

Acknowledgments

Acknowledgments to Edition 4.5

Everyone who has used Kyma over the past year and provided us with feedback has contributed to the improvements in this latest version of the software and the documentation. We would like to thank each of you and to invite you to continue to use Kyma and to keep talking to us! Your feedback has had a measurable impact!

Due to space limitations, this note must, of necessity, fall short of acknowledging everyone who has helped us this year, so please refer to the web site for a more complete listing of Kyma users (and please send us any news of your exploits): <http://www.SymbolicSound.com>.

A few individuals regularly contributed so much time and energy above and beyond the ordinary towards the improvement of Kyma, we wanted to mention them by name:

Barry O'Dell coordinated all the ordering of parts, and made everything come together at exactly the right time in exactly the right way so that your Capybara could be manufactured and tested.

Chip Leh is the friendly voice you hear on the telephone when you call Symbolic Sound, and he stress-tested each of the tutorials and all of the examples for this book.

Pete Johnston at the Tape Gallery spent many hours of transatlantic telephone time talking with us about how we could improve the morphing features in Kyma. Thanks to you, Pete, morphing sounds much better and is infinitely easier to do than it was a year ago. We hope you will continue to pressure us to make it even better (and easier) next year!

J. P. Jones was always the first to install and test each new beta version and the first to call in with bug reports, suggestions, and appreciative enthusiasm for the new features, followed immediately by an inquiry as to when the *next* revision would be coming out and whether we could send it by something faster than ordinary air mail. Thanks for your cheerful enthusiasm, your nontrivial sense of humor, your understanding and appreciation of what Kyma is about, your input and suggestions on both hardware and software developments, and thanks for being sonically insatiable; it helps keep us plummeting forward!

Francois "resistance is futile" Blaignan used Kyma on the sound track of several films and TV shows (including *Star Trek*), and we've spent many hours talking at length on the telephone in Champaign and in his studio in Hollywood about different sound design, synthesis and processing techniques. Thanks for your hospitality, thanks for your friendship and moral support, and thank you for hours of stimulating discussions!

Joel Chadabe, founder of the Electronic Music Foundation (www.emf.org) saw the potential of Kyma from the earliest days and has been continuously supportive of our efforts at many different levels. Thanks for making Symbolic Sound a part of history by including us in your book! (*Electric Sound: The Past and Promise of Electronic Music*, Prentice Hall).

Dennis Miller is another early supporter who has been unfailingly generous in sharing his knowledge of and enthusiasm for Kyma in print, on-line, and in person. We know we can always trust you to tell us we've done something great and to just as readily point out when some shareware program out there does one thing or another a little better (thus shaming us into improving Kyma all the time). Thanks for your enthusiasm and appreciation and thanks for never letting us get complacent about anything.

John Platt uses Kyma in both his teaching and in his research in psychoacoustics at McMaster University and it was his tenacious use of the early version of control panels that inspired us to come up with the current (and vastly improved) version that we call Tools. Thanks for your easy-going brilliance and fiendishly clever kludges, and thanks for never letting us forget about all the scientific applications of Kyma!

Lippold Haken has been our friend and colleague since 1980. Thanks for continuing to use Kyma in your research, and remember to never give up on sound!

Bill Rust posted several (unsolicited) endorsements on the Internet that sounded as if they could have come from our marketing department — actually they were even better and more enthusiastic than the marketing copy we generate. Thanks, Bill!

Thanks to a few weeks of low level, late night hacking, emailing, and phone calling between Mark Gorlinsky and Kurt Hebel, all Power Mac users can now run Kyma with or *without* virtual memory (so why can't all your other music software run with virtual memory turned *on* like Kyma can?)

Agostino Di Scipio has probably used Kyma in more live performances than any other individual. Thanks for pushing this aspect of Kyma!

Marcus Hobbs has pushed the development of the real-time evaluator and tools through his intensive and creative application of these features to implement xenharmonic tunings with colleagues Stephen Taylor and Erv Wilson.

Fred Szymanski made suggestions for how to “fatten” the emulated analog sounds and engaged us in stimulating discussions of nested control and controller feedback.

Thanks to all the Kyma 4.5 beta testers for living on the edge and providing essential feedback and suggestions: Francois Blaignan, Robert DeFord, Agostino Di Scipio, Larry Fritts, Vance Galloway, Lippold Haken, Marcus Hobbs, Pete Johnston, J. P. Jones, Chip Leh, Eric Lyon, John Mantegna, American McGee, Kelvin Russell, Bill Rust, Fred Szymanski, Lance Massey, John Platt, Andy Todd, and Yasushi Yoshida.

Everyone who attended the summer immersion weekends interacted with us in a hot house atmosphere where ideas were being exchanged so fast it was hard to tell who came up with what. The one thing that was certain was that Kyma was always a little bit better after each of the workshops. Thanks to all those who came through Champaign for workshops or consulting:

Dennis Miller, Chip Leh, Jeff Stolet, Karlheinz Essl, Bruno Liberda, Francois Blaignan, J. P. Jones, Agostino DiScipio, Nadine Miner, John Platt, Antonio Barata, Brian Belet, David Mooney, Pete Johnston, Larry Fritts, Mike Radentz, Bill Sequeira, Fred Szymanski, Will Chessner, Todd Watkins, Dragana Barac-Cikoja, Chris Czopnik, Marcus Hobbs, Laura Tedeschini-Lalli, Roberto D’Autilia, Greg Hunter, and Eugenio Giordani.

Thanks to Kevin Jones, Godric Wilke, Jason Edge, Joran Rudi, Adelhard Roidinger, and Bruno Liberda for inviting us to give workshops in Europe, thanks to Frank Serafine for letting us use his studios for a mini-workshop in L.A., and, to Lloyd Billing, thanks for letting us use the studios at Tape Gallery for a mini-workshop in London (and thanks for exploring those smaller, out-of-the-way booths at the AES show in New York).

And to our parents, thank you for understanding why we care so much about Kyma and Symbolic Sound, and thank you for *always* having believed in us, irrespective of the whims of fortune, politics, or popular opinion.

Carla Scaletti and Kurt Hebel
August, 1997

Acknowledgments to Edition 4.1

Addendum 4.1 of the Reference Manual was written by Carla Scaletti and Kurt Hebel.

Thanks to Lippold Haken, Barry O’Dell, and Chip Leh for their contributions to Kyma 4.1 and the Cypara-33.

Feedback from Kyma users was invaluable in shaping this new software release. We’d like to thank, in particular, Pete Johnston, John Platt, Agostino Di Scipio, Jonathan MacKenzie, Ron Kuivila, Fred Szymanski, John Dunn and Robert Austin, and all the others, too many to enumerate here, who had an influence on the shape and content of this release.

We would like to dedicate our work on this software release to our friend and colleague — Salvatore Martirano.[†] Whenever we had a new break-through we used to go over to Sal’s studio to show it to him,

[†] Salvatore Martirano, composer, teacher, and friend, died on the night of November 17, 1995 from complications due to ALS (Lou Gehrig’s disease). Throughout this difficult illness, he never lost his fantastic, unfettered musical imagination, his generous spirit, or his ironic sense of humor.

knowing that he would understand and appreciate all of its implications (as well as how much work had gone into it). It's hard to get used to the idea that we can't do that anymore. We miss you, Sal.

Acknowledgments to Edition 4.0

Version 4.0 of the Reference Manual was written by Jon D'Errico, Carla Scaletti wrote the Introduction, Tutorials, and Prototype Reference, and Kurt J. Hebel brought it all together through many hours of proofreading, rewriting, editing, and test-driving of the entire manual. We take full responsibility for any errors, so please report them to us so that they can be corrected in version 4.1.

Special thanks to Lippold Haken, Mark Smart, and Bryan Holloway for their contributions to Kyma 4.0 and the Capybara-33.

