Elk Workshop ADC ‘22

Hassle-Free Embedded Development on Your Computer

Elk Workshop ADC ‘22

Workshop repository: github.com/elk-audio/adc-22.git

Docs	elk-audio.github.io/elk-docs
Code	github.com/elk-audio
Community Plugins	elk-audio.github.io/elk-community
Website	elk.audio
Forum	forum.elk.audio

Outline - First part

Elk Audio Introduction (talk)

Demonstrations:

1. Set up a chain of plugins
2. Write a control application that uses SUSHI's gRPC API
3. Implement the control of your embedded device using
   a. Physical controls
   b. Remote GUIs
   c. End-user development tools
4. Use additional tools to monitor performance and problems
5. Q&A

Outline - Second part

Work independently, with our assistance

Using the same tools we demonstrated

Create a prototype of an embedded audio device:

- A simple synthesiser
- Or stompbox pedal

A few Elk hardware units will be available if you want to run your experiments on real hardware

Requirements

Laptop

- macOS (10.15 or later)
- Linux with a recent distribution and JACK audio server

Basic knowledge of one of these two languages:

- Python (recommended)
- C++

Requirements

Optional

- Small MIDI controller
- JUCE dev environment
- Elk Audio OS SDK - if you want to cross-compile

Elk
Audio OS

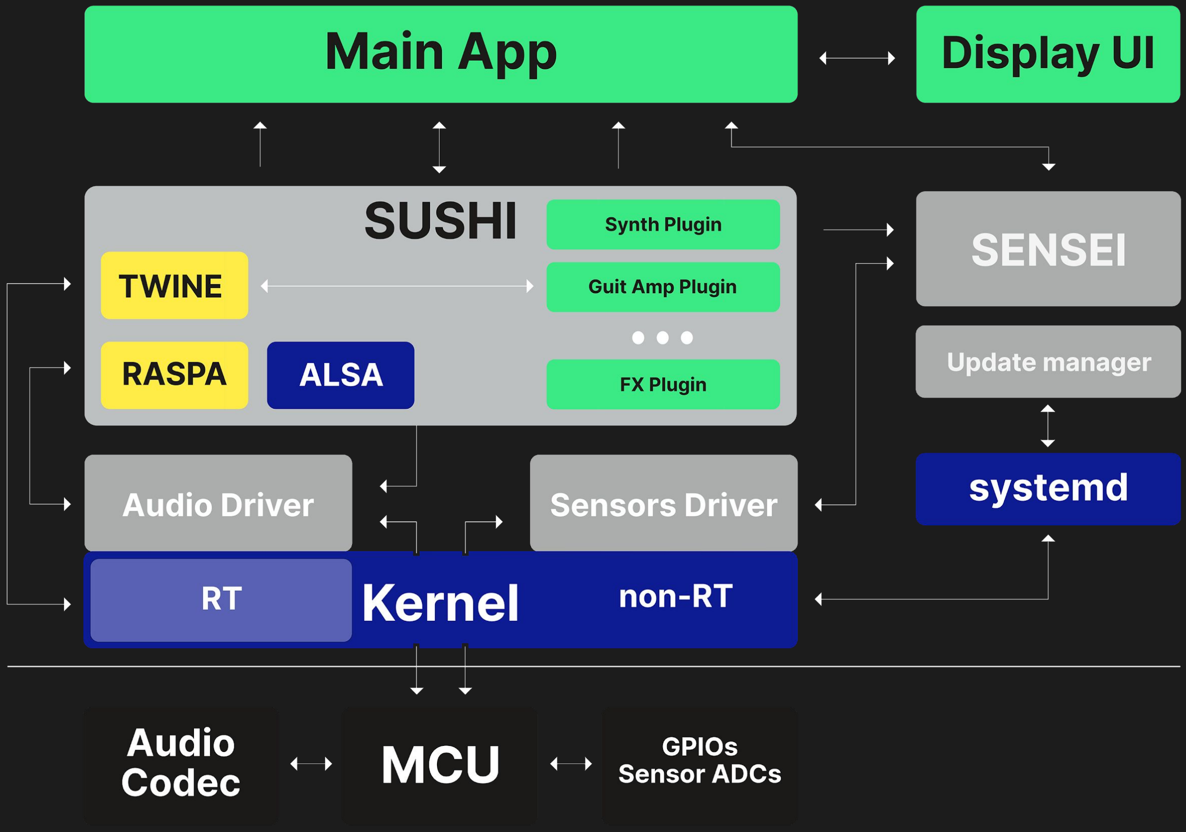
Technology

Custom Linux Distribution
Xenomai realtime Kernel
ARM & x86

Connectivity: WiFi, BLE
Plugin support: VST2, VST3, LV2
CV & Gate I/O
Ableton Link

