LiVES C++ Language Bindings: - Programmable Video Control with LiVES

In this document I am going to explain some of the features of the LiVES C++ interface, which was first introduced in LiVES 2.4.0

(The full API is documented at http://lives-video.com/doxygen/liblives/)

Note: before beginning the tutorial, it is a good idea to run LiVES at least once normally. This is because the first time LiVES is run, it will go through a setup procedure in order to install itself. If you have not run LiVES on the system, you can either activate it from the menu or enter "lives" in a terminal. After the installation has completed you can exit and continue with this tutorial.

It will also be helpful if you open a couple of video clips in LiVES and save these as a Set before exiting.

Let's get started with a simple example. Open your preferred text editor and enter the following:

```
#include <liblives.hpp>

#include <unistd.h> // for sleep()
#include <iostream> // for cout

int main() {
        using namespace lives;
        using namespace std;

         livesApp *lives = new livesApp;

         sleep(40);

         set cset = lives->getSet();

         cout << "Set name is " << cset.name() << endl;

         sleep(20);

         delete lives;
}
```

Save this as hellolives.cpp.

Let's take a look at the code piece by piece.

```
#include <liblives.hpp>


#include <unistd.h>
#include <iostream>
```

You need to include the header file, liblives.hpp in every file which uses lives binding functions. This will define the “lives” namespace, which we can use here optionally, to avoid having to type lives:: everywhere.

Here we will also include unistd.h which defines the sleep() function we will use in this first example, and iostream so we can print some output.

```
int main() {
        using namespace std;
        using namespace lives;

        livesApp *lives = new livesApp;
```

Here we create a new livesApp. When the livesApp constructor is called, by default the LiVES application will start up as a desktop application, similar to if you typed “lives” from the commandline. We could create it more simply as: livesApp lives;
but here I want to show what happens when we delete it later on.

```
        sleep(40);

        set cset = lives->getSet();

        cout << “Set name is “ << cset.name() << endl;
```

The code above will wait 40 seconds, then get the current set name. If you have a set loaded in LiVES then the name will be shown, otherwise the name will be empty.

```
        sleep(20);

        delete lives;
}
```

Finally, we wait a further 20 seconds, the the livesApp is shut down cleanly. If the user has any clips open they will be saved as a set. The user will first be prompted for a set name if necessary.

How to compile:

First of all we need to build liblives. For this you will need the LiVES sources (either from a package or from subversion).

Then you need to configure the source with a special option:

*$ **./configure --enable-liblives***

then the usual:

*$ **make***

*$ **sudo make install***

by default this will install liblives.a in /usr/lib. (If you want to install in another location, eg. /usr/local/lib you can specify this on the configure line, eg.:

$ ./configure –prefix=/usr/local –enable-liblives

).

Then you need to compile your script, dynamically linking it with liblives and other libs (libgtk+, libpulse, etc). To do this you can simply run:

*$ **g**++ hellolives.cpp -**I/usr/include/liblives** -**llives** -**o** hellolives*

This will compile hello.cpp and create an executable called hellolives. We can make it runnable with:

*$ **chmod a+x** ./hellolives*

Note: of course you can add extra parameters to the compiler command, for example -*O3* to produce optimised code, or -*g* to produce output that can be debugged with gdb.

The code for this program can be found in the LiVES sourcecode in liblives/examples/hellolives.cpp.

Running the program

You can run your newly compiled program simply with:

$ ./hellolives

You should see the LiVES application open; you may be prompted to open a default set, if you saved one last time. After one minute, the application will shut down cleanly. If you have any open clips, the current set will be saved; lives will prompt you for a set name if necessary.

Note that *hellolives* is a standalone application, meaning you do not need to have the lives executable installed to run it. However you still need the lives plugins (video decoders etc), plus the various libraries (liblives, gtk+, libpulse, etc) which were linked in at compile time. The command:

$ ldd hellolives

should give a clue as to which libraries are necessary. (Replace *hellolives* with the name of your executable of course).

Doxygen Documentation

If you have Doxygen installed, then after running *make install* you should find doxygen files in

/usr/share/doc/lives/html/liblives/*

Next example: adding callbacks and info boxes

Now, it would be nice to get rid of all those sleep() calls in our example, and have things run nice and smooth. There are a couple of ways we can do this. The first is to add one or more callbacks. Callbacks are functions which are triggered when events happen in the LiVES application. There are various events which can be used, such as loading a new clip, a new set, or even synching on frame playback.

Suppose we want to get notified when LiVES startup is complete. We can do this by adding a callback for Mode Change. LiVES has two modes: clip edit mode and multitrack mode, and when the user or a script changes between these modes a Mode Change event is emitted. A Mode Change is also emitted when LiVES startup is finished.

Here is the example code to do this:

```
ready = false;

lives->addCallback(LIVES_NOTIFY_MODE_CHANGED, modeChangedCb, NULL);

while (!ready) sleep(1);
```

Here we added a function, modeChangedCb, which should be called when LiVES emits a Mode Changed event. The final parameter (NULL in this case) is void * data which can be passed to the callback.