All of our encounters and experiences with Kyma users have contributed towards the improvements found in Kyma 4.0. We invite you to continue to interact with us and with each other, so that future versions will be even closer to your ideal system. In particular, we would like to acknowledge the help of our life-on-the-edge beta testers: Salvatore Martirano, Frank Tveor Nordensten, Jeff Stolet (and his students at the University of Oregon), Joel Chadabe, and Nathaniel Reichman. There are others, too numerous to name here, who have helped us get to the final release version of Kyma 4.0; to you we say, "Thanks!", and we hope to get your name in here for the next version!

Acknowledgments To Edition 2.05

Many thanks to Jon D'Errico for helping complete this edition of the manual. The first edition was written by Carla Scaletti; Kurt Hebel joined her in revising the second edition.

Acknowledgments to the Second Edition

Dick Robinson provided valuable, detailed feedback on the first edition of the manual. Thanks to Frank Nordensten, John Mantegna, Mark Smart, and Paul Christensen who tried out all of the new tutorials. Thanks to all of the participants in the Intensive Workshop in Sound Computation who also contributed many ideas for this second edition. And thanks to all of the Kyma users who have provided us with feedback, suggestions, and encouragement over the years.

Acknowledgments to the First Edition

Mark Lentczner provided a large number of insightful suggestions particularly with regard to the user interface. His enthusiasm, positive outlook and quick grasp of the system made him a pleasure to work with. Brian Belet and Antonio Barata read through the entire manual suggesting several clarifications and penciling in some amusing comments. Bill Walker suffered through the very first version of this manuscript and some of the earliest versions of the program. Kelly Fitz has been a consistent and intelligent user of the more advanced features of Kyma and has contributed to their development. Jon Sigurjonsson, Alec McLane, Insook Choi, Ben Cox, and Robin Bargar all took part in the first summer workshop on Kyma and were among the first composers to complete compositions and studies using the system.

Lippold Haken developed the Platypus, the first signal processor that enabled Kyma to do software synthesis in real time.

Introduction

What's in a Manual

Question: What do you call someone who reads manuals from cover-to-cover?

Answer: Power user or Guru.

If you set aside a few minutes each day — over breakfast, at the end of the workday, on a long trip, as a comforting bedtime story — and read the overview and work through the tutorials, you will become the envy of your colleagues and an idol to your friends. And as an extra added side-benefit, you will also be making more powerful and effective use of Kyma in your work.

You can think of this manual as three separate references:

Overview and Tutorials Everyone should read through the overview to see how all of the parts of the system relate to each other, and work through the tutorials to get an idea of what examples came with the system.

Quick Reference and Prototypes Reference Keep this manual next to your computer and opened to this section to quickly refresh your memory on what you can do in each of the editors.

Reference Manual and Appendices Most people will probably prefer to look things up in this section as needed, rather than reading straight through it.

Introduction and Overview

The overview is intended to give an overall sense of what Kyma is, what it contains, and how it all works together. Hidden within it is **Tutorial 0**, a tutorial on the basics of getting around in Kyma's graphic user interface. Reading the introduction and overview will give you an outline of the extent of Kyma, so it will give you an overall sense for what is available before you begin focusing on the details.

Tutorials: Kyma in 24 Hours!

The tutorials are intended as an orientation to Kyma and its user interface, and they give you a chance to explore Kyma with a little tutor at your elbow explaining everything and giving you hints along the way. The tutorials are intended as a starting point for your own creative work, so don't feel that you *have* to stick with them diligently to the end; ideally, they should be just enough to get you started on your *own* explorations through the environment. On the other hand, if you systematically complete one tutorial per day, you will be completely conversant in Kyma in just one month! (You can even take every seventh day off to rest, putting aside the manual to play in Kyma).

Examples

When you first start to use Kyma in your own work and anytime you begin a new project using Kyma, set aside some time to look through the example Sounds provided in the **Examples** folder; this has two benefits: the first is that you can use many of these synthesis and processing Sounds exactly as is, with minor tweaking, or as starting points for your own designs; the second benefit is that studying other people's Sounds is an excellent way to get new ideas and to learn new Kyma tricks that you can apply to your own sound designs.

Reference

The *Kyma Quick Reference* (starting on page 206), the *Prototypes Reference* (starting on page 218) and the main *Reference* (starting on page 417) are intended to serve, as their names imply, as reference books. In other words, if a question comes up as you are working, you should be able to quickly locate the answer on one of these references by using the *Index* or the *Table of Contents*.[§]

[§] By the way, if you can't find an answer to a simple question in the index, then please let us know about it so that we can fix it in the next edition of this manual (symsound@SymbolicSound.com).

These sections tend to be graphical summaries, tables, and extremely concise descriptions, which is exactly what you want out of a reference, though they don't necessarily make for the most entertaining straight-through reading. (However, if you do read through these sections, your reward is that you will be a POWER USER, and we may have a job for you on our support staff. So if you sit down and read through the entire book, let us know!)

Kyma Users' Group

When you do have questions about Kyma, don't overlook one of the best resources of all: the collective experience and intelligence of your fellow Kyma-ites. When you write a note to kyma@SymbolicSound.com, you can access the collective knowledge of not only the entire Symbolic Sound staff, but an international network of Kyma users who have been using Kyma to make music and sounds since 1990 (even earlier, if you include the developers).

Teaching Kyma to Others

If you are using Kyma in a one-semester beginning course in sound synthesis algorithms or electro-acoustic music, you could use the tutorials as part of the homework assignments (about two per week, in whatever ordering that makes sense with your lecture topics).

If it is an advanced course in which Kyma is to be used as the language, you could spend the first two to four weeks of class lecturing on the material in the overview and assign one tutorial per day for the students to complete during studio or lab time. After this intensive introduction to Kyma, the rest of the semester could be devoted to higher-level topics and Kyma could be used for in-class demonstrations, projects, labs, and assignments.

Peeling Back the Layers of Kyma

Kyma is a deep program that you can access in layers: at the highest level, immediately producing complex and interesting sounds using the examples found in the **Examples** folder, and by using the Tools; at the next level, by designing new sounds using the graphical Sound editor, wave editor, spectrum editor, and file organizer.[‡]

Whether and when you decide to proceed from one level to the next is entirely up to you. You can make a large variety of interesting and effective sounds simply by familiarizing yourself with the “factory presets” of Kyma — the processing and synthesis examples found in the **Examples** folder, and the Sounds at the *FTP* site contributed by your colleagues. You may find that you need never even use the Sound editor except to substitute your own source material.

Everyone who uses Kyma will use the top layer. For many, the top layer is all they will ever need. For others, the top layer is the introduction to the programming layer. There is no better way to learn about sound design than by studying the designs of others and tweaking them in small ways until you understand the effect of each parameter. Once you start designing new sounds, you can add them to the examples at the top layer, building and expanding your own personal sound library. You may even want to take a few of your less-proprietary examples and put them up at the Symbolic Sound *FTP* site for your Kyma colleagues to use (while saving the best sounds for yourself of course!)

Inevitably, though, you will find yourself tweaking a parameter here and there, even on the preset examples. This is the first step down the path to programming your own Sounds, a delightful addiction from which you may never recover (if you’re not already hopelessly addicted due to earlier experiences with modular synthesizers and software synthesis). This is the layer at which the real power of Kyma becomes apparent. Kyma is extremely open-ended and modular. In this respect it is both liberating and at times even slightly intimidating.

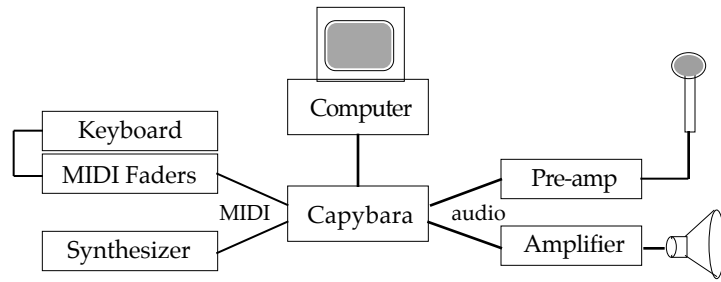
Kyma is definitely a program for consenting adults. Once you start getting into sound programming, there are so many possibilities, so many decisions, so many choices to make, that it requires you to set your own guidelines and your own constraints. In that respect it is like facing the empty page just before starting to compose music, write a paper, or work on any creative project.