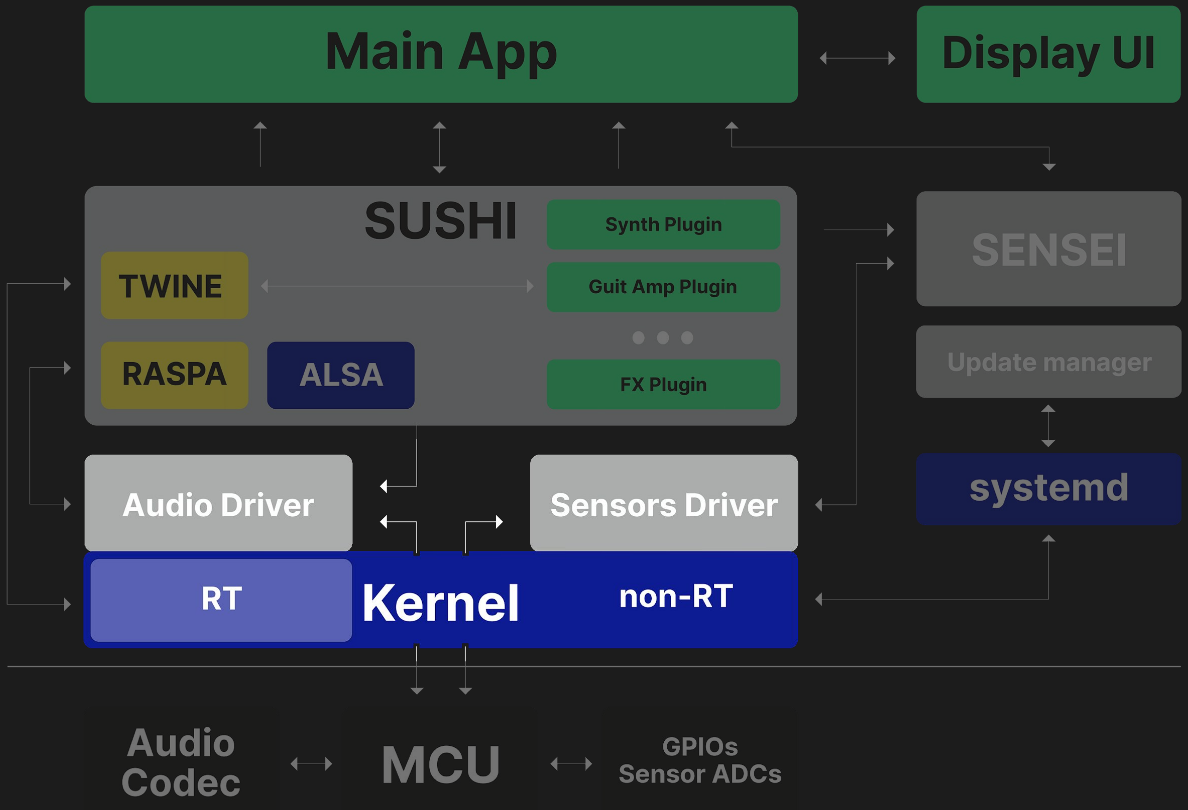
Architecture

Architecture



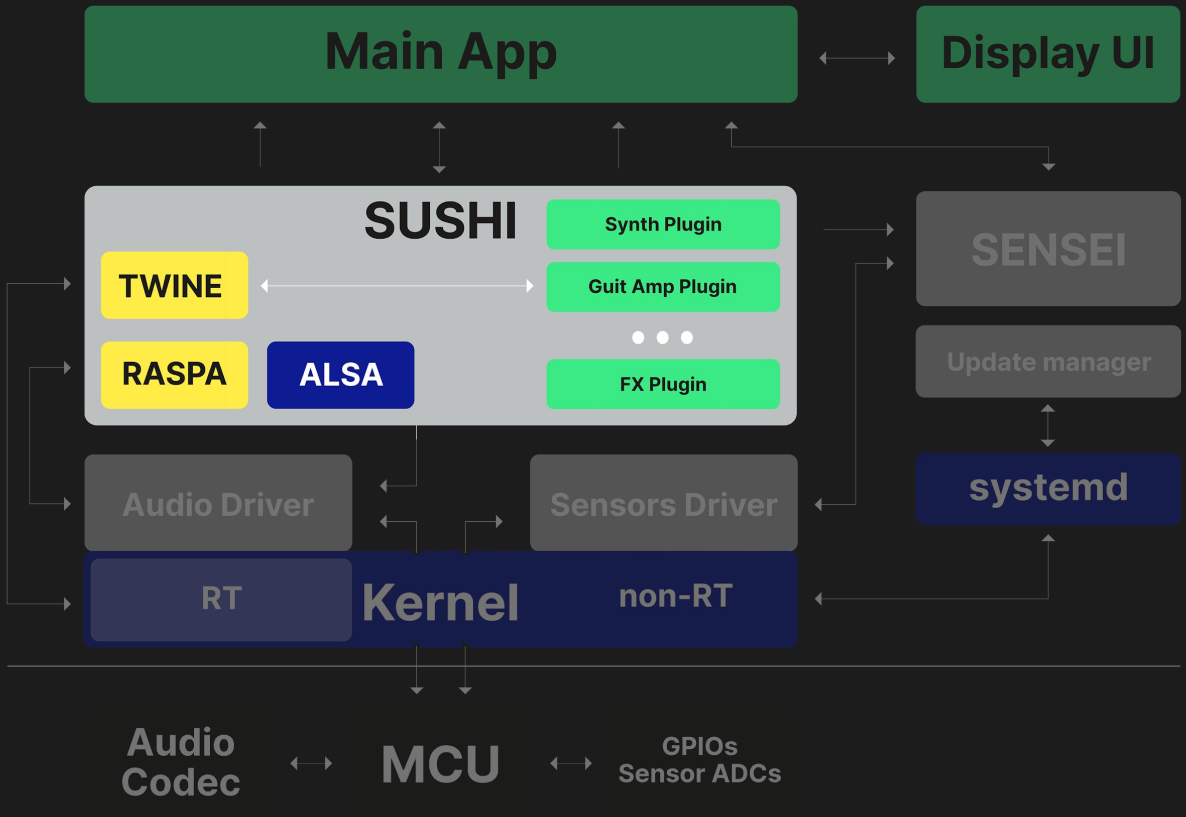
Architecture

Dual Kernel



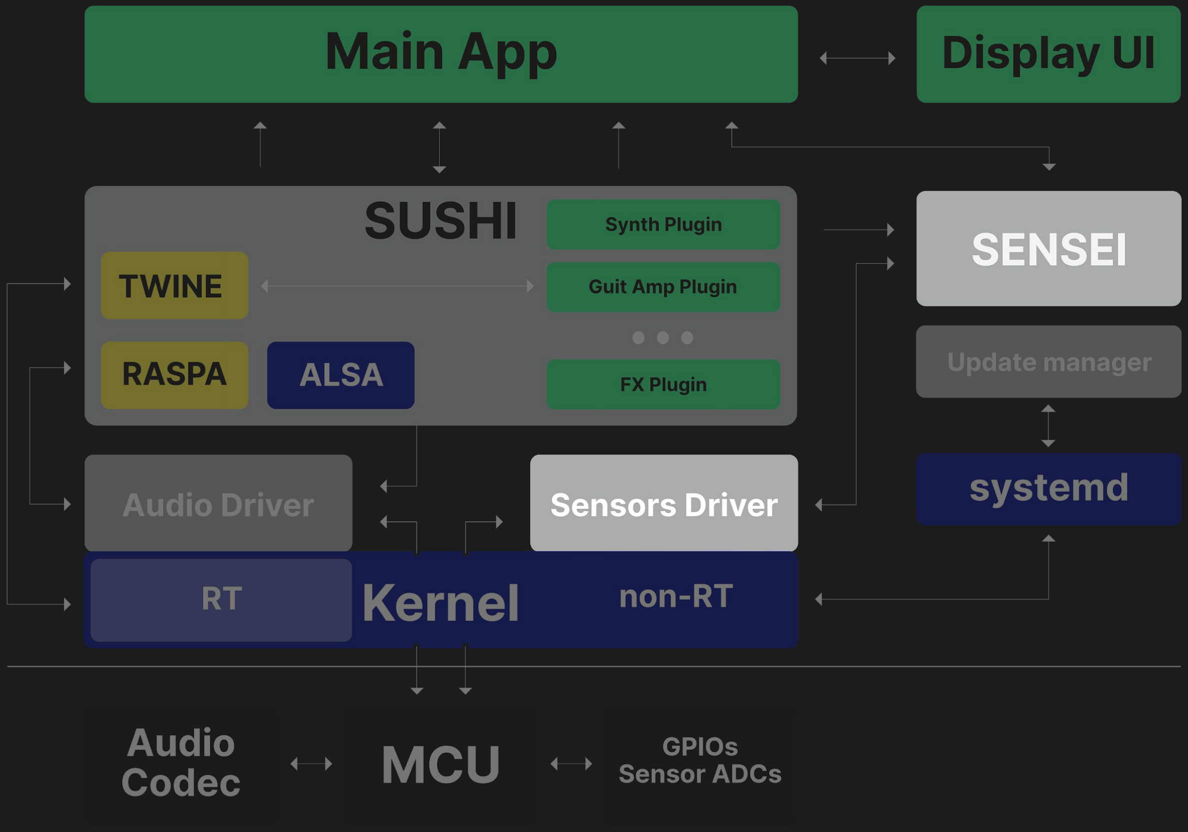
Architecture

Dual Kernel
Plugin Host



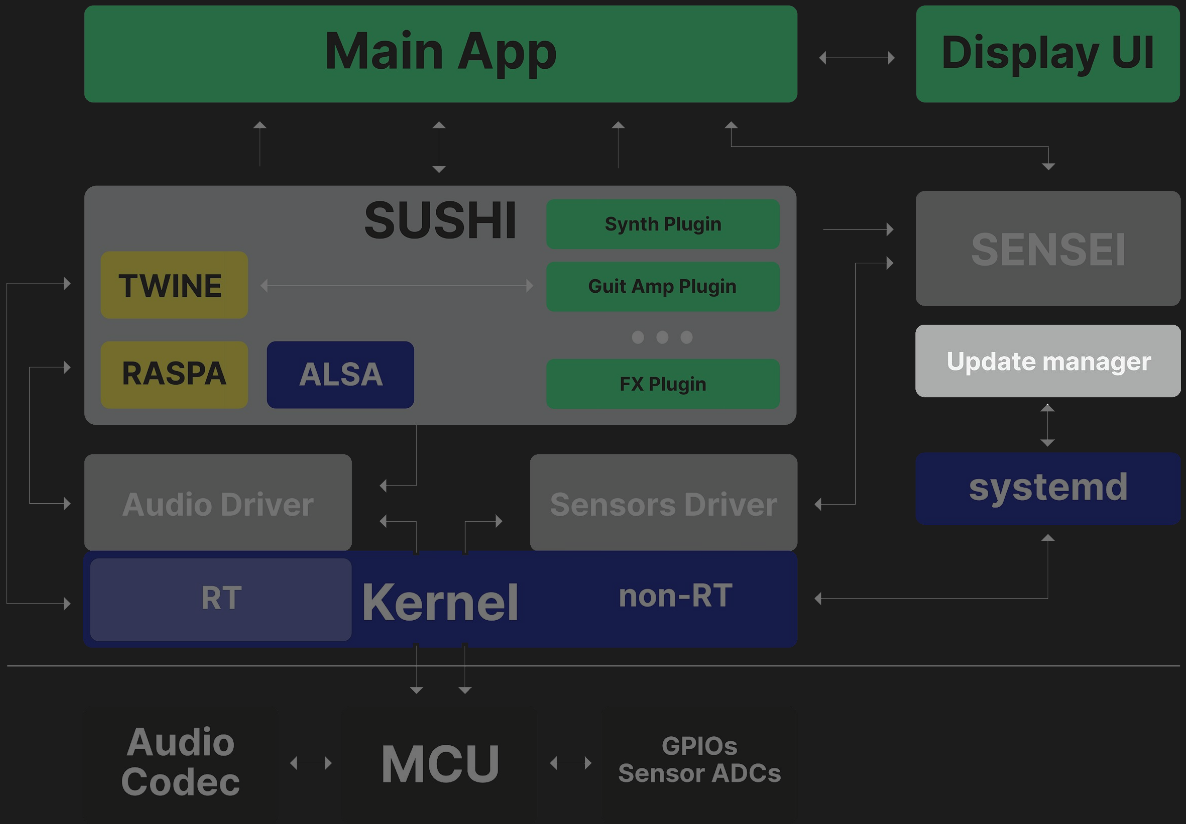
Architecture

Dual Kernel
Plugin Host
Sensor Daemon



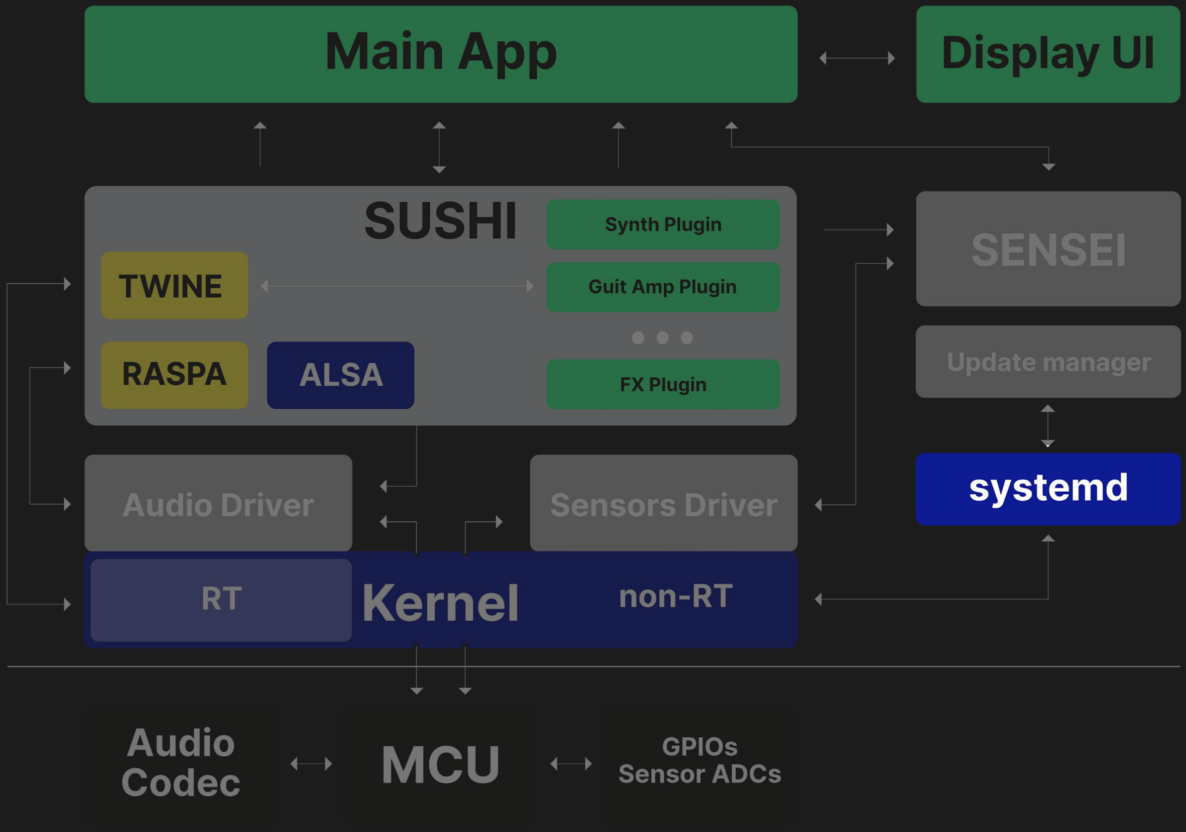
Architecture

Dual Kernel
Plugin Host
Sensor Daemon
Software Update



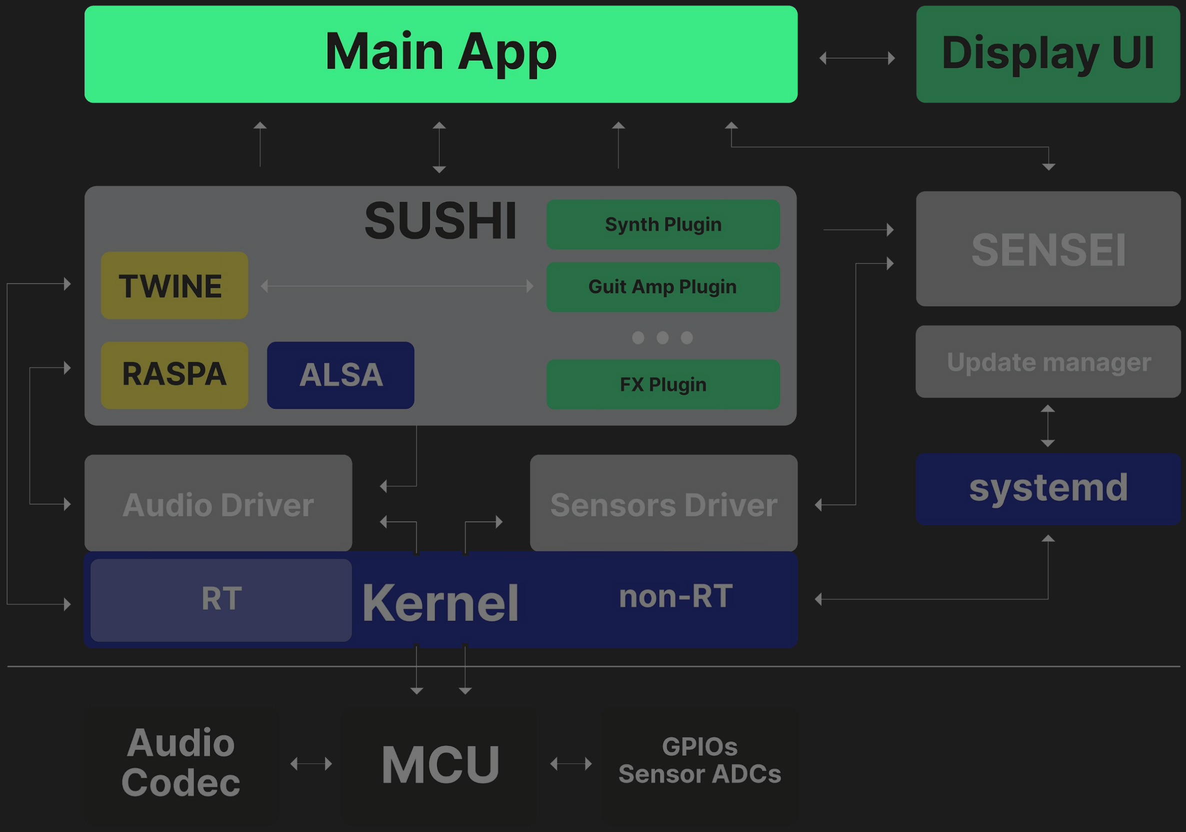
Architecture

Dual Kernel
Plugin Host
Sensor Daemon
Software Update
Systemd



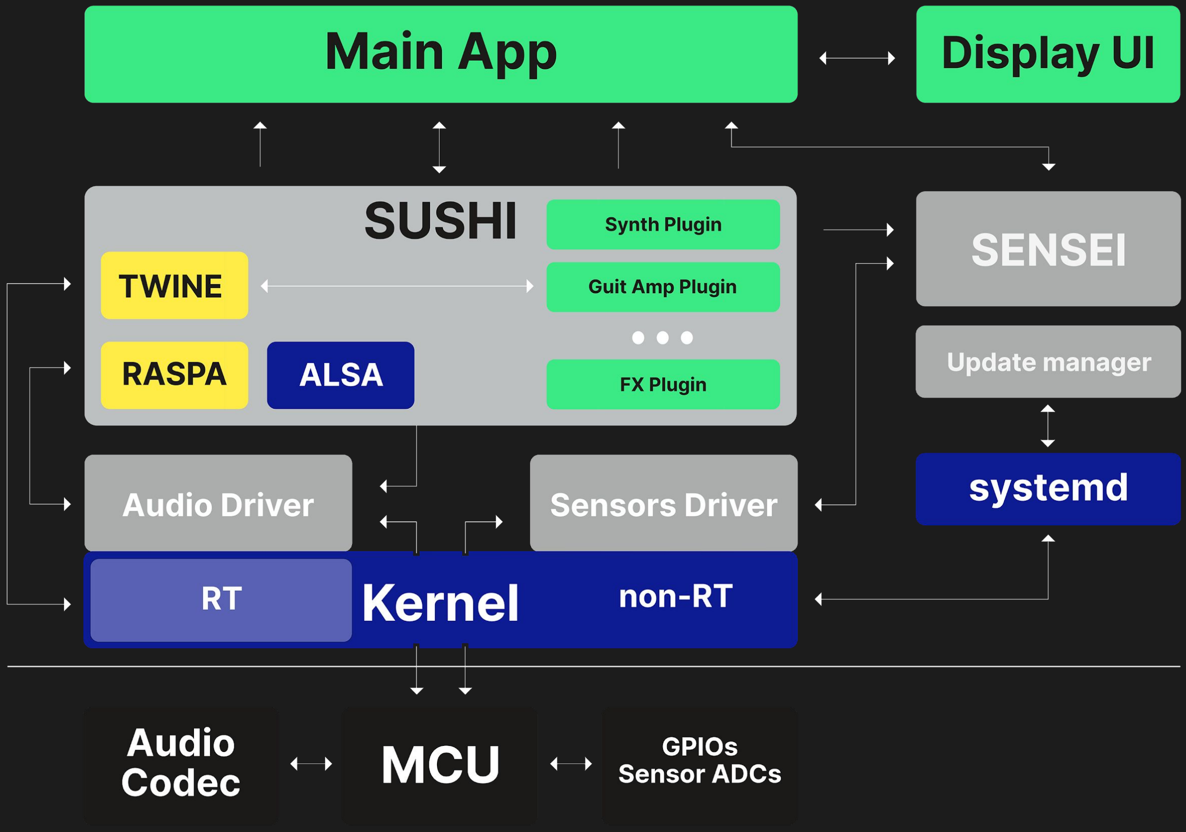
Architecture

Dual Kernel
Plugin Host
Sensor Daemon
Software Update
Systemd
Main App



Architecture

Dual Kernel
Plugin Host
Sensor Daemon
Software Update
Systemd
Main App



Sushi DAW overview

Headless host with full run-time control

MIDI, gRPC, OSC

Hosts VST 2.4, VST 3.7 and LV2 plugins

Low latency performance

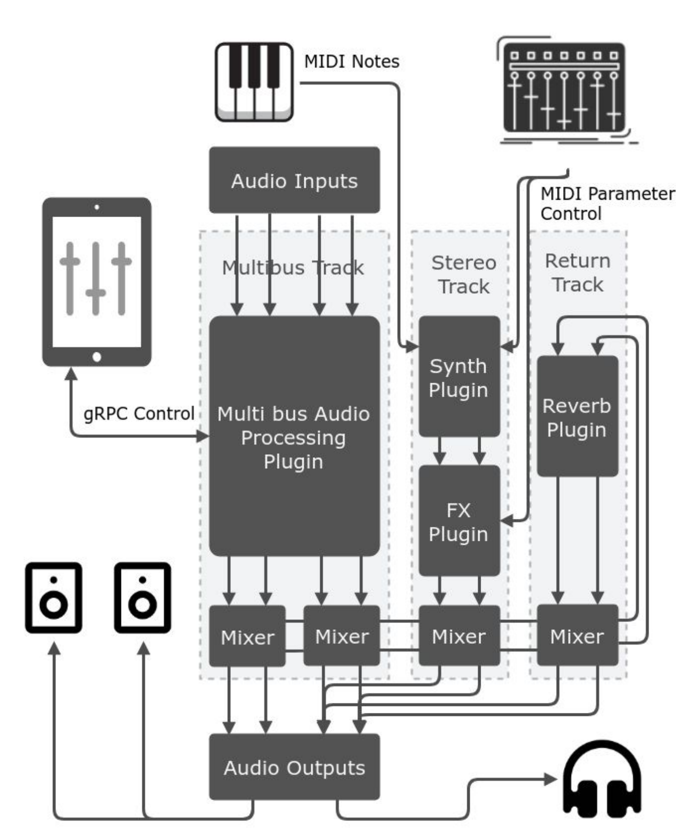
Multithreaded audio processing support

Ableton Link support

Audio connections through

Raspa, Jack, Portaudio, and file I/O

Simple scripting configuration



Sushi DAW configuration

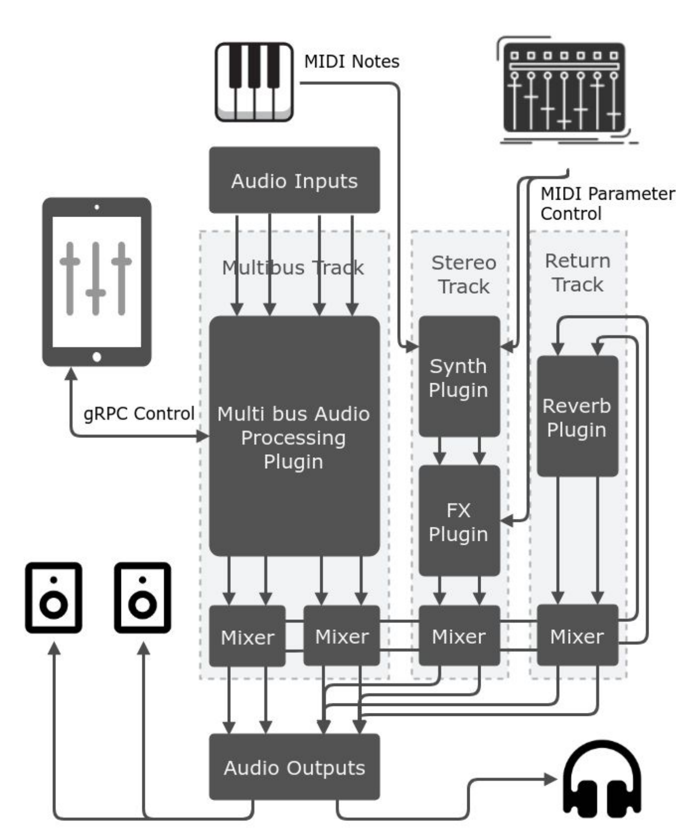
JSON configuration file

example coming in the next slides

gRPC

as the final exercise

https://elk-audio.github.io/elk-docs



Distributed Systems

Messaging Patterns

Transport Layer