Here is the callback function itself:

```
static bool ready;

bool modeChangedCb(livesApp *lives, modeChangedInfo *info, void *data) {
  ready = true;
  return false;
}
```

After our callback function has been called, LiVES will be ready for action. We can for example pop up a message box for the user:

```
lives->showInfo("Ready to play ?");
```

When the program is run, this will cause a message box to pop up for the user with the specified text. The code will pause here until the user clicks “OK”.

Next, we can play some video (assuming there is some loaded – otherwise nothing will happen):

```
 player p = lives->getPlayer();

 p.play();

 sleep(10);

 p.stop();
```

The complete code now looks like:

```
#include "liblives.hpp"

#include <unistd.h>
#include <iostream>

static int ready;

using namespace std;
using namespace lives;


bool modeChangedCb(livesApp *lives, modeChangedInfo *info, void *data) {
 ready = 1;
```

```
  return false;
}



int main() {

  livesApp *lives = new livesApp;

  set cset = lives->getSet();

  ready = 0;

  lives->addCallback(LIVES_CALLBACK_MODE_CHANGED, modeChangedCb, NULL);

  while (!ready) sleep(1);

  cout << "Set name is " << cset.name() << endl;

  lives->showInfo("Ready to play ?");

  player p = lives->getPlayer();

  p.play();

  sleep(10);

  p.stop();

  sleep(10);

  delete lives;

}
```

The code for this program can be found in the LiVES sourcecode in liblives/examples/simpleplayer.cpp.

Next steps

Here are some more things we can do:

- since we are creating “lives” in our main function, we don't need to allocate it. Just be aware that when it goes out of scope, the LiVES application will quit.

- if LiVES has open clips, we can save them as a set and start with a blank slate:

```
 cset.save(“”);
```

to let the user choose the name via the GUI,

or we can provide the name:

```
if (! cset.name().empty()  ) cset.save( cset.name() );
else cset.save(“newsetname”);
```

[Note that “newsetname” must not be the name of an existing set, otherwise the command will fail, unless you do lives.getSet()->save(“newsetname”, true); which will force the current clips to be appended to Set “newset”] (See availableSets()).

After this the app should be empty of clips. We can verify this with:

```
int nclips = cset.numClips();
cout << “lives has “ << nclips << “ clips open.” << endl;
```

Indeed, nclips should be zero.

Opening new files

Now let's take a look at opening new files. First of all we can let the user choose a file to open:

```
livesApp lives;
```

```
livesString dirname = prefs::currentVideoLoadDir(lives);
livesString filename = lives.chooseFileWithPreview(dirname,
        LIVES_FILE_CHOOSER_VIDEO_AUDIO);
```

then we can try to open it:

```
if (! filename.empty() ) {
        clip c = lives.openFile(filename);
        if ( c.isValid() )
                cout << "got a file with " << c.frames() << " frames" << endl;
}
```

A Note on Interactivity

So far the kinds of scripts we have been looking at have been fairly simple, and if you were writing them for yourself there would be no problem running them.

But suppose you are making a script for somebody else. You might not want them clicking around the menus and opening and closing files themselves while you script is running.

To prevent this, you can start LiVES in non-interactive mode. This means that the menus will be hidden and keyboard accelerators will be blocked, and various button disable, for example the Cancel button during file loading.

To enable this from the start you can pass in a special option when you create the livesApp object, just as if you were passing a commandline option to the LiVES application.

Here is an example:

```
#include "liblives.hpp"
#include <unistd.h>
#include <iostream>

int main() {
 using namespace std;
 using namespace lives;
 char *argv[2];
 argv[0]="-noninteractive";
 argv[1]=”-noset”;
livesApp lives(2, argv);
```

You can see here we start LiVES with two options: “-noninteractive”, which disables interactivity, and “-noset” which will force LiVES to start with no clips loaded. As an alternative to the modeChanged callback we looked at before, there is another way to wait for LiVES startup:

```
while (lives.status() != LIVES_STATUS_READY) sleep(1);
```

It is also possible to check the value of lives.isReady() to see when that becomes true.

And, although we started with no clips because of the “-noset” argument, there is one condition we need to take into consideration. If the last run of LiVES crashed for some reason the user will be asked if they want to recover the last clips. You should give them a chance to do so and then save the recovered clips as a set:

```
set cset = lives.getSet();
cset.save("");
```

This will do nothing if there are no clips open (the usual case), but if clips were recovered from a prior crash, then the user can choose a name and those clips will be saved as a set.

Finally, after running a script, you might want to let the user make their own adjustments:

```
lives.setInteractive(true);
```

And lastly, instead of exiting straight away and thus forcing LiVES to shut down, it is possible to wait for the user to click on Quit themselves, either by:

```
while (lives.isValid()) sleep(1);
```

or:

```
while (lives.status() != LIVES_STATUS_INVALID) sleep(1);
```

or, alternately, by adding a callback for an appQuit event:

```
finished = false;
lives.addCallback(LIVES_CALLBACK_APP_QUIT, quit_callback, NULL);
while (!finished) sleep(1);
```

and setting *finished* to **true** in the callback.

