Nervebox:

A Control System for Machines That Make Music

Andrew Albert Cavatorta

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning

in partial fulfillment of the requirements for the degree of Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author May 7, 2010

Certified **Tod Machover**
Professor of Music and Media
MIT Media Lab
Thesis Supervisor

Accepted **Pattie Maes**
Chairperson, Departmental Committee on Graduate Studies
MIT Media Lab

Nervebox: A Control System for Machines That Make Music

Andrew Cavatorta

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
on May 7, 2010 in partial fulfillment of the requirements for the degree of Master of Arts and Sciences

Abstract

The last 130 years of musical invention are punctuated with fascinating musical instruments that use the electromechanical actuation to turn various natural phenomena into sound and music. But this history is very sparse compared to analog and PC-based digital synthesis.

The development of these electromechanical musical instruments presents a daunting array of technical challenges. Musical pioneers wishing to develop new electromechanical instruments often spend most of their finite time and resources solving the same set of problems over and over. This difficulty inhibits the development of new electromechanical instruments and often detracts from the quality of those that are completed.

As a solution to this problem, I propose Nervebox — a platform of code and basic hardware that encapsulates generalized solutions to problems encountered repeatedly during the development of electromechanical instruments. Upon its official release, I hope for Nervebox to help start a small revolution in electromechanical music, much like MAX/MSP and others have done for PC-based synthesis, and like the abstraction of basic concepts like oscillators and filters has done for analog electronic synfthesis. Anyone building new electromechanical instruments can start with much of their low-level work already done. This will enable them to focus more on composition and the instruments' various aesthetic dimensions.

The system is written in Python, JavaScript and Verilog. It is free, generalized, and easily extensible.

Thesis Advisor: Tod Machover, Professor of Music and Media

Nervebox: A Control System for Machines That Make Music

Thesis Committee

Advisor		**Tod Machover** Professor of Music and Media MIT Media Lab
Reader		**Cynthia Breazeal** Associate Professor of Media Arts and Sciences MIT Media Lab
Reader		**Leah Buechley** Assistant Professor of Media Arts and Sciences MIT Media Lab
Reader		**Joe Paradiso** Associate Professor of Media Arts and Science MIT Media Lab

Acknowledgements

I am very thankful to -

Tod Machover for his inspiration and support

Leah Buechley for her practical guidance

Cynthia Breazeal and Joe Paradiso for their patience and inspiration

Pattie Maes for her invaluable support during my application process

Dan Paluska and Jeff Leiberman for sharing the details of their spectacular machines

And Marina Porter for more than I can list

Table of Contents

List of Illustrations

List of Photos

List of Figures

1 Introduction

This thesis documents Nervebox, a hardware and software platform providing a general control system for electromechanical musical instruments.

Since the time of Thaddeus Cahill's Telharmonium, musical experimenters have generally spent more of their time re-solving the same technical problems than creating music [1]. This has had a detrimental effect on the whole field of experimental electronic and electromechanical music in two ways. First, time spent on technical problems is time not available for musical and aesthetic experimentation, though there is a small potential overlap. Second, the difficulty of the technical problems has created a barrier to entry for many potential musical pioneers.

This was the state of PC-based sound synthesis before it was revolutionized by mature software like MAX/MSP, Chuck, Supercollider, cSound, and others. These have freed experimental musicians from needing to each re-invent low-level synthesis before being able to start making music [2].

I am hopeful that bringing a similarly-enabling platform to the field of electromechanical music will catalyze a slow but ever-growing explosion in new types of music and expression.

An effective platform for developing electromechanical instruments must include a way to abstract the system's necessary internal complexity into a set of simpler concepts that combine in powerful ways. While electromechanical musical instruments vary wildly in their designs, there are commonalities among nearly all of them that can be used to simplify the ways we imagine and create them. Such a system must also be able to represent musical data in a way that is rich enough to encompass the expressive dimensions of the input devices and open enough to accommodate the musical subtleties of never-before-imagined instruments.

This abstraction of the elements of electromechanical music, with a focus on representation, is the subject of this research.

I think of it as a nervous system that brings music into machines.

2 Electromechanical Musical Instruments

2.1 Definition

All musical instruments are cultural artifacts, and can be categorized into a boundless number of ontologies. For example — musical styles, tuning systems, note ranges and timbres, cultural origins, or the mechanics of sound production. The definitions of these categories serve to describe their location in an ontology and differentiate them from their ontological neighbors.

As all musical instruments are machines, they can be categorized by their underlying technologies. It is into this ontological tree that I am placing my definition of electromechanical musical instruments.

Definitions exist for many types of instruments using modern technologies: electo-acoustic instruments, hybrid digital-acoustic percussion instruments[3], prepared pianos, etc. I have not found in the literature a clear general definition of electromechanical musical instruments, perhaps because they are often taken for granted as a superset of more specifically-defined types of instruments. So I will originate a definition for the purposes of this thesis.

I am defining electromechanical musical instruments as instruments that use electromechanical actuation to produce motions that generate musical signals.

These signals may be acoustic, directly generating sound. They may be electronic, made audible through an amplifier and loudspeaker. Or they may exist in various other media, such as wave energy in water or resonating strings.

This definition is intentionally broad, but different from its ontological neighbors. Analog or digital synthesizers are not electromechanical musical instruments because they do not generate their musical signals using electromechanically-induced motion. There is an overlap between electromechanical musical instruments electro-acoustic instruments[4]. But electro-acoustic instruments that generate their musical signals using synthesizers, samples, or recordings do not fit this definition of electromechanical musical instruments. Prepared pianos, on the other hand, are a subset of electromechanical musical instruments.

2.2 Selected Historical Examples

Elisha Gray is generally credited with inventing the first electromechanical musical instrument, the Musical Telegraph, in 1876 [5]. The Musical Telegraph was a small keyboard instrument which used a series of tuned primitive oscillators to vibrate a series of metallic bars. In the language of the patent, in which it is called the "Telephonic Telegraph", we can see Mr. Gray needing to explain ideas and abstractions that we can call by single-word names today.

The patent begins:

> "Be it known that I, Elisha Gray, of Chicago, in the county of Cook and State of Illinois, have invented certain new and useful improvements in the art of and apparatus for generating and transmitting through and electric circuit rhythmical impulses, undulations, vibrations, or waves representing composite tones, musical impressions, or sounds of any character or quality whatever, and audibly reproducing such impulses, vibrations, or waves, of which art and apparatus the following is a specification."

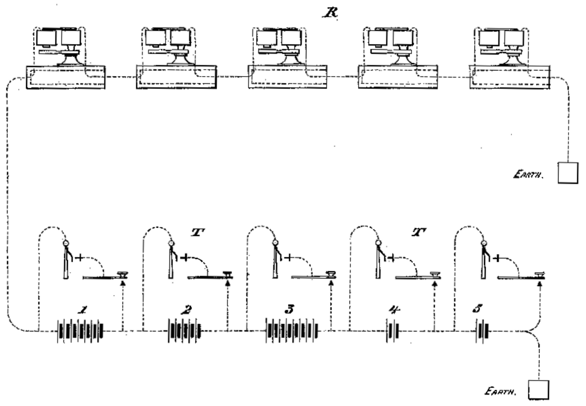


Illustration 1: From patent for Elisha Gray's Musical Telegraph, showing an array of buzzers on top and an array of batteries and primitive oscillators below.

The Musical Telegraph contained the seeds of the modern synthesizer: a keyboard, oscillators, and a predecessor of the loudspeaker. It also contained the seeds of the telephone, for which he famously lost the patent rights by submitting his patent one hour later than Alexander Graham Bell's.

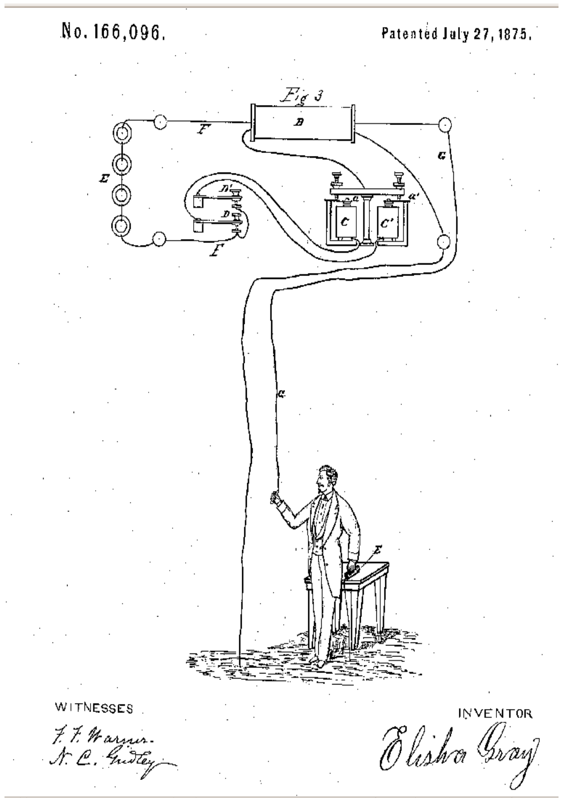


Illustration 2: Elisha Gray's patent for the "Art of Transmitting Musical Impressions or Sounds Telegraphically"

Mr. Gray was prescient enough to see the potential for transmitting music over distances and to multiple receivers. He also filed a patent for an "Electric Telegraph for Transmitting Musical Tones" [6]. This leveraged the ubiquity of telegraph lines, using them as a transmission network for music.

Thaddeus Cahill extended that concept in 1897 with the completion of his first Telharmonium[7], the Mark I. The Telharmonium, also called the Dynamophone, leveraged the telephone and telephone network for music transmission.

Music was played by live musicians on unique and complex keyboards that were inspired by the consoles of church organs[8]. Pressing the keyboard keys closed circuits between enormous electromechanical dynamos and telephone lines. The music could be heard through the telephone by asking a telephone operator to connect you to the Telharmonium.

The instrument preceded the invention of the electrical amplifier, requiring a signal generation process which switched a volume of electrical power unusual for any musical instrument. He describes the signal generation in his 1895 patent application:

> "By my present system, I generate the requisite electrical vibrations at the central station by means of alternating current dynamos, or alternators, as we may briefly term them. ... The musical electrical vibrations which I thus throw on the line are millions of times more powerful, measured in watts, than those ordinarily thrown upon the line by a telephone microphone of the kind commonly used, ..."

The alternators produced clean, sine-like waves. The sound was pure and sweet, but lacked character and timbral variety. The Telharmonium could produce more complex timbres by borrowing a technique from pipe organs. Pipe organ consoles feature a control interface called organ stops, which open and close the airflow to ranks of pipes which vary by timbre or octave range. Opening different stops will cause any note pressed on the keyboards to be expressed on different ranks of pipes, thereby producing different timbres. Multiple stops can be opened simultaneously to produce complex combinations of timbres.

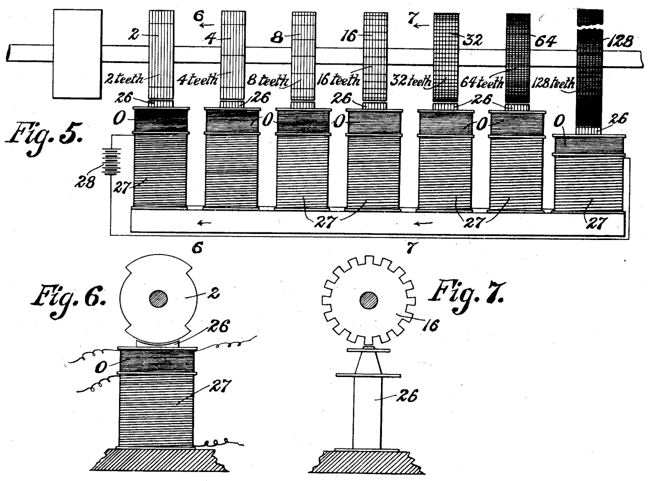


Illustration 3: The alternators of Thaddeus Cahill's Telharmonium

Cahill's patent includes a set of sliding drawbars, an affordance enabling players to add various harmonics to any note played on the Telharmonium. The additive synthesis of multiple harmonics is acoustically similar to the simultaneous sounding of multiple ranks of organ pipes.

The Mark I weighed 7 tons. It was followed by the Mark II and Mark III, which each weighed 200 tons.

The enormous mass of these instruments echoes the enormity of the challenges facing early pioneers of electromechanical music. The illustrations from the patents remind us that these inventions came from a time when every component had to made by hand from a limited palette of materials. These economics and the general lack of knowledge about electricity are enough to explain the sparse development efforts during the early years of electrical invention.

These instruments may seem a bit crude and naïve. But the times were not naïve mechanically or musically. This was the short-lived golden age of mechanical music, in which the concepts of the player piano and the barrel organ combined and mutated into the orchestrion — a pneumatically-actuated whole-orchestra-in-a-box, including piano strings, organ pipes, woodwind instruments, drums, cymbals, wood blocks, and more.

The most sophisticated models contained 3 or 4 full-sized violins, which were fingered by felted mechanical paddles and bowed by an ingenious circular horsehair bow. The speed and pressure of the bow, the fingering of notes and even vibrato, all of this musical expressivity was actuated by pneumatically-powered mechanical components. The score was encoded in holes punched on a wide paper roll which was read pneumatically.

We may have seen the development of more sophisticated, electrically-actuated orchestrions if it were not for the explosion in popularity of radio in the early 1920s. The Musical Telegraph, the Telharmonium, the Phonograph, the orchestrion, and the radio were all attempts to provide music without the need for musicians. Each had their drawbacks. But radio was the clear winner by the 1920s [8].

The mid-20th Century brought the Hammond Organ, which borrowed many ideas from the Telharmonium. Laurens Hammond's 1934 patent[9] entitled "Electrical Musical Instrument" shows an instrument featuring racks of spinning tonewheels which power "alternators", drawbars controlling additive synthesis of harmonics, and complex custom keyboards inspired by pipe organs.

Unlike previous electromechanical instruments, which were all commercial flops,   Hammond organs were wildly popular. The

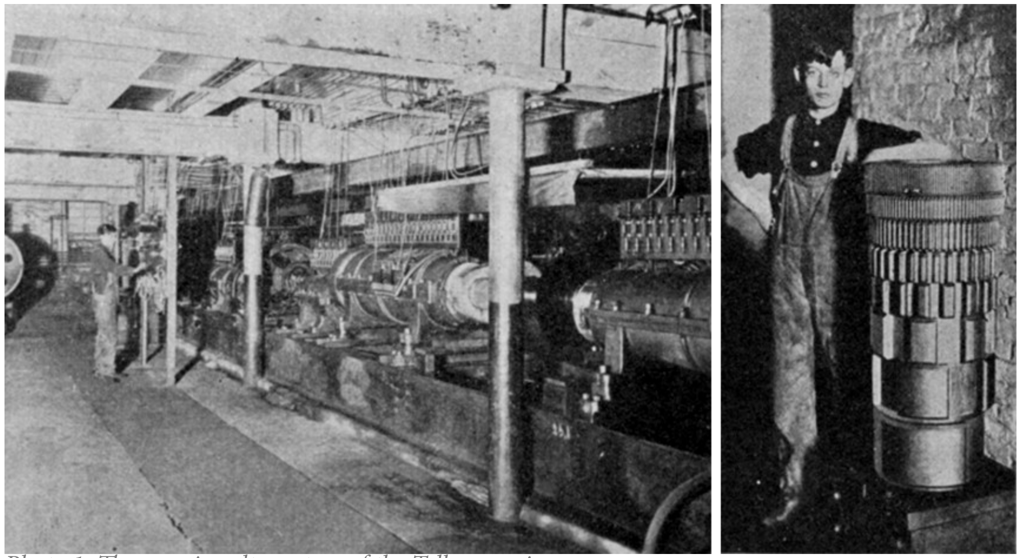


Photo 1: The massive alternators of the Telharmonium

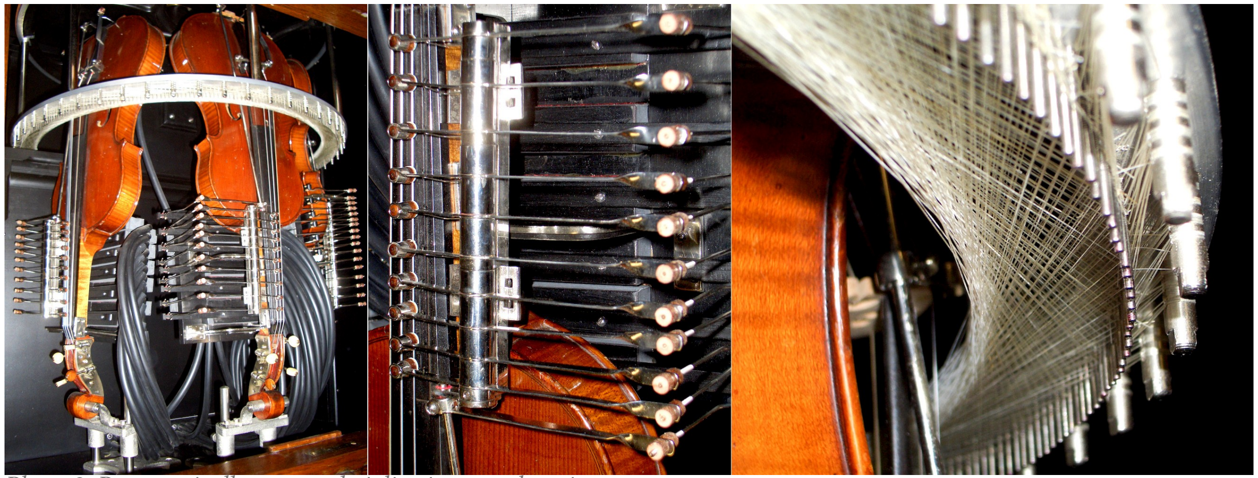


Photo 2: Pneumatically-actuated violins in an orchestrion.

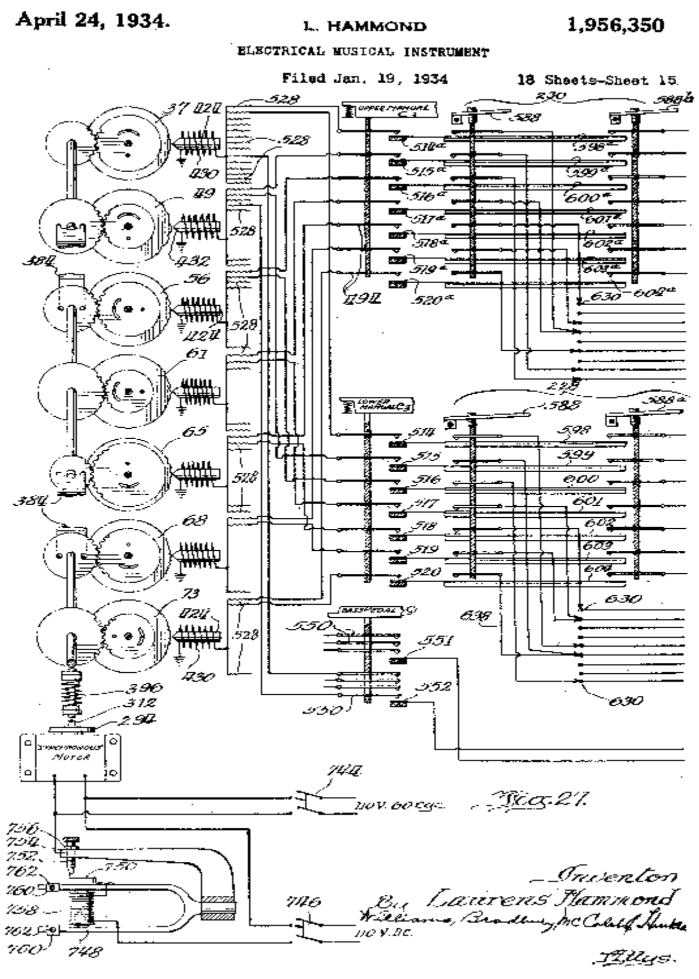


Illustration 4: Laurens Hammond's 1934 patent shows how the Hammond tonewheels and alternators echo designs used in the Telharmonium

Hammond Organ Company produced 31 major electromechanical models between 1935 and 1974.

Many models included other electromechanical features such as a Leslie rotating speaker cabinet and *vibrato scanner* [10]. The Hammond vibrato scanner produces a vibrato effect through an impressive electromechanical method involving a primitive electronic memory written to via the capacitive coupling of rotating plates.

The 1960s brought Harry Chamerlin's Mellotron, a keyboard instrument in which each key triggered playback of samples of approximately 8 seconds each[11][12]. This instrument's sound generation process seems less physical, as it is essentially a multichannel tape player connected to a keyboard. But it is interesting as a link between the golden age of electromechanical instruments and the present age of music composed of samples.

The Hammond Organ, Mellotron, and other electromechanical instruments of the mid-20th century eventually fell out of fashion. They were heavy, delicate, and expensive to develop and maintain. They were also, to some degree, novelty instruments. And new novelties continued to arrive.