Messaging inherently depends on transport layer

UDP - TCP

Each has advantages and disadvantages

Transport Layer

UDP - Connectionless:

- Messages are broadcasted
- Receipt not guaranteed
- No confirmation expected
- No reply
- Nor is order guaranteed

Transport Layer

TCP - Connection Oriented:

- No Broadcasting
- Potentially high latency and jitter
- But reliable messaging

Each has its advantages and disadvantages

A conscious choice is needed!

Message Passing

The choice is made to:

Use the advantages of connectionless messaging

...With the disadvantages this entails

It’s for direct messaging, with no reply or confirmation expected - and the lowest possible latency.

Think of MIDI as an example!

Request / Response

Client requests → server replies

Like a function invocation

With receipt confirmation

With return value

Synchronous or asynchronous

Think of a web 1.0 server

Publish / Subscribe

Clients subscribe to server notifications

Servers notify on each change

Notifications stop when:

- A server is offline
- Clients unsubscribe

Think of web 2.0: push notifications

Protocols and Libraries

Messaging - Open Sound Control

25 years old now!

Originally for music performance data

- Shared between instruments & computers
- In widespread use today

Official spec defines socket messaging

Common for request / response, and publish / subscribe:

- Building on official spec
- Add-ons also exist, e.g. OSC Query

OSC is widely supported

Control applications:

TouchOSC - TWO - OpenStageControl - OSC/Pilot - Lemur

DAW's:

Ableton Live - BitWig - Reaper - MOTU Digital Performer - LV2 Ingen

Plugins:

Native Instruments' Reaktor - FAW Circle^2 ...

Visuals, Stage Lighting & Projection Mapping:

Resolume - TouchDesigner - Unreal Engine - Notch - VDMX - MadMapper ...

OSC - Messages for a synth instance

/synth/oscillator_1/frequency, f, 440.0

/synth/filter/cutoff, f, 65.0

/synth/envelope, ffff, 0.5, 1.0, 1.0, 0.1

/synth/interpolate_to_preset, si, “lately_bass”, 1000

“Address Pattern”

“Arguments”

“Type Tag String”

OSC - Simple synthesizer namespace

- synth
  - oscillator_1
    - frequency: f
    - shape: i
  - filter ← *"Node"*
    - cutoff: f
    - resonance: f
  - envelope: ffff ← *"Type Tag String"*
    - attack: f
    - decay: f
    - sustain: f
    - release: f ← *"Type Tag"*
  - interpolate_to_preset: si

OSC Query

Adds rich discovery features for OSC

In its core is a simple HTTP server providing:

- Existence discovery
- Namespace & State

Much optional extended functionality

LGPL C++ library available: **libossia**

RPC Protocols

A Remote Procedure Call protocol is needed

For native request / response & publish / subscribe

