Embedded Comparo: Small DSP Systems Side-by-Side

Francesco Di Maggio
Eindhoven University of Technology
Eindhoven, Netherlands
f.di.maggio@tue.nl

Bart Hengeveld
Eindhoven University of Technology
Eindhoven, Netherlands
b.j.hengeveld@tue.nl

Atau Tanaka
Goldsmiths University of London
London, United Kingdom
Bristol Interaction Group
Bristol, United Kingdom
a.tanaka@gold.ac.uk

Abstract

This paper presents a comparative analysis of four embedded platforms designed for real-time audio processing: Bela, Daisy, OWL, and Raspberry Pi. These platforms have become integral tools in the field of digital musical instrument design, offering a variety of workflows, programming environments, and deployment methods. Although each system carries its own distinct strengths and constraints, the current workflow to embed DSP code across multiple devices lacks standardized approaches. To address this challenge, we develop a methodology that focuses on deploying Pure Data patches across all four platforms. Our study is structured around four test patches. Our findings highlight the trade-offs in latency, processing power, and memory constraints across the selected platforms. As a result, we propose a streamlined workflow to deploy Pd patches on each board using Plugdata, the Heavy Compiler, and their respective Web IDEs. As an ongoing contribution to the NIME community, we document our methodologies, workflows, and best practices in an open source repository, which serves as a continuously evolving resource for future research in the hands of musicians, researchers, and developers working with embedded musical systems.

Keywords

Embedded Platforms, Single-Board Computer, Microcontroller, DSP, Plugdata, Bela, Daisy, OWL, Raspberry Pi

1 INTRODUCTION

The increasing computational power of single-board computers (SBCs) and microcontroller units (MCUs) has enabled musicians, developers, and researchers to build custom embedded musical instruments with low latency and high-quality sound. Typically, working with embedded systems required an in-depth understanding of low-level programming languages such as C/C++, making them largely inaccessible to those without specialized expertise. However, modern computer music programming environments such as Pure Data (Pd), Max gen~ and RNBO, FAUST, and SuperCollider have significantly reduced this barrier, allowing a broader range of users to create and deploy DSP code directly on embedded hardware. Furthermore, the active community-driven effort to share open source resources, works, and best practices has further reduced these barriers, empowering musicians without specialized programming expertise to deploy their projects on embedded systems.

Despite these advancements, embedded platforms present specific constraints in workflow, computational capacity, debugging, and hardware integration. The proposition seems straightforward: musicians can continue to develop their systems using familiar high-level software tools and run them on the embedded system. While the *lingua franca* of Max or Pure Data is appealing to musicians, the SBC and MCU versions of these languages are not full implementations and have a limited subset of the patchable objects found in the desktop computer version. Patches may require extensive modification to accommodate the different architectures, toolchains, and DSP optimizations. This can pose challenges for those looking to develop entangled, versatile, and reusable instruments across platforms.

In this study, we compare four popular systems: two single-board computers – the Raspberry Pi and BeagleBone; and two microcontroller systems – the Daisy and OWL. We use complementary products (some offered by the manufacturers): audio interfaces, expansion boards, and supplementary hardware that make these embedded processors interface to music systems via analog audio input/output and control input/output via MIDI and 0-5V control voltage (CV). We developed a standard set of patches across the four systems to assess their workflows and performance. We have created a common workflow that abstracts the platform-specific elements, allowing the user to create a core synthesis patch that can then be adapted to each of the different targets. In addition to the insights published here, we have made our patches and tutorial documents publicly available on GitHub.

The paper is organized as follows. We first present prior work by others that report on running audio on embedded systems. We then present the four hardware platforms that we studied, including the auxiliary interfacing systems. We present the common computer music languages that can be used, make our choice for this study, and then present the deployment workflow for each system. We then describe the four patches we tested on the system of increasing complexity. We discuss implementation issues and performance comparing the systems, before concluding with perspectives for continuing work.

2 RELATED WORK

The integration of embedded platforms into the design of digital musical instruments has been a recurring theme within the NIME community. Researchers have explored the possibilities and implications of using MCUs and SBCs for real-time audio processing, interactive sound synthesis, and modular instrument design. Over the past decade, several studies have demonstrated how embedded systems can be adapted for use in musical contexts, from standalone synthesizers to intelligent musical instruments.

Webster et al. [25] developed the OWL programmable effects pedal, one of the first microcontroller-based systems for audio. They offer a C++ programming environment for audio signal processing and point to the future possibility of compiling code written in SuperCollider or patches from Pure Data. More recently, Wakefield [24] presented a gen~ to the Daisy workflow to deploy Max patches on the Daisy microcontroller.

NIME '25, June 24–27, 2025, Canberra, Australia

Embedded implementations of OSC and audio networking are described in [3, 20]. MacConnell et al. [6] describe the running of Pure Data on the BeagleBone SBC. McPherson and Zappi [9] describe a hardware audio interface for the BeagleBone to enable low-latency audio, and Moro et al. [13] describes the use of Pure Data on the platform. Drinkwater [5] describes the running of Pure Data patches on the Raspberry Pi and provides a step-by-step tutorial to users.

Beyond Pure Data, Michon et al. [11] presented a framework for embedding FAUST code into microcontrollers and SBCs, demonstrating the efficiency of FAUST for high-performance and low-latency audio processing. Yee-King [26] describes running SuperCollider on SBCs. In the commercial sphere, Cycling '74 has recently provided a dedicated export section for Raspberry Pi using Max/RNBO.

In embedded DMI design, Momeni et al. [12] present a platform for rapid embedded prototyping. Marasco [7] explored the potential of the Norns Shield, a Raspberry Pi-based device, as an accessible tool for ensemble-based music applications. Sullivan et al. [21] present a series of embedded performance instruments.

More recently, the possibility of an embedded DSP has led to the concept of the Internet of Musical Things (IOMT) [22] and the exploration of embedded AI for NIME in a workshop [15, 16]. Comparisons of different architectures and platforms for embedded DMIs have been carried out by Meneses [10] for Linux implementations and Vignati et al. [23] in the case of IOMT.

Building on this existing body of research, our study aims to contribute to the ongoing NIME discourse on embedded musical systems by proposing a common methodology that streamlines the deployment of Pd patches across multiple embedded platforms.

3 EMBEDDED HARDWARE

As seen in Table 1, single-board computers are fully functional systems that integrate a processor, RAM, storage, and I/O interfaces on a single board. They typically run operating systems such as Linux, providing support for high-level programming languages and real-time audio environments. SBCs are ideal for complex audio applications that require greater computational power, multithreading, or extended memory management. Raspberry Pi and BeagleBone fall into this category [13]. They support running audio applications natively, meaning that code can be modified, loaded, and executed directly on the board. However, since SBCs are based on general-purpose operating systems, they often experience higher latency unless special-purpose software [2] or accompanying hardware [8] are used.

