

Using FAUST with ROS

(version 0.0.07)

GRAMME
Centre National de Création Musicale

December 2014

Contents

1	Introduction	5
1.1	FAUST	5
1.1.1	Design Principles	5
1.1.2	Signal Processor Semantic	6
1.2	ROS	6
1.2.1	What is it ?	6
1.2.2	Concepts	7
1.3	Using FAUST with ROS	9
1.4	Audio Server	10
2	Compiling <i>FAUST</i> Program for <i>ROS</i> Use	11
2.1	Compiling in a FAUST Archive	11
2.2	Compiling in a Workspace	12
2.3	Example	13
3	Using <i>FAUST</i> Nodes	15
3.1	Run the Master	15
3.2	Run a FAUST Node	16
3.3	To Which Topics is a FAUST Node Subscribing ?	16
3.4	How to Escape from a Running Node ?	18
4	Metadata	19
4.1	DSP writing	19
4.2	Compilation	20
4.3	Run	20

5	Common Error Messages	21
5.1	The command does not output anything	21
5.2	No such file or directory	21
5.3	[roslaunch] error	21

Chapter 1

Introduction

FAUST (*Functional Audio Stream*) is a functional programming language specifically designed for real-time signal processing and synthesis. FAUST targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.

ROS (*Robot Operating System*) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

1.1 FAUST

1.1.1 Design Principles

Various principles have guided the design of FAUST:

- FAUST is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. FAUST is, as much as possible, free from implementation details.
- FAUST programs are fully compiled, not interpreted. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.
- The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn't depend of any DSP library or runtime system. It has a very deterministic behavior and a constant memory footprint.
- The semantic of FAUST is simple and well defined. This is not just of academic interest. It allows the FAUST compiler to be *semantically driven*. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal performance.

- FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (`:`, `~`, `<:`, `:`, `>`).
- Thanks to the notion of *architecture*, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the FAUST code.

1.1.2 Signal Processor Semantic

A FAUST program describes a *signal processor*. The role of a *signal processor* is to transform a group of (possibly empty) *input signals* in order to produce a group of (possibly empty) *output signals*. Most audio equipments can be modeled as *signal processors*. They have audio inputs, audio outputs as well as control signals interfaced with sliders, knobs, vu-meters, etc...

For more informations about FAUST, please see *faust-quick-reference.pdf* and the tutorials in FAUST documentation.

1.2 ROS

1.2.1 What is it ?

This section's content (1.2 ROS) is taken from ROS documentation. It can be found on [ROS official website](#) and [ROS wiki](#).

Creating truly robust, general-purpose robot software is *hard*. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

As a result, ROS was built from the ground up to encourage *collaborative* robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work, as is described throughout this site.

1.2.2 Concepts

Filesystem level

The filesystem level concepts mainly cover ROS resources that you encounter on disk, such as:

- **Packages** are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.
- **Metapackages** are specialized Packages which only serve to represent a group of related other packages.
- **Services** : Service descriptions, stored in `my_package/srv/MyServiceType.srv`, define the request and response data structures for [services](#) in ROS.
- **Messages** : Message descriptions, stored in `my_package/msg/MyMessageType.msg`, define the data structures for [messages](#) sent in ROS.

Computation Graph level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

- **Master** : The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Nodes** : Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client [library](#), such as [roscpp](#) or [rospy](#).
- **Topics** : Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given [topic](#). The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

- **The Parameter Server :** The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- **Messages :** Nodes communicate with each other by passing [messages](#). A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structures).