Before You Get Started

As you read through the introduction and tutorials, there are suggestions that you try out certain examples. Many of these examples assume that you have a microphone connected to the left audio input channel of your Capybara and that you have some kind of MIDI input device or controller hooked up to the MIDI input on the back of the Capybara. Some examples require a MIDI synthesizer to be connected to the MIDI output.

[‡] There is an even lower level to Kyma, and that is the third-party development level. By creating sound libraries and placing them at the *FTP* site, you are already acting as a third party developer in the community of Kyma users. It is also possible to program new Tools and to write new assembly language modules for Kyma. To do this, you will need the developers’ kit. In order to get the developers’ kit you must apply to become a third party developer at either the Tool level (Smalltalk programming) or the DSP assembly language level. Of necessity, we have to limit the number of people in the developers program, because Smalltalk and DSP programming require intensive support and training. One of the requirements for being accepted is experience programming in Smalltalk (for Tools) or in 56002 assembly language (for new modules). If you have this kind of experience and are interested in developing for Kyma and the Capybara, send us electronic mail at info-kyma@SymbolicSound.com describing your experience and plans.

Go ahead and set this up now, so that you will be ready to try the examples as they come up.



Your studio may have audio and MIDI patchbays to make it more easily reconfigured. If you connect the Capybara up to the patchbays, you can make this configuration by moving patch cords at the patchbay, rather than moving cables at the back of your equipment. If your patchbay is computer controlled, use the software for the patchbay to make sure it is configured correctly.

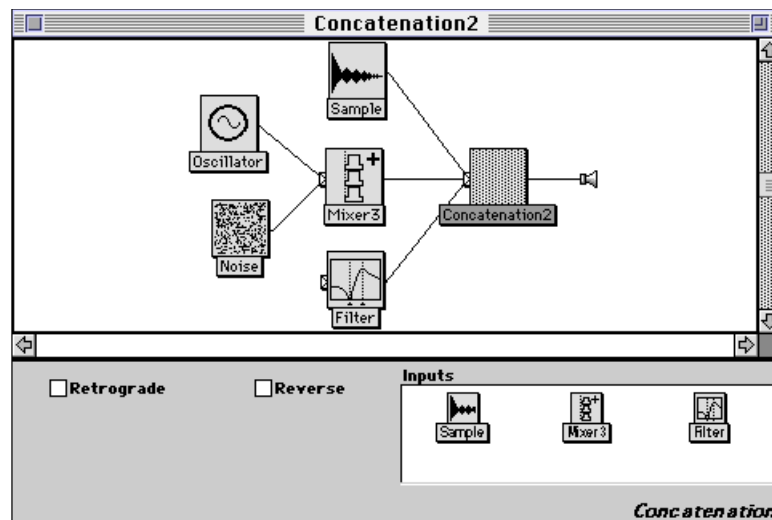
Kyma: a Language for Creating and Manipulating Sound

A language is a set of words and a set of rules for combining those words into expressions. Despite the fact that both the set of words and the set of rules are finite, the number of expressions you can generate by combining the words is infinite. In English, for example, the set of words can be enumerated in a book called a dictionary (which, while it is a very large book, is not infinitely large), and the set of rules for combining those words can be found in a smaller book on English grammar. In actual practice, we rarely need refer to these books, using an even smaller working set of words and rules in our heads, to produce a seemingly endless stream of written and spoken expression.

Kyma, too, is a language — a language for creating and manipulating sound. It provides a vocabulary (a set of basic modules or Sounds)



and a means for combining those elementary modules into an infinite number of arbitrarily complex Sounds



Languages vs. Devices

Of the many programs you use daily on your computer, some function more as “devices” and some function more as “languages”.

A software “device” is typically a complex, high-level program that does a specific and limited set of tasks for you (and if it’s a good program, it does these tasks quickly, easily, and repetitively, and probably throws in a flashy color interface for your aesthetic edification). It does *not* let you get inside the device, modify it, and use it to do some other set of tasks.

A language, on the other hand, is a set of elementary modules and some means for combining the modules. Unlike a device, a language doesn’t do anything by itself. It waits, only hinting at its mysterious promise and potential; it waits for *you*.[§]

A language doesn’t do anything by itself. But what you *say* in a language is your own.

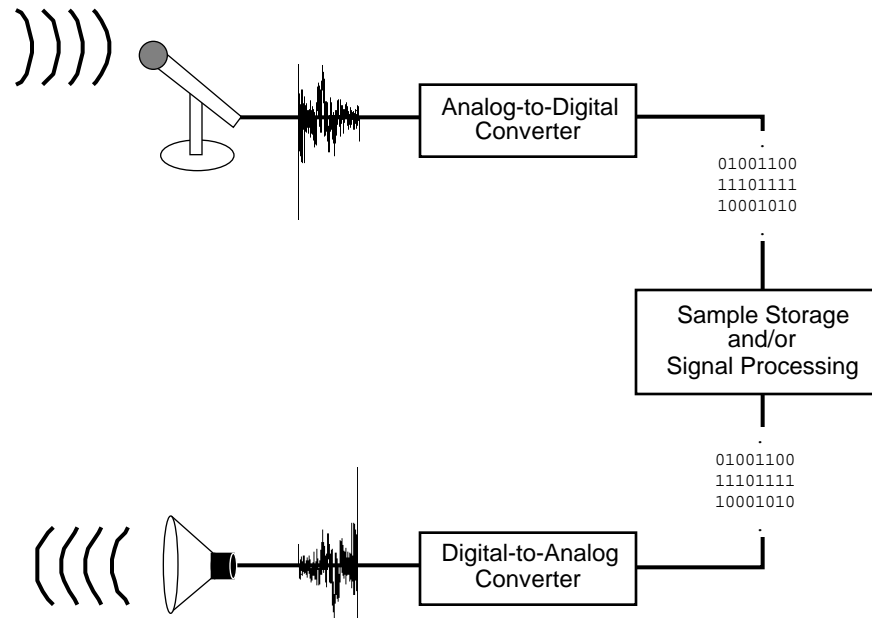
[§] One of the many myths surrounding technology is the idea that, using a computer, *anyone* can be a composer or sound designer or audio researcher, without having to invest any time or energy in their own education. If you buy into this myth, then Kyma is *not* for you. We assume that our users are serious about sound, that it is either their profession or their intense avocation, and that they are voracious auto-didacts.

First, Kyma asks you to invest some time in learning the language. Your reward will be a fluency for creating an infinite variety of new sounds that no one has ever heard before! Once you learn the basic vocabulary of Sounds and the few simple rules for combining them, you will achieve a kind of critical mass when your knowledge and facility in the language will start increasing at an exponential rate.

At some point, you will find yourself thinking in Kyma, dreaming in Kyma, designing all of your sounds in Kyma, wondering how you ever got along without it, wondering why your friends are agonizing over how to do things that you would find simple to whip up in Kyma.

Sound on the Computer

What is the domain of this language? What exactly are we synthesizing and manipulating in Kyma?



Digital audio can be thought of as a symmetric process of turning acoustic air pressure variations into a voltage signal, into a stream of numbers, and then reversing the process to get from the stream of numbers, to a voltage signal, and back into a changing air pressure.

Physical sound is a variation in air pressure. You can detect these changes in air pressure using a transducer like a microphone which has a diaphragm inside that moves back and forth in response to changes in air pressure and translates this variation in air pressure into a continuously varying voltage.

You can use an analog-to-digital converter to measure (or “sample”) the value of this continuously varying voltage at evenly spaced time intervals to produce a stream of numbers corresponding to the instantaneous amplitudes of that sound at those points in time.

As you convert the continuous voltage into a stream of discrete numbers, you can save them onto a hard disk or a CD.

Later, you can read those numbers off the CD in the same order, feed them to a digital-to-analog converter which filters or interpolates between the discrete values, turning them back into a continuously varying voltage.