In contrast, microcontroller units are lightweight embedded platforms designed specifically for task-specific real-time processing. Although the decreasing size of SBCs and the increasing power of MCUs begin to blur their distinctions, MCUs have traditionally been differentiated from general-purpose computer systems by not running an operating system, not handling hierarchical file systems, and not having an onboard memory management unit (MMU). Today, the reality is that MCUs are often able to access data on external storage devices such as SD cards, and run low overhead operating system-like kernels such as a Real-Time Operating System (RTOS) or Application Programming Interfaces (API) such as the Portable Operating System Interface (POSIX). For an early review of RTOS for MCUs, see [1]. For our purposes here, the MCU does not separately run the signal processing application environment (like Pd) and open files (patches). Instead, MCUs require the patch and the application to be compiled into low-level executable code to be deployed on the firmware. This improves low-latency DSP execution, but limits run-time modification and interactive debugging.

In this study, we chose to examine the deployment workflows and constraints of the following four platforms: Raspberry Pi and BeagleBone (SBC); Daisy and OWL (MCU). Each requires a subsystem for audio and control input/output. We chose readily available kits and products that have been developed around each of the platforms, allowing musicians to integrate the results into performance setups using MIDI or Eurorack standards. We selected Eurorack modular adapters for the most part, providing line-level audio I/O and 0-5V CV control.

We also wanted to find systems that offered high-level software deployment tools, either in the form of an easy-to-use interactive development environment (IDE) or integration into authoring environments like Pure Data. While all this hardware can be programmed using software compilers and low-level firmware flashing tools, we wanted to find environments that offered high-level workflows for musicians not comfortable with working using a shell and command line interface.

We used the Lich[1] module from Befaco in their collaboration with Rebel Technologies, creators of the OWL system. For Daisy, we used two variants of the Daisy Seed microcontroller board inside a Pod evaluation board; and the Daisy Patch Submodule inside the Patch.init Eurorack adapter[2]. For BeagleBone, we used the Bela low-latency audio"cape" [8] inside a Pepper Eurorack adapter[3]. Selecting a system for the Raspberry Pi opened up more options, as discussed below. We settled on the Organelle[4].

3.1 OWL

OWL (currently mk3) from Rebel Technology is an ARM Cortex M7 microcontroller unit that supports a variety of programming languages, including C++, Pure Data, Max gen~, and FAUST.

- 8-96kHz sampling rate
- 48kHz 24-bit stereo audio codec
- 480MHz ARM Cortex M7 microcontroller
- 3500 operations per sample at 48kHz
- 8+1 MB RAM / 8+1 MB flash memory
- 32-bit floating point audio processing

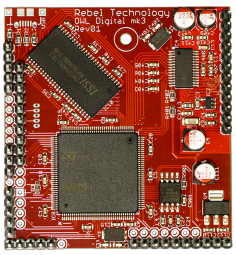

Figure 1: The OWL board.

3.1.1 Lich. The Befaco Lich is a programmable Eurorack module based on the OWL platform, featuring CV integration and DC-coupled stereo I/O.

[1] https://www.befaco.org/lich
[2] https://electro-smith.com/collections/daisy
[3] https://learn.bela.io/products/modular/pepper
[4] https://www.critterandguitari.com/organelle

Table 1: Single-Board Computer (SBC) vs. Microcontroller Unit (MCU)

	Single-Board Computer (SBC)	Microcontroller Unit (MCU)
OS Support	OS (e.g., Linux), RAM, storage	No operating system, no file system
Programming	Runs high-level software	Requires compilation
Power	High (multi-threading, large buffers)	Low (optimized for efficiency)
Memory	High (RAM, external storage)	Limited (flash memory, SRAM)
Live Patching	Yes (real-time editing, e.g., in Pd)	No (requires firmware flashing)

Figure 2: The Befaco Lich module.

3.2 Daisy

Daisy is an ARM Cortex M7 microcontroller platform for embedded audio development, with several variants, including Daisy Seed and Daisy Patch. Provides stereo audio I/O that supports programming in C++, Max/MSP gen~, and Pure Data. The Daisy Seed development board is available in two versions: one with 1MB and another with 64MB of SDRAM. Both include 8MB of flash memory onboard. The Daisy Submodule is a variant designed for Eurorack module design and development.

- Stereo audio I/O
- 96kHz 24-bit audio hardware (AC-Coupled)
- x31 GPIO
- x12 ADC inputs (16-bit),
- x2 DAC outputs (12-bit, DC-Coupled)
- ARM Cortex-M7 MCU, running at 480MHz
- 1MB or 64MB of SDRAM
- 8MB external flash memory
- SD card interface and PWM outputs
- SPI, UART, SAI/I2S, I2C
- Dedicated VIN pin for external power

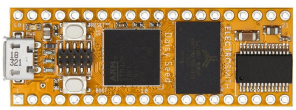

Figure 3: The Daisy Seed development board.

3.2.1 Pod. Daisy Pod is a compact and programmable audio development board for the Seed and provides stereo audio I/O and GPIO pins for control. The Daisy patch.Init() is a Eurorack adapter for the Submodule and provides line-level audio I/O and CV integration. We ran our tests on both the Pod and patch.Init().

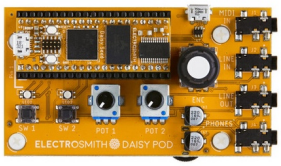

Figure 4: The Daisy Pod development board.

3.3 Bela

The Bela system consists of a BeagleBone Black single-board computer paired with a custom audio interface expansion board, or "cape", which delivers low-latency audio and control processing [8]. The Bela software stack includes a custom Linux distribution optimized for real-time audio, including a real-time kernel, compilers for C++ and FAUST, and support for creative coding environments like Max gen~ and Pure Data. In particular, Pd can operate in two distinct modes on Bela: either as an application running directly on the BeagleBone or by compiling patches into efficient C++ code using the Heavy Compiler[5] for optimized performance.

- Stereo audio I/O with ultra-low latency
- BeagleBone Black 1GHz AM3358 ARM Cortex-A8
- 16 digital I/O
- 8 x 16-bit analog inputs
- 8 x 16-bit analog outputs
- 2 speaker amplifiers
- 4GB of internal memory

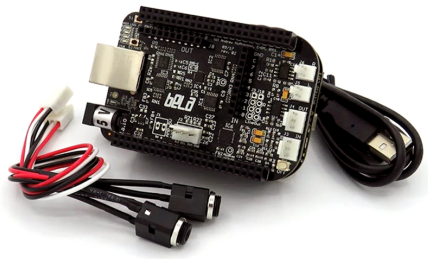

Figure 5: The Bela board.