The arrival of commercial modular synthesizers by R.A. Moog

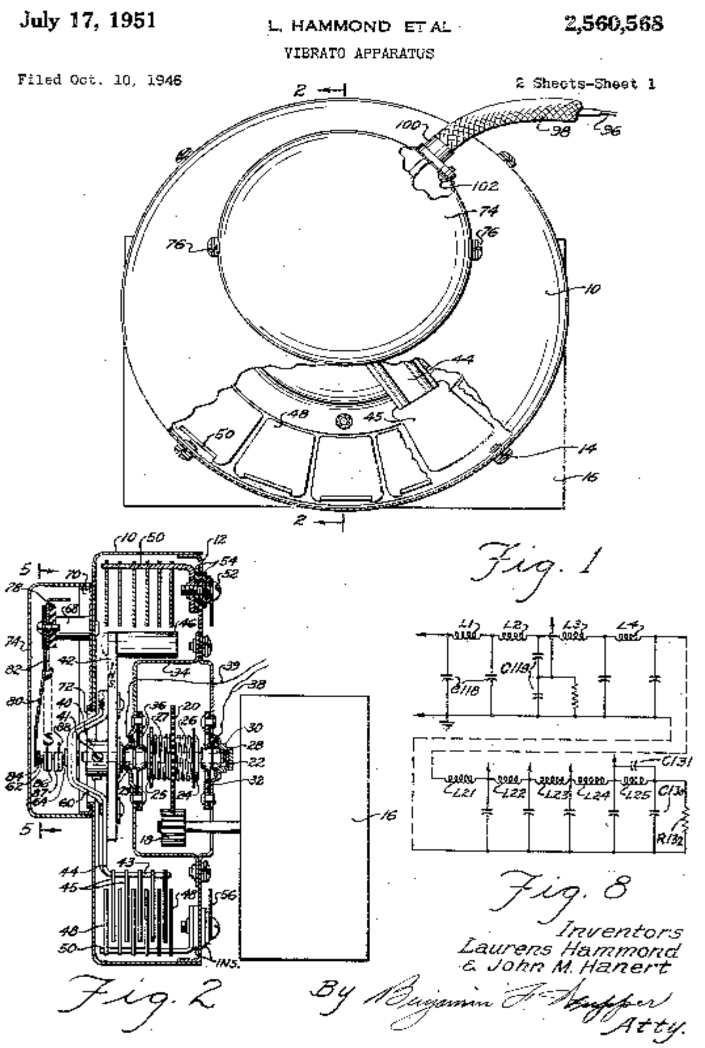


Illustration 5: Detail from patent for "Hammond Vibrato Apparatus".

Company and Buchla & Associates in 1973 introduced a new direction in keyboard instruments that was more portable and offered exciting new sonic frontiers [13]. The first commercial digital samplers were introduced in 1976 and 1979. By the late 1980s, a new sample-based popular music aesthetic was overtaking the synth-pop of the early- and mid-1980s. By the late 1990s, PC-based music composition and performance was providing far more options than any dedicated sampler or sampling keyboard.

2.3 Art, Maker Culture and Electromechanical Music

Surprisingly, we are entering another age of electromechanical music — one of greater experimental and creative breadth than any before it.

These new instruments are not intended for mass markets. They are unique and individual, emerging from the intersection of sound art, installation art, robot fetishism, maker culture, and musical innovators pushing beyond the world of laptop music.

It is misleading to post just a few examples, as there are more new machines than I can ever keep up with. But here are 4 interesting examples:

Tim Hawkinson's Überorgan [14] features 11 suspended air bladders the size of city buses and forces air from them through various devices and actuated membranes to produce sound and music. The score is painted on a very long plastic sheet (at right in Photo 5, below) and read as the sheet is scrolled by motors across an array of photosensors. Part of its appeal is the absurdity of it size and its exaggerated physicality.

LEMUR's Guitar-bot [15] is comprised of 4 identical units which play together as a single instrument under computer control. Each unit can pluck a guitar string and mechanically actuate fingering and glissando along a fretless fingerboard. It does not represent a new way to make music. But it is fascinating to watch and is clearly informed by a heavy dose of robot fetishism.

Ensemble Robot's Whirliphon [16] spins 7 corrugated tubes at precisely controllable speeds to produce 3 octaves of continuous musical notes. It's interesting because it is the first playable instrument to create music in this way. Its unique timbre has been described to me as "a chorus of angry angels" and "kind of like sniffing a whole fistful of magic markers".

Dan Paluska and Jeff Lieberman's Absolut Quartet [17] is comprised of 3 multi-segment instruments. The most memorable and impressive is the Ballistic Marimba, which launches rubber balls in parabolic arcs, landing them on specific marimba bars at precise times. This adds a unique performative value: the pleasure of tension, expectation and resolution in both the visual and aural modalities.

2.4 Electromechanical Music vs. Electronic Synthesis

Why would musicians and musical inventors bother to create electromechanical musical instruments in 2010, when digital samplers and digital synthesis are so accessible, ubiquitous, easy and inexpensive? In place of a scientific explanation, I offer 4 arguments from personal observation.

2.4.1 Acoustic Innovation

Electromechanical instruments open the potential to create music in entirely new ways. There are natural phenomena that create sound, but require the precision control of a machine to make music. To name just a few: spinning corrugated tubes, polyphonic musical saws, synchronized water droplets, artificial larynges, the chamber resonance of architectural spaces, and the highly-expressive-but-nearly-impossible-to-play daxophone[].

2.4.2 Performance: visible creation vs. music from a laptop

Digital performances using sequencers or other software can face a

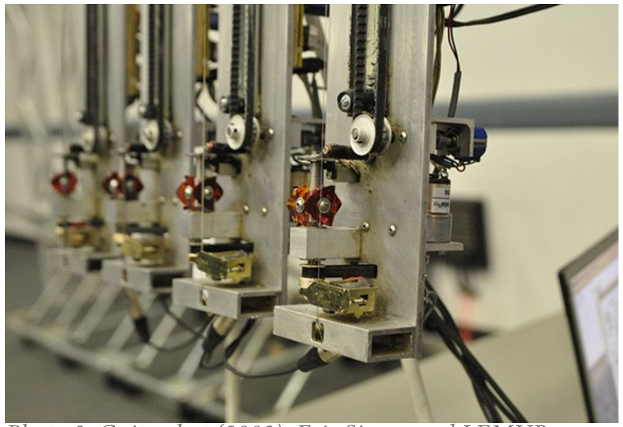


Photo 3: Guitar-bot (2003), Eric Singer and LEMUR

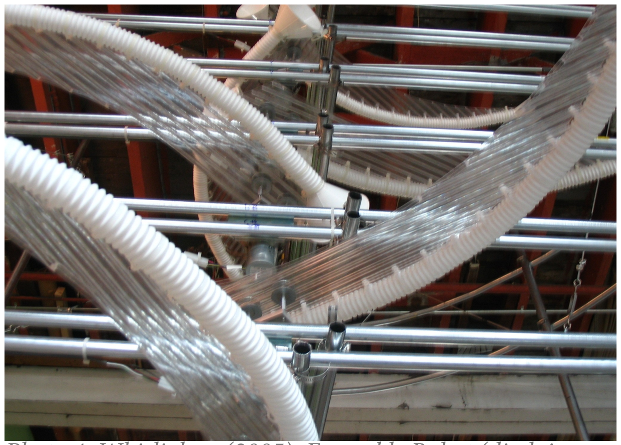


Photo 4: Whirliphon (2005), Ensemble Robot (disclaimer: I designed this instrument)

Photo 5: The Uberorgan (2000), Tim Hawkinson at MassMoCA [Photo by Doug Bartow]

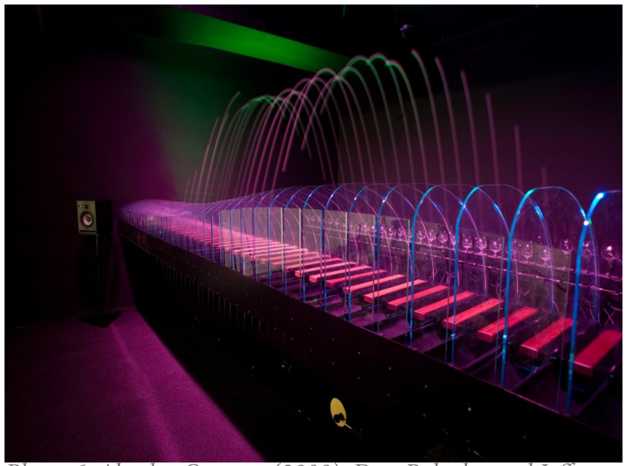


Photo 6: Absolut Quartet (2008), Dan Paluska and Jeff Lieberman

serious problem: The audience cannot see digital music being created. There is no visual causation. This can leave an audience feeling disconnected from the performance. Some performances add light shows, dancers, live experimental projections, etc. But a feeling that nothing is "happening" can persist.

In many of the new generation of electromechanical musical instruments, the audience can see the physical motions that create the music. This can be very compelling, and at its best, downright wondrous and hypnotic.

Dan Paluska and Jeff Lieberman's Absolut Quartet and LEMUR's Guitar-bot both demonstrate this hypnotic quality very well.

2.4.3 Acoustic Richness: [electro]acoustic vs. digital

The naturally rich acoustic sounds of the physical world have a complexity and physicality that many digital sources strive unsuccessfully to match. These rich sounds of the physical world are full of emotional associations, making them musically accessible and semiotically numinous.

The Whirlyphon is an excellent example of this. Much of its unusual timbre comes from the glassy-sounding interaction of upper harmonics. There are many arguments about which complex sounds can be reasonably synthesized. But they are moot in this case, as even high quality speakers cannot reproduce this highly spatialized sound — including the way in which the geometry of the Doppler effect on the spinning tubes changes with the listeners' proximity to the instrument.

2.4.4 Contribution: new instruments vs. software with new configurations

Electromechanical musical instruments remain a relatively unexplored frontier. There is still the opportunity to create profoundly new and compelling instruments, sounds, music, and performance experiences. The excitement created by Tim Hawkinson's Überorgan is among the best examples of success based on spectacle..

2.5 The Barrier

These are all good reasons to make electromechanical music. So why , then, would musicians and musical inventors not want to create electromechanical musical instruments?

Creating an instrument of expressive quality, as opposed to a sound effect, can be an arduous undertaking. The creation of articulate sound is an art and a science. And it is also technically challenging. Section 2.5.1 shows a real-world example of the problems that are solved over and over again.

The technical challenge has had a detrimental effect on the whole field. It sets a high technical barrier to entry for musical explorers. It limits the production of high-quality instruments because their creation requires a high degree of technical and aesthetic skill. And it limits the quality of the music created, as most of an explorer's finite time, attention and ingenuity go into engineering rather than composition. [1]

2.5.1 Example: Absolut Quartet

Dan Paluska was kind enough to send me a summary of the control system he and Jeff Lieberman developed for the Absolut Quartet. It makes an excellent example of the set of problems facing creators of electromechanical musical instruments. Dan explained their control system to me as a list of *electronic paths*, as shown below.

Figure 1: Electronic Paths in the Absolut Quartet System

🟦				1	Flash interface receives melody input from user
🟦				2	Max/MSP patch receives text packet of notes and times
🟦				3a	Computer analyzes some and expands into ~2 1/2 minute song using an equation composition template.
🟦				3b	MIDI score is appropriately filter for note ranges, allowed speed of note firings(reload time).
🟦		🟨		3c	Pre-delays are added to account for air time of the rubber balls.
🟦				4	Computer outputs data as MIDI
	🟥			5	Doepfer MIDI-to-TTL interface converters MIDI notes into on/off signals
	🟥			6	Custom buffer board queues TTL signals and routes them
	🟥			7	Control network routes signals to actuation sites.
		🟨	🟩	8	Custom boards local to each ball shooter, wine glass, or percussion element that take TTL pulse and do some local control specific to the instrument.
		🟨	🟩	9a	Marimba Shooters: a sequence of 4 timed operations which fires and then reloads the shooter.
			🟩	9b	Wine Glasses: solenoid pull
			🟩	9c	Percussion: solenoid pull with 8 levels of strength for midi volume.

Key to color tags in Listing 1:

🟦	mapping input data to an internal musical representation
🟥	routing the music data to multiple output devices
🟨	mapping the musical data into actuation control
🟩	actuation circuitry

The color tags above show how the tasks of the electronic paths can be abstracted into tasks common to all electromechanical musical instruments: mapping input data to an internal musical representation, routing the music data to multiple output devices, mapping the musical data into actuation control, actuation circuitry.

Developing solutions to handle these tasks required commercial data conversion products and multiple custom circuit boards, the invention of an internal data format (on top of MIDI), custom circuitry to map the musical data to actuation, custom motor controllers, and the solving of many smaller problems within each task.

3 Nervebox

3.1 The Big Idea

While electromechanical musical instruments vary wildly in their designs, there are commonalities among nearly all of them that can be used to simplify the abstractions by which we imagine them and to expedite the processes by which we create them.

To that end, I present Nervebox, a hardware and software platform, as a generalized control system for machines that make music.

3.2 Abstractions and Processes: Evolution of Electronic Music

Abstractions matter, intellectually and economically. For instance, the collective development of higher abstractions in electronics has enabled an economy of portable ideas and modular components. Shared, portable ideas are needed to build a culture which supports a technology. And modular components representing those abstractions transform the design and development processes, empowering experimenters and engineers with to build with greater complexity and speed.

We can see the evolution of abstractions and processes in electronics in the patents already referenced.

Though this diagram (Illustration 6) of the Telharmonium's alternators does contain some symbols for electrical abstractions such as wires and inductive coils, it is mostly defined in very physical terms: materials, tolerances, springs, blocks, diameters of wire, numbers of windings. Cahill could not treat these parts as modular components because every component had to be made and tested by hand [8].

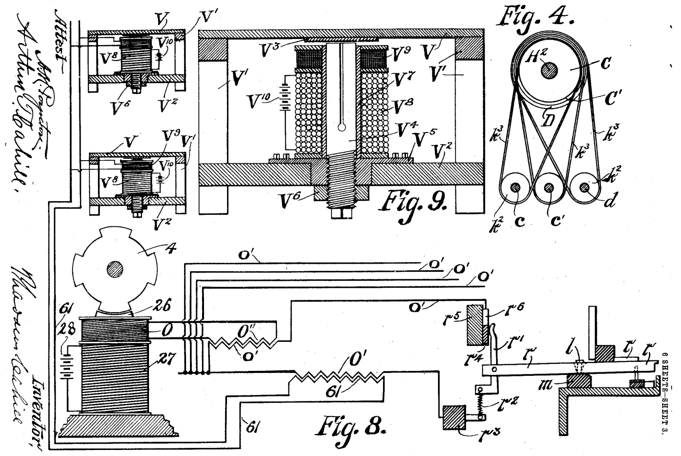


Illustration 6: diagram of alternator circuits from 1897 Telharmonium patent

37 years later, this diagram (Illustration 7) of the Hammond organ's alternators is more schematic and abstract, focusing more on electrical concepts and taking most of the materials and components for granted. This level of abstraction describes far greater complexity than the

previous diagram.

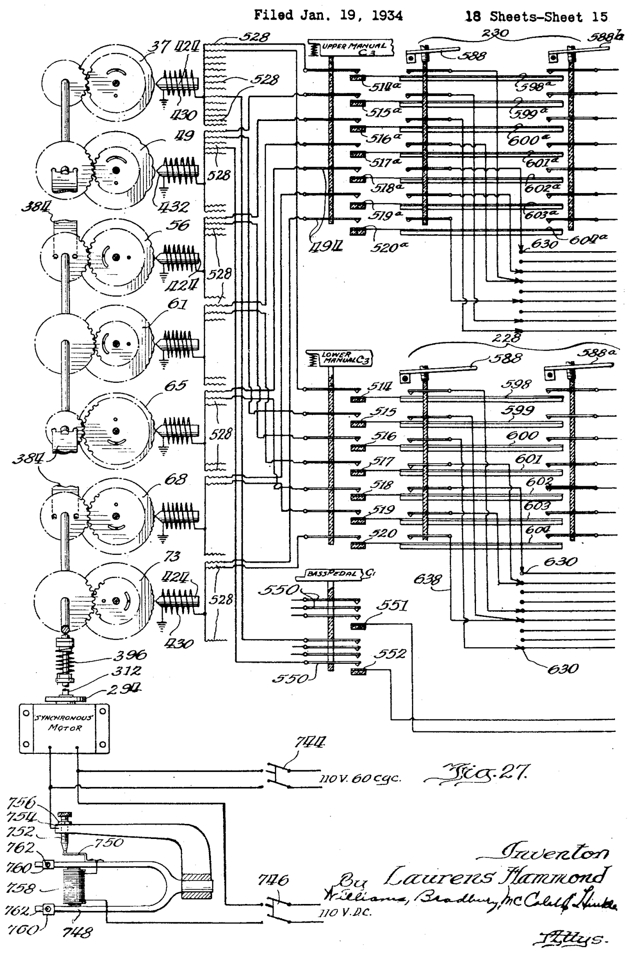


Illustration 7: diagram of alternator circuits from 1934 Hammond patent

41 years later, in 1975, we see the continuing evolution of abstractions in Robert Moog's patent for his first commercial modular synthesizer. The schematic diagram in Illustration 8 describes the circuitry almost entirely in modular blocks, high above the level of by-then-cleanly-abstracted standard electronic components. Once again, this level of abstraction describes at least one order of magnitude more complexity than the diagram in the previous patent.

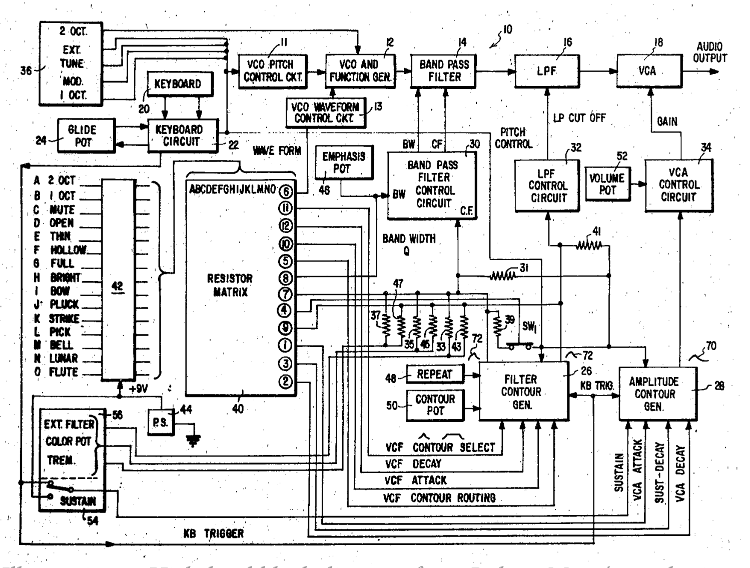


Illustration 8: High-level block diagram from Robert Moog's synthesizer patent

It also echoes advances in the design process. Wrapping complex circuits in reductive abstractions frees engineers and experimenters from needing to invest their time and ingenuity in lower-level tasks, such as making precise resistors from scratch, or stable voltage-controlled oscillators. Portable abstractions such as various types of oscillators, amplifiers, and filters continue to co-evolve with commercially available standardized components, enabling engineers and experimenters to think and build at increasing levels of abstraction and complexity.

A similar evolution has taken place in the field of digital synthesis. In 1966, when Paul Lansky was beginning to compose music on digital computers, the very basics of digital synthesis were just being developed[18]. Making music with digital computers required a significant knowledge of algorithms, music theory, and the workings of mainframe computers. His work process involved writing instructions on stacks of punch cards, waiting for his job to write the instructions to digital tape, and carrying the tape across the street to "play" on another computer. Composing his first piece took one and a half years. He was so surprised and disappointed by the results that he destroyed all evidence of the piece.

Today, anyone with access to a PC can compose music in real-time with digital synthesis. No knowledge of algorithms, music theory, or computer science is necessary. Various music software packages such as SuperCollider, Digital Performer, cSound, and PureData hide these complexities under the surfaces of high-level abstractions. This simplicity, which brings computer-based composition processes within the reach of millions, has precipitated a boom in new music and musical ideas[19].

Electromechanical music technology, by comparison, has not gone though a similar evolution in the last 50 years. It still lacks the level of empowering abstraction found in analog and digital synthesis technologies. One result is that musical explorers working with electromechanical music must invest significant time and ingenuity solving low-level problems from scratch.

3.3 Nervebox Abstraction

The Nervebox platform encapsulates the inherent complexity of control systems for electromechanical music into a set of general abstractions that can be used to bring music into nearly any electromechanical musical instruments, musical robots, or sound installations. It is not limited to any particular type of music, actuation, or sound-producing natural phenomena.

Illustration 9 shows the Nervebox platform's abstraction of the

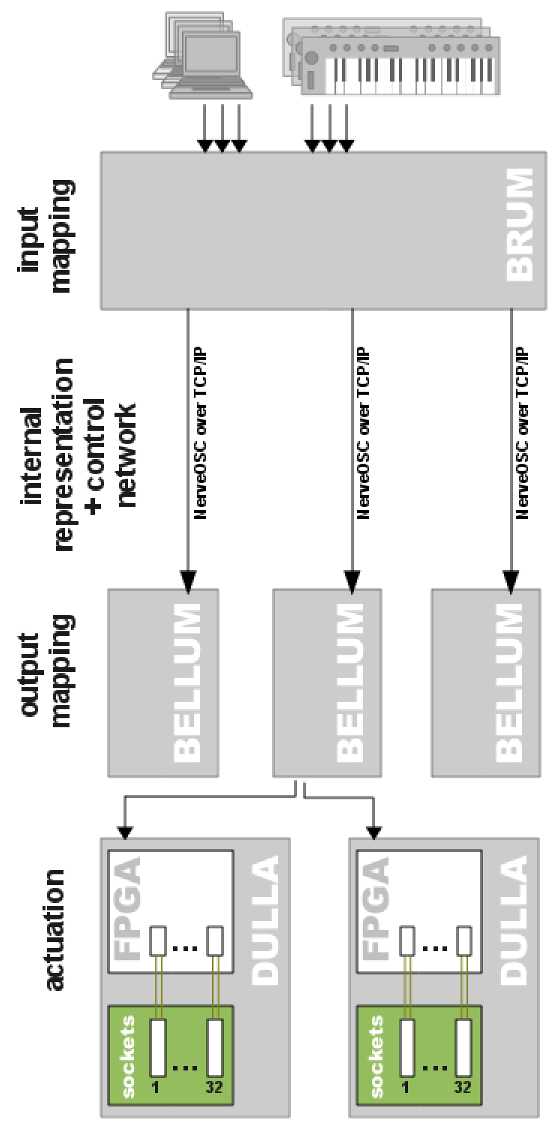


Illustration 9: Top-level view of Nervebox abstraction

functions that are common to almost all electromechanical instruments. These are abstracted into 5 components: input mapping, internal representation, control network, output mapping, and actuation.