So, a standard script might look something like the following:

```
#include "liblives.hpp"
#include <unistd.h>

int main() {
  using namespace lives;
  char *argv[2];
  argv[0]="-noninteractive";
  argv[1]="-noset";
  livesApp lives(2, argv);
  while (lives.status() != LIVES_STATUS_READY) sleep(1);
  set cset = lives.getSet();
  cset.save(“”);

  // rest of code goes here....

  // done...

// optional to allow user to Quit by themselves:
  lives.setInteractive(true);
  while (lives.isValid()) sleep (1);
  return 0;
}
```

Reloading a clip set

So, now that we have seen how to save clip sets, how about reloading them ? That is easily achieved.

```
livesString sn=lives.chooseSet();
lives.reloadSet(sn);
```

The first line lets the user choose a set from a list of available sets, and the second line loads it.

Applying Realtime Effects

One of the most fun things to do with LiVES is to apply realtime effects during playback. In this example we will start by playing two clips simultaneously, overlaying one on the other.

First we need to make sure we are in clip edit mode, and have at least 2 clips open:

```
lives.setMode(LIVES_INTERFACE_MODE_CLIPEDIT);
set cset = lives.getSet();

if (cset.numClips() < 2) {
```

```
  lives.showInfo("We need at least two clips loaded for this demo.\nLet's load some more.");


  while (cset.numClips() < 2) {
    livesString fname = lives.chooseFileWithPreview(prefs::currentVideoLoadDir(lives),
LIVES_FILE_CHOOSER_VIDEO_AUDIO);
    lives.openFile(fname);
  }


}
```

Now let's get the first two clips, and switch to the first of these.

```
clip clip1 = cset.nthClip(0);
clip clip2 = cset.nthClip(1);


// set clip 1 as our foreground, clip 2 as our background
clip1.switchTo();
```

Now we are going to clear the effect mapping, and map an overlay (transition to effectKey 1, mode 0).

```
effectKeyMap fxmap = lives.getEffectKeyMap();
```

```
fxmap.clear(); // clear the default mapping

effect chromablend(lives, "", "chroma blend", "salsaman");   // get an instance of chroma blend effect

fxmap[1].appendMapping(chromablend);  // map chromablend to effectKey 1

// enable the effect and start playing
fxmap[1].setEnabled(true);
```

Next we will set clip2 as our background (setting background only works if we have a transition effect enabled, which we now do).

```
clip2.setIsBackground();

player p = lives.getPlayer();

p.play();
```

The code for this program can be found in the LiVES sourcecode in liblives/examples/simpleplayer.cpp.

Playing in a Separate Window

The player object has the capacity to play in a separate, detached window and also to play fullscreen.

Setting both modes (separate window, fullscreen) will result in activation of the current video playback plugin. [pref TODO] Here is an example of such playback:

```
player p = lives.getPlayer();

p.setFS(true);

p.play();

// wait for pb to start
while (lives.status() != LIVES_STATUS_PLAYING) sleep(1);
// we can also use: while (!lives.isPlaying() ) sleep(1);

// wait for pb to stop
while (lives.status() == LIVES_STATUS_PLAYING) sleep(1);
// we can also use:  while (lives.isPlaying() ) sleep(1);

p.setFS(false);
```

Note that we first need to wait for playback to start (as it is not triggered instantly by player::play()), and then wait for playback to finish.

Multitrack Mode

Before going to look at some more features of the Clip Editor, let us take a quick glance at Multitrack mode.

To switch modes from Clip Editor to Multitrack, and vice-versa we can use:

```
lives.setMode(LIVES_INTERFACE_MODE_MULTITRACK);
```

and:

```
lives.setMode(LIVES_INTERFACE_MODE_CLIP_EDITOR);
```

Actually, changing to multitrack mode we can supply more parameters (e.g. frame width and height and framerate, as well as audio details). If these are not specified then either the user will be prompted for these, or else defaults will be used [pref TODO].

Assuming we have at least one clip open, we can insert frames into the Multitrack layout:

```
clip c = lives.getSet().nthClip(0);
lives.setMode(LIVES_INTERFACE_MODE_MULTITRACK);
multitrack mt = lives.getMultitrack();
mt.setCurrentTrack(0);
```

```
mt.setCurrentTime(0.);

block b = mt.insertBlock(c);

cout << “inserted a block from clip “ << c.name() << “ in track “ <<  \
mt.trackLabel( mt.currentTrack() ) << “ at time “ << b.startTime() << “ duration “ << b.length() << \
endl;

lives.getPlayer().play();

while (lives.isPlaying() ) sleep(1);

mt.wipeLayout(true);
```

The code for this program can be found in the LiVES sourcecode in liblives/examples/multitrack1.cpp.