[5]https://github.com/enzienaudio/hvcc

3.3.1 Pepper. The Pepper is an expansion board designed as an extension to the Bela system. Provides audio input and output connectivity and control voltage (CV) interaction in a Eurorack format. Although we used the Pepper in this study, all results and tests can also be run on the Bela system alone without the Pepper.

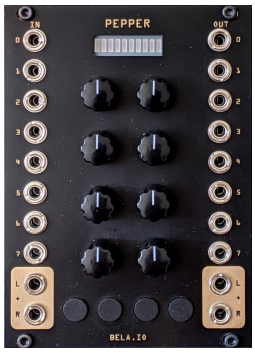

Figure 6: The Bela Pepper Eurorack module.

3.4 Raspberry Pi

Raspberry Pi[6] is a family of low-cost single-board computers popular in the DIY electronics scene. The different models include - a full comparison can be found here[7]:

- **3, 4+ and 5**: Quad-core ARM Cortex-A72 processor, 8GB of RAM, dual HDMI outputs, and USB 3.0 ports. Requires an external audio interface.
- **Zero 2 W**: Quad-core 64-bit ARM Cortex-A53 processor clocked at 1GHz and 512MB of SDRAM. Built-in WiFi (2.4GHz) and Bluetooth 4.2. MicroSD slot. No audio I/O.

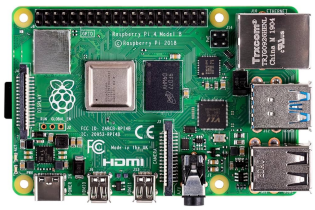

Figure 7: The Raspberry Pi 4 board.

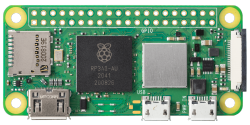

Figure 8: The Raspberry Pi Zero 2 W board.

To transform the Raspberry Pi into a music platform, dedicated audio add-ons are necessary, such as Pisound[8], Raspberry Pi DAC+[9], and Patchbox OS[10]. The latter is a specialized operating system that includes pre-installed software such as Pure Data, SuperCollider, and MODEP (a virtual pedalboard), streamlining audio-focused workflows.

[6]https://www.raspberrypi.com/
[7]https://en.wikipedia.org/wiki/Raspberry_Pi
[8]https://blokas.io/pisound
[9]https://www.raspberrypi.com/products/dac-plus
[10]https://blokas.io/patchbox-os

3.4.1 Organelle. Although there are different Eurorack adapters for different versions of Raspberry Pi hardware[11][12], none offer a high-level software deployment environment. We therefore chose to use the Organelle, a standalone synth built on the Raspberry Pi, with line-level audio I/O, MIDI, and a graphical user interface (GUI)-based patch manager. The Organelle is built on a Raspberry Pi Compute Module 3+.

Figure 9: The Critter & Guitari Organelle M.

4 SOFTWARE ENVIRONMENTS

Embedded platforms offer a variety of programming environments to develop and deploy real-time audio applications. These environments accommodate a wide range of skill levels, from musicians and artists to software developers, providing the flexibility to design and implement custom instruments, effects, and audio processes. A key challenge when working with multiple embedded platforms is adapting code and deployment workflows across different hardware architectures, toolchains, and namespace conventions.

Table 2 summarizes the different programming environments that run on these platforms. As shown in Table 2, Pure Data is the programming environment supported by all platforms. The actual Pd implementation may differ (whether by cross-compilation or by the platform running Pd itself), and we discuss this. Importantly, for the end user, this provides a familiar development environment and provides us, for the purposes of this study, a way to deploy the same patch across the different platforms to assess computational power, as well as workflow.

4.1 Pure Data

The Pure Data (Pd)[13] audio programming environment, in addition to its default end-user software package, can be deployed across multiple platforms using different compilation and optimization methods:

- **libpd**[14]: A lightweight library version of Pure Data that allows Pd patches to be embedded in C/C++ projects. It is used on platforms such as iOS, Android, Unity, and OpenFrameworks.
- **Heavy Compiler**[15]: This tool compiles Pd patches into optimized C++ code for deployment on embedded platforms.
- **Plugdata**[16]: A GUI front-end for Pd written in JUCE, offering the compilation of Pd patches to VST plug-ins, the preparation for deployment on mobile operating systems, the compilation to C++ (using hvcc), and the direct export of patches to Daisy.

[11]https://github.com/Deftaudio/Midi-boards/tree/master/Eurorack_RPi
[12]https://github.com/Allen-Synthesis/EuroPi
[13]https://puredata.info
[14]https://github.com/libpd/libpd
[15]https://github.com/Wasted-Audio/hvcc
[16]https://plugdata.org

Table 2: Embedded Hardware and Their Relative Software Environments.

	Bela	Daisy	OWL	Raspberry Pi	Organelle
Web IDE	Yes	Yes	Yes	No	No
Pure Data	Yes	Yes	Yes	Yes	Yes
Plugdata	No	Yes	No	No	No
Max/gen~	Yes	Yes	Yes	Yes	No
Max/RNBO	Yes	No	No	Yes	No
C++	Yes	Yes	Yes	Yes	Yes
FAUST	Yes	No	Yes	Yes	No
Arduino	Yes	Yes	No	Yes	No
Supercollider	Yes	No	No	Yes	Yes

Each platform examined in this study offers different methods for deploying and running Pure Data patches, reflecting variations in hardware architecture, processing power, and memory constraints. While some platforms support direct uploading of Pd patches, others require cross-compilation into optimized C++ code. The process of adapting a single patch to run on different platforms involves adjusting audio/control mappings, hardware integration, and buffer management to ensure compatibility.

4.2 Deployment Methods

The deployment process varies depending on the platform. Bela, for example, allows Pd patches to be compiled using its own browser-based interactive development environment (IDE)[17] or by compiling them into optimized C++ code using the Heavy Compiler. Daisy and OWL require conversion of Pd patches into firmware through the Heavy Compiler. This can be done on the Daisy via Plugdata or pd2dsy, a Pd-to-Daisy conversion tool. OWL runs Heavy in the cloud and makes it accessible via its cloud-based Web IDE[18]. The Organelle runs Pure Data natively and connects to the local network over WiFi to make its file system available to the user. In order to compile our Pd patches onto Daisy and OWL, we used the Heavy Compiler (hvcc). Heavy operates by analyzing Pd patches and translating their signal processing components into optimized C++ code, reducing CPU load and memory overhead. Heavy compiles the entire Pd patch into a standalone, pre-optimized DSP code, significantly improving CPU efficiency, latency, and resource management.