The names of some of the abstractions are inspired by names of brain structures: cerebrum, cerebellum and medulla. The Brum interprets diverse inputs and abstracts them into a common representation. It manages the user interface (Nervebox UI), stores mappings and configurations, and coordinates the actions of the Bellums. Each Bellum receives abstracted musical data from the Brum and converts it into machine control commands appropriate to its instrument. Since each type of instrument is different, each Bellum is configured differently. This pushes the various instruments' differences out to the periphery of the architecture. The Dulla is the actuation interface, where the bytes meet the volts. It controls motors and other actuators. It also reads data from sensors for closed-loop operations.

3.3.1 Input Mapper - The Brum

In this system, mappings convert one form of data to another, and often serve musical and aesthetic purposes in the process. The Nervebox platform assumes there will be one or more simultaneous streams of input. Capturing and encoding these streams is the first function of the Brum, or input mapper. Different stream types are handled by different

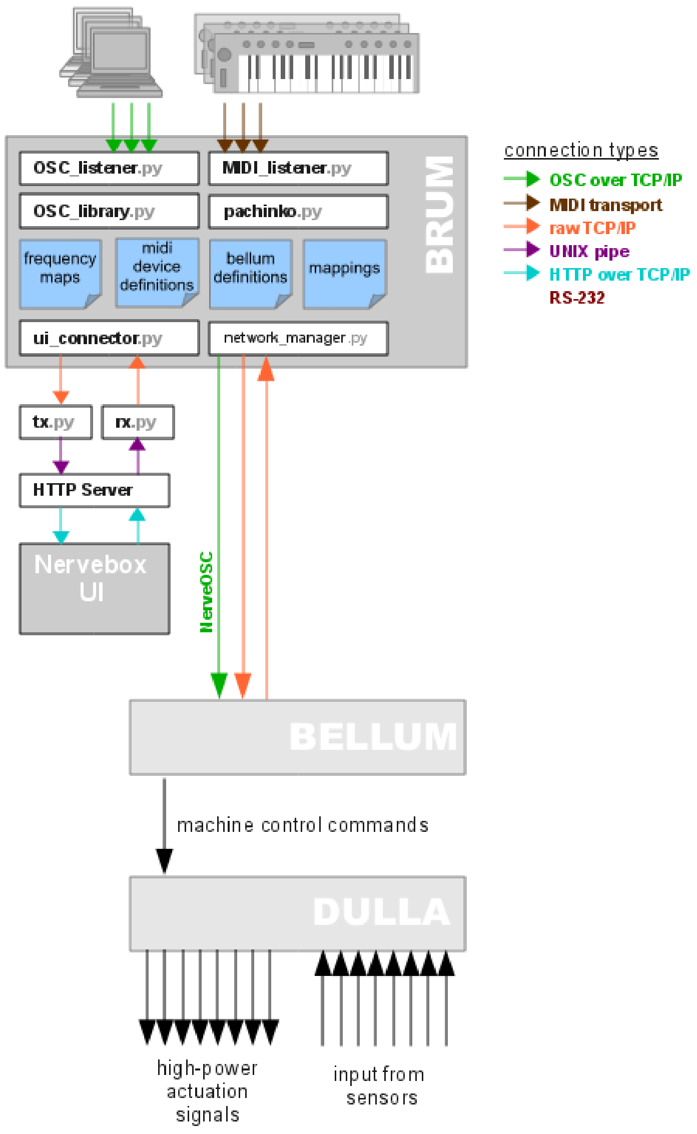


Illustration 10: Detail of Brum

modules, making it easily expandable to new input types. The next function of the Brum is to convert elements of the incoming data into musical events and assign them to one or more instruments. The output of the Brum is a stream of musical events encoded in a unified format that serves as the Nervebox platform's internal musical representation.

3.3.2 Internal Music Representation - NerveOSC

In all electronic and electromechanical musical instruments, music is abstracted into an internal data representation that can be processed, manipulated, mapped and routed. This may be analog or digital, single- or multichannel, serialized or real-time.

MIDI is a great standard and has enabled a revolution in electronic music. But MIDI's reductiveness and limitations cause many musical inventors to find it necessary to create their own formats. Even when these formats piggy-back on top of MIDI, they are often proprietary, ad-hoc, time-consuming to create, and not portable.

The Nervebox platform represents data in a unique flavor of the Open Sound Control format [20], called NerveOSC. I chose OSC over MIDI because its address patterns and flexible data arrays make possible a data format which can describe complex musical concepts within the clear semantics of the format, as opposed to the ad-hoc and convoluted hacks of MIDI. NerveOSC is intended to be able to reasonably represent all the richness of musical expression created by input devices and all of the musical and timbral possibilities of any instruments used as output. This is covered in greater detail in section 3.4 - Detail of NerveOSC.

3.3.3 Control Network - TCP/IP

Most systems require an electronic network to route their inputs and internal signals to multiple devices or actuators. For example, the Telharmonium used the telephone network and the Hammond organ used matrices of wires from the manuals to the tonewheels. NerveOSC is built on top of OSC, which uses TCP/IP and UDP as its wire-level protocols. Basing Nervebox's control network around TCP/IP eliminates the need to create a proprietary wire-level protocol.

3.3.4 Output Mappers - The Bellums

This mapping layer takes NerveOSC data as input and maps it to machine control commands that drive the electromechanical actuation that creates music. In doing so, it abstracts the mechanical and electronic details away from the rest of the system. One Bellum will exist for each instrument or major component thereof. And separate code modules will be required by different types of instruments (see 3.8 Development Process below).

3.3.5 Actuation Control - The Dulla

This abstraction layer is the final stage where the bytes meet the volts (that drive the machines that make the notes). Here I define actuation as the mechanisms that convert machine control signals into musically vibrating air. This could be an electric organ's motorized tone wheels and speaker, motors spinning corrugated tubes, solenoids striking resonant metal chimes, or the bellows and pneumatic valves of a church organ. The possibilities are boundless. Actuation has 2 components: the acoustic machinery that vibrates the air and the electromechanical systems that control that machinery.

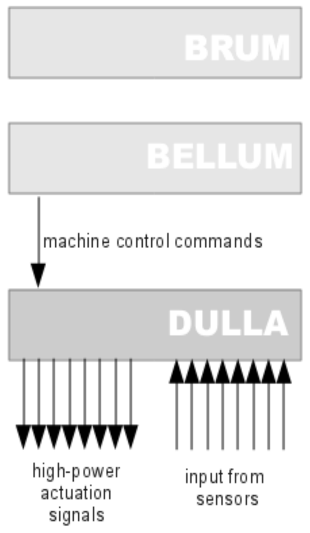


Illustration 11: Black box view of Dulla

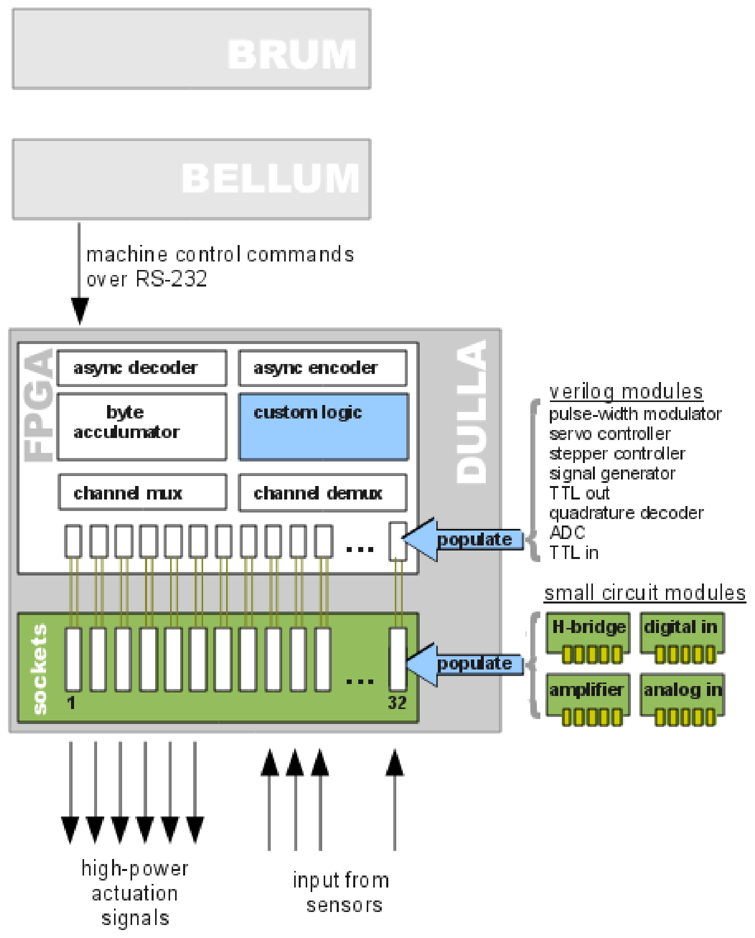


Illustration 12: Detail view of Dulla

Any attempt to standardize the acoustic machinery that vibrates the air will be working against the innovative spirit I'm seeking to support and promote.

But the electronic control of the machinery can be abstracted in this way: From a gross perspective, the Dulla is a black box that receives standardized machine commands from the Bellum and produces the precisely-timed high-current signals that drive the instrument's actuators. In closed-loop actuation systems, there are also lines of sensor data running from the instrument back to the black box.

Within that black box are 2 layers. The first is an FPGA that receives machine control commands from the Bellum. Almost all machine control circuitry is created within the FPGA: signal generators, PWM sources, H-bridge logic, stepper motor controllers, A/D converters, quadrature decoders, and more. Compared with microcontrollers, FPGAs are well suited here because of their ability to perform multiple time-sensitive tasks literally simultaneously. Compared to discrete electronic components, FPGAs are compact and very power-efficient.

But most importantly, they enable this platform to use one standard set of electronic hardware to perform any and all machine control tasks. And FPGAs are configured with Verilog or VHDL code, making complex circuitry as portable and easily reproducible as software.

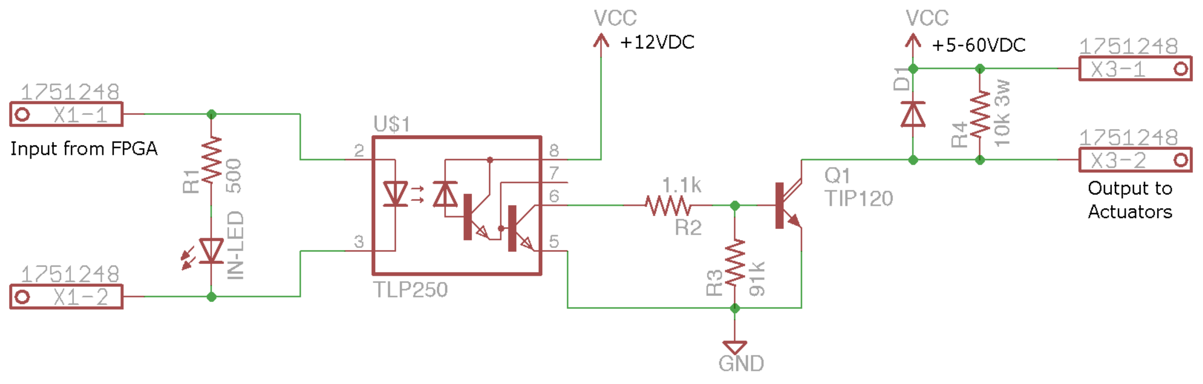


Illustration 13: general-purpose amplifier and H-bridge for Dulla

The second layer is simply multiple channels of high-current switches that amplify the low-current output of the FPGAs to the high-current signals that drive the actuators.

Nervebox presents a standard amplifier circuit and standard H-bridge circuit, freeing musical experimenters from the need to design their own.

Illustration 13 shows the schematic diagrams for the amplifier. For simplicity and ruggedness, I presently use TIP120 NPN bipolar junction transistors in both designs rather than MOSFETS. As they've been used only in all-on/all-off modes, heat dissipation has not been a problem. But in the future I may upgrade to a more mature MOSFET-based design.

While the development of new instruments will still require new actuation code to be written, the Dulla handles many underlying functions and enables standardization of hardware and circuit designs that are easily portable and quickly reproducible. Also, a future online library (see section 5) of Verilog modules could help ease and speed development time.

3.4 Detail of NerveOSC

As mentioned above, this system's internal musical representation is called NerveOSC. It is a unique flavor of the flexible OSC protocol. OSC supports some features missing from MIDI [21].

3.4.1 Structure

Where a typical MIDI channel voice message has the following 3-byte structure:

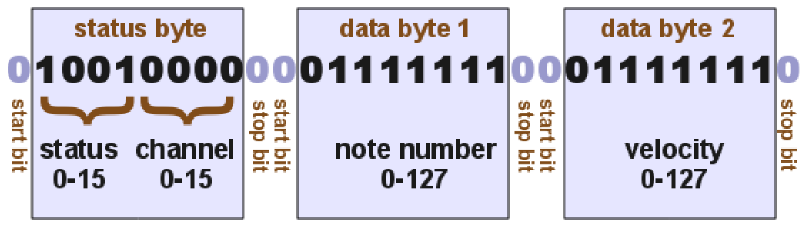


Illustration 14: midi message structure

A typical NerveOSC packet has this structure:

device/subsystem [**eventID, frequency (Hz), amplitude, timbre data array**]

3.4.2 Address Patterns

Using OSC's address pattern feature, NerveOSC can address any number of uniquely-named devices. And it can address subsystems within each, such as a specific string or a group of strings. This offers far more, and more transparent, address space per event than MIDI's 16 channels.

NerveOSC adds 3 more useful features: arbitrary frequencies, eventIDs, and timbre data.

3.4.3 Arbitrary Frequencies

Arbitrary frequencies are described in Hz with 16 bits of precision, making it easy to use any tuning system without employing hacks.

In contrast, MIDI defines notes as numbers from 0 to 127, with each explicitly representing a note in 12-Tone Equal Temperament @A=440Hz. It is possible, at the receiving end of a MIDI message, to interpret MIDI note numbers in any way desired. But if one is using a tuning system such as 31-tone equal temperament, MIDI's full 128 note range barely describes 4 octaves. It would be possible to send other octave ranges on other MIDI channels, or to accompany every single note with a another MIDI message, a pitch_bend command that modifies its frequency. But using a representation system that can describe any frequency directly and without ad-hoc hacks is much simpler.

3.4.4 EventIDs

EventIDs are used for mundane but important purposes. Their main function is to connect initial events (like pushing a key on a keyboard) to corresponding update events (like rolling the pitch or mod wheels while the key is down). In this case, the initial and update events would carry the same eventID, making them logically connectible downstream. This makes it much easier to describe dynamic tones with glissando, portamento, tremolo, and changes in timbre. It also helps to prevent crosstalk between music events that originate from different input devices.

3.4.5 Timbre

The third new feature of NerveOSC is timbre data. Timbre values are added to the end of the data array in NerveOSC. The considerations for the encoding timbre are summarized in the next section.

3.5 Timbre and Representation

3.5.1 The Negative Definition

Timbre is often negatively defined, as a sort of musical chaff left over after loudness, pitch and duration have been extracted. For instance, the American National Standards Institute defines timbre as "[...] that attribute of sensation in terms of which a listener can judge that two sounds having the same loudness and pitch are dissimilar". In the absence of an authoritative positive definition, much highly original research has attempted to characterize timbre from different perspectives .

These efforts generally fall into two categories, physical measurements and perceptual classification. Though much of the research shows that it is difficult to fully separate the two.

3.5.2 Physical Analysis

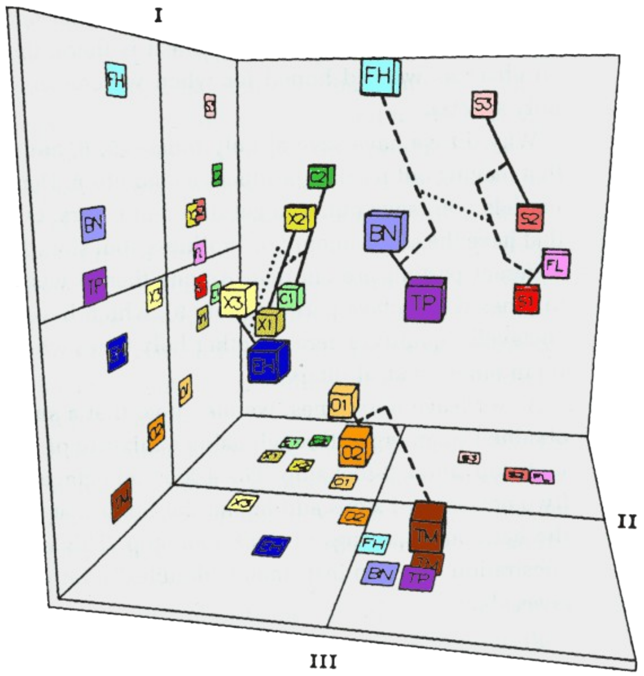
In physical terms, timbre can be defined as the change in a sound's spectra over time. The complexities of raw sound — each frequency, phase and amplitude, plus their individual distortions and aperiodicities — present an unmanageably large data set. Therefore, much of the work in physical analysis has focused on representing the perceptually important aspects of timbre within a reduced number of dimensions[22].

The fundamental modern work on timbre is J.M. Grey's Timbre Space [23], which used human subjects to quantify the perceptual difference between pairs of sounds of various orchestral instruments. These relationships of perceived difference showed very promising correlation when represented in a 3 dimensional graph of quantitative sound properties developed by Grey. This work is the foundation cited by the majority of subsequent work on timbre.

Following Grey's initial research, many reductive models parse timbre into distinctly spectral and temporal aspects. The two primary spectral characteristics are a wide vs. narrow distribution of spectral energy and high vs. low frequency of the barycenter of spectral energy [24].

Temporal aspects are slightly more complex, as they deal with changes over time. Much research has focused on the attack portion of a

Timbre Spaces



Dimension I: spectral energy distribution, from broad to narrow

Dimension II: timing of the attack and decay, synchronous to asynchronous

Dimension III: amount of inharmonic sound in the attack, from high to none

Illustration 15: Grey's Timbre Space

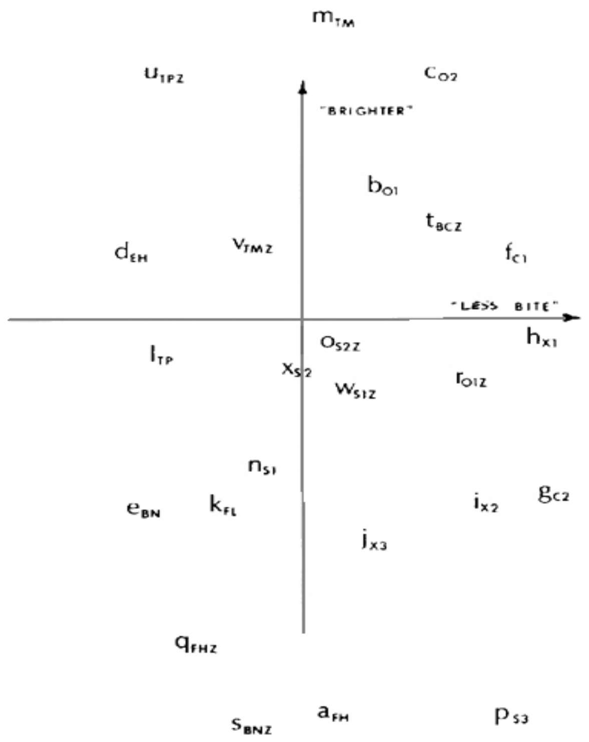


Illustration 16: Wessel's 2-Dimensional Timbre Space

BN - Bassoon
C1 - E flat Clarinet
C2 - B flat Bass Clarinet
EH - English Horn
FH - French Horn
FL - Flute
O1 - Oboe
O2 - Oboe (different instrument and player)

S1 - Cello, muted sul ponticello
S2 - Cello
S3 - Cello, muted sul tasto
TM - Muted Trombone
TP - B flat Trumpet
X1 - Saxophone, played mf
X2 - Saxophone, played p
X3 - Soprano Saxophone

sound's envelope, because that period has been shown to play an inordinately important role in how we identify sounds [25]. The primary temporal characteristic used by Grey is whether the high or low frequencies emerge first during the attack period.

A highly reductive 2-dimensional timbre space was developed in 1978 by David Wessel [26] for use as a timbre-control surface for synthesis. The idea was that by specifying coordinates in a particular timbre space, one could hear the timbre represented by those coordinates. Such a 2-dimensional timbre controller brings to mind the "basic waveform controller"from Hugh LeCaine's 1948 Electronic Sackbut [27]. Where LeCaine's 2-dimensional timbre controller uses "bright <-> dark" and "octave <-> non-octave" as its axes, Wessel's timbre space uses "bright <-> dark" and "more bite <-> less bite". The term "bite" in this case refers to a collection of characteristics of a sound's onset time.

In 2004 Geoffroy Peeters and others from the Music Perception and Cognition and Analysis-Synthesis team at Ircam collected timbral description systems from all available literature and extracted 71 timbral descriptors. Nervebox does not use these 71 timbral descriptors, but I've listed them in Appendix B because they provide a sense of the number of quantitative dimensions that affect timbre. Peeters and company used incremental multiple regression analysis to reduce the 71 timbral descriptors down to an optimal set of 5 psychoacoustic descriptors:

1. spectral centroid
2. the spectral spread
3. the spectral deviation
4. the effective duration and attack time
5. roughness and fluctuation strength

This is interesting as an attempt to incorporate all known timbral descriptions. But its effectiveness in predicting perceptual timbral differences has not yet been tested.

3.5.3 Perceptual Classification

All of this research into the physical aspects of timbre can help us better understand the perceptual aspects. For instance, sounds having a higher-frequency barycenter of spectral energy are generally said to sound 'brighter'.

But perceptual classification and the creation of useful timbral description systems are much more difficult. Some interesting attempts have been made, such as The ZIPI Music Parameter Description Language [28] and SeaWave [29]. But timbre, like consonance, seems to be at least partly a cultural construct [30][19] —making it even more difficult to find an unbiased solid ground on which to build a classification system.

Quietly lurking behind most of this work is the subject of identity — identifying individual musical instruments out of an orchestra or specifying exact timbres out of the palette of all possible sounds. Carol L. Krumhansl's research[22] revealed the existence of uniquely-recognizable perceptual features for certain instruments, such as the odd-harmonic of a clarinet, the mechanical "bump" of a harpsichord, coining the term specificities.