Think of invoking remote methods

As if they are local

Google Remote Procedure Call

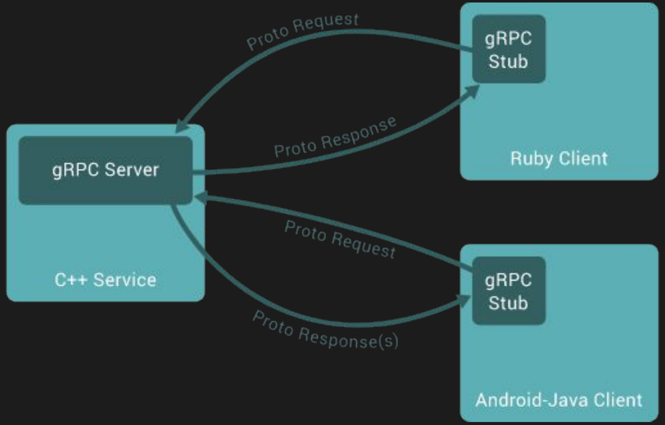
Requires a statically defined API using:

"Protocol Buffer Interface Description Language"

.protobuf

Describes methods and data structures

Transcompiled for each language



gRPC - protocol buffer file

```
service Plugin {
    rpc SetParameter (SetRequest) returns (SetReply) {}
}

// Request message with float value:
message SetRequest {
    float value = 1;
}

// Response message with value as string:
message SetReply {
    string value = 1;
}
```

gRPC - In Use

Server

```
#include "api.grpc.pb.h"


Status SetParameter(ServerContext* context,
                    const SetRequest* request,
                    SetReply* reply) override {
   reply->set_parameter(_to_label(request->value()));

   return Status::OK;
}
```

gRPC - In Use

Client

```
#include "api.grpc.pb.h"

std::string SetParameter(float value) {
    (...)
    request.set_parameter(value);
    Status status = stub_->SetParameter(&context, request, &reply);
    if (status.ok()) return reply.value();
    else             return "RPC failed";
}
```

Implementation Challenges

Concurrency

Control handling implementation in separate thread:

It cannot be in the UI or audio processes

The usual concurrency issues arise

Thankfully solutions are extensively covered!