You can feed this voltage to a speaker which translates the voltage changes into movements of a diaphragm which pushes the air around, thus recreating the air pressure variations.

This entire process is, by now, ubiquitous and familiar to everyone as the process of digital recording.

However, once you have converted the acoustic sound into a stream of numbers, you open up all kinds of possibilities for manipulating that stream of numbers on the computer.

You can do arithmetic on the numbers before sending them to the digital-to-analog converter to be turned back into sound again. This is what is meant by digital signal processing or digital effects *processing*.

From there, it is only a small leap of faith to just drop the entire top half of our diagram, and *generate* the stream of numbers ourselves. This is what is meant by software sound *synthesis*.

For that matter, we could take the stream of numbers from some other sources, say, the result of a scientific experiment or the position of someone's head in a virtual environment, and use *that* stream of numbers to generate sound or to control other parameters of the sound. This is the idea behind an emerging application of computer-generated sound called *data-driven sound*.

Kyma seeks to provide a single, uniform framework for dealing with *all* of these aspects of sound on the computer: sampling, processing, and synthesis, together with composition and performance.

Kyma's Sound Object

Kyma is based on elements called Sound objects. You see evidence of these Sounds everywhere in the Kyma user interface in the form of graphic icons.



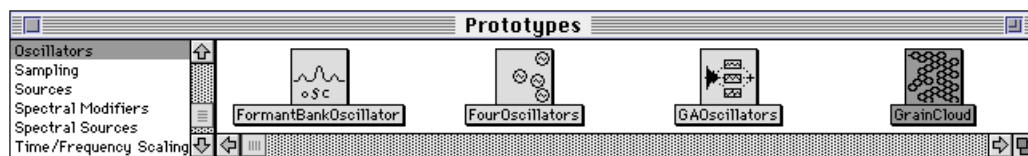
A Sound represents a stream of numbers like the streams of instantaneous amplitude values we talked about in the previous section. That stream of numbers could come from a digital recording read from the disk, or it might be purely synthesized, or it could be the result of modifying another stream of numbers.

Another way to think of a Sound is that it represents an algorithm or a program. The algorithm might describe a procedure for synthesizing the stream of numbers, or it might describe the process for reading the stream off of the disk, or it might describe some arithmetic to perform on an input stream.

Yet another way to think of a Sound is as a virtual module, analogous to the modules like oscillators, delay lines, or filters that you might find in a synthesizer or effects processor, but implemented entirely in software, rather than hardware.

The Words

Take a quick look through the *Prototypes Reference* in this manual beginning on page 218. This is the "dictionary" of "words" (that is, the Sounds) in the Kyma language. They are arranged in alphabetical order and list the name of the Sound followed by a definition. All of these Sounds are also found in the palette called Prototypes that appears across the top of the screen in Kyma:

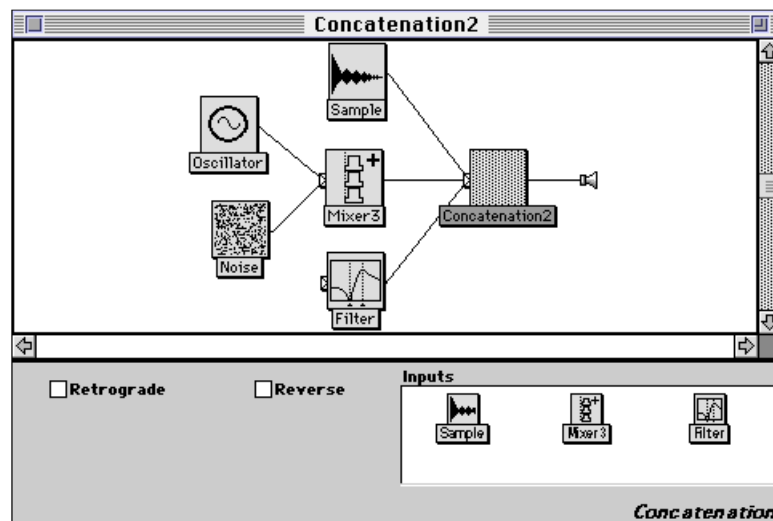


The Sounds are categorized in order to make them easier to find. Select one of the category names from the list on the left, and all the Sounds in that category appear as icons on the right. To get at the definition of a Sound on-line, select the Sound icon in the prototypes window, and choose **Describe sound** from the **Info** menu. To search for a Sound by name, use **Find prototype** from the **Action** menu, and enter part of the name of the Sound when prompted.

There is nothing special about the Sounds in the prototypes window, by the way. The prototypes window is just a collection of Sounds: one example of each type of Sound in Kyma. It is convenient to have one of each Sound type available in a palette like this, but wherever the manual talks about dragging a Sound from the prototypes, you should know that any other Sound (from the Sound file window or even from another open Sound editor) would serve as well.

Grammar

Kyma “sentences” are constructed in the Sound editor window:

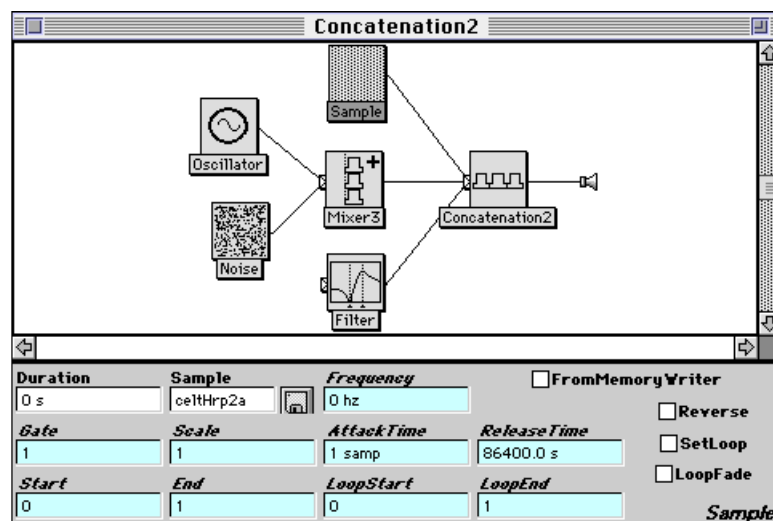


The basic rule for creating complex Sounds out of simpler ones is contained in the definition for the Kyma Sound Object.

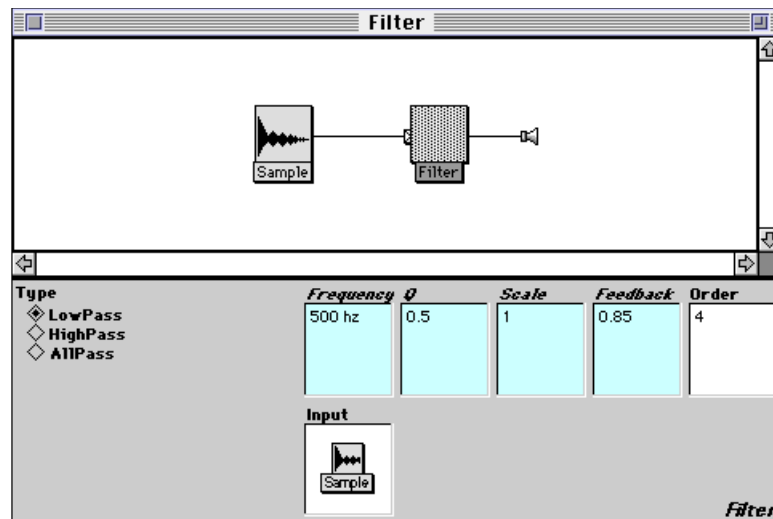
A Sound is:

- a source of sound, or
- a modifier or processor of sound, or
- a combiner of several sounds

A Sound with no inputs is a source of sound. *Noise*, *Oscillator*, *GenericSource*, *FormantBankOscillator*, *SumOfSines*, *DiskPlayer*, and *Sample* are all examples of synthetic or sampled sound sources in Kyma.