4.2.1 OWL Lich. OWL utilizes its Web IDE to compile Pd patches via hvcc. Programming is done via a cloud-based online IDE where the HTTP server and compilers run on the web, and recognize over USB-MIDI the connected device.

The steps to load a Pd patch into Befaco Lich are the following:

(1) **Prepare the patch**: Modify the patch according to the OWL namespace, where CV inputs and outputs are accessed using predefined mappings. For example,"r Button_1 @owl B1" is used to map to the left button.
(2) **Upload via OWL IDE**[19]: Use the OWL Patch Library or Lich IDE on-line to upload the patch. The IDE automatically compiles the Pd patch for the OWL platform.
(3) **Create an account**: Go to https://www.rebeltech.org/my-account, and sign in.
(4) **Upload the patch**:
   (a) Go to *Patches* → *My Patches* → *Create new patch.*
   (b) Upload or download a Pd patch to use as a starting point (e.g., Pure Data template[20]).
   (c) Fill in the details and upload your Pd patch or link it to a GitHub URL.
(5) **Save and Compile**: → *Device* → *Connect to Device.*
(6) **Load it up to the OWL**: Use OwlControl to load the patch or store it directly on the board.

4.2.2 Daisy Pod. The Daisy software ecosystem includes tools like pd2dsy[21] for Pure Data, Plugdata direct export support, and oopsy[22] for Max, which streamlines the deployment of developers working in these environments. Daisy requires that Pd patches be converted into low-level code before execution. We used the Export feature of Plugdata, which calls the Heavy compiler to generate C++ code with which it flashes the pod. The steps to load a Pd patch into Daisy Pod are the following:

(1) **Install the Daisy Toolchain** and configure the environment following the main guidelines[23].
(2) **Prepare the patch**: Daisy's namespace for physical controls (e.g. knobs, buttons) must be defined before proceeding to the next process. Ensure that they are mapped to the correct hardware controls. For example, the Heavy Compiler uses "r Knob1 @hv_param" to map to the left knob. Use Daisy Pod's documentation and template Pd patch for guidance.
(3) **Upload and Compile**: To compile while Daisy is in DFU mode, use one of the following two methods:
   (a) **Direct export from Plugdata**: Open the patch in Plugdata. Use the direct export function to convert the Pd patch into a firmware file for Daisy Pod.
   (b) **Deploy via Daisy Web Programmer**: Upload the Pd patch to Daisy using the Daisy Web Programmer.

4.2.3 Bela Pepper. Bela supports two modes of Pd deployment:

- **libpd mode**: Runs Pd patches using its browser IDE, where a C wrapper is compiled before execution to run the patch via lidpd.
- **Heavy mode**[24]: Compiles Pd patches into C++ using hvcc to run the low-level code.

The Bela Linux distribution runs an HTTP server, allowing a development workflow based on a browser IDE, accessed over the local network using a standard browser, allowing users to upload, edit, and manage programs on the Bela hardware.

[17] http://bela.local
[18] https://www.rebeltech.org/patch-library/
[19] https://www.rebeltech.org/patch-library/patches/latest
[20] http://hoxtonowl.com/patch-library/patch/Pure_Data_Template
[21] https://github.com/electro-smith/pd2dsy
[22] https://github.com/electro-smith/oopsy
[23] https://github.com/electro-smith/DaisyWiki/wiki
[24] https://learn.bela.io/tutorials/pure-data/advanced/using-the-heavy-compiler

Table 3: Heavy vs. Non-Heavy Compiler Mode.

	Heavy Compiler (Bela, Daisy, OWL)	Native Pd (Bela, Organelle)
Deployment	Compiles Pd to C++	Interprets Pd patches at runtime
Latency	Low latency	Latency based on buffer size
Efficiency	Reduced load	Higher CPU usage
Flexibility	Less flexible, requires recompilation	More flexible, allows live editing
Memory Usage	Optimized for low-memory	Higher memory

The steps to load a Pd patch to Bela Pepper in native mode:

(1) **Prepare the patch**: Ensure that ADC/DAC are mapped to the appropriate hardware controls. For example, CV inputs are routed through Bela's ADCs (3-10), and outputs through DACs (3-10). Use Bela's documentation and template Pd patch for guidance.
(2) **Upload to browser IDE**: Access Bela's Web IDE at http://bela.local. Create a new project and choose Pure Data. Upload (or drag-and-drop) the Pd patch as the _main.pd file.
(3) **Compile**: Click the **Run** button. The browser IDE invokes libpd, compiles, and deploys the patch to the board. Adjust hardware connections to ensure compatibility.

4.2.4 Organelle. The Organelle runs Pd patches natively on its Raspberry Pi 3. The device has a 21 character x 6 line monochrome OLED screen and a rotary encoder knob for navigating the file system. Pd patches can be loaded from a USB storage device or over WiFi. While a keyboard and mouse can be connected to the USB port, and an external monitor to the HDMI port, allowing use as a computer, the Organelle has also been designed to allow access to its patch manager without standard computer input peripherals.

Raspberry Pi can run Pd patches natively on its Linux-based operating system, without additional compilation.

The steps to load a Pd patch into Organelle are the following:

(1) **Prepare the patch**: Ensure that the patch conforms to Organelle's patch namespace and control structure. Assign knobs and buttons within the Pd patch to Organelle's hardware inputs (e.g., `knob1`, `button1`).
(2) **Transfer to Organelle**: Organelle's WiFi implementation works in two modes, as an access point (AP) and as a client. Starting in AP mode allows the user to connect their computer to the Organelle's WiFi. Here, the user can configure the Organelle to join an existing WiFi network by storing the SSID and WPA code. From that point on, the Organelle can be used in client mode, joining known WiFi networks using the encoder knob and the built-in OLED screen. The Organelle runs an HTTP server that provides access to its file system via the URL http://organellem.local. Users can upload Pd patches using the browser interface.

5 TEST PATCHES

To establish a baseline for deployment and performance evaluation, we developed four (4) Pure Data test patches covering basic synthesis, sampling, and live processing: 1) Hello World, 2) Simple FM synthesizer, 3) Granular synthesizer, and 4) Live cloud generator.

5.1 Hello World

The Hello World patch has minimal audio (sine wave) and serves as a proof-of-concept to deploy the "same" patch on our four target platforms. This patch serves as a test bed to understand the deployment workflows, platform-specific adaptations, and hardware integration for each board, as well as to ensure that each platform can process real-time audio and control input consistently.