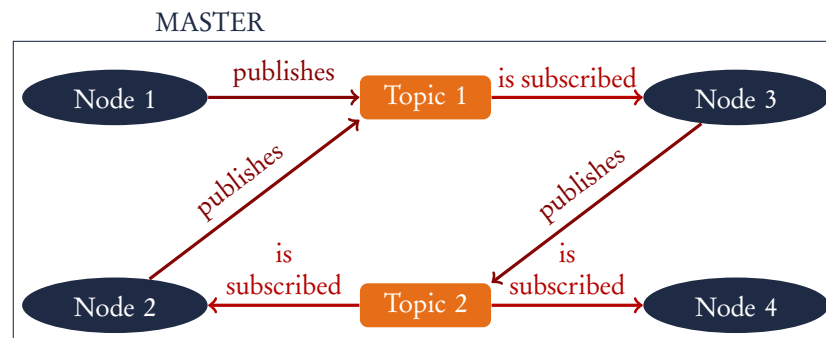


Figure 1.1: ROS Concepts in a Diagram

Names

Names are really important in ROS. Valid names have these characteristics :

- first character is an alpha character : `[a-z][A-Z]`
- subsequent characters can be alphanumeric : `[a-z][A-Z][0-9]`, underscores : `_` or forward slash : `/`
- there is at most one forward slash : `/`

For more informations on ROS and tutorials, please have a look to the website : www.wiki.ros.org.

1.3 Using *FAUST* with *ROS*

The idea of using FAUST modules with ROS could be summed up in the following diagrams.

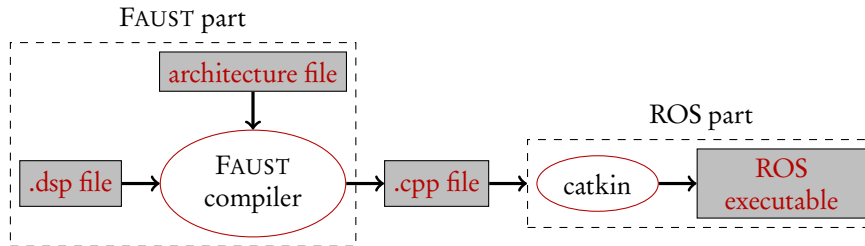


Figure 1.2: Compilation process

As shown on figure 1.2, the dsp file is compiled into a C++ file thanks to the FAUST compiler. Then, the C++ file can be compiled with *catkin* in a ROS package to create a ROS executable, that you can run with [roslaunch](#).

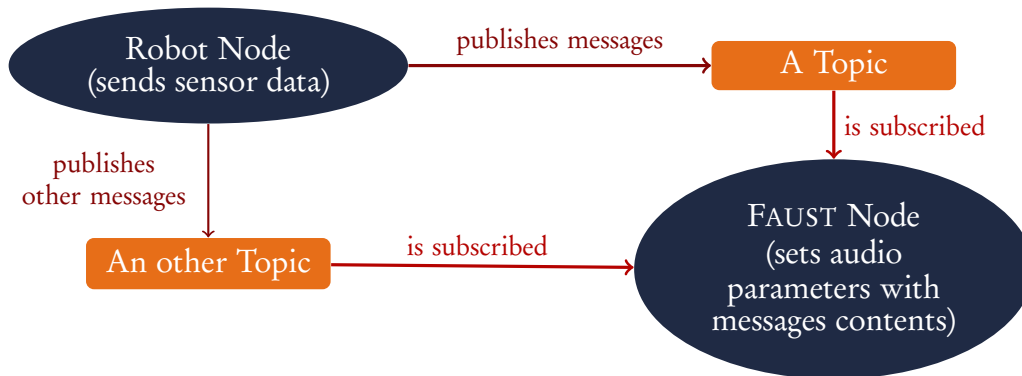


Figure 1.3: Robot using FAUST

Once the executables coming from DSP files compiled, you can run and combine them with robotic applications (figure 1.3).

1.4 Audio Server

FAUST applications use the jack audio server. Make sure it is installed on your machine.

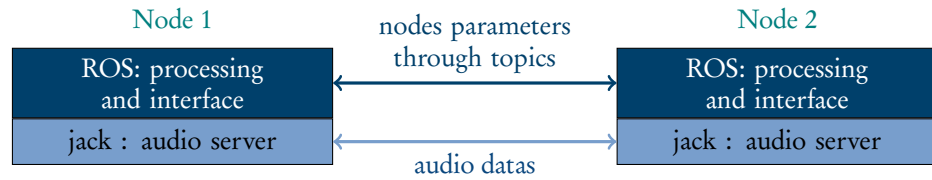


Figure 1.4: APIs used by FAUST nodes

Chapter 2

Compiling *FAUST* Program for *ROS* Use

To compile a FAUST program for a ROS use, you can use either the `faust2ros` command, or the `faust2rosgtk` one, which adds a gtk graphic user interface to the simple `faust2ros` command. Note that all the FAUST compilation options remain.