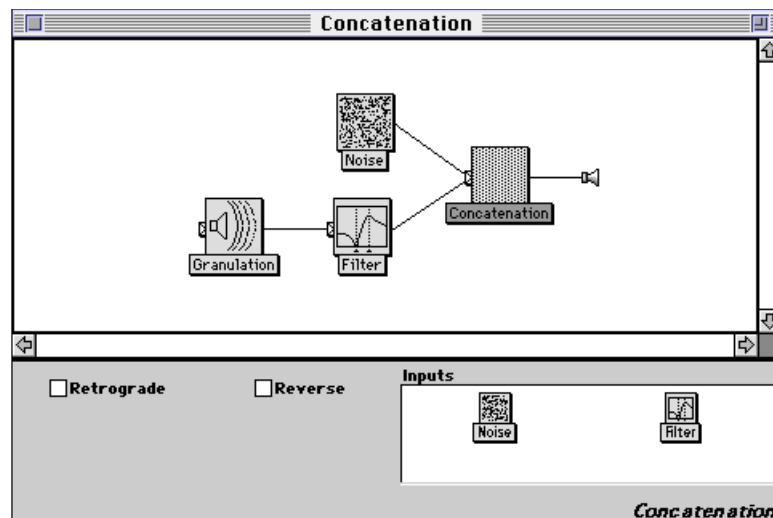


A Sound with a single input is a modifier or processor of the sound that comes from its input. In the Sound editor, the input is always shown to the left of the Sound that modifies it. You can think of the signal as flowing from the left to the right.



This is by far the largest category of Sounds in Kyma, and includes Sounds like *Filter*, *Vocoder*, *Delay-WithFeedback*, *DiskRecorder*, *TimeFrequencyScaler*, and others.

A Sound with several inputs is a combiner. Two special examples of combiners are the *Mixer* (which causes all of its **Inputs** to occur simultaneously) and the *Concatenation* (which causes its **Inputs** to occur one after another in a sequence).



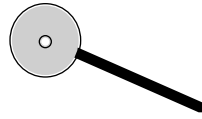
A combiner can also have more complex specifications for how its inputs come together; in a *Script*, for example, you write a script specifying when and how all the inputs are to occur with respect to one another.

If the Sound objects are the words, then, in some sense, the Kyma language contains only *nouns*. The sound sources are like objects (oscillator, noise generator), the sound modifiers are also objects (filter, delay line), and even the combiners function as objects (mixer, concatenation).

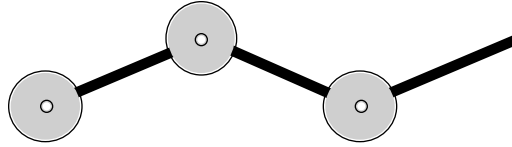
A Sound is a Sound is a Sound...

So why do we insist on calling everything a Sound (even things like filters and delay lines)? We do it in order to underscore the idea that Sounds are completely interchangeable and infinitely chainable.

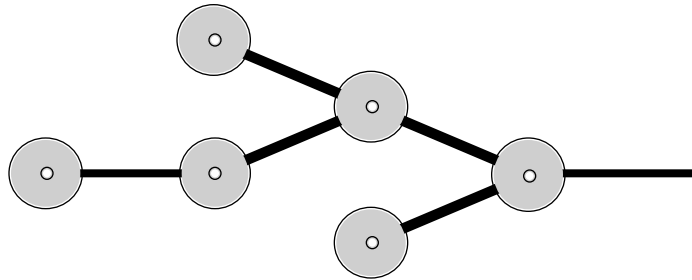
A Sound, no matter how complex, can always serve as the input to another Sound. Think of Sounds as something like highly abstracted audio Tinker Toys. Once you have constructed one hub and spoke



you can plug that construction into another hub, and that one, in turn, into yet another, *ad infinitum*:



At any point you could also fan out and connect several spokes



No matter how complex a sub-construction of Tinker Toys you have made, you can always plug it in at the end of a spoke, just the same as if it were a single hub.

The same is true for Sounds. Anywhere you can use a sound source, you could also use a modifier of a source or a combiner of a source; Sounds are uniform and interchangeable. Even a Sound that contains a complex script that functions as a score or reads a standard MIDI file can be used as the input to another chain of modifiers and combiners. This is one of the things that makes Kyma different from most other music software: a “score” or an algorithm for generating events is part of the Sound object and is not necessarily at the “top level” of the hierarchy; there can be several “scores” distributed throughout a complex signal flow diagram. (And by the way, the entire signal flow diagram is itself a Sound).

We call all of these objects Sounds in order to emphasize the uniformity and interchangeability of Sound objects. But having said all that, we can now relax a bit and sometimes use the word *module* to describe Sounds.

Sound Parameters

All Sounds, whether they are sources, modifiers, or combiners, also have *parameters* or settings. These have nothing to do with the signal flow, but are local adjustments made to that Sound alone and affecting the way that Sound does its generating or processing or combining of other Sounds.

Sound parameters can be:

- Constants (for example, numbers, strings, or sample names)
- Functions of time (for example, ramp functions or triggers from a metronome)
- Hot controls (supplied externally from a MIDI controller or internally from Kyma)
- Sounds (for example, using an *Oscillator* to control the parameter of another Sound)
- Arithmetic expressions involving any combination of the above

Structure

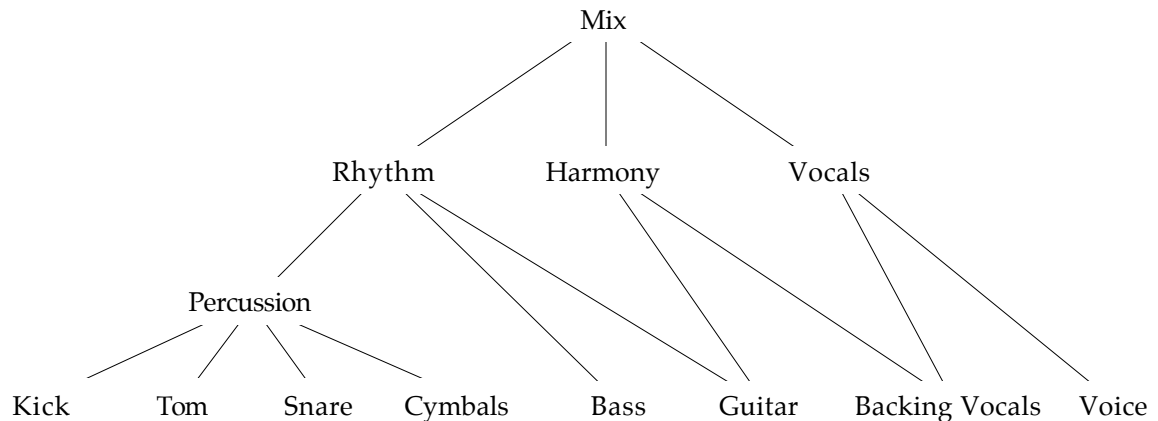
To restate the definition of Sound in a slightly different way, a Sound is

a Sound, or

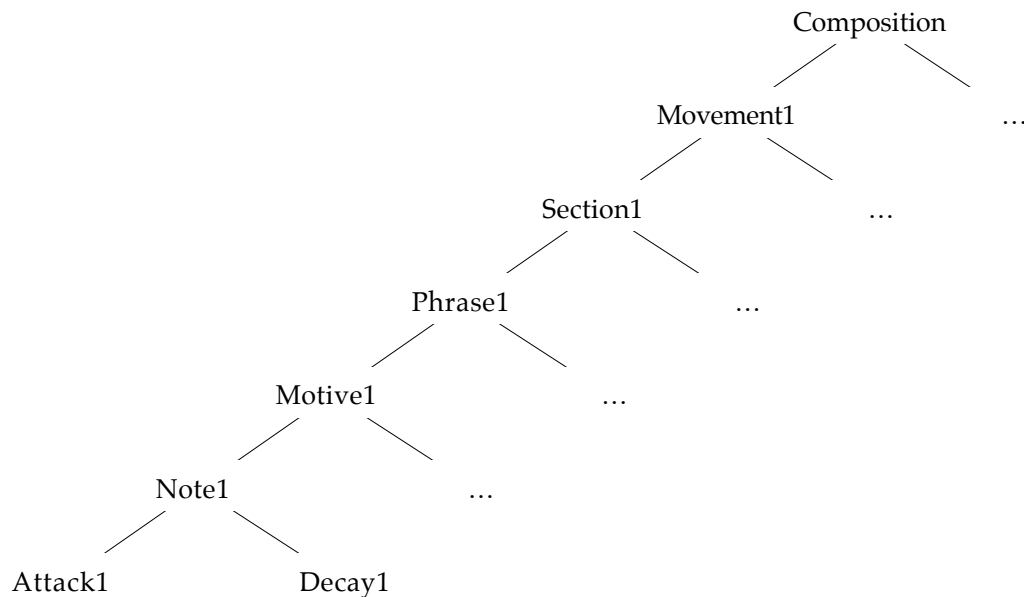
a collection of Sounds that functions as a single entity

This is a familiar idea in the multitrack recording studio where you might have one or more microphones on each performer, each recorded on a separate track.