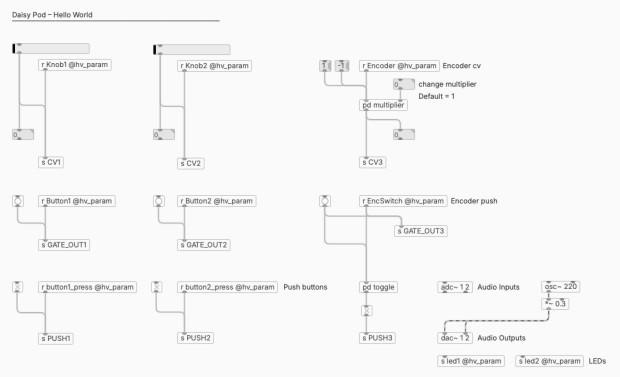

Figure 10: The Hello World Pd patch.

The Hello World patch also serves as a template, showing the abstraction of control input and showing how, through a simple messaging system, a core patch can be adapted to the target in question by replacing one object in the patch.

5.2 FM Synthesizer

The FM synthesizer patch generates a basic FM tone, where the user can control the carrier frequency, modulator frequency, and modulation index. The patch is designed to interact with the hardware's ADC, DAC, and physical controls (e.g., knobs, buttons, CV inputs) across platforms. In the following, we outline the deployment workflows for each board, highlighting the necessary adaptations for namespace differences and the use of physical inputs and outputs.

The Pd patch contains:

- **Carrier Oscillator**: A sine wave whose frequency is controlled by a knob or CV input.
- **Modulator Oscillator**: Another sine wave modulating the carrier frequency.
- **Modulation Index**: Controlled by a knob or CV input that defines the depth of modulation.
- **Audio Output**: Routed to the board's DAC.
- **Control Inputs**: Mapped to hardware controls (knobs, buttons, or CV inputs) for live interaction.

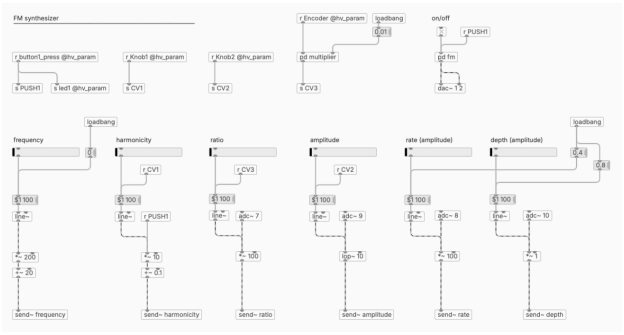

Figure 11: The FM synthesizer Pd patch.

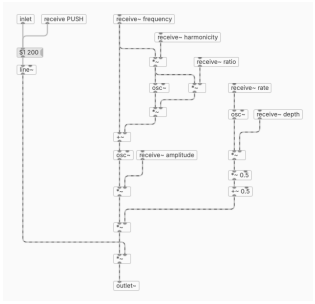

Figure 12: The FM Synthesizer DSP.

5.3 Granular Synthesizer

The granular synthesizer demonstrates the feasibility of reading audio samples from a buffer, windowed granular manipulation, and real-time control mapping. The granular synthesizer is a Pure Data readaptation of Sakonda's "MSP Granular Synthesis v2.5" (2000) [19]. This patch serves as an excellent benchmark, as it was originally developed as a Max patch running on some of the first laptops capable of running MSP. It has then been ported to Pd and has been run on early mobile platforms [14]. Each grain voice uses a ramp wave (phasor) to read a sample stored in a buffer, at a speed determined by the sampling rate, transposition, and time-stretch parameters. It uses a sample and hold so that the phasor frequency is updated at grain boundaries. Multiple instances of the grain voice are offset by an offset phase depending on the number of voices, to be multiplied by a triangle window (from another buffer), and overlap summed to produce continuous output. Our implementation here was a four-grain version of the patch.

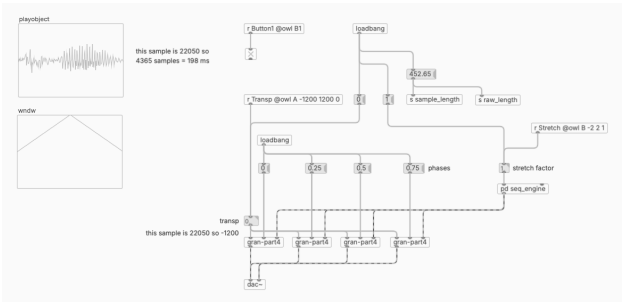

Figure 13: The Granular Synthesizer Pd patch.

The patch introduces the following:

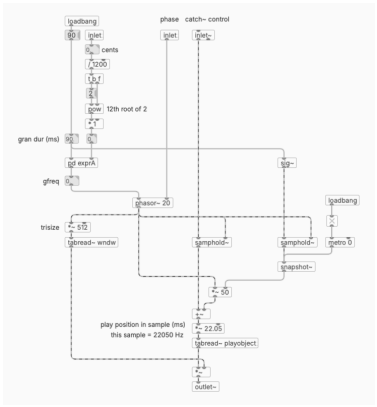

Figure 14: The Granular Synthesizer DSP.

- **Real-Time Sample Playback/Freeze**: Allows for loading audio samples for manipulation.
- **Adjustable Playback Position**: Enables control over where the sample is read in real-time.
- **Duration/Speed Controls**: Users can modify how long each grain lasts and playback speed.
- **Grain Size/Density**: Adjusts the distribution of grains over time.
- **Transposition/Pitch Shift**: Enables live tuning of grains, useful for creative sound design.

We adapted the Granular synthesis patch following the same workflow as in the FM synth:

(1) **Ensure proper audio/control mapping** to accommodate platform-specific namespaces.
(2) **Modify sample handling**: Some platforms require external storage or buffer optimization.
(3) **Optimize controls**: Mapped to CV, MIDI, or physical knobs depending on the board.
(4) **Deploy and Test**: The DSP is loaded on each platform, ensuring efficient DSP performance.

This test ultimately demonstrates the scalability of our deployment workflow, providing a structured method for embedding more advanced audio synthesis techniques into hardware.

5.4 Live Cloud Generator

The live cloud generator synthesizer demonstrates live audio input using a double buffering mechanism and object-oriented dynamic instantiation of grains. It is based on Cipriani and Giri's Microsound Asynchronous Granular Synthesis example published in their Max lesson book [4] that demonstrates Curtis Road's principles of granular synthesis [18], [17]. A set of generative parameters creates a "cloud" of grains: grain density, time stretch, grain duration, transposition, pitch quantization. All parameters have a programmable aleatoric variance. The Cipriani and Giri patch is written in Max and instantiates an arbitrary number of granular voices written in gen~. A new grain is invoked when a prior grain is 'busy' and overflows.