BE CAREFUL ! To run these commands, you need to have ROS installed on your machine. They are indeed using `catkin_make` and `roslaunch`, which are ROS commands.

2.1 Compiling in a *FAUST* Archive

In order to compile a DSP file into a FAUST archive, just type the command followed by your file :

```
faust2ros file.dsp
```

It should output :

```
file.zip;
```

and the resulting `file.zip` folder should contain a package called `file`, which contains a `.cpp` file corresponding to the DSP file.

If the DSP file is not in the current directory, make sure to type the right path. For instance :

```
faust2ros ~/faust/examples/myfile.dsp
```

Comments:

- If you want to use the `faust2rosgtk` command, the output will have a `_gtk` extension. For instance :

```
faust2rosgtk file.dsp
```

should output :

```
file_gtk.zip;
```

- The zip file is located in the current directory.

2.2 Compiling in a Workspace

Thanks to the option `-install`, you have the possibility to create a package from your DSP file directly in a workspace you choose. Just type :

```
faust2ros -install faust_ws file.dsp
```

It should output :

```
file.cpp;
```

and you should have a `faust_ws` repository looking like this :

```
faust_ws
├── build
├── devel
├── src
│   └── file : File Package
│       ├── include
│       ├── src
│       │   └── file.cpp : File generated with the Faust compiler
│       ├── CMakeLists.txt
│       └── package.xml
```

2.3 Example

Here is an example of a three files compilation.

Input :

```
faust2rosgtk -install foo_ws -o foo1 file1.dsp
              -install foo_ws -o foo2 file2.dsp
              -install bar_ws -o bar file3.dsp
```

Output :

```
~
├── foo_ws
│   ├── foo1
│   └── foo2
└── bar_ws
    └── bar
```


Chapter 3

Using *FAUST* Nodes

Once your DSP files are compiled into ROS executables, you can run them into a ROS master.

3.1 Run the Master

A FAUST node needs a master to run. You can check if a master is already running by typing :

```
rostopic list
```

Then, there are two possibilities :

- either you get the following message :

```
ERROR: Unable to communicate with master!
```

which means there is no master running

- or you get :

```
/rosout  
/rosout_agg
```

which means a master is already running.

To run a master, you have to type the following command :

```
roscore
```

3.2 Run a *FAUST* Node

Now that your master is running, you can run your FAUST node. It is quite simple. Type :

```
roslaunch mynodepackage mynode
```

For instance, if your node name is *foo*, then type :

```
roslaunch foo foo
```

If you get an error message looking like this :

```
[roslaunch] Couldn't find executable named foo below /path/to/myworkspace/src/foo
```

then refer to section 5.3

3.3 To Which Topics is a *FAUST* Node Subscribing ?

Once your FAUST node is running, it automatically subscribes to topics corresponding to the parameters you can modify, and to the widgets the gtk graphic interface has. For instance, if you use a FAUST node generated from the *noise.dsp* file (in the *examples* directory), the *noise_gtk* node will subscribe to the topic `noise_gtk/Volume` and the graphic interface will look like this :

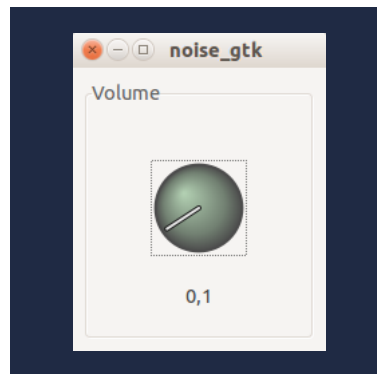


Figure 3.1: *noise_gtk* graphic interface

A more complex example like the harpe.dsp file, which contains three widgets, can generate several topics to subscribe to :

```
/harpe_gtk/attenuation
/harpe_gtk/hand
/harpe_gtk/level
```

and the graphic interface can look like this :