3.5.4 In Electromechanical Instruments

The requirements of timbral data description in NerveOSC are more focused. We are controlling physical instruments with natural timbral dimensions, not synthesizers. For practical purposes, we're interested in only the aspects of an instrument's timbre which are variable and controllable via actuation. The timbral variations of any one instrument should generally be expressible in a small number of dimensions. In fact, the timbral parameters of the attack time are unlikely to vary for any one instrument, according to Dr. Shlomo Dubnov : "This effect, which for time scales shorter than 100 or 200 ms is beyond the player, is expected to be typical of the particular instrument or maybe the instrument family." [25]

3.5.5 Perceptual Classification and Nervebox

I'd like to have built the NerveOSC timbral data format on top of the physical analysis of timbre because of the precision it provides. But I built it on top of the perceptual classification of timbre, because users of the system are unlikely to have access to the tools or knowledge necessary for physical analysis.

I believe that any perceptual ontology of timbre will grow unwieldy in size long before it becomes inclusive and detailed enough to be useful for this purpose. So Nervebox users are able to define their own collection of timbral terms for each instrument. I expect to see terms with names implying a boundless number of possible classification schemes, for instance: *pinkness*, *maraca*, *sidetoside*, *heavenly*, *those that belong to the Emperor* [31] and *rusty*.

Users developing a new instrument are responsible for finding and naming the timbral variations that can be made via actuation. Users creating new input mappings will be able to map selected ranges of the expressive dimensions of input devices to selected ranges of the user-defined timbre values. This is covered in more detail in section 3.8 — Development Process.

In this way, the timbre data format can represent the expressive capabilities of nearly any input devices, the timbral capabilities of nearly any experimental instruments, and the mapping of the former onto the latter. In section 4 I will be evaluating success in this based on tests of Nervebox's expressivity and fidelity.

3.6 Nervebox UI

The purpose of Nervebox's user interface (Illustration 17) is to enable users to create new mappings between streams of musical input such as MIDI keyboards, composition software, network streams, or custom devices and various instruments. It can also be used to debug mappings and connections and to test all instruments prior to a performance.

The user interface enables users to create new mappings for the Brum without writing any code or needing to understand the inner working of the instruments. This high level of abstraction greatly speeds and simplifies the process of composing and performing. I am describing it here in some detail because improvements in abstraction and process are much of the motivation behind Nervebox.

3.6.1 Mapping Mode

Mappings are created using a patchbay metaphor in the main area (1). Right-clicking on the workspace brings up a menu of available modules (1.a). Clicking a menu item causes a module's interface element to be created at the click's coordinates. So far I've only written the modules for mapping MIDI inputs. Modules for OSC and other input formats will be written in the next version. Modules can be dragged by their blue top bars (1.b) and deleted by clicking their "x" buttons (1.c).

The green connector (1.d) at the top of a module is its main inlet. The one or more green connectors (1.e) at the bottom of a module are its outlets. Connections between modules (1.f) can be created by sequential mouse clicks, causing the outlet of one module to be routed to the inlet of another. The connections can be destroyed by clicking on the connection line itself.

The green connectors (1.g) on the right side of of the MIDI-to-OSC modules are timbre inlets, setting timbre values that will be sent to the Bellums with each NerveOSC packet. Each type of instrument has a different set of timbre inlets, representing each instrument's timbral dimensions.

Mappings are listed, created, loaded, saved, and deleted in the panel labeled Manage Mappings (2).

3.6.2 Debug Mode

The Enable Trace and Enable Debug features greatly simplify the debugging process by causing the internal behavior of the mapping

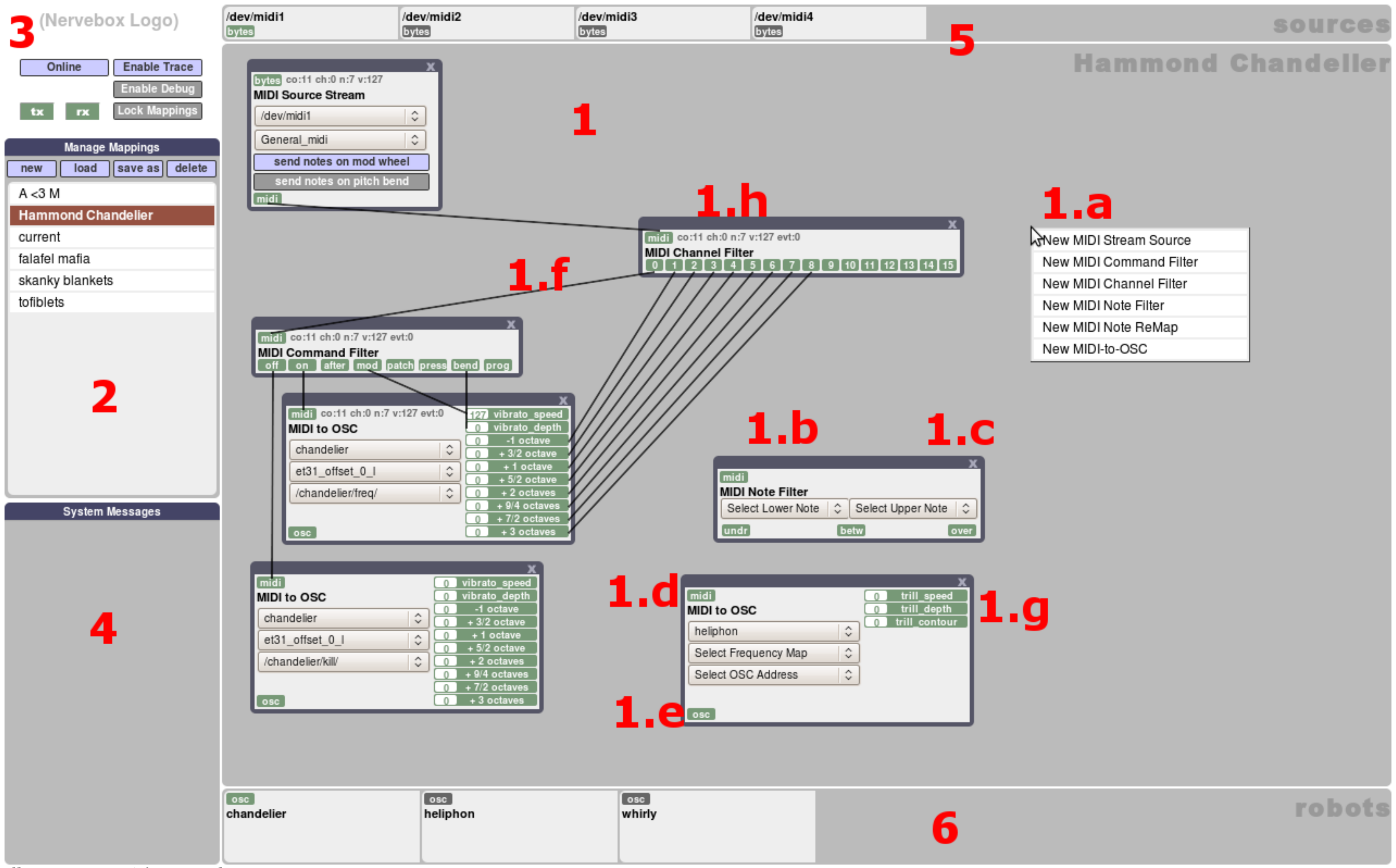


Illustration 17: The Nervebox UI

process to be shown in the UI.

The Control Panel (3) in the upper left corner enables a user to set global functions for the interface. For instance, the Enable Trace button is blue, indicating it is in its "true" mode. When Enable Trace == true, the contents of messages passed between modules are displayed (1.h) next to the inlet connectors of each module.

Enable Debug causes internal system messages to be displayed in the System Messages (4) pane.

3.6.3 Go Mode

When preparing for a performance, this interface can be used to show in real-time which input devices (5) and instruments (6) are connected. The next version will show more data about the exact status of each Bellum, such as whether its Dulla, amplifiers, and senors are connected and responding.

It is also expected that each Bellum will feature built-in test sequences, allowing users to run thorough checks of each instrument's tuning, timing, etc., prior to a performance.

3.6.4 Example Mapping

A walk though the flow of a mapping may help clarify what these mappings can do and how they work.

The mapping in this example is called "Hammond Chandelier", as indicated by the label in the upper right and by the highlight in the list of mappings. It is created for the Chandelier, an instrument envisioned by Tod Machover which is capable of playing rich and complex harmonics.

A MIDI-to-OSC module's timbre inlets reflect the timbral dimensions of the selected instrument. In this case, the Chandelier is selected, so the timbre inlets reflect the Chandelier's timbral dimensions: vibrato depth, vibrato speed, an undertone, and the first 7 steps of the harmonic series. This mapping enables a player to adjust the harmonics added to each note played on the keyboard by using controls on the keyboard that are mapped to different MIDI channels. In my tests I use a keyboard featuring assignable sliders, which I use to mimic the drawbars of a Hammond organ.

The Sources pane shows one MIDI source (0) with a green light, indicating the one MIDI interface that is plugged into the Brum.

A MIDI Source Stream module (1) is set to listen to the MIDI stream that is present: "/dev/midi1". This module parses MIDI messages from the input stream and adds appropriate eventIDs to each. Its outlet is connected to the inlet of a MIDI Channel Filter module (2).

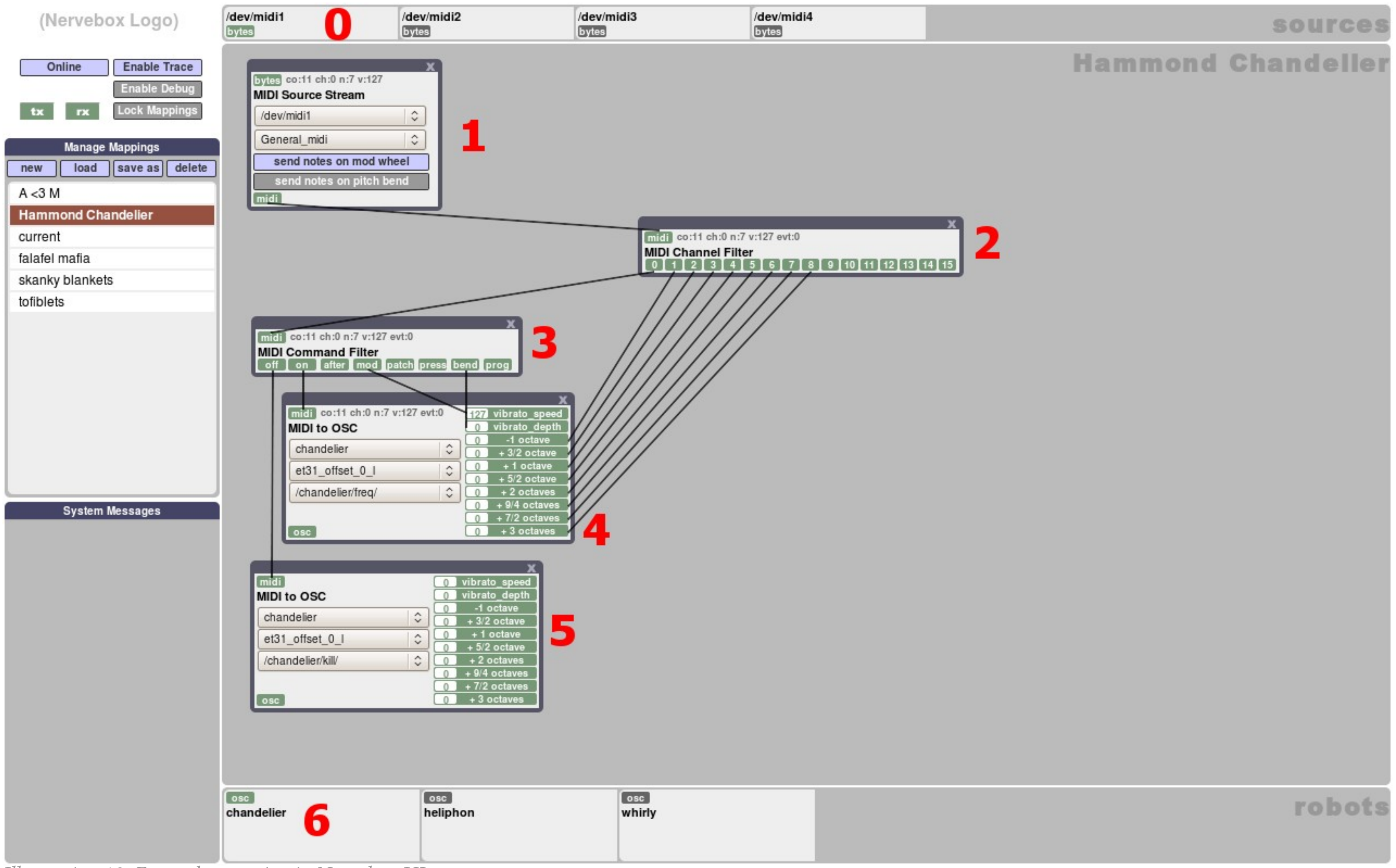


Illustration 18: Example mapping in Nervebox UI

In the MIDI Channel Filter, MIDI messages having a channel value of 0 are routed to the main inlet of a MIDI Command Filter module (3). Messages having channel values of 1-8 are routed to the timbre inlets of a MIDI to OSC module (4).

The MIDI Command Filter module (3) is routing MIDI messages with command values of "note off" to the main inlet of a MIDI-to-OSC module (5) and messages with command values of "note on" to the main inlet of another MIDI-to-OSC module (4).

Messages with command values of "mod wheel" and "pitch bend" are routed to the timbre inlets of MIDI-to-OSC module (4), enabling the player to change the depth and speed of the vibrato by rolling the keyboard's mod wheel and pitch bend wheel.

The MIDI-to-OSC modules (4, 5) convert incoming MIDI messages to NerveOSC messages with this format:

device/subsystem [eventID, frequency (Hz), amplitude, {timbre data}]

A packet from MIDI-to-OSC module (4) in this mapping might look like this:

'/chandelier/freq/' [1, '75.216257354', 100, 127, 0, 0, 0, 0, 0, 0, 0, 0, 0]

This music data is sent to the Chandelier and converted into music.

A mapping like Hammond Chandelier can be created in less than 2 minutes. Even mappings controlling complex interactions between multiple input streams and instruments can be created quickly and easily using these high-level abstractions.

3.7 Implementation — General

So far, my explanation of Nervebox has been largely conceptual. But I'll need to explain details of my present implementation to provide context for the upcoming major sections: Development Process, Evaluation, and Conclusion.

3.7.1 Hardware

The Brum and Bellums of Nervebox are built to run on commodity PCs. I've been using a variety of laptops and netbooks from Dell and HP. I chose Dell netbooks for their excellent Linux support and because their low price can help keep Nervebox accessible to other users. The Dullas are currently built with Xilinx Spartan 3-AN development boards.

3.7.2 Operating System

The Brum and Bellums are built on top of Ubuntu Linux 9.10, and should be forward-compatible with future versions. I chose Linux

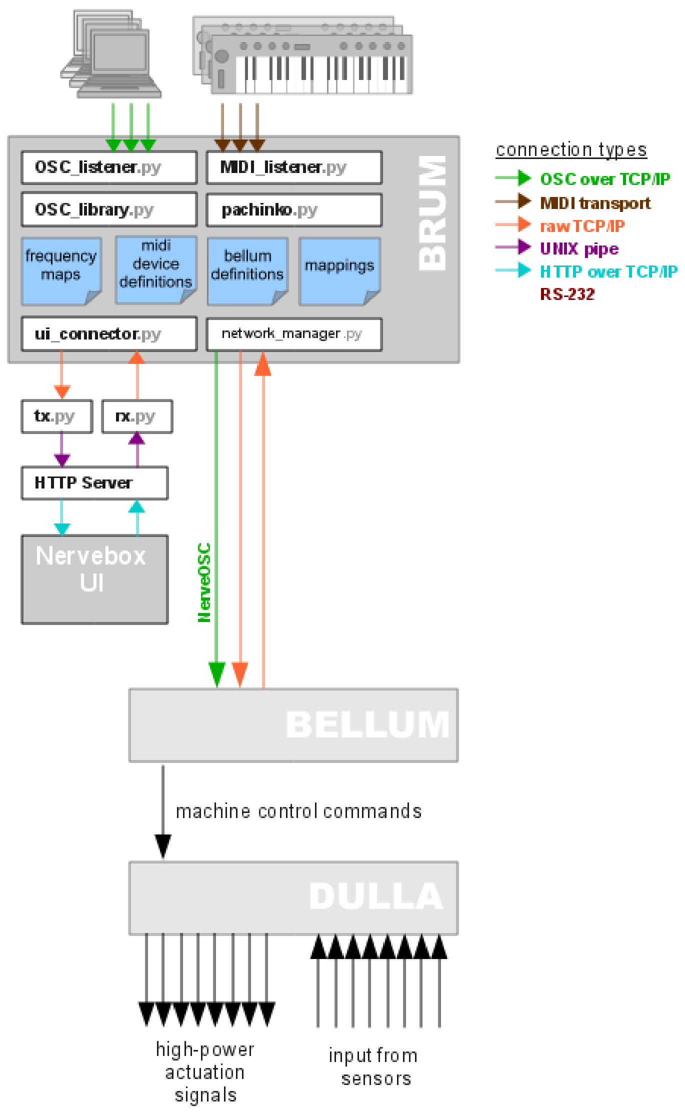


Illustration 19: Python modules of the Brum

because it's easy under Linux to access byte-level I/O from any peripheral device, such as MIDI and RS-232 interfaces. It is also easy to set priorities for individual processes — which is important because music performance software must have the highest possible process priority to ensure the lowest possible latency.

3.7.3 Languages

The Brum and Bellums are written in Python 2.6.4 and are expected to be forward-compatible with Python 3.x. I chose Python because of its ever-growing popularity and its potential accessibility to inexperienced programmers.

The circuitry of the various Dullas is defined using Verilog. I chose Verilog because the only other mature option, VHDL, is frightful to behold.

Nervebox UI is written purely in JavaScript. I chose Javascript for Nervebox UI because I prefer for user interfaces to run in a browser. The Web paradigm inherently supports multiple users and can be run instantly from any modern computer without installers and drivers.

3.7.4 Brum Implementation

The Brum is the switchboard at core of Nervebox. It handles the connection and disconnection of devices, such as MIDI sources, OSC

Figure 2: example mapping

```
[       # modules
            {action:"new", type:"modules", name:"0", param:"MIDI_Source_Stream",
            client_x:23, client_y:14},
            {action:"new", type:"modules", name:"1", param:"MIDI_Filter_Command",
            client_x:27, client_y:251},
            {action:"new", type:"modules", name:"2", param:"MIDI_to_OSC",
            client_x:55, client_y:321},
            {action:"new", type:"modules", name:"3", param:"MIDI_Filter_Channel",
            client_x:383, client_y:159},
            {action:"new", type:"modules", name:"4", param:"MIDI_to_OSC",
            client_x:26, client_y:476},
            # functions
            {action:"setSendOnPitchBend", type:"function", name:"0", param:false},
            {action:"setOSCPath", type:"function", name:"4", param:"/chandelier/kill/"},
            {action:"setFreqMap", type:"function", name:"4", param:"et31_offset_0_l"},
            {action:"setInstrument", type:"function", name:"4", param:"chandelier"},
            {action:"setFreqMap", type:"function", name:"2", param:"et31_offset_0_l"},
            {action:"setOSCPath", type:"function", name:"2",
            param:"/chandelier/freq/"},
            {action:"setInstrument", type:"function", name:"2", param:"chandelier"},
            {action:"setSendOnModWheel", type:"function", name:"0", param:true},
            {action:"setMIDIDevice", type:"function", name:"0",
            param:"General_midi"},
            {action:"setPath", type:"function", name:"0", param:"/dev/midi1"},
            # connections
            {dest_inlet:0, dest_name:"2", type:"connection", action:"add",
            src_name:"1", src_outlet:1},
            {dest_inlet:0, dest_name:"3", type:"connection", action:"add",
            src_name:"0", src_outlet:0},
            {dest_inlet:0, dest_name:"1", type:"connection", action:"add",
            src_name:"3", src_outlet:0},
            {dest_inlet:1, dest_name:"2", type:"connection", action:"add",
            src_name:"1", src_outlet:3},
            {dest_inlet:2, dest_name:"2", type:"connection", action:"add",
            src_name:"1", src_outlet:6},
            {dest_inlet:0, dest_name:"4", type:"connection", action:"add",
            src_name:"1", src_outlet:0},
            {dest_inlet:3, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:1},
            {dest_inlet:4, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:2},
            {dest_inlet:5, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:3},
            {dest_inlet:6, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:4},
            {dest_inlet:7, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:5},
            {dest_inlet:8, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:6},
            {dest_inlet:9, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:7},
            {dest_inlet:10, dest_name:"2", type:"connection", action:"add",
            src_name:"3", src_outlet:8}
]
```

sources, Bellums and browsers. And it manages multiple persistent channels of communication with each — via raw sockets, UNIX character devices, and OSC and HTTP over TCP/IP. It stores mappings and system states; serves and stores data for Nervebox UI; consumes several sources of configuration data — conf files, frequency and keyboard maps, instrument specifications, and MIDI and OSC input device specifications.

One of its more complex functions is the metaprogramming module called pachinko.py. This module converts the text-based mappings into runnable code. For instance, the example mapping from Illustration 16 is dynamically generated by Nervebox UI and is stored on the server as the text below.

pachinko.py creates a runnable mapping by instantiating runnable code for each module defined in the "# modules" section above. It then configures the modules using parameters from the "# functions" section and creates a flow control network based on the flow control implied in the rules of the "# connections" section.

3.7.5 Bellum Implementation

The function of the Bellum is to convert incoming NerveOSC messages into machine control commands, which are sent to the Dulla.