For this study, we made two ports of the Cipriani/Giri patch: Max RNBO and Pd. Our RNBO version allowed us to test the deployment across our platforms to consider it for this study. The Pd version allowed us to use more commonly available free software to implement the same principle. We used the clone object in Pd to instantiate grain voices.

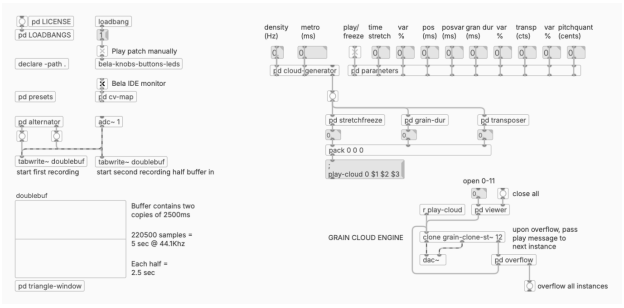

Figure 15: The Live Cloud Generator Pd patch.

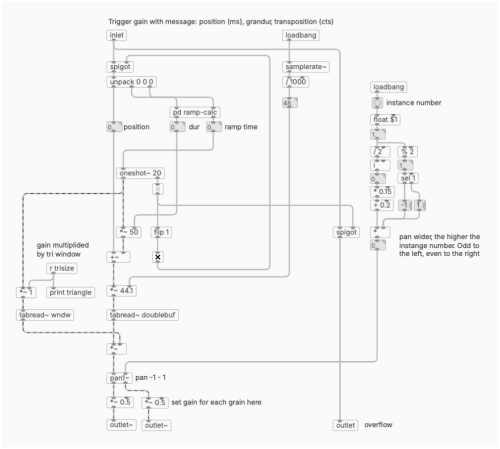

Figure 16: The Live Cloud Generator DSP.

While Cipriani and Giri's original patch granulates sampled sounds from a buffer, our adaptation allows real-time granulation of live audio input using a circular double buffer. The buffer is continuously written into by two alternating write objects: one writes to the left half, while the other, offset by half the buffer duration, writes to the right half, resulting in a seamless loop.

6 RESULTS AND DISCUSSION

We deployed our four test patches on each of our target platforms. Each patch was adapted to the target in question by replacing the Control abstraction. This process highlighted differences, inconveniences, and, at times, problems in their respective workflows. Although this is not a formal quantitative benchmarking, our observations nonetheless point to performance differences on the different platforms, across SBC and MCU systems, between compiled and interpreted patches.

Hardware specifications, features, and support can change quite rapidly. Therefore, this paper only represents a "snapshot" of these four boards that is valid at present (April 2025), as things may change in the future. In the time of our study, the OWL MCU board in Lich was updated from Cortex M4-based OWL2 to M7-based OWL3. In addition, the diversity of architectures and deployment methods makes direct, quantitative, and technical comparisons difficult. The performance results published in the following should be seen as indicative and do not represent a formal benchmarking. A summary of this comparison will be kept up-to-date in our online repository.

Although all platforms successfully executed Hello World, FM synth, and granular synth, each platform required modifications to ensure efficient playback, especially in audio buffer access and grain processing. As we will describe below, the cloud generator did not run on all platforms, and we will discuss why.

6.1 Deployment Considerations

Each platform demonstrates different trade-offs in terms of implementation and deployment. The Bela and Organelle, as SBC systems, allowed us to load Pd patches as would run on a conventional computer. All patches were run, including the cloud generator using the clone instantiation object.

In contrast, the MCU systems, Daisy and OWL, had more constraints to consider when creating a patch. They depend on the hvcc compiler, which is a limited implementation of Pd, with a list of supported[25] and unsupported[26] objects. Even with the convenience of web IDE workflows, these constraints produced errors that had to be tracked and worked around in the source patch. Overall, our findings suggest that while Pd-based workflows are effective across all platforms, additional DSP optimizations are necessary for low-power microcontrollers such as Daisy and OWL to efficiently manage intensive audio processing tasks.

6.2 Performance Considerations

Each platform required specific modifications to handle sample playback, as granular synthesis is more CPU- and RAM-intensive than FM synthesis. Adjusting the sampling rate and signal vector size for each platform was crucial in making things run without dropouts.

The following overview summarizes the sampling rates and buffer sizes for each system:

- **OWL**: 22.05 kHz, buffer size unknown
- **Daisy**: 8/16/32/48/96 kHz, 1-256 buffer size
- **Bela**: 22.05/44.1/88.2 kHz, 2-4096 buffer size
- **Organelle**: 44.1 kHz, buffer size unknown

Each platform has a means of reporting CPU or MCU load in percentage (%). The OWL reports it in the Web IDE as the patch is running, reported by the device over its USB connection. The Bela is similar, when running the patch with the Web IDE connected. The Organelle reports CPU load on an Info screen on its OLED display along with other system information such as IP address, Wifi SSID, host name, and file directory information. Daisy does not report the MCU load directly to a high-level user. For C/C++ developer programming on Daisy, the manufacturer offers a library with a CpuLoadMeter class and low-level functions for monitoring load[27]. In all cases, given the different audio sampling rates and DSP buffer sizes, the performance figures we report here do not constitute a systematic comparison.

The granular synthesizer on the OWL occupies 49% CPU with 25.99 KB of memory. The same patch on Organelle reports 13% CPU load. Bela reported 37% running at 44.1 kHz and 39% running at 22.05 kHz. This seems counterintuitive, and we are investigating this, but in the first instance it seems to be connected to the fact that setting the sampling rate automatically changes the number of analog channels, doubling from 4 to 8 when going from 44.1 to 22.05 kHz. In either case, the setting that has more bearing on performance is the audio vector size ("block size" in the Bela IDE). Although changing the block size does not necessarily have an apparent bearing on the CPU load, patches can experience buffer underrun on smaller block sizes, causing clicks.

[25] https://wasted-audio.github.io/hvcc/docs/09.supported_vanilla_objects.html
[26] https://wasted-audio.github.io/hvcc/docs/10.unsupported_vanilla_objects.html
[27] https://electro-smith.github.io/libDaisy/classdaisy_1_1_cpu_load_meter.html

Table 4: Deployment Considerations for Each Platform.

Bela	Daisy	OWL	Raspberry Pi
Linux-based, runs Pd standalone or as C++ export	Direct export in Plugdata. Heavy Compiler, unsupported objects	Eurorack integration. Hvcc, unsupported objects	Has native Pd support. Some Pis require I/O audio add-ons

Table 5: Performance and Latency Considerations.