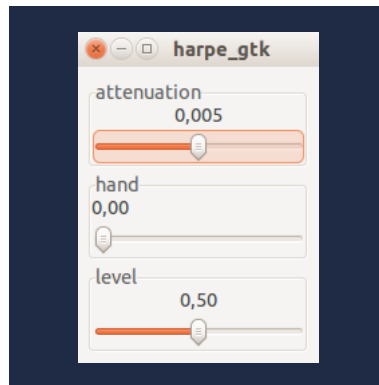


Figure 3.2: harpe_gtk graphic interface

If you want to change the topic name, just remap them while running your node :

```
roslaunch myfaustpackage myfaustnode /topicname:=/
newtopicname
```

For instance, to remap the /harpe/hand topic to /play, then run the harpe node like this :

```
roslaunch harpe harpe /harpe/hand:=/play
```

Comment: The FAUST nodes subscribe to topics using std_msgs message types. Depending on the widgets you use, you can subscribe to default topics using either Float32 or Bool messages.

Widget	Message type
Button	std_msgs/Bool
Check Button	std_msgs/Bool
Slider	std_msgs/Float32
Num. Entry	std_msgs/Float32

3.4 How to Escape from a Running Node ?

To close a node running in ROS, you have two possibilities, depending on the graphic interface :

- If your node has a graphic interface, then quit by clicking on the red cross in the corner of the window.
- If your node does not have any graphic interface, then quit by typing Ctrl+C in the node's terminal window.

Chapter 4

Metadata

You might not want to use the float or bool standard messages for your topic, and compile the dsp file directly with the right topic name. In order to accomplish this, you can use ROS metadata.

4.1 DSP writing

It all starts in the widgets definition. Until now, maybe that you only wrote :

```
param = hslider("level", etc);
```

To use ROS metadata, you simply have to add square brackets with your topic parameters :

```
param = hslider("level [ros:/my/topic/name msg_type  
msg_name field_name]", etc);
```

As an example, if you intend to use integers in a foo/bar/baz topic, you can type :

```
param = hslider("level [foo/bar/baz std_msgs Int32 data]  
", etc);
```

And if you intend to use the y field of a Point32 geometry_msg, then type :

```
param = hslider("level [foo/bar/baz geometry_msgs  
Point32 y]", etc);
```

4.2 Compilation

To compile your dsp file, just do like you used to do before to find out this wonderful chapter about metadata : `faust2ros` or `faust2rosgtk`. It will add a `RosCallbacks` class in your C++ file, containing specific callbacks.

Comment : Even without any declared ROS metadata, a `rosCallbacks` class is created, but it does not contain any callback implementation. It is just an empty class.

You can then build your executable using `catkin_make`.

4.3 Run

To run your node, just do as usual, using `roslaunch` or a launch file you wrote. The FAUST node creates two kinds of topics :

- Default topics, using Float32 and Bool standard messages
- Customized topics, created from ROS metadata.

Chapter 5

Common Error Messages

Compiling can fail. Here are some common mistakes and how to solve them.

5.1 The command does not output anything

If, after typing your command followed by a file name, your terminal does not output anything like `myfile.zip`; or `myfile.cpp`; and returns only a blank line, make sure you are in the correct directory or you entered the correct path to reach the DSP file.

5.2 No such file or directory

If you used the `-install` option, make sure you typed the complete workspace path. For instance, instead of typing this :

```
faust2ros -install myworkspace ~/path/to/myfile.dsp
```

you should type :

```
faust2ros -install path/to/myworkspace ~/path/to/myfile.dsp
```

5.3 [roswin] error

If, while trying to run a FAUST node (called *myname*), an error message showed up saying :

```
[roswin] Couldn't find executable named myname below /  
path/to/myworkspace/src/myname
```

Then you have to source your workspace :

- If your workspace is only a test workspace, then type :

```
source path/to/myworkspace/devel/setup.bash
```

in your terminal.

- If your workspace is going to be your current ROS workspace, you can add it to the source directories :

```
echo "source path/to/myworkspace/devel/setup.bash" >>  
~/.bashrc
```