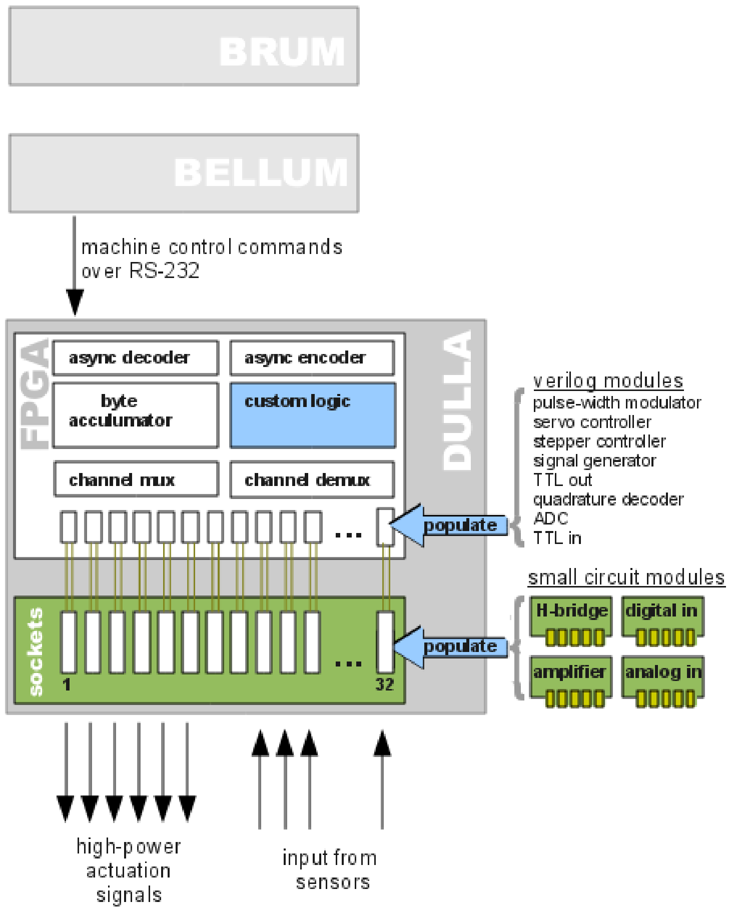


Illustration 20: Detail of the Dulla

Each Bellum features a core of generic code that handles all of the common features. These include a socket connection for receiving NerveOSC messages and 2 unidirectional raw sockets for

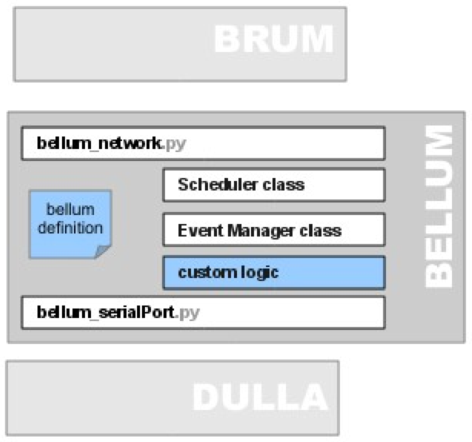


Illustration 21: Python modules of the Bellum

communication with the the Brum. It also manages communication with one or more Dullas via RS-232 ports. Each Bellum also features code that is specific to the instrument it controls. See 3.8 Development for more details.

3.7.6 Dulla Implementation

The present Dulla implementation is functional. But its inspiration lies in a design concept that was beyond the scope of this thesis. Here I describe the Dulla's inspiration and its present state.

The Dulla is conceived as an all-purpose PC peripheral for reading data from virtually any sensors and for controlling virtually any type of actuation. This design is is, in part, a reaction to my frustration with the exorbitant costs and limited functionality of commercial motor control products.   At its core is an FPGA (Field-Programmable Gate Array), not a microcontroller. I chose FPGAs because they can operate in a parallel fashion without encountering clock division problems.

The Dulla design is conveniently modular, with pre-designed current-switching circuits to amplify the small signal from the FPGA into high-power signals for driving actuators. These circuits are very simple and inexpensive because all processing functions occur within the FPGA. For instance, the pulse-width-modulated signals output by the H-bridge will be generated by soft PWM circuitry within the FPGA. The H-Bridge is just switching power.

The important result is that users can control their new instruments' actuators without designing and creating new hardware. This removes a substantial barrier; users with no knowledge of circuit design can create their own electromechanical musical instruments.

Of course users may create their own circuit modules. But the basic 4 should be enough for most projects: amplifier, H-bridge, digital input, ADC input.

The main difference between the current implementation and the design concept is that the design concept features a mainboard with the FPGA and 32 slots for small daughter boards. These daughter boards would hold the aforementioned circuit modules.

The Dulla's mainboard and daughter boards have not yet been designed and fabricated, as that is beyond the scope of this thesis.

Currently the Dulla exists in the form of Xilinx development boards and circuits occupying number of breadboards. I have written and tested Verilog modules for RS-232 communication, packet accumulation, channel demultiplexing, PWM and signal generation, and quadrature decoding. And I've breadboarded and tested the amplifier, H-bridge, and digital input circuits.

I've been using the Xilinx XC3S700AN device from the non-volatile Spartan 3-AN family. It runs at 50MHz and features 372 general-purpose I/O pins and 700,000 system gates. The chip costs about $40 and requires few supporting components.

3.7.7 Nervebox UI Implementation

There are 3 main components that make Nervebox UI work: the Brum, the HTTP connections, and the Client.

Figure 3: remote script names

getInputs	getMapping	ping_client
getBellums	getMappingNames	trace_source
getMidiDevices	getCurrentMappingName	trace_component
getNoteMaps	deleteMapping	trace_timbre
getFreqMaps	saveMappingAs	
	saveBlankNewMapping	

The Brum does not serve up the Client like a series of web pages. The Client is a persistent, free-standing program, running in the browser. The Brum and Client exchange only data, formatted as JSON (JavaScript Object Notation) [32]. The Brum pushes data about Nervebox's configuration and state to the Client. And the Client sends data about changes to mappings and Client state to the Brum. Figure 3 shows a list of Brum functions called by the Client.

Nervebox UI's HTTP connections do not use the normal HTTP request/response cycle. They use two unidirectional connections, a receive and a persistent transmit.

Requests are sent from the Client to Apache, the HTTP server, as usual. Apache is configured with mod_python, enabling it to run python scripts as subprocesses of its main process. Incoming HTTP requests are passed off to a small script, rx.py, which parses requests and passes them to the Brum via a TCP/IP socket connection. The Brum does not return a response to the request at this point. The Brum returns only a

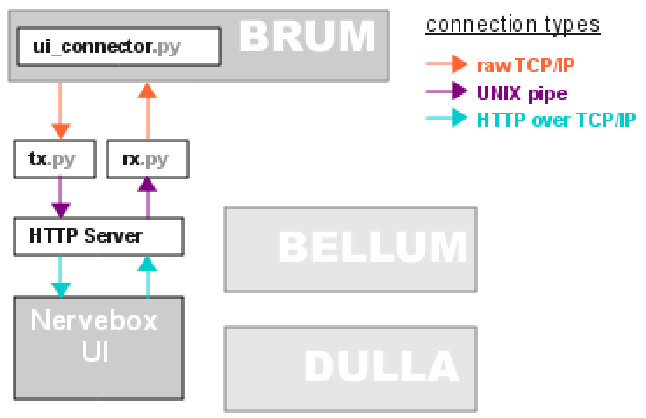


Illustration 22: Nervebox UI's communication cycle

JSON-encoded "true" for any request; or an error message if an exception was encountered.

Responses to the request return to the Client via a persistent HTTP connection, also known as HTTP server push. This is maintained through tx.py, another script that runs as a subprocess of Apache and connects to the Brum via TCP/IP socket connections.

The server push channel exists because the server constantly needs to send data to the client that the client did not request. In HTTP (prior to HTML5), the client is intended to the Client only when requested. A nontrivial amount of hacking and fine tuning is required to make server push work reliably.

The server push channel is used to send all data. Even data that could travel in the response to a request from the Client. This is done partly for the simplicity that comes with consistency. But it is also intended to prevent connection deadlock. Browsers can only keep a limited number of connections open to any one server. Since the server push connection is already persistently open, I'm ensuring all other connections are as short as possible, lessening the chance that the browser will reach its connection limit.

The client is written in entirely in JavaScript, with styles defined with Cascading Style Sheets. It does not use 3rd-party libraries like jQuery, Dojo, or ext.js. Instead, it uses a framework called mrClean that I wrote previously and finished for this project.

mrClean is a framework for creating rich, desktop-like applications that run inside a browser. It provides core libraries for HTTP communication, error handling and reporting, saving and restoring GUI state, drag and drop, skins, event routing, and more. Much of its functionality is dedicated to desktop-like user interaction. It also includes a library of constructors for 33 JavaScript object, from floating dialog boxes to date-manipulating libraries to folder trees.

All of the rich and responsive interactivity you see in Nervebox UI comes from mrClean.

3.8 Development Process

Again, I'm proposing that Nervebox's value is the way in which it empowers musical experimenters to create new musical machines more quickly and easily. This section covers the development process on a practical and detailed level.

3.8.1 Creating New Mappings

The most common development activity will be the mapping of various inputs to various instruments, as I expect that each instrument developed will likely be used for more than one composition or performance context.

I covered the process of creating mappings in detail in sections 3.6 and 3.6.*. These mappings leverage many underlying systems of the Brum as discussed above — functionality that would otherwise take many days to code from scratch.

Using the abstractions presented in Nervebox UI, complex mappings can be created, tested, and debugged within minutes. No coding is required. And robust tools exist to help in debugging.

3.8.2 Creating New Pachinko Modules

Nervebox currently supports 6 types of mapping modules. So far I've been able to build all if the mappings I've needed using only these. But future users will inevitably want others, particularly modules for filtering and routing OSC inputs or raw audio streams.

To create new pachinko modules, new code must be written in pachinko.py and the Web client files nervebox.js and app.css. I can create a new module in under an hour. But new users will face a daunting learning curve in the metaprogramming of pachinko.py, the pure-JavaScript GUI architecture of mrClean, and the unusual HTTP communication technique that connects them. So the development of new pachinko modules will currently be difficult for users.

A future version of Nervebox UI may include a way for users to create new pachinko modules without needing to understand the underlying architecture. A purely graphical method will be included in Nervebox UI 2.0.

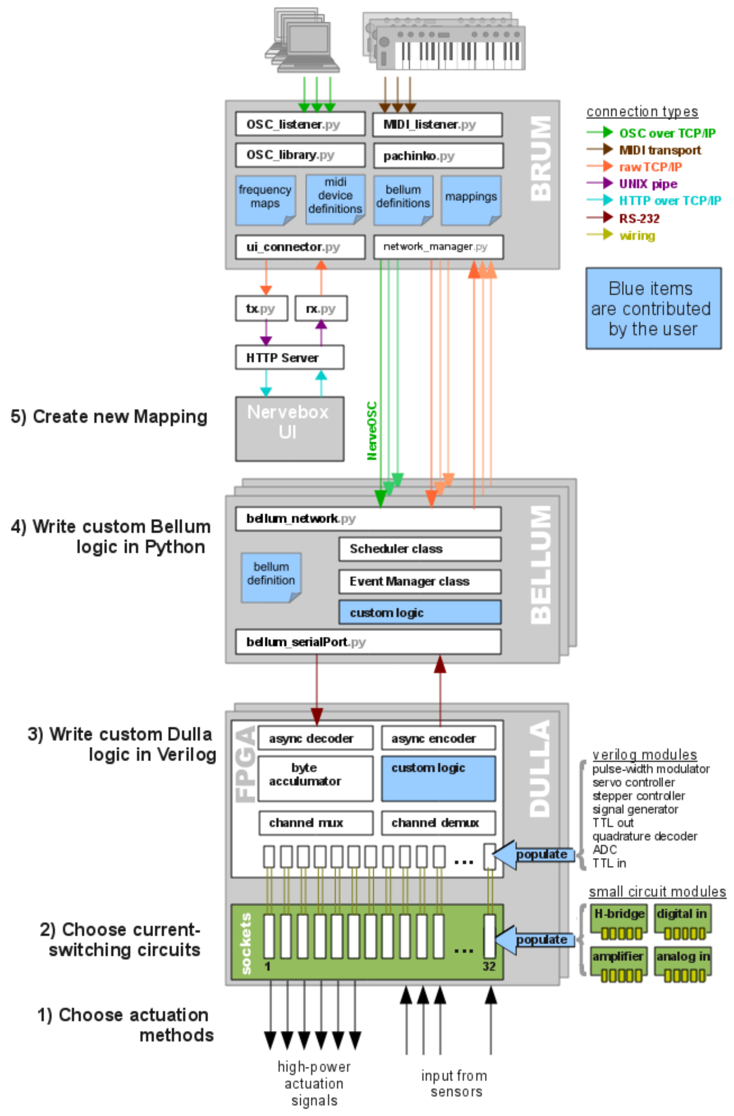


Illustration 23: The Nervebox actuation path

3.8.3 Creating a New Instrument

Unlike the creation of new mappings and new pachinko modules, the creation of control systems for new instruments requires some engineering.

Nervebox provides hardware and software that greatly expedite the process of developing control systems for new electromechanical instruments. But I don't believe the convex hull of all these instruments' possibilities can be realistically predicted. And any attempt to limit those possibilities would be working against the exploratory spirit I'm seeking to support and promote with Nervebox.

The design of new instruments requires a chain of decisions that starts at the instrument and works backwards towards the flow of incoming musical data. I will use the Chandelier as an example of the process of creating an actuation path.

a. Choose actuation methods

The FPGA in the Dulla is able to generate almost any type of control signal for electrically-controlled actuators: stepper and servo motors, solenoids and electromagnets, electro-pneumatic and electro-hydraulic valves and more. So users are free to choose any type of actuator that suits their instrument.

The Chandelier is designed to use 48 separate electromagnets to excite 48 strings. And it uses 48 brushless DC motors to engage or release padded levers that can damp the strings. The electromagnets are driven by square waves of varying frequencies. And the damper motors are engaged when a simple DC current is on, and disengaged via spring return when the DC current is off. This makes for 96 channels of actuation.

b. Choose current-switching circuits

The function of the current-switching circuits is to amplify the low-power control signals generated by the output pins of the FPGA into high-power signals for driving actuators, or to act as a safe electrical interface between incoming sensor data and input pins of the FPGA.

Figure 4: Verilog module for variable-frequency square wave generator

```
module square_waves (
   input clock, // wire from system clock
   input [23:0] period, // 24 wires setting value for square wave period
   output square_wave_pin_out // wire to FPGA output pin
);
  reg [24:0] period_counter = 0; // 25-bit register for period counter
  reg wave_bool = 0; // boolean value sent to pin square_wave_pin_out
  always @(posedge clock) // at the positive edge of every clock cycle
     period_counter <= ( period_counter > period*2)?0: period_counter+1;
     // increment register period_counter, reset  to 0 when it exceeds period*2
  always @(posedge clock) // at the positive edge of every clock cycle
     wave_bool <= (period_counter  > period)?1:0;
     // set register wave_bool  to 1 if  period_counter  > period, otherwise 0
  assign  square_wave_pin_out = wave_bool;
     // continuously assign value of wave_bool to square_wave_pin_out

endmodule;
```

The current-switching circuit modules of the Dulla (from section 3.7.6) should be able to power and control almost any actuators drawing up to 60V @ 8A. So users generally won't need to design their own circuits.

But they will need to create the circuits on circuit boards or breadboards. Section 6 includes ways future Nervebox versions could expedite the creation of circuit boards.

The Chandelier uses the same simple amplifier circuit for all 96 of its actuators.

c. Write Dulla configuration to produce actuation signals

The function of the Dulla's FPGA is to convert incoming motor control commands from the Bellum into signals that control the actuators. The Dulla is configured using Verilog.

I'm aware that FPGAs and Verilog are not part of the current standard hacker toolkit. This is likely to be the most challenging part of the development process. Nervebox contains a few Verilog modules, such as an RS-232 receiver, that will help expedite common tasks. And section 6 covers ways this could be made easier in the future.

In the Chandelier, each of the 48 electromagnets and 48 damper motors is controlled by the output of a separate pin on the the Xilinx XC3S700AN. The signals for the electromagnets are all square waves of different frequencies. Listing 4 shows an example of the Verilog code from which each variable square wave oscillator is created.

A more complete listing of the Chandelier's Verilog code can be found in Appendix A.

d. Write Bellum logic to convert music data into actuation commands

Many of the complex functions of the Bellum are already built into the platform code:network and RS-232 communications, OSC parsing, event management, and the formalities of registering with and unregistering from the Brum. And there is a growing library of musical logic such as multithreaded classes for vibrato, tremolo, arpeggio, and the future scheduling of events.

The task of the users' code is to convert the NerveOSC input into the machine-control commands consumed by the Dulla. This is where the music meets the machinery. This conversion process contains the musical logic of the instrument, which may be very simple or very complex.

I'll continue to use the Chandelier as an example for consistency, even if it is a rather complex example.

The Chandelier's rich sound is the result of the use of harmonics and a slow, shallow vibrato. Illustration 21 shows the meaning of the values in an example packet of NerveOSC.

The OSC **address** ends in 'freq', indicating that the **note** value should be interpreted as a frequency in Hz.

While the **vibrato speed** and **vibrato depth** values are both set to 0, the Chandelier Bellum still uses a baseline vibrato. So a single, sustained note event arriving as a packet of NerveOSC is converted into a constant stream of changing frequencies sent to the Dulla until the Bellum receives a NerveOSC packet with a matching eventId and an address of 'chandelier/kill/'.

The harmonics array has non-zero entries for the second and sixth harmonics, indicating that additional notes are to be sounded concurrently with the fundamental frequency. These notes have amplitude values of 64/128 and 32/128, adjusting for zero-based counting. Like the fundamental note, each of these harmonics will also be converted by the vibrato process into a stream of ever-changing frequencies.

'/chandelier/freq/' [	**1,**	**'75.216257354',**	**100,**	**0,**	**0,**	**0, 63, 0, 0, 0, 31, 0, 0]**
address	**eventId**	**note**	**amplitude**	**vibrato speed**	**vibrato depth**	**harmonics**

Illustration 25: example NerveOSC packet for the Chandelier

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**

red bits encode the id of the target string, in this case string 1
blue bits encode the period of the string in 50MHz clock cycles, in this case 28409 cycles, or a frequency of 440Hz.

Illustration 24: Bellum -> Dulla data format for Chandelier

The user must also decide on the data format to be sent from the Bellum to the Dulla. For instance, data is sent from the Chandelier's Bellum to its Dulla is in the format shown in Illustration 24.

The standard Bellum code includes functions to simplify the process of encoding binary data for the Dulla.

See Appendix A for the Python code that performs these operations.

4 Evaluation

4.1 Measuring Generality, Expressivity, and Fidelity

The initial goals of Nervebox will be satisfied if it provides a platform encapsulating the complex technical problems encountered in the development of electromechanical musical instruments behind a set of high-level abstractions that can be combined to control almost any such instrument. I label this ability to control many different types of instruments the *generality* of Nervebox.

I evaluated the generality of Nervebox by using it as a platform upon which to build control systems for 2 very different electromechanical musical instruments — the Chandelier and Ensemble Robot's Heliphon. I then tested these systems to determine their fidelity and expressivity.

I am considering any control system's fidelity to be a measurement of its ability to reproduce the intentions of the composer or player to the best or its instrument's ability. Put more simply, the fidelity is the measure of the correctness of a control system, the inverse of the measure of its errors or artifacts.

And I am considering a control system's expressivity to be a measurement of its ability to define and exploit the full expressive range of the instrument it is controlling — frequency range, dynamics, timbres, textures, and specificities. Extra credit: adding new, compound expressivities that are not naturally inherent to the instrument, such as the additive harmonics of the Hammond Chandelier mapping in Illustration 16 and section 3.6.4.

4.2 The Chandelier

I've already used the Chandelier in earlier examples. For this section, I'll provide a more more thorough description of the instrument and the implementation of its controller.

Tod Machover's group has built 3 different versions of the Chandelier. The first one was was built by Mike Fabio and was the subject of his 2007 thesis The Chandelier: An Exploration in Robotic Instrument Design. This Chandelier was an instrument featuring 4 groups of 4 strings, each group being actuated in a different way.

The second version is commonly referred to as the Chandelier Testbed. It is the embodiment of a long series of prototypes developed in the process of exploring functional and musical possibilities for the final version. The Chandelier Testbed is a large steel Unistrut frame

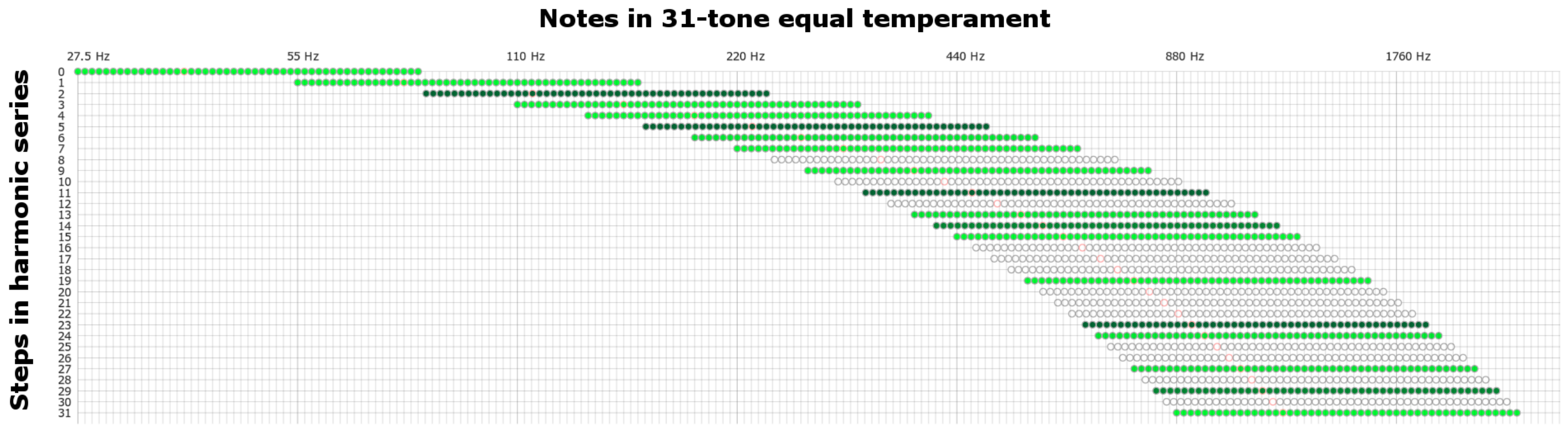


Illustration 26: Intersection of 31-tone equal temperament and frequencies created with upper harmonics

featuring 32 piano strings tuned in 31-tone equal temperament, actuated into vibration by powerful electromagnets. Electric guitar pickups are used to capture and amplify the sounds of the Chandelier.