Sometimes you want to treat the tracks independently, giving each its own processing or position in the stereo field. At other points, you might want to take a “submix” (say, all of the backing vocals) and treat it as a single entity, routing it through its own effects chain and EQ. And other times, you might want to treat the entire mix as one object, putting it through an effects processor to simulate room ambiance.



To use traditional musical terminology, you could think of a composition as a sequence of notes, but in reality, the notes are grouped into meaningful motives, the motives into phrases, *etc.* Sometimes you may want to treat a note as an independent entity, sometimes you treat a phrase as one entity (subjecting the entire phrase to transposition or augmentation), and sometimes you want to go even deeper than the note level, modifying the attack and decay time or the timbre of an individual note.



Whether you are composing or designing sound, you are constantly popping up and down to these different organizational levels, without even having to be consciously aware that you are organizing sound on several different time scales and submixes almost simultaneously.

Kyma's Sound object is meant to facilitate this kind of work — to make time-scale-switching and sub-grouping natural and visually apparent.

In a Kyma Sound structure you might have a mix of

- a noise generator feeding through some highly tuned filters, the resonance of which you are controlling from a MIDI keyboard,

- some oscillators resynthesizing vocal sounds from an analysis and repeating an algorithmically generated harmonic pattern in an alternate tuning (the algorithm for generating the pattern being part of the *MIDI Voice* module),

- and a sequence of different filters, processing, and modulation on a *GenericSource*, so that you can speak or sing into the microphone and the processing will be different at different times.

Synthesis/Processing Techniques

Despite the fact that Kyma provides modules covering a wide range of synthesis and processing techniques, that in itself is not half as exciting as the fact that you can combine these algorithms in entirely *new* ways in order to create your *own* synthesis and processing algorithms. In Kyma you will be using a direct manipulation iconic interface to do, almost as a matter of routine, what others labor over for months: come up with new, never-before-heard synthesis and processing algorithms!

Synthesis algorithms in Kyma range from sample and disk recording/playback to oscillators and envelope generators, to nonlinear distortion like waveshaping, ring modulation and frequency modulation, to wavetable synthesis, to full-blown additive synthesis, to noise generators and a variety of filters for subtractive synthesis, cross synthesis using either RE filters or the vocoder, resynthesis and spectral warping using the *SumOfSines* or the GA oscillators, plus combinations of any of the above. Processing, while difficult to truly separate from synthesis, includes delay, modulation, reverberation elements, distortion, filtering, compression/expansion, vocoding, manipulations to the spectrum (in the spectrum editor), and whatever brilliantly twisted new combinations of all of the above that you can dream up!

Current Implementation

Kyma is a language, and there have been several implementations of that language in software and in software-augmented-by-hardware-accelerators.

In the current implementation, the graphic interface, structural manipulations, and reading from/writing to disk are handled by the host computer — at this point, either a Macintosh or a Windows PC. Sound synthesis and processing takes place on the Capybara, a multiprocessor sound computation engine designed and built by Symbolic Sound. The Capybara is a general-purpose computing engine, so in reality “the software *is* the synthesizer”. *You* program the Capybara to behave as a synthesizer, sampler, effects processor, or combination of all, but the programming is all done graphically, by connecting modules to each other on the screen of your Macintosh or PC.

Brief History

The first version of Kyma was a software-only version designed and written by Carla Scaletti in the fall of 1986. It was written in Apple Smalltalk and ran on a Macintosh 512 K. In 1987, she extended Kyma to make use of the CERL Sound Group's Platypus (a discrete-component DSP developed by Lippold Haken and Kurt Hebel in 1983) for real-time sound synthesis.

The real time version of Kyma was shown at the International Computer Music Conference in the summer of 1987, and Kyma (along with the Motorola DSP56000, granular synthesis, and physical modeling) was identified by Bob Moog as one of the technologies-to-watch in an article he wrote on that conference for *Keyboard* magazine (December 1987 issue).

By 1990, the sound-generation and processing part of Kyma had been ported to the multiprocessor Capybara, then to the Capybara-33 in 1995, and to the Capybara-66 in 1996.

In 1992, the graphic interface was ported to Windows computers. In 1996, Kyma and the Capybara hardware was extended to include the PCI-bus Macintoshes and PCs. Similarly, PC-Card (also called PCMCIA card) support was added in 1997.

The software has been rewritten several times in order to increase the speed and add new features, tools and algorithms. In 1995, version 4.0 was twice the speed of the previous version and provided hot parameters for the first time. Version 4.1 in 1996 added MIDI scripts, GA synthesis, and RE synthesis. In 1997, Version 4.5 added the spectrum editor, the file organizer, the tools, the vocoder, and several new sound synthesis/processing algorithms.

The point of all this is to give you an idea of the rate at which Kyma is evolving, so you can see that you have invested in something that will continue to evolve and improve with time.

Kyma Evolves

Why and how does Kyma evolve over time? In our minds, we have an ideal Kyma, and we try to realize this ideal with the technology available to us at the time. In the process of implementing the ideas and using the language, we learn more about the ideas and we refine our ideal.

By putting Kyma into the hands of others and listening to what they have to say about it, we further refine and expand upon the ideal, and this feeds back into the next implementation. So, it is important for you, as a Kyma user, to interact with us; we rely on your feedback in this continual process of implementation and refinement.

How is Kyma Being Used?

As a Kyma user, you are part of an international community of progressive musicians, sound designers, and researchers. You can read about some of their activities in the *Eighth Nerve*, the on-line Kyma newsletter that you can access from our web site (<http://www.SymbolicSound.com>). Incidentally, if you haven't sent us your biography yet, please avail yourself of this opportunity to have a presence on the world wide web (or a link from our site to your own).

Music

Kyma is being used in all kinds of music, from live performances to the sound tracks for music videos, to tape music and musique concrete, and including installations and performance art. See the web site for up to date news on specific artists, upcoming events, and references to books, CDs, and videos.

Teaching

Kyma's graphical representation of the signal flow and its highly interactive interface lend themselves to classroom demonstrations and the real time exploration of *what-if* scenarios. So it is not surprising that Kyma is used extensively in teaching about sound in universities all over the world and in courses ranging from psychoacoustics to music composition.

Sound Design

Kyma was created as a language for sound design, and it is being used by sound designers for film, television, advertising, radio, and music as part of the international entertainment industry including Hollywood, New York, St. Louis, London, Toronto, Tokyo, and elsewhere.

Research

Electrical engineers, computer scientists, and psychoacoustics and speech researchers are using Kyma for algorithm development, for generating stimuli for perception experiments, and for other research projects.

Data-driven Sound

Patterns in experimental data or in data generated by models and simulations can often be made immediately apparent when you map that data into sound. Engineers and researchers are exploring the idea of using sound to help uncover patterns in data that might otherwise go unnoticed if represented visually.

