- Audio Filter Graphs Using Gstreamer -

Goal:
During this exercise we want to create some filter graphs using the Gstreamer framework. Also we want to develop a stand-alone player program.

Exercise 9.1: Preparation for GStreamer

Check / install on host and on target the following packages:

- **`libgstreamer0.10`**
- **`libgstreamer0.10-dev`**
- **`gstreamer0.10-plugins-good`** und **`-bad`** (Codecs to play mp3,etc.)
- **`gstreamer0.10-tools`**

Now use **`gst-inspect-0.10`** to examine existing filters and verify that we have off- and mp3 codecs installed.

Exercise 9.2: Up and Running

Create your first filter graph to test the audio chain of your systems using the CLI tool gst-launch (hint: single line! Otherwise use “\” as a line separator):

```
gst-launch-0.10 audiotestsrc ! audioconvert ! audioresample !
alsasink
```

This should work on both host and target. Check, that your headphones are plugged in correctly.

Exercise 9.3: Simple Decoding

Now create a simple filter graph to decode and playback an ogg/vorbis audio file:

```
gst-launch-0.10 filesrc location=<my_file.ogg> ! oggdemux !
vorbisdec ! audioconvert ! audioresample ! alsasink
```

Can you propose any simplifications? You might consult the gstreamer documentation.

Exercise 9.4: Ripping / Transcoding of an audio file

Now, we transcode an mp3 audio file into an ogg/vorbis file using:

```
gst-launch-0.10 filesrc location=<my_file.mp3> ! decodebin !
audioconvert ! vorbisenc ! oggmux ! filesink
location=<my_file.ogg>
```

Compare the runtime on host and on target when transcoding the same file.

Exercise 9.5: Ripping and Playback in parallel
Now we want to playback and transcode (rip) at the same time. This can be accomplished by the following pipeline:

```
gst-launch-0.10 filesrc location=<my_file.mp3> ! decodebin !
tee name=t ! queue ! audioconvert ! vorbisenc ! oggmux !
filesink location=<my_file.ogg> t. ! queue ! audioconvert !
audioresample ! alsasink
```

Exercise 9.6: Our own stand-alone playback program (Host version)
The CLI-based approach is completey useless for any serious stand-alone applications. For this, we need a compiled executable.
Create a C project in Eclipse and copy the code of `gstreamer01_uri.c .` To be ready to build, you first need some additional parameters in Eclipse. To find out, which ones are required, you can use the utility:

```
pkg-config --cflags --libs gstreamer-0.10
```

Insert the appropriate details into the tool configuration of your Eclipse project: In the project view open the project properties with ALT ENTER. Then → C/C++ Build → Settings → GCC-C-Compiler → Includes → Include paths. Then add the libraries in the GCC C-linker panel. On http://webuser.hs-furtwangen.de/~coe you also find some Ogg files to test your program. Make sure to add an appropriate URI as a parameter in your run configuration. Work yourself through the source code and understand its mechanisms. These will be needed in the next exercise..
Hint: additional information and a nice gstreamer tutorial you can find on
http://docs.gstreamer.com

Exercise 9.7: Your own playback-programm (Target version)
Now we want to cross compile our program `gstreamer01_uri.c` in Eclipse and then we remote execute it on the target using an appropriate remote run configuration.

Hint: why does the Linker option `sysroot`, known from the last exercise, not work immediately? Apparently, the linker needs an additional library. Find out, which one, by analyzing the linker output in Eclipse (you also might want to check the Internet). After adding this library into the linker parameters (at the correct position!), cross compiling will work again as usual.