The third version is commonly referred to as the Real Chandelier. This is the full-scale 48-string instrument that will be used as a dramatic set piece and musical instrument in Tod Machover's upcoming opera Death and the Powers.

My control system was designed to control the third version of the Chandelier. But my tests have been performed using the second version, as the third and final version is currently still in production. I refer to the Chandelier Testbed as simple the Chandelier hereafter.

4.2.1 Expressive Dimensions of the Chandelier

Tonal Range

The tonal range of the Chandelier starts at 27.5Hz, also known as double pedal A. This note is near the bottom of the human hearing range. Determination of the upper limit of its range has been musically unimportant, as its range extends beyond the upper limit of the human hearing range.

The Chandelier's 32 strings are tuned in 31-tone equal temperament, their fundamentals covering the range from 27.5Hz to 55Hz. These notes are sounded by using magnetic pulses from the electromagnets to set the strings resonating at their fundamental frequencies. Because we're driving them with electromagnets, we can also sound each string at frequencies from that string's upper harmonic modes. So each string can produce a range of notes, with frequencies corresponding to the harmonic series, originating with each strings' fundamental frequency.

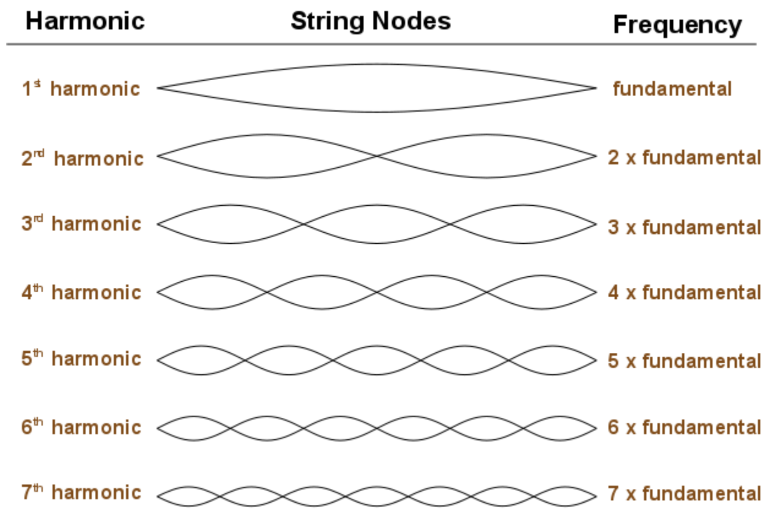


Illustration 27: Harmonic Modes and the harmonic series

The notes in each string's harmonic series do not necessarily correspond to notes in any equal tempered temperament. Illustration 26 shows a model of notes producible by the Chandelier's 48 strings, calculated up to each string's 32nd harmonic.

The horizontal scale denotes frequency. The circles indicate the notes that can be produced. The vertical scale corresponds to steps in each note's harmonic series. So the top row of green notes shows the fundamentals, or first harmonics, starting at 27.5Hz. The next row down shows the notes produced by each string's second harmonic, which lie an octave above the fundamentals. The third row shows the third harmonic, 1.5 octaves above the fundamentals. Each note-circle's color indicates how in- or out-of-tune it is compared to 31-tone equal temperament. 5 colors of green are used, corresponding to the number of cents (1200ths of an octave) each note's frequency deviates from its nearest match in 31-tone equal temperament. Bright green shows a perfect match. The darkest green show a deviation of 4 cents. White circles have a deviation of 5 or more cents. A difference of 6 cents or less is considered to be imperceptible by most humans [33]. So this illustration shows that upper harmonics can be used to create more-than-full coverage of the notes in 31-tone equal temperament.

Timbre and Specificities

The electromagnetically-driven strings of the Chandelier feature very little timbral variation. Slight shades of upper- and sub-harmonics can be introduced by changing the placement of the electromagnet along the length of the string, thereby changing its location relative the string's nodes and anti-nodes. But the dominant sound from each string is a simple, sine-like wave.

These electromagnetically-driven strings have one, very interesting specificity — a throbbing tremolo that increases with the amplitude of the string's vibration. This happens because the tension on a string increases with its displacement, thus increasing the frequencies of the resonant modes of the string, and temporarily decreasing the resonant coupling between the string and electromagnet. This slow oscillation

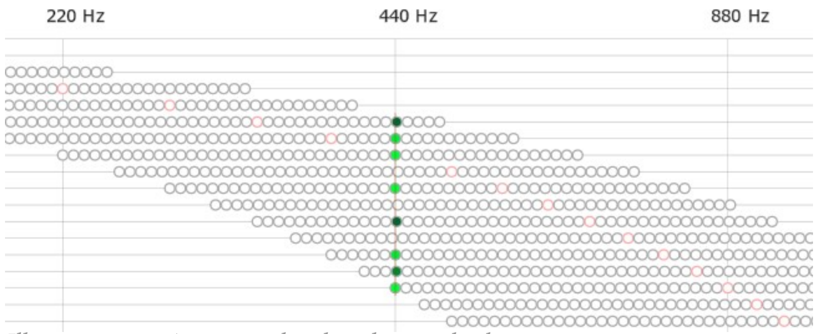


Illustration 28: A-440 can be played on multiple strings.

occurs as a string with low-amplitude gains resonant coupling with the electromagnet, then gains energy and increases amplitude, then increases its natural resonant frequency and loses resonant coupling then becomes a string with a low amplitude, restarting the cycle.

Dynamics

I define the Chandelier strings' amplitude as the ratio between a string's length and it's maximum displacement while resonating. The strings of the Chandelier can be played in a continuous dynamic range from zero displacement up to the point where they reach a physical limit to their displacement, such as the limit of physical clearance, the limited power of the electromagnets, or aforementioned tremolo specificity. In the current Chandelier setup, the maximum amplitude is around 2%, at which point the vibrating strings strike the electromagnets. This dynamic range, from 0% to 2%, provides more than enough dynamic range for purposes of musical expressivity.

4.2.2 Extra Credit: Synthetic Expressive Dimensions of the Chandelier

A good controller should be able to add some additional expressive dimensions that are not inherent to the physical structure of the instrument. I call these synthetic expressive dimensions.

As mentioned above, the Chandelier's strings tend to sounds like simple, sine-like waves. This sound is pure, but musically dull. I've found 3 synthetic expressive dimensions that greatly enrich the sound of the Chandelier.

Slow Vibrato

Driving a string with electromagnetic pulses that are slightly out of phase with the string's resonance will cause rich harmonics to bloom in the string's sound. And effective way to keep the pulses continually out of phase with the string is to slowly and shallowly change the frequency of the pulses. The difference in frequencies must remain within a safe

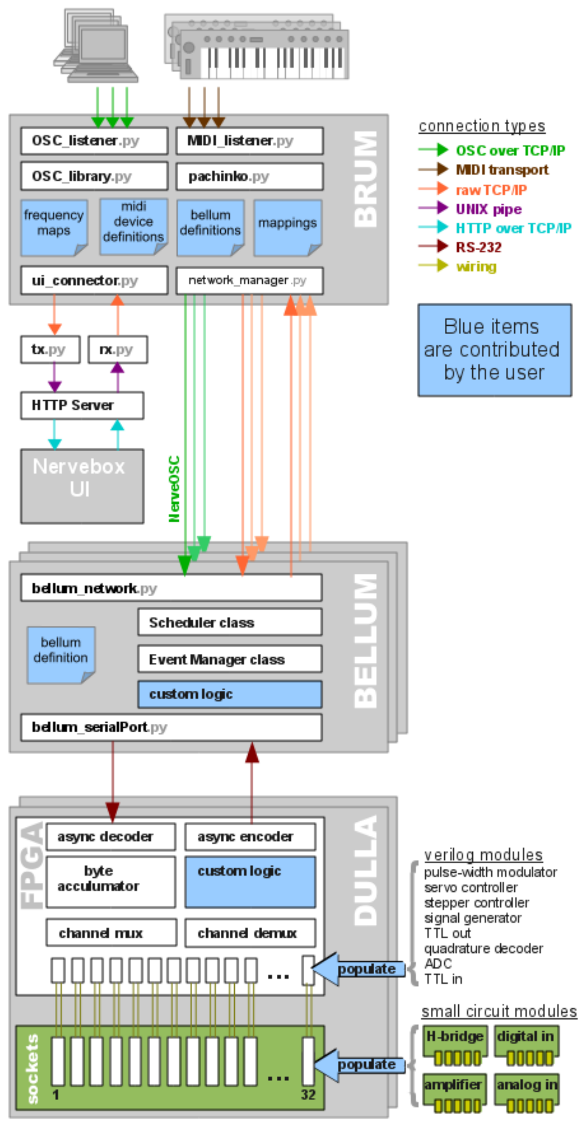


Illustration 29: all details contributed by user, shown in context

band that is shallow enough that it does not interfere with the resonant coupling of the pulses and the string. Slowly changing the frequency up and down within this safe band — effectively a long, shallow vibrato — is an effective way to add ringing harmonics and produce a richer sound.

Multiple Strings per Note

One effect of the Chandelier's complex tonal space (Illustration 28) is that notes from above the first harmonic can be played on multiple strings. For instance, Illustration 26 shows how an A-440 can be played on the 16th harmonic of string 1, the 15th harmonic of string 4, the 14th harmonic of string 7, and so on.

These notes all ring at slightly different frequencies very close to 440Hz, as is reflected by the range of colors representing them. Sounding all of them at the same time creates a lush sonic fabric full of meshing and un-meshing phases.

Harmonics

One more synthetic expressive dimension that can enrich the sound of the Chandelier is the use of carefully controlled additional harmonics — as is done with pipe organs and Hammond organs.

4.2.3 Expressivity of Nervebox-based Chandelier controller

Here we test the Nervebox-based controller's ability to exploit and control all of the Chandelier's expressive dimensions.

Illustration 29 shows, in context, the 5 components of a Nervebox-based controller.

a) Dulla: selection (and assembly) of current switching circuit modules

b) Dulla: custom FPGA configuration written in Verilog

Figure 5: Verilog for pulse-width modulator

```
/* pulse-width modulation module */


module PWM(
        input clock,// wire from system clock
        input [7:0] PWM_in, // 8 wires setting value for duty cycle
        output PWM_out // wire to FPGA output pin


);
reg [8:0] PWM_accumulator;  // 9-bit register for accumulating PWM cycles
always @( posedge clock) // at the positive edge of every clock cycle


        PWM_accumulator <= PWM_accumulator[7:0] + PWM_in;
// continuously assign value of 9th bit of PWM_accumulator to PWM_out


assign PWM_out = PWM_accumulator[8];
endmodule;
```

c) Bellum: definition.py (instrument definition file)

d) Bellum: custom instrument behavior written in Python

e) Brum/Nervebox UI: mapping created with Nervebox UI

This is how the Nervebox platform is configured to exploit and control all of the Chandelier's expressive dimensions.

Tonal Range

The Chandelier's tonal range is encoded in the Chandelier Bellum's definition.py file, which is summarized in Appendix A2. The file contains a list, freqs_l, of 991 frequencies (31 tones * 31 harmonics) found in the tonal space shown in Illustration 26. These 991 frequencies appear again in a structure called strings, which groups the

Figure 6: Augmented Verilog module "square_waves"

```
/* variable frequency square wave generator module */
module square_waves (
        input clock, // wire from system clock
        input [23:0] period, // 24 wires setting value for square wave period
        // 24 wires setting value for square wave duty cycle
        input [23:0] duty_cycle,
        output square_wave_pin_out // wire to FPGA output pin
);
reg [24:0] period_counter = 0; // 25-bit register for period counter
reg wave_bool = 0; // boolean value sent to pin square_wave_pin_out
always @(posedge clock) // at the positive edge of every clock cycle
        period_counter <= ( period_counter > period*2)?0:
period_counter+1;
        // increment register period_counter, reset  to 0 when it exceeds
period*2
always @(posedge clock) // at the positive edge of every clock cycle
        wave_bool <= (period_counter  > duty_cycle )?1:0;
        // set register wave_bool  to 1 if  period_counter  > period,
otherwise 0
assign  square_wave_pin_out = wave_bool;
// continuously assign value of wave_bool to square_wave_pin_out
endmodule;
```

frequencies by the strings that can play them.

The custom instrument behavior written for the Chandelier Bellum contains a class called TonalStructure which maps the notes of

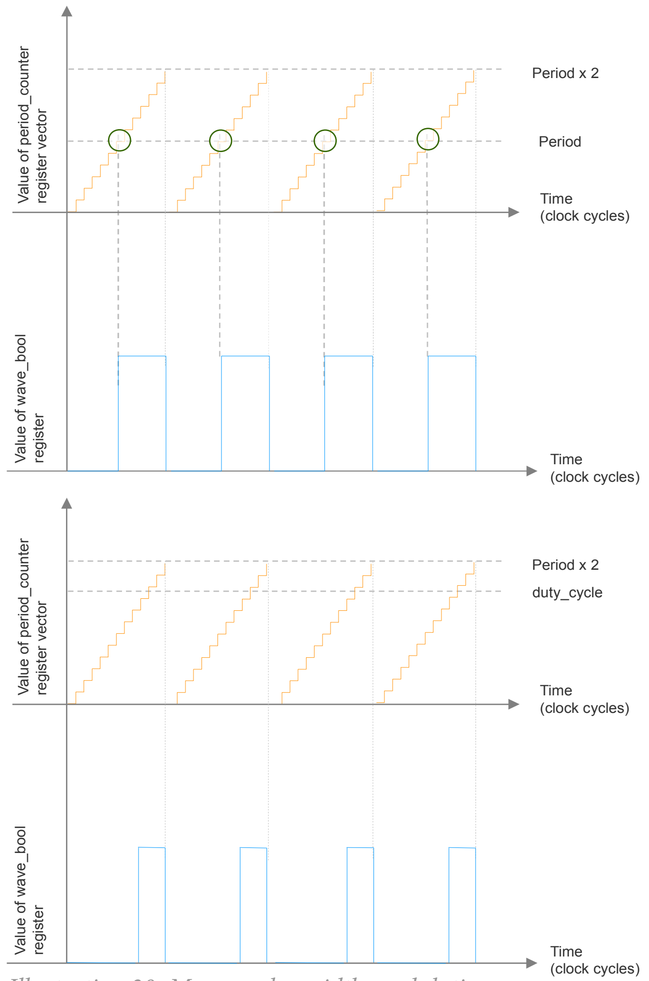

Value of period_counter register vector
Period x 2
Period
Time (clock cycles)
Value of wave_bool register
Time (clock cycles)
Value of period_counter register vector
Period x 2
duty_cycle
Time (clock cycles)
Value of wave_bool register
Time (clock cycles)

Illustration 30: Macro pulse-width modulation

incoming OSC messages to the 991 defined notes of the Chandelier.

In this way, an arbitrary number of octaves of the Chandelier's unusual tonal space can be easily mapped.

Timbre and Specificities

As mentioned above, the electromagnetically-driven strings of the Chandelier offer very little timbral variation. Its natural tremolo varies with the string's amplitude and can therefor be controlled via the dynamics.

Dynamics

My current implementation of the Chandelier controller did not control the Chandelier's dynamics when I started writing this section. This is because the Chandelier was underpowered during much of its development. And the focus was on producing the largest string amplitudes possible for the available current. Here I describe how this was added for purposes of evaluation.

The dynamics can be controlled very directly by varying the strength of the magnetic pulses that drive the string. This can be done very simply by adding a pulse-width modulator module to each oscillator.

But high-frequency PWM signals could have complex interactions with the electromagnet, which is a large solenoid. And a low-frequency PWM could disrupt the sensitive rhythms of the audio-frequency signals that set the string resonating.

I chose a simpler solution - modifying the duty cycle of the slow, audio-

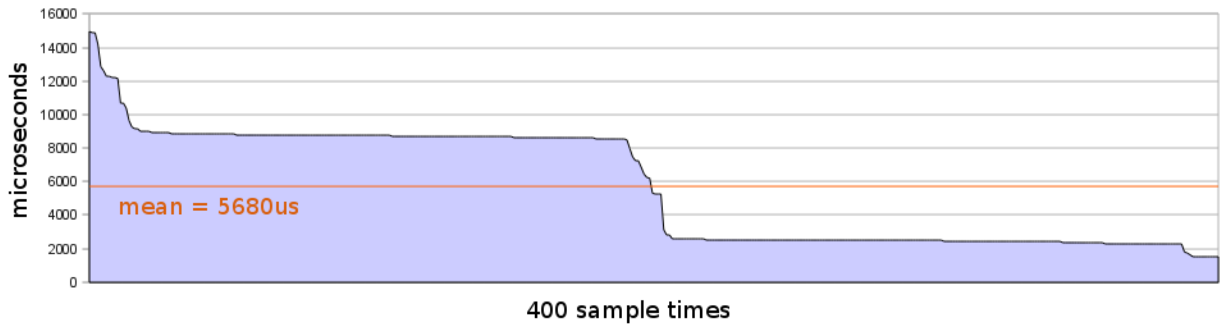


Illustration 31: latency for note-on and note-off events

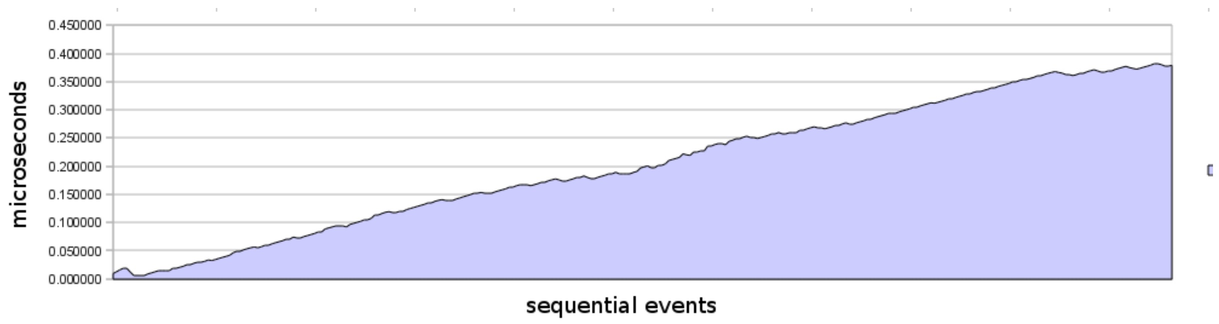


Illustration 32: rising latency, showing the slow flooding of the controller

frequency square waves that drive the electromagnets. The square waves originally had a 50% duty cycle. Duty cycles lower than 50% will impart less energy to the string, changing the amplitude.

Illustration 30 shows how the audio-frequency pulse widths were modulated by adding one new wire vector, duty_cycle, to the current square_waves module in the FPGA. Listing 6 shows the code that generates the new circuit.

Figure 6 below shows a new version of the Verilog module square_waves (Appendix A5) augmented to use a variable duty cycle. The wire vector duty_cycle is printed in red, to show where changes have been made.

So only a very small change was needed to enable the current Chandelier controller to exploit and control the Chandelier's natural dynamic range.

Slow Vibrato

The creation of a slow vibrato requires updating 2 separate files.

The custom instrument behavior written for the Chandelier Bellum contains a class called Vibr. This class calculates a slow, global vibrato that can be applied to all current notes, thereby driving the strings out of phase. Details of the Vibr class can be seen in Appendix A4.

The Chandelier's definition.py file (Appendix A2), contains a list called inlets_l, which defines the elements of the timbral data array. The first 2 elements are vibrato_speed and vibrato_depth. Their entry in inlets_l causes them to show up as mappable timbres for the Chandelier in Nervebox UI (as seen in Illustration 16) and also to occupy the first 2 positions in the timbral data array of NerveOSC packages addressed to the Chandelier's Bellum.

Values for vibrato_speed and vibrato_depth received by the Chandelier Bellum will change the parameters of the Vibr class and accordingly alter the speed and depth of the vibrato.

Multiple Strings

The aforementioned TonalStructure class (Appendix A4) in the Chandelier Bellum maps the frequencies of notes in incoming NerveOSC packets to the complex tonal space of the Chandelier. It took only a few lines of code to modify it to return all matches, on all strings, within a certain number of cents.

Harmonics

The addition of Hammond Organ-like harmonics is achieved in 3 steps.

First, the values "-1 octave", "+ 3/2 octave", "+ 1 octave", "+ 5/2 octave", "+ 2 octaves", "+ 9/4 octaves", "+ 7/2 octaves", and "+ 3 octaves" are added to inlets_l in definition.py. This causes them to become mappable timbres in Nervebox UI (see Illustration 16).

Second, a mapping is created that assigns values to the new timbre parameters.

Third, the parseOSC function in the Chandelier Bellum's custom instrument behavior is extended to create and play new musically appropriate notes for each mapped harmonic. See the parseOSC function in Appendix A4.