Related to this idea of data-driven sound is the idea of complete virtual environments, both in research and in the entertainment industry. Game developers and virtual environment creators are beginning to realize that simply triggering samples is not enough. Immersive environments require sound-generating models that are more interactive and parameterized so that they can respond to the actions of the viewer/player.

Overview of the Interface

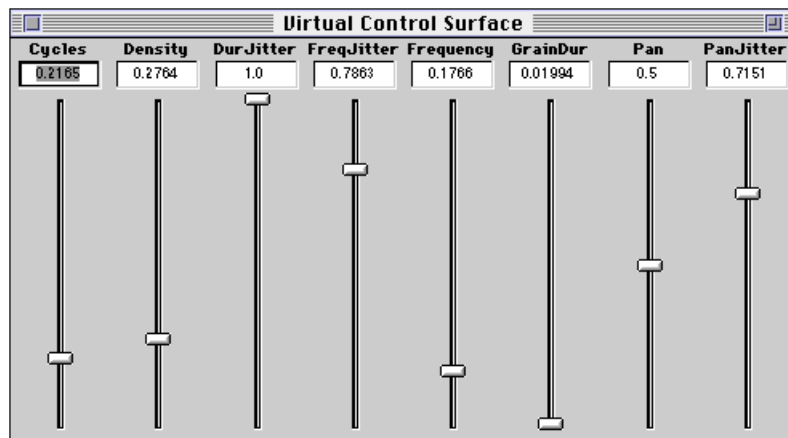
Kyma Sounds are abstract objects in the memory of the computer, but you interact with these objects in a fairly concrete way by manipulating graphics on the computer screen using the computer keyboard and mouse. The graphic interface of Kyma has several components:

Production and Performance

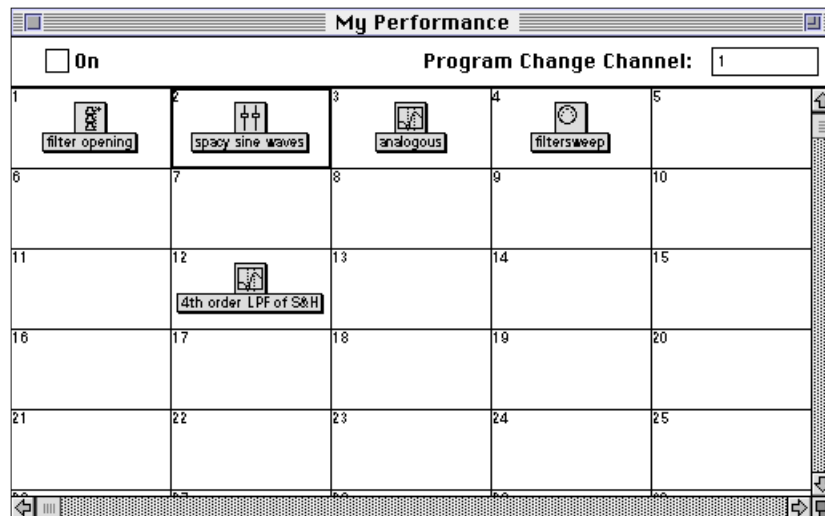
During live performance or when working under a production deadline, you can draw upon the previously-designed library of Sounds — including Sounds you have designed yourself, Sounds provided by Symbolic Sound, and Sounds contributed to the *FTP* site by your fellow users.

Editors and windows in support of performance and production include:

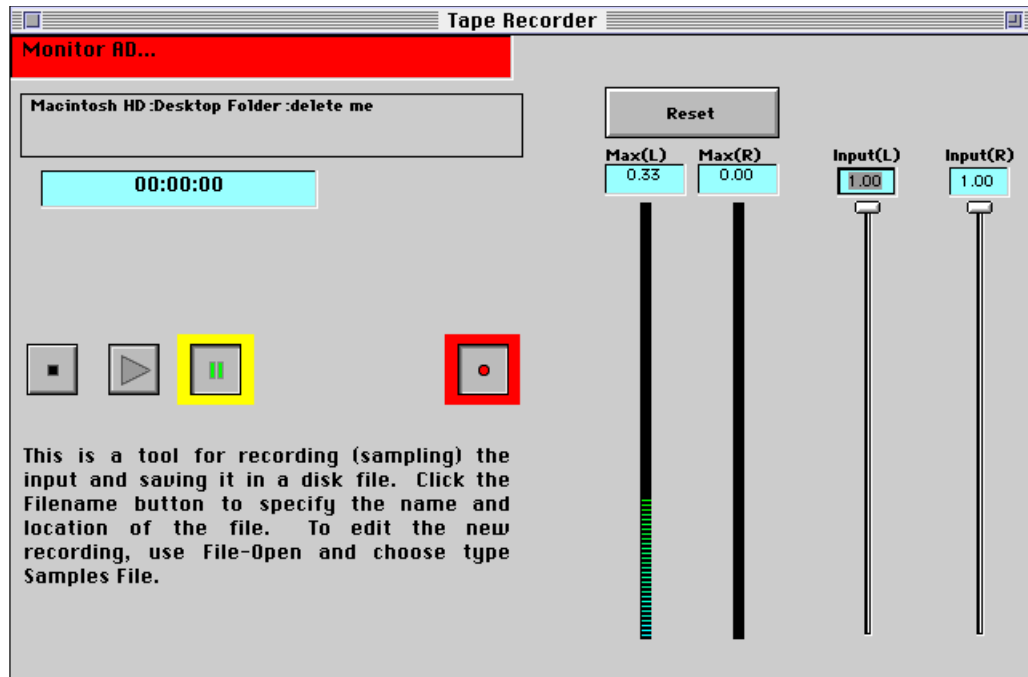
The virtual control surface that pops up automatically whenever you have specified hot parameters in one of your Sounds (we will talk more about hot parameters later in this introduction). This serves two purposes: one is to let you know the names and values of any hot parameters in your Sound, and the other is to provide graphical controls like faders and buttons so you can adjust the values of the hot parameters while the Sound is playing. Alternatively, you can control these hot parameters from a MIDI sequencer or using MIDI controllers.



The compiled Sound grid provides a way to quickly download precompiled Sounds in response to a MIDI program change message, a mouse click, or by tabbing into one of the squares of the grid. This can be useful in live performances or other presentations of Sounds that you have designed earlier and precompiled for the performance.



The items in the **Tools** menu provide support for sound design and development. A Tool could be described as a cross between a virtual-device front panel, a plug-in, and a “wizard”, the step-by-step guides provided in Microsoft programs. Any item in the **Tools** folder shows up as a choice in the **Tools** menu, so the tools can be updated by visiting the *FTP* site and downloading the latest set.

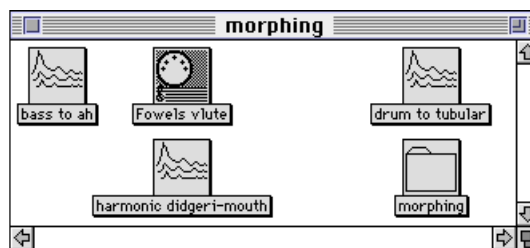


The Tape Recorder tool, for example, provides a quick means for sampling the Capybara input and storing the results in a sample file on the disk of your host processor.