	Bela	Daisy	OWL	Raspberry Pi
Granular Playback	Real-time sample loading	External sample buffer	Limited by patch size	Full sample streaming
CV Modulation	Multi-CV control	Pot mapping	Eurorack CV	MIDI & GPIO
Long Audio Buffers	RAM limitations	64MB SDRAM	1MB of memory	Disk-based streaming
Dynamic Patch Load	Supported	Supported	Supported	Requires restart
Overall Capabilities	Lowest latency	Higher RAM capacity	Flexible CV integration	Higher CPU power

Adjusting to a larger block size alleviates this. Daisy also allows for setting the buffer size. This is an important setting that would be useful in OWL and Organelle.

Memory size was another issue when working with audio buffers, especially on MCU systems. All code must be compiled down to one file, with which the firmware is flashed, including audio files. While an audio file can remain unloaded alongside the main patch on an SBC system, buffers need to be allocated (and loaded with data) for all audio to be used in an MCU system. This, plus the smaller memory footprint of some MCU systems, creates a kind of double bind. The OWL, for example, will not load code compiled that is greater than 512kB, severely limiting the length of samples that can be used. Whilst the Granular synth was compiled and run, we were limited in sample buffer size. With the triangle window occupying 512 samples of memory, through a process of trial and error, the maximum audio buffer we could load was 4365 samples. At a sampling rate of 22.05 kHz, this corresponds to 198 ms of sample.

Bela allowed for low-latency grain manipulation, though buffer size limitations had to be considered. Daisy, with its 64MB of SDRAM, was better suited for longer audio buffers, but required the Pd patch to be adapted with the list of supported Pd-Heavy objects. OWL, while optimized for CV-based modular synthesis, faced restrictions in patch size due to its limited memory (1 MB) as well as Heavy patching limitations. Raspberry Pi, with its ability to handle disk-based sample playback, provides computational flexibility, allowing for complex granular synthesis with minimal constraints.

6.3 Compatibility Issues

The Bela and Organelle were able to run the cloud generator while the two MCU systems failed in this patch. Hvcc does not implement the clone object, so we made a simplified version of the patch where a fixed number of grain voices were explicitly instantiated. The overflow from one voice to another occurred through a cascading chain of patching. To simplify the test further, we made a version with just one grain voice.

The original gen~ patch of Cipriani and Giri depends on a feature in the Max implementation of a phasor that it can play in a nonrepeating mode where the phasor stops when it reaches 1.0. It reports as busy when it is running. The Pd implementation of the phasor~ does not have this mode. We made an abstraction, called oneshot~ that implements this using a ramp. A line~ object goes to its destination. For the busy flag, we used the threshold~ object to give a logical output of whether the maximum had been reached. Although this workaround works in Pd and in our SBCs, hvcc does not support the threshold~ object. We tried creating a busy flag detector in the control domain, and although this seems to operate sufficiently on computer systems, it fails on the MC systems. In the end, we succeeded in compiling the patch without error in Plugdata but the compiled file does not produce sound, neither on the OWL nor Daisy. Debugging continues.

7 CONCLUSION AND FUTURE WORK

The comparison of different systems began during the *Brain-Body Digital Musical Instrument* project[28], where we implemented the granular synthesizer patch in Lich and Organelle. The cloud generator was ported from gen to RNBO to Pd and ultimately ran on the Bela Pepper. This has been presented live in concert on several occasions, including in Tanaka's *Déplacement for trio and electronics*, premiered in 2024 in Brussels at the festival Ars Musica[29], where each member of a chamber trio had their own Bela subsystem for live sampling and cloud generation. Tanaka has performed with the Bela Pepper / Cloud Generator system in a network concert connecting London and Newcastle in November 2024, and in solo concerts in Bristol and Paris in March and April 2025. We used a simplified version of the concert patch for deployment across the four systems described here.

This study looked at the deployment of Pure Data (Pd) patches on multiple embedded microcontrollers and single-board computer music platforms, demonstrating a unified workflow adaptable to various hardware architectures. We successfully implemented and evaluated four test patches of increasing complexity, achieving compatibility with three out of four patches on all tested platforms. We identified issues primarily related to differences between microcontroller-based systems lacking operating system support for external file access, limitations in memory footprint, and compatibility constraints imposed by third-party cross-compilers.

Our results highlight the significant potential of embedded musical applications, as well as the ability to deploy identical code, developed in a high-level programming environment, onto various target hardware platforms. This capability represents a significant step forward in increasing accessibility and flexibility for musicians and digital musical instrument (DMI) designers.

[28] https://bbdmi.nakala.fr/
[29] https://www.arsmusica.be/nl/evenement/eramaa-trio-quartetto-prometeo

All materials, including test patches and comprehensive step-by-step documentation, are publicly available in our online repository[30]. Our goal in sharing these resources openly is to support the broader music technology community by simplifying entry barriers for newcomers and providing clear guidance to practitioners who might lack direct access to diverse hardware systems.

Future work will further expand the toolkit presented by introducing modular widgets explicitly designed for interoperability across multiple hardware platforms. These modules will include reusable abstractions for data acquisition, advanced filtering techniques, versatile control inputs, and output formats such as Control Voltage (CV) and MIDI.

We are confident that we will be able to make the cloud generator function on microcontrollers!

We remain committed to maintaining and continuously updating our repository, ensuring that it reflects advances in embedded technologies, new Pd patches, evolving user needs, and practical case studies. Extending our efforts beyond Pure Data, we plan to incorporate additional audio-oriented programming languages and environments such as RNBO, FAUST, and SuperCollider, thereby broadening the applicability and relevance of our approach. We hope that this paper and our online resource will help musicians select the appropriate embedded system for their project, and arrive at a certain interoperability across different platforms, disentangling what can otherwise be a confusing landscape of similar systems. We hope that the insights gained here will be useful to DMI builders and to research in emerging areas such as the Internet of Musical Things (IOMT).

8 ACKNOWLEDGEMENTS

The research that led to these results has received generous funding from the Agence nationale de la recherche (ANR, France) under grant ANR-21-CE38-0018; and the Arts & Humanitics Research Council (AHRC, UK) grant AH/V009567/1.

9 ETHICAL STANDARDS

This research adheres to the ethical and professional conduct principles outlined in the NIME guidelines. No human participants, personal data or sensitive information were involved in the study.

The development and evaluation of this work was conducted openly and transparently to ensure reproducibility and accessibility. All hardware and software components used in this study are open source, clearly documented, and accessible to the public.

The authors declare no conflicts of interest, financial or otherwise. Specifically, the authors have no affiliations, partnerships, or other relationships with any of the companies or organizations behind the platforms reviewed in this research. All data and materials generated or analyzed during this study have been openly shared, adhering to established open research standards and community practices.