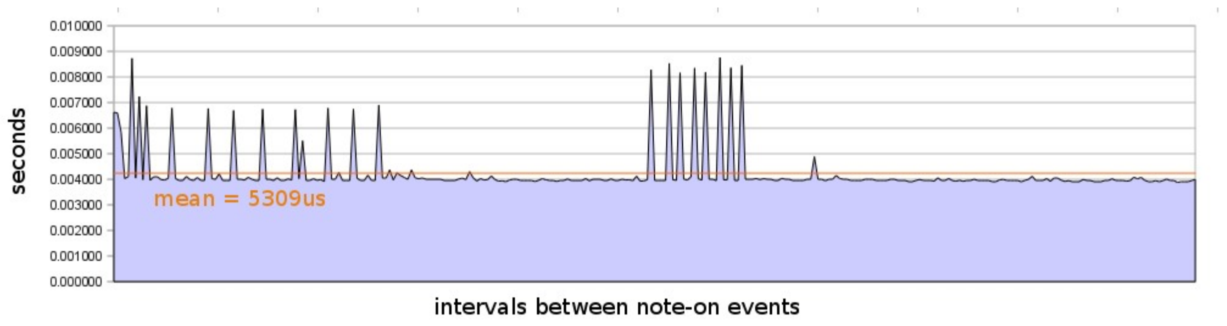


Illustration 33: measurement of minimum intervals between note-on events

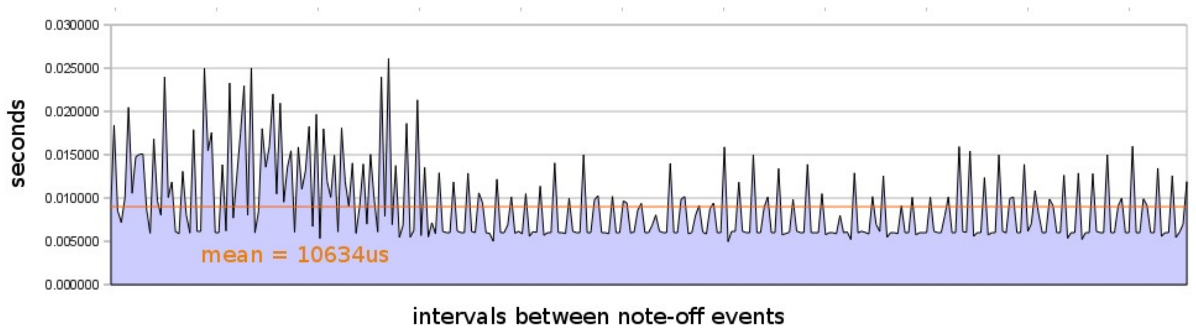


Illustration 34: measurement of minimum intervals between note-off events

4.2.4 Fidelity of Nervebox-based Chandelier controller

To measure the fidelity of the Chandelier controller, I measured its errors, latency, and the limits of its throughput.

I performed these tests on a Dell Inspiron 1525 laptop with 2GB of RAM and a 1.66GHz Intel Core2 Duo processor. The laptop was running Ubuntu 9.10 and Python 2.6.4.

The test harness for these measurements records the time, in microseconds, when MIDI events first enter the Brum and when they leave the Bellum via its serial port. I would have preferred to take measurements from the very end of the chain, from the Dulla's current-switching modules. But I did not have the means with which to sync the microsecond precision of processor-based measurements with any time measurements of the current-switching side of the Dulla. Nonetheless, these timing measurements span the components of the Chandelier controller that do the complex processing and heavy lifting.

First I measured the total end-to-end latency of events. Illustration 29 shows the distribution of latency in 200 note-on and 200 note-off events. The note-off events took considerably less time than the note-on events. This is expected, as the note-on events require the Bellum to to scan the Chandelier's tonal space multiple times for each event and each harmonic. Illustration 29 shows this disparity by displaying these latencies sorted from high to low. The mean latency for note-on events is 8921 microseconds and the mean latency for note-off events is 2448 microseconds. The mean latency for both note-on and note-off events is is the one that affects performance, since they occur in pairs. This value is 5680 microseconds, which I consider to be comfortably small.

Next I measured the maximum end-to-end throughput. I did this by adding a function to the test harness that generates MIDI notes slightly faster than the Chandelier controller can process them. Illustration 30 shows how the latency of a stream of 300 events slowly increases when MIDI notes are entering the system at a rate slightly higher than the maximum throughput. The slow increase in latency demonstrates that

Photo 7: The Heliphon

the controller is saturated with events during the testing period. The rate at which events emerge from the other end of the stream is a good measure if the maximum throughput.

The throughputs for note-on and note-off events were noticeably different in early testing. So I created new tests that show the two patterns separately.

Illustration 33 shows the intervals between 300 sequential note-on events. The mean interval value is 5309 microseconds. This

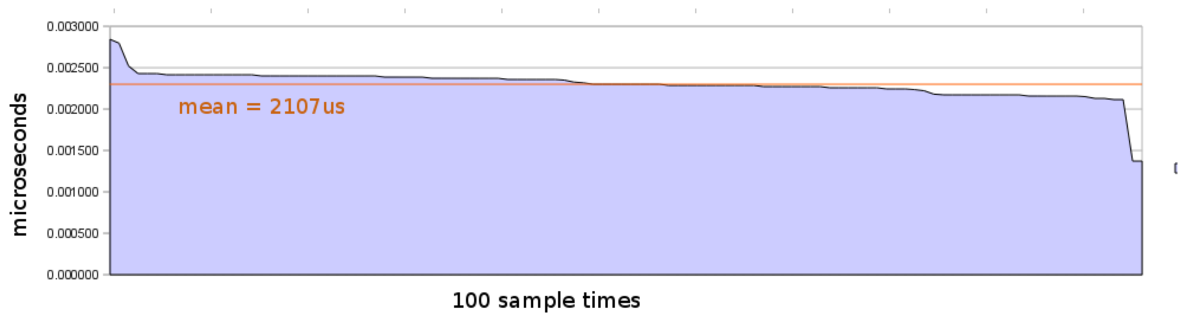


Illustration 35: latency for note-on and note-off events

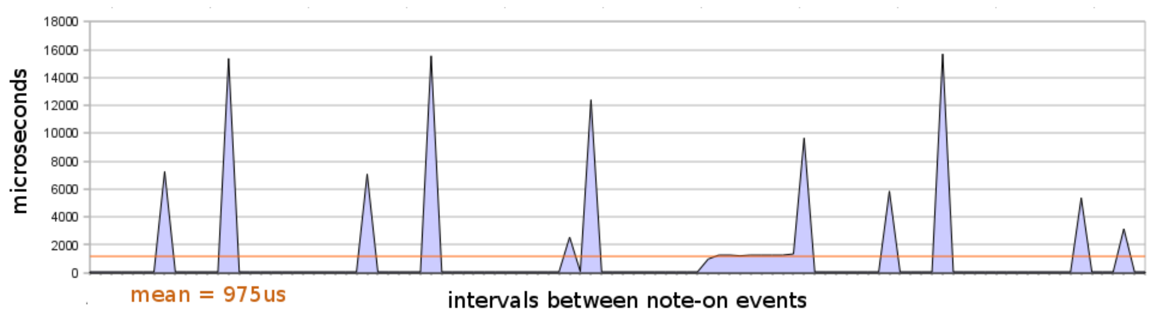


Illustration 36: measurement of minimum intervals between note-on events

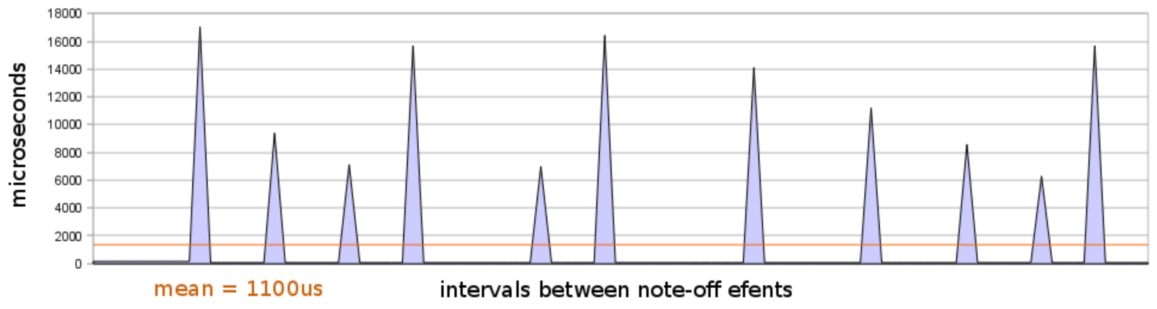


Illustration 37: measurement of minimum intervals between note-off events

corresponds to a throughput of 188 events per second.

Illustration 34 shows the intervals between 300 sequential note-off events. The mean interval value is 10634µs. This corresponds to a throughput of 94 events per second.

The test harness measuring the input and output of the system also scanned for dropped packets, incorrect ordering, and incorrect values. The total count for each of these types of errors was zero.

4.2.5 Conclusion

The system latency is acceptably low — especially for a controller that must perform so many tonal calculations for every note.

The throughput is surprisingly low. The two note-on and note-off values average out to about ~141 events per second. This is fine for the Chandelier, which has a very slow attack time.

It is surprising that the throughput for note-off events is lower than that for note-on events, as they require fewer calculations.

An error rate of zero, even when the controller is saturated with messages, is a pleasant surprise. Though it is clear that if the input rate exceeds the maximum throughput for too long, then buffers somewhere in the chain will overflow and packets will be lost. I'm not interested in measuring this threshold, as the important rule is to prevent the input rate from exceeding the maximum throughput.

4.3 The Heliphon

The Heliphon (see Photo 8) is a simple electromechanical musical instrument developed by Ensemble Robot. It features 25 tuned metal bars that are struck by 25 linear solenoid actuators. The Heliphon is far simpler than the Chandelier, both musically and mechanically.

4.3.1 Expressive Dimensions of the Heliphon

Tonal Range

The Heliphon plays 25 discreet, unbendable notes, from G3 to G5.

Timbre and Specificities

The only timbral dimension I've been able to identify in the Heliphon is a certain plinkiness that increases with the amplitude. This is caused by an increase in the duration of contact between the bar and the solenoid rod.

Dynamics

The Heliphon has a small dynamic range which can be accessed by charging a bar's solenoid for different periods of time. This causes the solenoid rod to strike the bar at different velocities. The effective range is small. If the charge period is lower than ~20ms, the solenoid fails to reach the bar. If the charge period is greater than ~55ms, the rod connects with the bar for too long, damping it and creating an inharmonic timbre.

4.3.2 Extra Credit: Synthetic Expressive Dimensions of the Heliphon

The Heliphon has one interesting dimension — speed. The instrument was built to play faster than any instrument in a Balinese Gamelan. It can play the same note up to 8 times per second. And the speed at which it can play sequences of different notes is limited only by this ~125ms return time for each note.

Trill

To exploit the Heliphon's speed, I've created a synthetic expressive dimension that I call trill. Trill is a term for several similar effects — single repeating notes, 2-note trills, and arpeggios.

Trill has 4 parameters — speed, depth, direction and contour.

There is no minimum speed required by the instrument. But it is not practical for the trill's top speed to exceed the instrument's top speed. Trill depth can range from a single note to an arpeggio of all active note to an arpeggio of all active notes plus an extrapolation thereof. Trill direction denotes the tonal direction of an arpeggio. And trill contour denotes whether the trill increases or decreases over time.

4.3.3 Expressivity of Nervebox-based Heliphon controller

As the Heliphon is a simple instrument, the evaluation of its expressivity is simpler than that of the Chandelier.

Tonal Range

Nervebox can exploit and control the tonal range from G3 to G5 in 12-tone equal temperament without any special code or mapping. The list of available frequencies are simply added to the Heliphon's definition.py file.

Timbre and Specificities

This instrument's one timbral dimension is linked to dynamics, and not independently controllable.

Dynamics

Once the range of valid solenoid charge durations had been measured, it was a simple matter to map the amplitude values of incoming NerveOSC packets to the solenoid charge durations in Bellum. The solenoid charge durations are passed to the FPGA, which handles the charge and discharge of the solenoids.

Corresponding note-off events are automatically generated for each note-on, with the delay in between based on amplitude values. So incoming note-off events are ignored.

Trill

Adding the trill dimension and its 4 parameters required adding "trill_speed", "trill_depth", "trill_direction" and "trill_contour" to the inlets_l list in the Heliphon's definition.py file. This made these timbral values available for mapping within Nervebox UI.

Custom code for the Bellum was added to parse these values from the timbral data array and pass them to a multithreaded class called Triller, with produces all of the trill effects listed in 4.2.2.

The extrapolation of a scale, which occurs when trill_depth is set to its maximum value, is achieved simply by using notes from an octave above or below to double each note.

4.3.4 Fidelity of the Nervebox-based Heliphon controller

I measured the errors, latency, and throughput of the Heliphon controller using the same experimental setup that was used for the previous Chandelier test.

The Heliphon controller is different from the Chandelier controller in that it uses a different mapping in the Brum, different custom code in its Bellum, a different FPGA configuration, and a different definition.py file.

The custom code in the Bellum is much simpler than that of the Chandelier. Its tonal mapping is a simple one-to-one list, as is its amplitude mapping. Its only non-trivial feature is the Triller class, which is much simpler and far less computationally expensive than the Chandelier's Vibr class.

I measured the total end-to-end latency of events. Illustration 33 shows the results, ordered by latency values. The simplicity of the Heliphon's behavior is reflected in the very low latency shown in the tests. The mean end-to-end latency of the Heliphon controller is only 2107 microseconds.

Next I measured the maximum end-to-end throughput using the same flooding technique used with the Chandelier. Note-on events emerged from the Bellum with mean intervals of 975 microseconds, as shown in Illustration 36.

The results for note-off events were similar, with a mean of 1100 microseconds.

The average of the mean intervals for note-on and note-off events is 1037.5 microseconds, which corresponds to a throughput of 963 events per second.

Once again, the test harness scanned for dropped packets, incorrect ordering, and incorrect values. The total count for each of these types of errors was zero.

4.3.5 Conclusion

A controller built with the Nervebox platform can easily exploit and control all of the expressive parameters of the Heliphon.

The systems latency is imperceptibly low. And again no errors were found during testing.

The throughput of 963 events per second is plenty for even a timing-sensitive instrument like the Heliphon.

To judge whether 963 events per second this is a good value, we can compare it to the nominal maximum value for MIDI transfers. MIDI's hardware transport has a nominal maxim throughput of 1042 messages per second.

So this controller's end-to-end throughput is nearly as high as data flowing unprocessed through a MIDI cable. And these NerveOSC events carry more musical data than MIDI messages.

5 Conclusion

The abstractions presented by the Nervebox platform seem to be well placed.

Using the Nervebox platform, I was able to relatively easily build control systems for two very different electromechanical musical instruments. Aspects the the development process that were common to all electromechanical instruments were neatly abstracted behind generalized software and hardware. These include the gritty details of input mapping, internal music representation, the control network, output mapping, actuation, and a user interface.

Development time was spent only on the unique aspects of the instruments. And the Nervebox-based controllers were able to exploit and control all of each instrument's expressive dimensions.

The current implementation could use improvement. The throughput for even complex musical processing should run at a speed that can keep up with precise music.

My personal experience as a user of Nervebox, rather than as a developer, was full of pleasant surprises. The Nervebox UI enabled me to create mappings in minutes that before had taken a day or more of hand-coding to write.

But I discovered that the idea of FPGAs is more appealing than the reality. Verilog is a powerful and elegant way to express the idea of a complex and time-sensitive machine. But the FPGAs themselves, from both major manufacturers, are full of strange quirks that can only be learned through experience.

Still, I found the process of developing control circuits with an FPGA faster and easier than with integrated circuits and discrete components — if only because I could test and iterate designs continuously without needing to buy or spec parts.

Of course, the most important test of Nervebox's usefulness will happen if and when other musical experimenters use it to build controllers for their own instruments.

6 Future: Openness and Community

It will take more than new technologies and abstractions to create a new boom in electromechanical music. It will take a community.

There are many individuals and small groups making electromechanical instruments. These instruments, ideas and technologies are evolving separately in isolation, like animals of the Galapagos Islands.

I hope that by building a website around an open-source version of Nervebox, I can help create a community of these far-flung groups and individuals.

Visitors will be able to download the Python, Verilog and Javascript code, as well as circuit board layouts in various popular formats such as DFX and PDF.

More importantly, visitors will be able to share their own modular code and circuits, and their machines, music, and inspirations.

It is my hope that the field of electromechanical music can finally enjoy the type of vibrant community already enjoyed by the fields of digital and analog synthesis.

Appendix A: Code and Circuits

A1: example mapping for Chandelier

```
[
# modules
{action:"new", type:"modules", name:"0",
param:"MIDI_Source_Stream", client_x:23, client_y:14},
{action:"new", type:"modules", name:"1",
param:"MIDI_Filter_Command", client_x:27, client_y:251},
{action:"new", type:"modules", name:"2", param:"MIDI_to_OSC",
client_x:55, client_y:321},
{action:"new", type:"modules", name:"3",
param:"MIDI_Filter_Channel", client_x:383, client_y:159},
{action:"new", type:"modules", name:"4", param:"MIDI_to_OSC",
client_x:26, client_y:476},
# functions
{action:"setSendOnPitchBend", type:"function", name:"0", param:false},
{action:"setOSCPath", type:"function", name:"4",
param:"/chandelier/kill/"},
{action:"setFreqMap", type:"function", name:"4",
param:"et31_offset_0_l"},
{action:"setInstrument", type:"function", name:"4", param:"chandelier"},
{action:"setFreqMap", type:"function", name:"2",
param:"et31_offset_0_l"},
{action:"setOSCPath", type:"function", name:"2",
param:"/chandelier/freq/"},
{action:"setInstrument", type:"function", name:"2", param:"chandelier"},
{action:"setSendOnModWheel", type:"function", name:"0", param:true},
{action:"setMIDIDevice", type:"function", name:"0",
param:"General_midi"},
{action:"setPath", type:"function", name:"0", param:"/dev/midi1"},
# connections
{dest_inlet:0, dest_name:"2", type:"connection", action:"add",
src_name:"1", src_outlet:1},
{dest_inlet:0, dest_name:"3", type:"connection", action:"add",
src_name:"0", src_outlet:0},
{dest_inlet:0, dest_name:"1", type:"connection", action:"add",
src_name:"3", src_outlet:0},
{dest_inlet:1, dest_name:"2", type:"connection", action:"add",
src_name:"1", src_outlet:3},
{dest_inlet:2, dest_name:"2", type:"connection", action:"add",
src_name:"1", src_outlet:6},
{dest_inlet:0, dest_name:"4", type:"connection", action:"add",
src_name:"1", src_outlet:0},
{dest_inlet:3, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:1},
{dest_inlet:4, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:2},
{dest_inlet:5, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:3},
{dest_inlet:6, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:4},
{dest_inlet:7, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:5},
{dest_inlet:8, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:6},
{dest_inlet:9, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:7},
{dest_inlet:10, dest_name:"2", type:"connection", action:"add",
src_name:"3", src_outlet:8}
]
```

A2: definition.py file for Chandelier

Red ellipses (...) indicate truncations in this 1081-line file.

```
definition={
 "name":"chandelier",
 "present_b":False,
 "network":False,
 "inlets_l":[
  "vibrato_speed",
  "vibrato_depth",
  "-1 octave",
  "+ 3/2 octave",
  "+ 1 octave",
  "+ 5/2 octave",
  "+ 2 octaves",
  "+ 9/4 octaves",
  "+ 7/2 octaves",
  "+ 3 octaves"
 ],
 "paths_l":[
  "/chandelier/freq/",
  "/chandelier/string/1",
  "/chandelier/string/2",

  ...

  "/chandelier/string/31",
  "/chandelier/kill/",
  "/chandelier/test/",
 ],
 "tuning":["equal_temperament", 31.0, 8.1757989156],
 "freqs_l":[
  ["27.5"],
  ["28.12"],
  ["28.76"],

  ...

  ["1721.08"],
 ],
 "strings":{
  "00":[
"27.5","55","82.5","110","137.5","165","192.5","220","247.5","275","302.5","
330","357.5","385","412.5","440","467.5","495","522.5","550","577.5","605",
"632.5","660","687.5","715","742.5","770","797.5","825","852.5","880" ],
  "01":[
"28.12","56.24","84.37","112.49","140.61","168.73","196.85","224.97","253.
1","281.22","309.34","337.46","365.58","393.71","421.83","449.95","478.07
","506.19","534.31","562.44","590.56","618.68","646.8","674.92","703.05","
731.17","759.29","787.41","815.53","843.65","871.78","899.9"],
  "02":[
"28.76","57.52","86.27","115.03","143.79","172.55","201.3","230.06","258.8
2","287.58","316.33","345.09","373.85","402.61","431.37","460.12","488.88
","517.64","546.4","575.15","603.91","632.67","661.43","690.18","718.94","
747.7","776.46","805.22","833.97","862.73","891.49","920.25"],

  ....

  "31":[
"53.78","107.57","161.35","215.14","268.92","322.7","376.49","430.27","48
4.05","537.84","591.62","645.41","699.19","752.97","806.76","860.54","914
.33","968.11","1021.89","1075.68","1129.46","1183.25","1237.03","1290.81
","1344.6","1398.38","1452.16","1505.95","1559.73","1613.52","1667.3","1
721.08"],
 },
}
```