Fabian Renn-Giles’ & Dave Rowland’s excellent ADC 19 talks:

“Real-Time 101” parts 1 & 2.

Reliability

For messaging on ethernet LAN, UDP is fine.

Problems arise for critical messages

For example Note On / Off

- If message is lost or out of order
- Logic needs to be able to recover

Feedback loops

Can arise between nodes - when both update each other of same change

Solution:

- UIs don’t echo on changes from remote messages
- Only when set from UIs’ controls

Complementary solution:

- For *continuous* properties, like e.g. volume
- Only send a notification if value changes

Where to implement remote control

In the hosting DAW - or the plugin itself?

Bypassing host plugin API allows complex control

But challenges arise

- Reflected parameters need to be synced between:
  - External control API
  - And host plugin API
- Collisions are possible and need resolving

Bandwidth / Rate-limiting

Despite small packets

Unsurmountable message counts are possible!

Best to rate-limit messages to 25-30Hz (UI refresh rate)

Rely on smoothing to interpolate over received values

Common also for plugin API parameters

OSC & gRPC
Together address all needs

Latency: OSC vs gRPC (simple messaging vs RPC)

OSC: UDP is standard

gRPC uses HTTP/2 by default

OSC is low-latency

(but stateless and connectionless)

Ideal for setting property state, and reflecting state changes

gRPC does complex architectures well

- Request / Response model
- Connection based
- Publish / Subscribe (gRPC’s streaming service)
- Synchronous and Asynchronous messaging

Meaning:

- Control a full DAW without compromise
- Discover namespaces dynamically
- Complex data structures

End-User Development

Depends on Run-Time editability:

- Human-Readable messages
- Modifying messages, and connections

With OSC you get both

Also a big ecosystem of end-user software supporting OSC

Messaging or RPC?

OSC or gRPC?

Yes!

Combine their strengths

Elk’s Tools

Sushi - gRPC excerp

```
service AudioGraphController {
    ...
    rpc CreateTrack (CreateTrackRequest)            returns (GenericVoidValue) {}
    rpc GetTrackProcessors (TrackIdentifier)        returns (ProcessorInfoList) {}
    ...
}

message CreateTrackRequest {
    string name = 1;
    int32  channels = 2;
}

message TrackIdentifier {
    int32 id = 1;
}

message ProcessorInfoList {
    repeated ProcessorInfo processors = 1;
}

message ProcessorInfo {
    int32  id = 1;
    string label = 2;
    string name = 3;
    ...
}
```

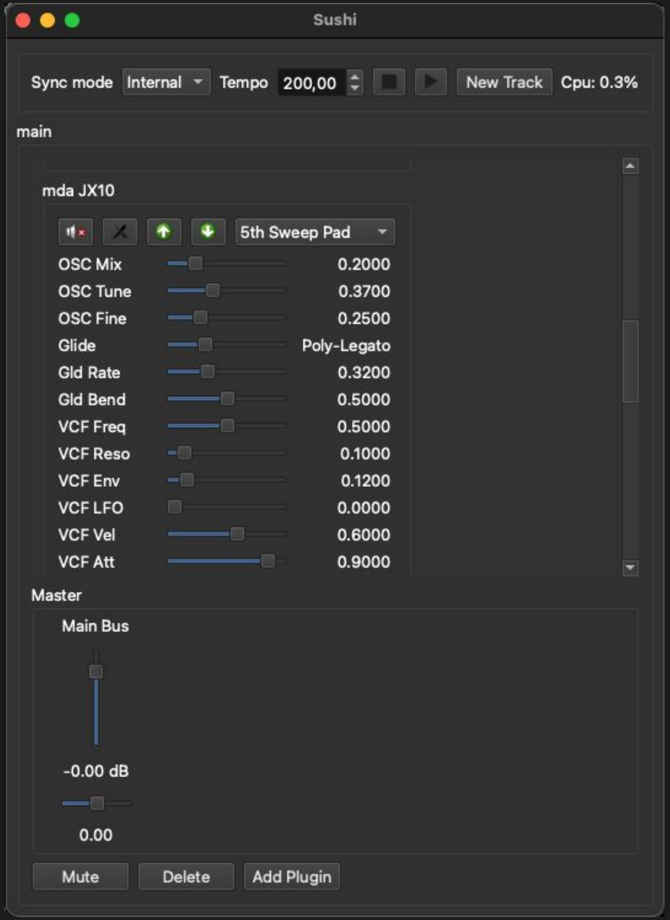
Sushi Example GUI & elkpy

Sushi example GUI

- GUI for controlling Sushi and hosted plugins
- Written in Python with QT (PySide6)

It uses the **elkpy** library:

- Wrapper around Sushi's gRPC interface
- Written in Python
- Simplifies gRPC use
- Corresponding **elkcpp** library for C++



Sushi - elkpy example code

```
def _load_plugin_on_track(self, track_id: int, plugin_spec: dict):
    path = plugin_spec['path']
    name = plugin_spec['name']
    p_type = plugin_spec['type']
    uid = plugin_spec['uid']

    try:
        self.audio_graph.create_processor_on_track(name, uid, path, p_type, track_id, 0, True)
    except Exception as e:
        print('Error loading plugin: {}'.format(e))
```

```
    self.set_pan_label(txt_value)

    value = self._controller.parameters.get_parameter_value(self._processor_id, gain_id)
    self.set_gain_slider(value)
    txt_value = self._controller.parameters.get_parameter_value_as_string(self._processor_id, self.gain_id)
    self.set_gain_label(txt_value)

    self._connect_signals()
```

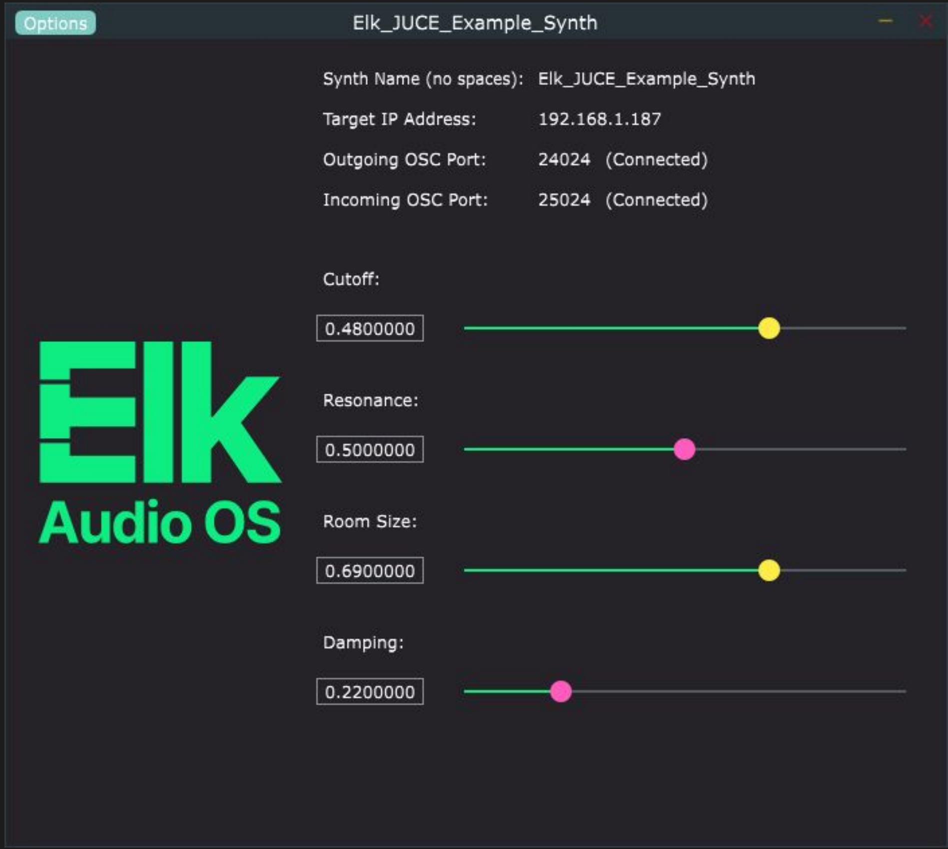
Elk JUCE Example

Our example plugin, in JUCE

Implementing OSC control

In parallel to the plugin API

It demonstrates bridging between these



What we’ve covered:

1. Contexts for headless audio
2. Distributed Systems patterns
3. Challenges and solutions to those
4. Available tools
5. Our concrete choices: OSC & gRPC

Demonstration

Demonstration

gRPC messaging Python example demonstrates:

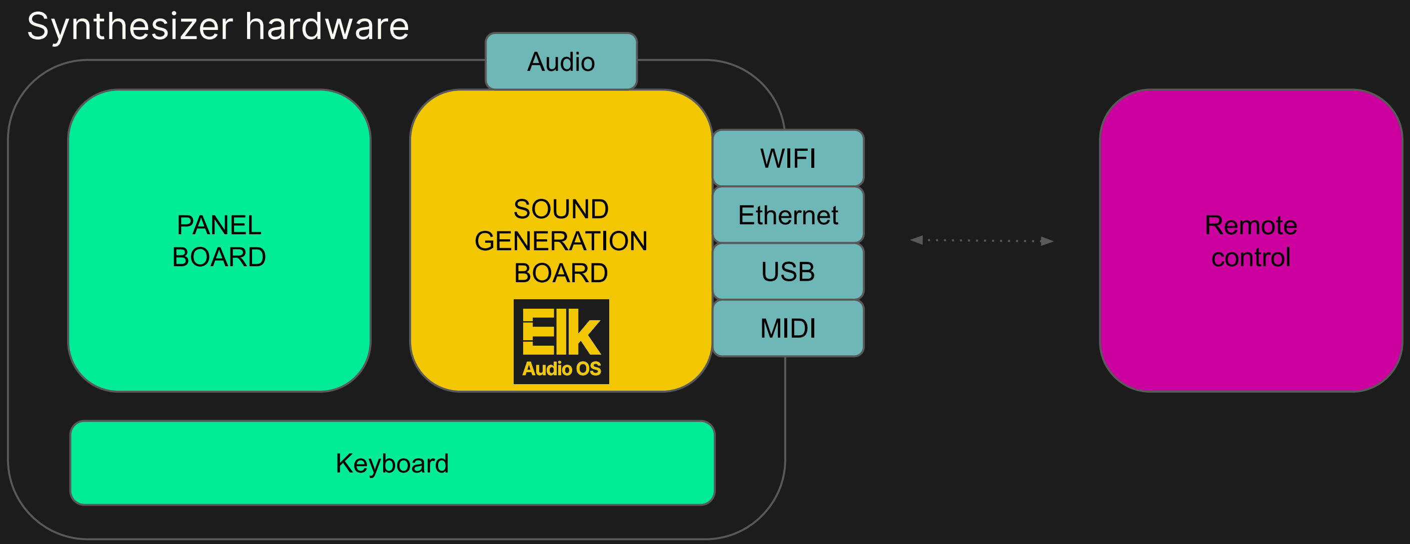
- request / response
- publish / subscribe
- asynchronous communication
- non-float-parameters

Elk example plugin using OSC demonstrates:

- Bypassing host plugin API
- OSC is human-readable
- Tools not made for each other can be used together, with run-time alteration
- Bridging plugin and remote API's

Build your Synthesizer

Synthesizer platform



Synthesizer prototyping

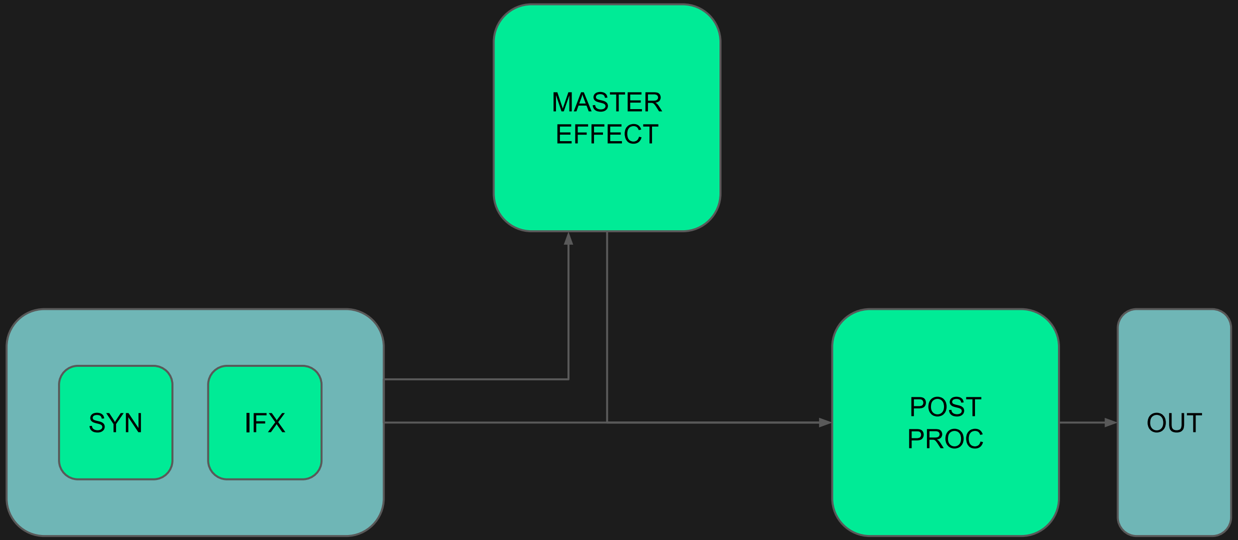
Developing algorithms on the target hardware

- Difficult
- Error prone
- Lack of debugging tools and connectivity

The solution

- Prototype the algorithms on your computer using sushi
- Run the same configuration on the target hardware

Synthesizer audio processing



Example: prototype a digital synthesizer before running it on embedded hardware

Run SUSHI

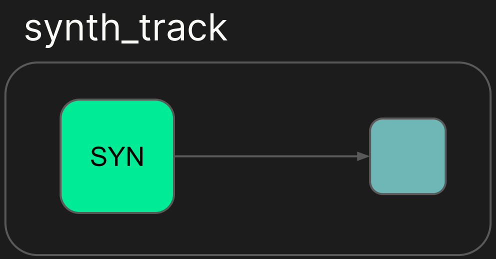
```
$ cd ~/elk-adc22/binaries
(extract binary for your platform)
$ ./sushi --portaudio -c ../config/01_synth.json
```