References

[1] Tran Nguyen Bao Anh and Su-Lim Tan. 2009. Real-time operating systems for small microcontrollers. *IEEE micro* 29, 5 (2009), 30–45.

[2] Edgar Berdahl and Wendy Ju. 2017. 2011: Satellite CCRMA: A Musical Interaction and Sound Synthesis Platform. In *A NIME Reader*, Alexander Refsum Jensenius and Michael J. Lyons (Eds.). Vol. 3. Springer International Publishing, 373–389. https://doi.org/10.1007/978-3-319-47214-0_24 Series Title: Current Research in Systematic Musicology.

[3] Edgar Berdahl, Spencer Salazar, and Myles Borins. 2013. Embedded Networking and Hardware-Accelerated Graphics with Satellite CCRMA. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 325–330. https://doi.org/10.5281/zenodo.1178476 ISSN: 2220-4806.

[4] Alessandro Cipriani and Maurizio Giri. 2023-02-05. *Electronic Music and Sound Design - Theory and Practice with Max 8 - volume 3: Vol. 3.* Contemponet.

[5] Robb Drinkwater. 2020. Small Sound: Pure Data on Raspberry Pi. In *Handmade Electronic Music: The Art of Hardware Hacking* (3 ed.), Nicolas Collins (Ed.). Routledge, 364–373. https://doi.org/10.4324/9780429264818

[6] Duncan MacConnell, Shawn Trail, George Tzanetakis, Peter Driessen, Wyatt Page, and N Wellington. 2013. Reconfigurable autonomous novel guitar effects. In *Proceedings of the International Conference on Sound and music Computing (SMC)*. Stockholm Sweden.

[7] Anthony T. Marasco. 2022. Approaching the Norns Shield as a Laptop Alternative for Democratizing Music Technology Ensembles. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.21428/92fbeb44.89003700 ISSN: 2220-4806.

[8] Andrew McPherson. 2017. Bela: An embedded platform for low-latency feedback control of sound. *The Journal of the Acoustical Society of America* 141, 5 (2017), 3618–3618. https://pubs.aip.org/asa/jasa/article-abstract/141/5_Supplement/3618/714049 Publisher: Acoustical Society of America.

[9] Andrew Mcpherson and Victor Zappi. 2015. An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black. *Journal of The Audio Engineering Society* (2015). https://www.semanticscholar.org/paper/An-Environment-for-Submillisecond-Latency-Audio-and-Mcpherson-Zappi/3050c75fb75680e3d7f731a263b37be8ac0890a6

[10] Eduardo Meneses, Johnty Wang, Sergio Freire, and Marcelo Wanderley. 2019. A Comparison of Open-Source Linux Frameworks for an Augmented Musical Instrument Implementation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 222–227. https://doi.org/10.5281/zenodo.3672934 ISSN: 2220-4806.

[11] Romain Michon, Yann Orlarey, Stéphane Letz, Dominique Fober, and Dirk Roosenburg. 2020-12. Embedded Real-Time Audio Signal Processing With Faust. In *International Faust Conference (IFC-20)* (Paris, France). https://hal.science/hal-03124896

[12] Ali Momeni, Daniel McNamara, and Jesse Stiles. 2018. MOM: an Extensible Platform for Rapid Prototyping and Design of Electroacoustic Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 65–71. https://doi.org/10.5281/zenodo.1302681 ISSN: 2220-4806.

[13] G. Moro, A. Bin, Robert H. Jack, Christian Heinrichs, and Andrew McPherson. 2016. Making High-Performance Embedded Instruments with Bela and Pure Data. In *International Conference on Live Interfaces*. https://www.semanticscholar.org/paper/Making-High-Performance-Embedded-Instruments-with-Moro-Bin/c94838fe19db5a2e9101edce3f68012b07e9819d

[14] Adam Parkinson and Koray Tahiroğlu. 2013. Composing Social Interactions for an Interactive-Spatial Performance System. In *Proceedings of the Sound and Music Computing Conference*.

[15] Teresa Pelinski and Charles P. Martin. 2024. Building NIMEs with Embedded AI. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://smcclab.github.io/nime-embedded-ai/

[16] Teresa Pelinski, Victor Shepardson, Steve Symons, Franco Santiago Caspe, Adan L. Benito Temprano, Jack Armitage, Chris Kiefer, Rebecca Fiebrink, Thor Magnusson, and Andrew McPherson. 2022. Embedded AI for NIME: Challenges and Opportunities. In *International Conference on New Interfaces for Musical Expression*. https://doi.org/10.21428/92fbeb44.76beab02

[17] Curtis Roads. 1996. *The Computer Music Tutorial*. The MIT Press.

[18] Curtis Roads. 2004. *Microsound*. MIT Press.

[19] Nobuyasu Sakonda. 2000. *Website*. http://formantbros.jp/sako/download.html

[20] Andrew Schmeder and Adrian Freed. 2008. uOSC : The Open Sound Control Reference Platform for Embedded Devices. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 175–180. https://doi.org/10.5281/zenodo.1179627 ISSN: 2220-4806.

[21] John Sullivan, Julian Vanasse, Catherine Guastavino, and Marcelo Wanderley. 2020. Reinventing the Noisebox: Designing Embedded Instruments for Active Musicians. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 5–10. https://doi.org/10.5281/zenodo.4813166 ISSN: 2220-4806.

[22] Luca Turchet, Carlo Fischione, Georg Essl, Damian Keller, and Mathieu Barthet. 2018. Internet of Musical Things: Vision and Challenges. *IEEE Access* 6 (2018), 61994–62017. https://doi.org/10.1109/ACCESS.2018.2872625

[23] Luca Vignati, Stefano Zambon, and Luca Turchet. 2022. A comparison of real-time linux-based architectures for embedded musical applications. In *Journal of the Audio Engineering Society*, Vol. 70. 83–93. https://www.aes.org/e-lib/browse.cfm?elib=21553 Publisher: Audio Engineering Society.

[24] Graham Wakefield. 2021. A streamlined workflow from Max/gen~ to modular hardware. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.21428/92fbeb44.e32fde90 ISSN: 2220-4806.

[25] Thomas Webster, Guillaume LeNost, and Martin Klang. 2014. The OWL programmable stage effects pedal: Revising the concept of the on-stage computer for live music performance. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 621–624. https://doi.org/10.5281/zenodo.1178979 ISSN: 2220-4806.

[26] Matthew Yee-King. 2025. SuperCollider on Small Computers. In *The SuperCollider Book (Second Edition)* (scott wilson; david cottle and nick collins, eds ed.). MIT Press. https://research.gold.ac.uk/id/eprint/38207/

[30] https://github.com/francesco-di-maggio/embedded-comparo