A3: Generic Nervebox Python code for Bellum

```
import sys
import osc
import time
import json
import math
import copy
import threading
import ConfigParser

try:
  sys.path.index('/opt/nervebox')
except ValueError: # if nervebox is NOT in the path
  sys.path.append('/opt/nervebox')
from bellums import bellum_network
from bellums import bellum_serialPort

import definition

config = ConfigParser.ConfigParser()
config.read(['/opt/nervebox/nervebox.cfg'])
BRUM_IP = config.get('network','BRUM_IP')
BROKER_PORT = int(config.get('network','BROKER_PORT'))
trace_enable = False
freqs_l = definition.definition["freqs_l"]
name_str = definition.definition["name"]

connections_d = None # global stub, instantiated in registerBellum
serial_port = bellum_serialPort.SerialPort()
serial_port.connect()

class EventManager:
  def __init__(self):
    self.events_d = {}
    self.lock = threading.Event()
  def add(self, event_d):
    self.lock.wait()
    self.events_d[event_d["event_id"]] = event_d
  def remove(self, event_int):
    self.lock.wait()
    del self.events_d[event_int]
  def get(self, event_int):
    self.lock.wait()
    if self.events_d.has_key(event_int):
      return self.events_d[event_int]
    else:
      return None
  def getAllKeys(self):
    self.lock.wait()
    return self.events_d.keys()

eventmanager = EventManager()

def brumListener(msg_j):
  msg_l = json.loads(msg_j)
  src_str = msg_l[0]
  action_str = msg_l[1]
  data = msg_l[2]
  if src_str == "system":
    if action_str == "trace_enable":
      global trace_enable
      trace_enable = data

def sendSerialData(binaryWord_str):
  if serial_port.connected:
    bwLen_int = len(binaryWord_str)
    if bwLen_int not in [8,14,18,24,25,30,35,40,44,48,52,56,60,64]:
      print "Error in makeSerialPackets, invalid length for
binaryWord_str:", bwLen_int
      return
    if bwLen_int == 8:
      stuffByteLength_int = 0
    elif bwLen_int <= 14:
      stuffByteLength_int = 1
    elif bwLen_int <= 24:
      stuffByteLength_int = 2
    elif bwLen_int <= 40:
      stuffByteLength_int = 3
    else:
      stuffByteLength_int = 4
    payloadLength_int = 8 - stuffByteLength_int
    packets_l = []
    packetNumber_int = 0
    while len(binaryWord_str) > 0:
      byteStuff_str = dec2bin(packetNumber_int, stuffByteLength_int) #
packet ordinal
      payload_str = binaryWord_str[0:payloadLength_int] # segment of
binary word
      binaryWord_str = binaryWord_str[payloadLength_int:] # truncate
binary word
      packetNumber_int += 1 # increment packet ordinal
      packet_int = int(byteStuff_str + payload_str, 2) # combine binary
strings and convert into base-10 value
      packet_chr = chr(packet_int)
      # packets_l.append(packet_int)
      serial_port.send(packet_chr)
  else:
    print "serial port not connected"

def dec2bin(n, fill):
  bStr = ''
  while n > 0:
    bStr = str(n % 2) + bStr
    n = n >> 1
  return bStr.zfill(fill)

def registerBellum():
  global name_str
  global connections_d
  connections_d = bellum_network.init(BRUM_IP, BROKER_PORT,
brumListener)
  connections_j = json.dumps(
    {"cmd":"register","data":{"name":name_str,
"server":connections_d["server"]["port"],
"client":connections_d["client"]["port"],
"oscServer":connections_d["oscServer"]["port"]}}
  )
  connections_d["client"]["thread"].send(connections_j)

class Scheduler(threading.Thread):
  def __init__(self):
    threading.Thread.__init__(self)
    self.queue_l = []
  def run(self):
    while True:
      self.timestamp = time.time() # create timestamp for NOW
      for evt_ord in range(len(self.queue_l)): # loop through all event ints
in queue
        try:
          evt = self.queue_l[evt_ord] # get reference to event
          if evt["timestamp"] < self.timestamp: # if event's timestamp is
earlier than NOW timestamp
            schedule_lock.set()
            executeOSC(evt["osc_data_d"])# send midi event
            schedule_lock.clear()
            self.queue_l.pop(evt_ord)# delete event
        except Exception as e:
```

```
        print "exception in main.Scheduler", e.args
      time.sleep(0.001)
  def add(self, osc_data_d, delay):
    timestamp = time.time() + delay
    self.queue_l.append(
      {
        "timestamp":timestamp,
        "osc_data_d":osc_data_d,
        "delay":delay
      }
    )

schedule_lock =  threading.Event()
registerBellum()
```

A4: Chandelier-specific Python code for Bellum

```
FPGAClock = 50000000
vibrato_rate = 64.0 # this default rate can be overwritten by MIDI
values from a keyboard mod wheel
vibrato_depth = 6 # in cents

class TonalStructure():
  intervalSearchOrder=[
    "tonic",
    "octave",
    "fifth",
    "majorthird",
    "minorseventh",
    "majorsecond",
    "tritone",
    "minorsixth",
    "majorseventh",
    "minorsecond",
    "minorthird",
    "fourth",
    "majorsixth",
  ]
  intervalToHarmonic={
    "tonic":[1],
    "octave":[2, 4, 8, 16, 32],
    "fifth":[3, 6, 12, 24],
    "majorthird":[5, 10, 20],
    "minorseventh":[7, 14, 28, 29],
    "majorsecond":[9, 18],
    "tritone":[11, 22, 23],
    "minorsixth":[13, 26, 25],
    "majorseventh":[15, 30, 31],
    "minorsecond":[17],
    "minorthird":[19],
    "fourth":[21],
    "majorsixth":[27],
  }
  def __init__(self):
    pass
  def calcCentsDiff(self, freq_lo_float, freq_hi_float):
    cents =  1200 * math.log( freq_lo_float/freq_hi_float ) / math.log(2);
    return cents
  def freqMatch(self, freq_float, tolerance_int):
    stringNames_l = definition.definition["strings"].keys()
    stringNames_l.sort()
    stringFreq_l = []
    for intervalName in self.intervalSearchOrder:# loop through
preferred intervals in order
      harmonics_l = self.intervalToHarmonic[intervalName]
      for h in harmonics_l: # loop through harmonic numbers, in order of
```

```
preferred intervals
    for stringName_str in stringNames_l: # loop through each string
      freqs_l = definition.definition["strings"][stringName_str]
      ch_freq = float(freqs_l[h-1])
      cents = self.calcCentsDiff(ch_freq, freq_float)
      if abs(cents) < tolerance_int:
        stringFreq_l.append([stringName_str, ch_freq])
  return stringFreq_l

tonalstructure = TonalStructure()

class _Vibr(threading.Thread):
 def __init__(self, depth):
  threading.Thread.__init__(self)
  self.depth = depth # vibrato depth, measured in cents
 def run(self):
  global vibrato_rate
  vibrato_increment_f = 0
  while 1:
   vibrato_increment_f = vibrato_increment_f + (vibrato_rate/512)
   vibrato_coefficient = math.sin(vibrato_increment_f)
   # get list of current frequencies from eventmanager
   eventmanager.lock.set()
   eventKeys_l = eventmanager.getAllKeys()
   eventmanager.lock.clear()
   for eventKey in eventKeys_l:
    eventmanager.lock.set()
    evt = eventmanager.get(eventKey)
    eventmanager.lock.clear()
    if evt != None: # if event exists.  it might not if delete immediately
before get()
     stringFreq_l = evt["freqs_l"]
     for sf_l in stringFreq_l:
      _freq = sf_l[1] if vibrato_rate == 0 else
self.vibrato_calculation(sf_l[1], vibrato_coefficient)
      bWord_str = makeBinaryWord(sf_l[0], str(_freq))
      sendSerialData(bWord_str)
      time.sleep(.02)
 def vibrato_calculation(self, freq, vibrato_coefficient):
   exponent = vibrato_coefficient * (float(self.depth) / float(1200)) #
there are 1200 cents per octave
   vFreq = float(freq) *  float(pow(2, exponent))
   return vFreq

_vibr = _Vibr(vibrato_depth)
_vibr.start()

def parseOSC(*raw):
 global trace_enable
 osc_data_l = raw[0]
 event_d = {
  "osc_addr":osc_data_l[0],
  "event_id":osc_data_l[2],
  "freq":osc_data_l[3],
  "amplitude":osc_data_l[4] if len(osc_data_l) > 5 else "",
  "vibrato_speed":int(osc_data_l[5]) if len(osc_data_l) > 5 else 0,
  "vibrato_depth":int(osc_data_l[6]) if len(osc_data_l) > 6 else 0,
  "-1 octave":int(osc_data_l[7]) if len(osc_data_l) > 7 else 0,
  "+ 3/2 octave":int(osc_data_l[8]) if len(osc_data_l) > 8 else 0,
  "+ 1 octave":int(osc_data_l[9]) if len(osc_data_l) > 9 else 0,
  "+ 5/2 octave":int(osc_data_l[10]) if len(osc_data_l) > 10 else 0,
  "+ 2 octaves":int(osc_data_l[11]) if len(osc_data_l) > 11 else 0,
  "+ 9/4 octaves":int(osc_data_l[12]) if len(osc_data_l) > 12 else 0,
  "+ 7/2 octaves":int(osc_data_l[13]) if len(osc_data_l) > 13 else 0,
  "+ 3 octaves":int(osc_data_l[14]) if len(osc_data_l) > 14 else 0
 }
 if event_d["osc_addr"] == "/chandelier/freq/":
  freq = float(event_d["freq"]) # convert freq string to freq float
  event_d["freqs_l"] = [] #
  sf_l = tonalstructure.freqMatch(freq, 5)
```

```
    if len(sf_l) > 0:
      event_d["freqs_l"].extend(sf_l)
    if event_d["-1 octave"] != 0:
      sf_l = tonalstructure.freqMatch((freq / 2), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 3/2 octave"] != 0:
      sf_l = tonalstructure.freqMatch((freq * (3/2)), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 1 octave"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 2), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 5/2 octave"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 3), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 2 octaves"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 4), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 9/4 octaves"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 5), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 7/2 octaves"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 6), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    if event_d["+ 3 octaves"] != 0:
      sf_l = tonalstructure.freqMatch((freq * 8), 5)
      if len(sf_l) > 0:
        event_d["freqs_l"].extend(sf_l)
    eventmanager.lock.set()
    eventmanager.add(event_d)
    eventmanager.lock.clear()
  if event_d["osc_addr"] == "/chandelier/kill/":
    eventmanager.lock.set()
    e_d = eventmanager.get(event_d["event_id"])
    eventmanager.remove(event_d["event_id"])
    eventmanager.lock.clear()
    if e_d == None:
      print "parseOSC /kill no event found"
      return
    f_2l = e_d['freqs_l']
    for f_l in f_2l:
      stringId = f_l[0]
      bWord_str = makeBinaryWord(stringId, "0")
      sendSerialData(bWord_str)

def makeBinaryWord(ch_str, f_str):
  """
  000 sssss (stringId)
  001 sffff (stringId, freq)
  010 fffff (freq)
  011 fffff (freq)
  100 fffff (freq)
  101 fffff (freq)
  """
  stringId_b_str = dec2bin(int(ch_str)+1, 6)
  if float(f_str) == 0:
    period_b_str = "000000000000000000000000"
  else:
    period_int = int((FPGAClock)/float(f_str))
    period_b_str = dec2bin(period_int, 24)
  word_b_str = stringId_b_str + period_b_str
  return word_b_str
```

```
def OSCBind():
  """ associate all paths in definition with mapper function """
  for path_str in definition.definition["paths_l"]:
    connections_d["oscServer"]["thread"].bind(parseOSC, path_str)

OSCBind()
```

A5: Verilog code for Chandelier Dulla

```
/* declare main module */
module chandelier(
        input clock,
        input RxD,
        output square_wave_pin_01,
        output square_wave_pin_02,
        output square_wave_pin_03,
        ...
        output square_wave_pin_48,
);

/* create 48 register vectors to hold period data*/
parameter periodBitWidth = 23;
reg [periodBitWidth:0] period_01;
reg [periodBitWidth:0] period_02;
reg [periodBitWidth:0] period_03;
...
reg [periodBitWidth:0] period_48;

/* create register vector longpacket to accumulate bits from  RS-232
deserializer */
reg [39:0] longpacket = 0;

/* create 48 variable square wave oscillators */
square_waves CHAN01(clock, square_wave_pin_01, period_01);
square_waves CHAN02(clock, square_wave_pin_02, period_02);
square_waves CHAN03(clock, square_wave_pin_03, period_03);
...
square_waves CHAN31(clock, square_wave_pin_48, period_48);

/* create RS-232  deserializer */
wire RxD_data_ready;
wire [7:0] RxD_data;
async_receiver deserializer(.clock(clock), .RxD(RxD),
.RxD_data_ready(RxD_data_ready), .RxD_data(RxD_data));

/* sort incoming packets and store in register vector longpacket */
always @(posedge clock) if(RxD_data_ready)
        begin

        /* sort which packet in 8-byte sequence */

        case(RxD_data[7:5])


        3'b000: longpacket[39:35] <= RxD_data[4:0];
        3'b001: longpacket[34:30] <= RxD_data[4:0];
        3'b010: longpacket[29:25] <= RxD_data[4:0];
        3'b011: longpacket[24:20] <= RxD_data[4:0];
        3'b100: longpacket[19:15] <= RxD_data[4:0];
        3'b101: longpacket[14:10] <= RxD_data[4:0];
        3'b110: longpacket[09:05] <= RxD_data[4:0];
        3'b111:
        begin
            longpacket[04:00] <= RxD_data[4:0];

        /* register vector longpacket is full */
        /* sort which string id */
        case(longpacket[38:33])
        /* copy period bits to appropriate period register vector */
        6'b000000:period_01 <= longpacket[23:0];
        6'b000001:period_02 <= longpacket[23:0];
        6'b000010:period_03 <= longpacket[23:0];
        ...

        6'b101111:period_48 <= longpacket[23:0];
        endcase
        end
        endcase
        end
endmodule

/* variable frequency square wave generator module */
module square_waves (
        nput clock, // wire from system clock
        input [23:0] period, // 24 wires setting value for square wave
period
        output square_wave_pin_out // wire to FPGA output pin
);
reg [24:0] period_counter = 0; // 25-bit register for period counter
reg wave_bool = 0; // boolean value sent to pin square_wave_pin_out
always @(posedge clock) // at the positive edge of every clock cycle
        period_counter <= ( period_counter > period*2)?0:
period_counter+1;
        // increment register period_counter, reset  to 0 when it exceeds
period*2
always @(posedge clock) // at the positive edge of every clock cycle
        wave_bool <= (period_counter  > period)?1:0;
        // set register wave_bool  to 1 if  period_counter  > period,
otherwise 0
assign  square_wave_pin_out = wave_bool;
// continuously assign value of wave_bool to square_wave_pin_out
```

```
endmodule;
```

A6: Schematic Diagram of Dulla amplifier module

Appendix B: Timbral Descriptors

Table 1 Harmonic descriptors I (from Peeters [00])
spectrum :
energy
spec Centroid (global mean spec)
spec.Centroid (global mean spec)
spec variation

harmonic :
spec energy
spec centroid
spec std
spec deviation (of the harmonic computed from the global mean spectrum)
spec slope
mean of the instantaneous energy
spec centroid computed on the vector composed of the maximum amplitude [lin] of cgsmax each harmonic over time
spec centroid computed on the vector composed of the mean amplitude [lin] of each cgsmoy harmonic over time
spec centroid computed on the vector composed of the rms amplitude [lin] of each cgsrms harmonic over time
mean of the instantaneous spec centroid [amp lin, freq lin]
mean of the instantaneous spec centroid [amp dB, freq lin]
mean of the instantaneous spec centroid [amp lin, freq log]
mean of the instantaneous spec centroid [amp dB, freq log]
spectral std computed on the vector composed of the maximum amplitude [lin] of each harmonic over time
spectral std computed on the vector composed of the mean amplitude [lin] of each harmonic over time
spectral std computed on the vector composed of the rms amplitude [lin] of each harmonic over time
mean of the instantaneous spec std [amp lin, freq lin]
mean of the instantaneous spec std [amp dB, freq lin]
mean of the instantaneous spec std [amp lin, freq log]
mean of the instantaneous spec std [amp dB, freq log]

Table 2. Harmonic descriptors II (from Peeters [00])
spectral std computed on the vector composed of the maximum of amplitude [dB] of each harmonic over time
spectral std computed on the vector composed of the mean of amplitude [dB] of each harmonic over time
spectral std computed on the vector composed of the rms of amplitude [dB] of each harmonic over time
mean of the instantaneous spec deviation [amp lin]
mean of the instantaneous spec deviation [amp dB]
mean of the instantaneous spec slope [amp lin]
mean of the instantaneous spec slope [amp dB]
spec flux using instantaneous spec centroid and cgsmax
spec flux using instantaneous spec centroid and cgsmoy
spec flux using instantaneous spec centroid and cgsrms
spec flux using instantaneous spec centroid and cgsi
harmonic spectral deviation
speed of variation of the spectrum
sum of the variations of the instantaneous harmonic from global mean harmonics
harmonic attack coherence

envelope :
log-attack time from [rms]
log-attack time from [max]
log-attack time from [smoothed rms]
log-attack time from [smoothed max]
effective duration
effective duration [norm by file length]

effective duration [norm by file length and f0]
effective duration [norm by file length and T]

skewness of the power spec
kurtosis of the power spec
slope of the power spec

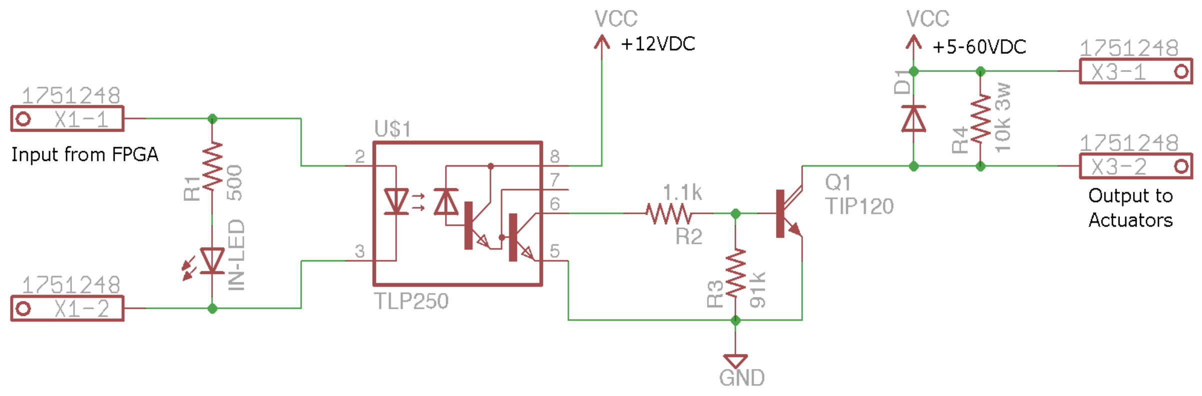


Table 3. Percussive descriptors (from Peeters [00])
log-attack time
temporal centroid
temporal std
effective duration
maximum value
ed*cgt
rms value of the power spectrum
rms value of the power spectrum [amp weighting dbA]
rms value of the power spectrum [amp weighting dbB]
rms value of the power spectrum [amp weighting dbC]
spec centroid of the power spec
spec centroid of the power spec [amp weighting dbA]
spec centroid of the power spec [amp weighting dbB]
spec centroid of the power spec [amp weighting dbC]
spec std of the power spec
spec std of the power spec [amp weighting dbA]
spec std of the power spec [amp weighting dbB]
spec std of the power spec [amp weighting dbC]

References

[1] Bitter Music: Collected Journals, Essays, Introductions, and Librettos, Harry Partch
University of Illinois Press, 2000

[2] Max at Seventeen, Puckette, Miller.
Computer Music Journal - Volume 26, Number 4, Winter 2002, pp. 31-43

[3] The Chandelier: An Exploration in Robotic Musical Instrument Design, Michael Fabio
Masters Thesis for MIT Media Lab, 2007

[4] Hybrid Percussion: Extending Physical Instruments Using Sampled Acoustics, Roberto Mario Aimi
PhD Dissertation for MIT Media Lab, 2007

[5] Telephonic Telegraph, Elisha Gray
U.S. Patent #233,345, 1880

[6] Electric Telegraph for Transmitting Musical Notes, Elisha Gray
U.S. Patent #166,096, 1875

[7] Music-Generating and Music-Distributing Apparatus, Thaddeus Cahill
U.S. Patent #1,107,261, 1914

[8] Magic Music from the Telharmonium, Reynold Weidenaar
The Scarecrow Press, 1995

[9] Electrical Musical Instrument, Laurens Hammond
U.S. Patent #1,956,350, 1934

[10] Vibrato Apparatus, Laurens Hammond and John M. Hanert
U.S. Patent #2,260,268, 1946

[11] Magnetic Tape Sound Reproducing Musical Instrument, Harry C. Chamberlin
U.S. Patent #2,940,351, 1960

[12] Sound Reproducing System, Harry C. Chamberlin
U.S. Patent #2,910,298, 1956

[13] Electronic Music Synthesizer, Robert A. Moog
U.S. Patent #4,050,343, 1977

[14] Tim Hawkinson's Überorgan, Getty Center
http://www.getty.edu/visit/events/hawkinson.html

[15] League of Electronic Musical Urban Robots
http://lemurbots.org/

[16] Ensemble Robot
http://ensemblerobot.com/VideoPages/whirly.shtml

[17] Absolut Quartet
http://www.absolut.com/absolutmachines

[18] Analytical Methods of Electroacoustic Music, Mary Simoni
Routledge, 2005

[19] Musimathics, Volume 1: The Mathematical Foundations of Music, Gareth Loy
The MIT Press, 2006

[20] Open Sound Control Protocol Spec

http://opensoundcontrol.org/spec-1_0

[21] What is the difference between OSC and MIDI?
http://opensoundcontrol.org/what-difference-between-osc-and-midi

[22] Why Is musical timbre so hard to understand?, Carol L Krumhansl
In Structure and Perception of Electroacoustic Sound and Music (eds. S.Nielzen & O Olsson), 1989

[23] An Exploration of Musical Timbre, J.M.Grey, 1975

[24] Tuning, Timbre, Spectrum, Scale, William Sethares
Springer, 1998

[25] Polyspectral Analysis of Musical Timbre, Shlomo Dubnov, Ph.D, 1996

[26] Timbre Space as a Musical Control Structure, David L. Wessel,
Rapport Ircam 12/78, 1978

[27] The Sackbut Blues: Hugh Le Caine, Pioneer in Electronic Music, Gayle Young
National Museum Of Science And Technology,1991

[28] The ZIPI Music Parameter Description Language, Keith McMillen, David L. Wessel, Matthew Wright
Published in Computer Music Journal 18:4 (Winter 94)

[29] SeaWave: A System for Musical Timbre Description, Russ Ethington, Bill Punch
Computer Music Journal Vol. 18, Issue 1 - Spring 1994

[30] Cross-Cultural Perception & Structure of Music, William H. Jackson , 1998
http://cybermesa.com/~bjackson/Papers/xc-music.htm

[31] The Analytical Language of John Wilkins, Jorge Luis Borges,
Essay. 1942

[32] JSON Specification, http://json.org/

[33] On the Sensations of Tone, Hermann Helmholz, 1863