CTRL+C to stop

Sushi JSON configuration example

Step 1 - the synth engine

Tracks

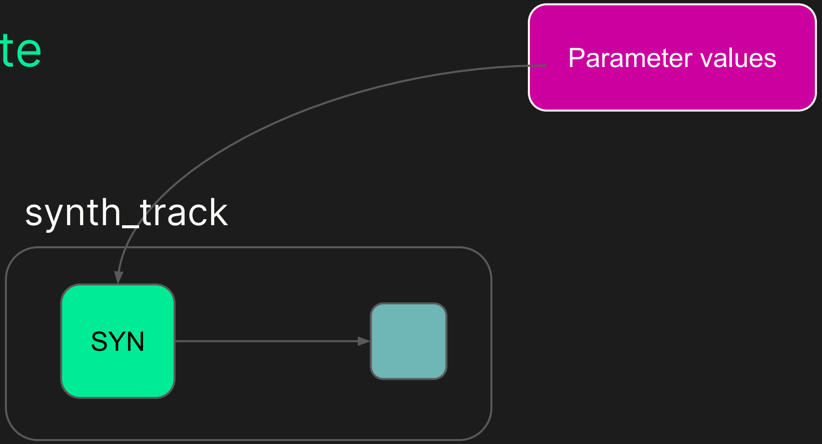


config/01_synth.json

Sushi JSON configuration example

Step 2 - initialize the parameters

Initial state

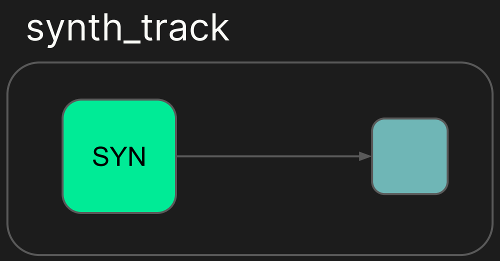


config/02_initial_state.json

Sushi JSON configuration example

Step 3 - use a sequencer for testing...

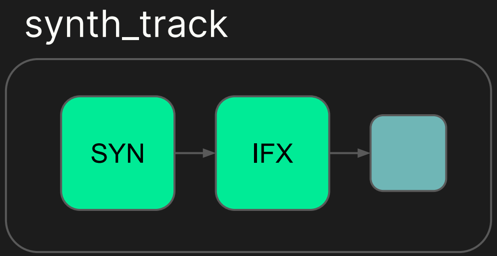
...and use initial_state to set the pitch of the steps



config/03_sequencer.json

Sushi JSON configuration example

Step 4 - add the insert effect

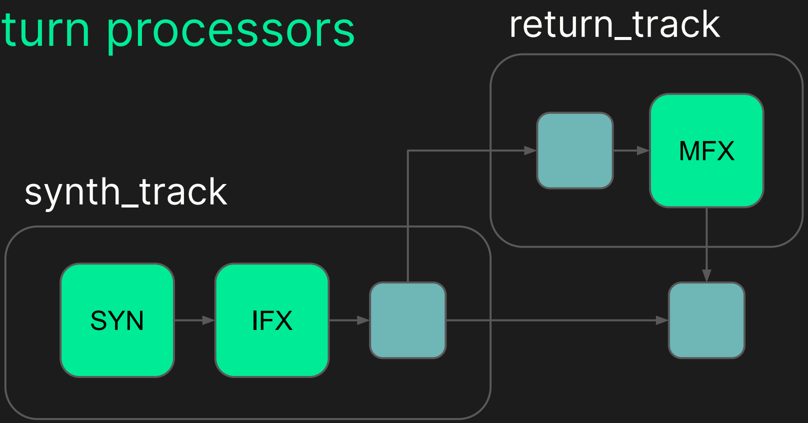


config/04_insert_effect.json

Sushi JSON configuration example

Step 5 - add the master effects

Send / return processors

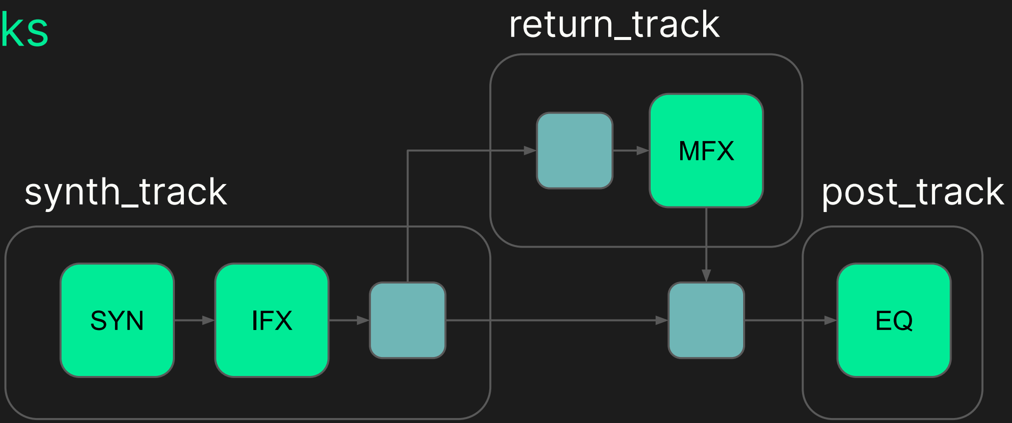


config/05_master_effect.json

Sushi JSON configuration example

Step 6 - add the post EQ

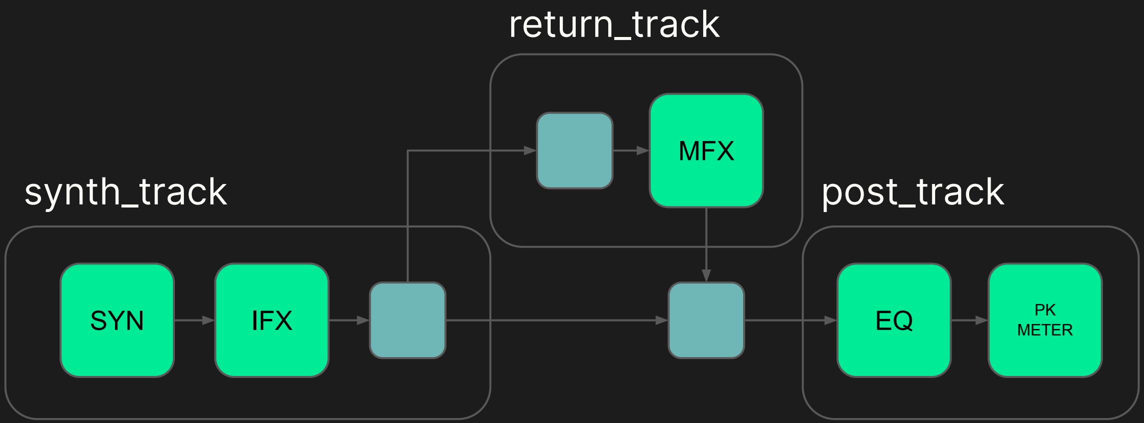
Post tracks



config/06_post_track.json

Sushi JSON configuration example

Step 7 - add OSC notifications



config/07_osc_notifications.json

Free exercise

Configure sushi using the gRPC API

- Create a synth chain similar to the example, but using the gRPC API
- Create some “magic buttons” to switch between different FXs / Generators

config/exercise.json


Elk
Thank you!