menu.Stack.PushOrReloadMenu("Primary",
"Expert.Chases.AppendStep")</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","FaderlessPlaybackSelect.0","Sta
ticPlaybacks",0)</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","NextFixture.0","NoGroup",0)</
step>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","NextFixture.0","NoGroup",0)</st
ep>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","KeypadAt.0","NoGroup",0)</ste
p>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","KeypadAt.0","NoGroup",0)</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","KeypadAt.0","NoGroup",0)</ste
```

```
p>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","KeypadAt.0","NoGroup",0)</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","Softkey.2","NoGroup",2)</step
>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","Softkey.2","NoGroup",2)</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","KeypadAt.0","NoGroup",0)</ste
p>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","KeypadAt.0","NoGroup",0)</step>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","NumericKeys.0","NoGroup",0)</
step>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","NumericKeys.0","NoGroup",0)</st
ep>
      <step
pause="0.001">Menu.InjectInput("OnButtonDown","KeypadEnter.0","NoGroup",0)</
step>
      <step
pause="0.001">Menu.InjectInput("OnButtonUp","KeypadEnter.0","NoGroup",0)</st
ep>
    </sequence>
  </macro>
</avolites.macros>
```

Again we only explain the functional steps within the sequence. (For all the other XML details please refer to Formats and syntax)

- all <step …> tags also contain a 'pause' property: pause="0.001". This is only a minor thing, and is discussed in Step Pause
- Menu.Stack.PushOrReloadMenu("Primary", "Expert.Chases.AppendStep") proves what was mentioned earlier: the macro refers to the state the software was in when the macro was recorded. in our example the macro was recorded in the 'Record Chase/Append Step' mmenu, thus Expert.Chases.AppendStep
- the next line Menu.InjectInput("OnButtonUp","FaderlessPlaybackSelect.0","StaticPlaybacks",0) is somewhat ophane and has something to do with how macros are recorded: the corresponding previous "OnButtonDown" actually triggered recording the macro - and now the key needs to be released again - hence this line is in the macro. I guess it does just nothing - very few actions are performed when a button is released.
- after this we have a series of Menu.InjectInput("OnButtonDown"… and subsequent Menu.InjectInput("OnButtonDown"… - again as mentioned earlier, the macro simply records key presses, and Menu.InjectInput is, by the way, the function to simply feign some input for Titan. The only interesting thing are the buttons which were pressed:
  - "NextFixture.0","NoGroup",0 is <Fix+1>
  - "KeypadAt.0","NoGroup",0 is <@>
  - "Softkey.2","NoGroup",2 is menubutton <B> which is **in this state** [Append

Step]
- “NumericKeys.0”,“NoGroup”,0 is the <0> key, and
- “KeypadEnter.0”,“NoGroup”,0 is the <Enter> key.

All we learn from this, for the moment, is that recorded macros really store button presses - nothing more, nothing less.

One little warning: DMX triggers are also regarded input which might make it into a recorded macro. **Never record macros with DMX input enabled! Else your macro will soon become really big, and the resulting behaviour can be unpredictable.** (I admit this info is back from version 7 or 8 and might have changed, though.)

Coding this macro?

Finally we try to code this macro (albeit I have to say that I really like recording this macro as an example 🙂). We go back to our advanced manually recording and try to find the suitable functions for these steps:

1. next fixture
2. at full (@@)
3. append step
4. at 0

1. “Next fixture” might work as `Selection.Context.Global.PatternNext()`
2. “at full” needs to be discussed separately - setting an attribute uses the function Programmer.Editor.Fixtures.SetControlValueById and goes along the lines Olie outlined in the forum. Another option is literally using @@, by `Command.RunCommand(“@@”)`, see Command.RunCommand
3. “append step” is the most tricky part: the function could be `Playbacks.AppendCue(Handle handle)` - but this would require us to know the handle
4. “at 0” → see above 2.

Conclusions

What have we learned from this?

- recorded macros are specific to the state the software is in
- sometimes a recorded macro might not only do the trick, but maybe the task is simply not to accomplish with code
- it's always a good idea to know how to program systematically: the first approach wouldn't fit for a macro at all

further readings

- Introduction to macros
- Console and simulator - how actions on the consoles are described
- Recorded vs. coded macros - both kinds: Country, AND Western
- Macro file format - what to observe when creating macro files

- Macro Folders - where exactly are the macro files stored
- Deploying macros - how to import a macro file into Titan
- XML format - a veeeery basic introduction into the format macro files are written in
- The Syntax of Functions - understanding how functions are described in general
- Control Structures - conditions and other means to control the flow
- Action and Menus - when a menu needs to be toggled in addition to the action
- Step Pause - a little delay might sometimes be helpful
- Active Binding - highlighting a macro handle as active
- Namespaces - a way to keep order of the functions, properties and other stuff
- Datatypes - numbers, words, yes & no: the various types of values
- Properties list - the affected system variables of Titan
- Function list - the functions mentioned in this wiki
- Examples list - all the contributed macros. And where is yours?

2017/10/13 15:12 · icke_siegen