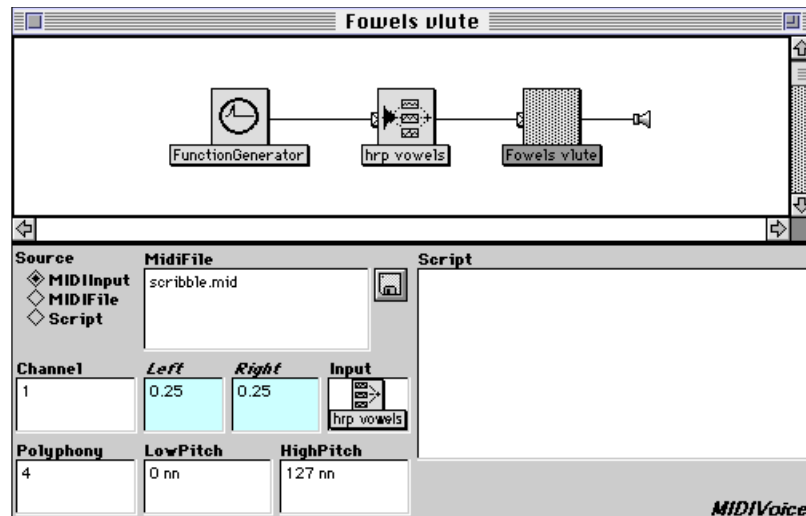
Preproduction

In pre-production exploratory phases, or whenever you are between projects, you can be developing your own Kyma Sounds, organizing them into Sound files, setting up precompiled Sound grids, trying out new synthesis and processing ideas, creating the sounds that no one has heard before. The purpose of Kyma is to provide tools for creative sound design, so quite a few of the tools and editors support of this activity:

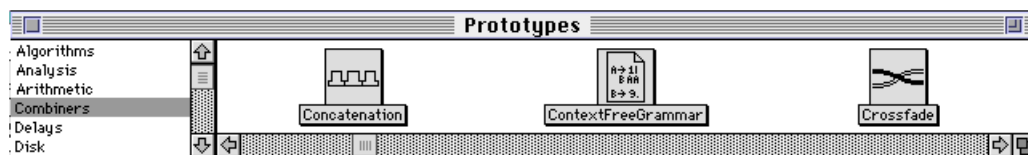
The Sound file window serves as a workspace when you are using Kyma, and provides a way to group Sounds from the same project or Sounds of a similar nature in a single disk file.



The Sound editor is where you design the signal flow and edit the parameters of individual modules.

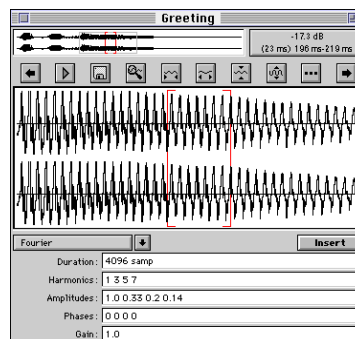


The system prototypes (“the prototype strip”) serves as a convenient source of template Sounds when you are constructing a complex signal flow diagram in the Sound editor. It contains one example of each type of Sound that comes with the system.

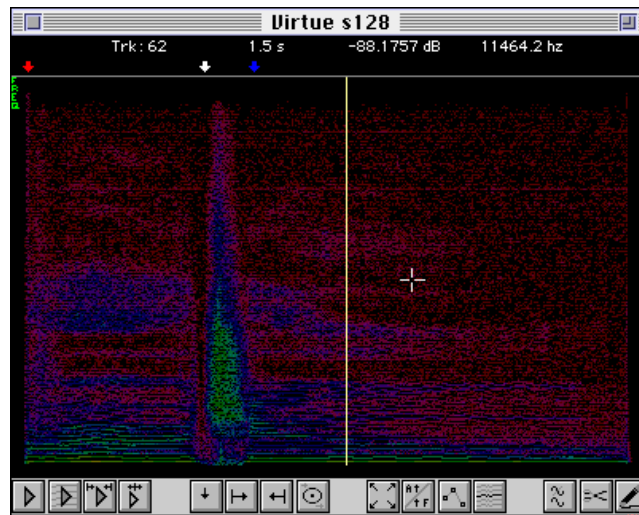


In addition to the Sound editor, there are four specialized editors, the file organizer, and two status windows:

The sample editor is a standard waveform editor, offering cut/copy/paste editing for digital recordings or “samples” as well as templates for generating your own wavetables algorithmically.



The spectrum editor is a two-dimensional editor for the amplitude and frequency envelopes of each sinusoidal partial of a sound that has been analyzed using Kyma's Spectral Analysis tool or the shareware Lemur program.[‡]



The global map editor is a text editor for specifying the mapping between the names of hot parameters and MIDI controller numbers.

```

"TimeCode is the time in seconds received from MIDI Time Code."
!TimeCode is: 'MIDI Time Code.'

"Some named controls used in Kyma examples and shared between this map and the default map."
!Attack is: 'MIDIController11.
!Decay is: 'MIDIController12.
!Release is: 'MIDIController13.

!Index is: 'MIDIController14.
!Start is: 'MIDIController15.
!Length is: 'MIDIController16.

!Frequency is: 'MIDIController18.
!Bandwidth is: 'MIDIController19.

!Feedback is: 'MIDIController21.
!Delay is: 'MIDIController22.
!Pan is: 'MIDIController23.

!Morph is: 'MIDIController25.
!Rate is: 'MIDIController26.

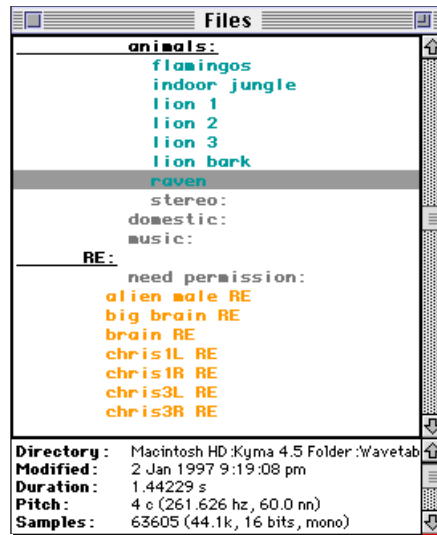
!Fader1 is: 'MIDIController11.
!Fader2 is: 'MIDIController12.
!Fader3 is: 'MIDIController13.
!Fader4 is: 'MIDIController14.
!Fader5 is: 'MIDIController15.
!Fader6 is: 'MIDIController16.
!Fader7 is: 'MIDIController17.

```

The text editor is a simple editor for entering or modifying text. For example, you could use this to create text files to be read and interpreted by Kyma Sounds or to test out a Smalltalk expression by typing the expression, selecting it, and choosing **Evaluate** from the **Edit** menu.

[‡] Available from the CERL Sound Group web site at <http://datura.cerl.uiuc.edu>.

The file organizer provides a color-coded list of all files that Kyma can work with, including samples, spectra, GA analyses, RE analyses, and MIDI files. You can select a file name and use **Ctrl+Space Bar** to hear the file, press **Enter** to open an editor on the file, or drag the file name into a Sound file window, Sound editor, or parameter field.



There are two status windows: one for monitoring or changing the status of the Capybara, and the other for monitoring the MIDI input and for monitoring and recycling memory on your computer.

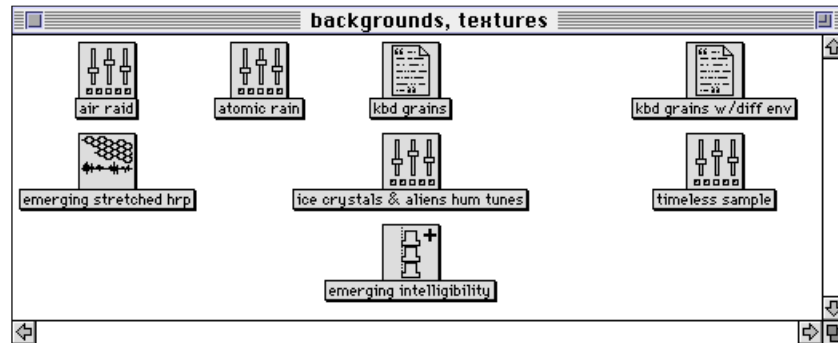


A particularly important Sound for design and development is the *Annotation*. This is actually a Sound rather than an editor (found in the system prototypes under **Variables and Annotations**). The purpose of an *Annotation* is to provide a brief explanation or reminder of what its input does. If you take some time to write a brief explanation of your Sound as you develop it, you will be able to remind yourself, six months in the future, just exactly what it was that you were doing.

Sound, Sound File, and Sound Editor

Now let's go back and revisit the concepts of Sound, signal flow, and parameter setting, this time in the context of how to actually manipulate these things using the graphic interface. (If you are near your computer, you might want to try out some of these things as we go along.)

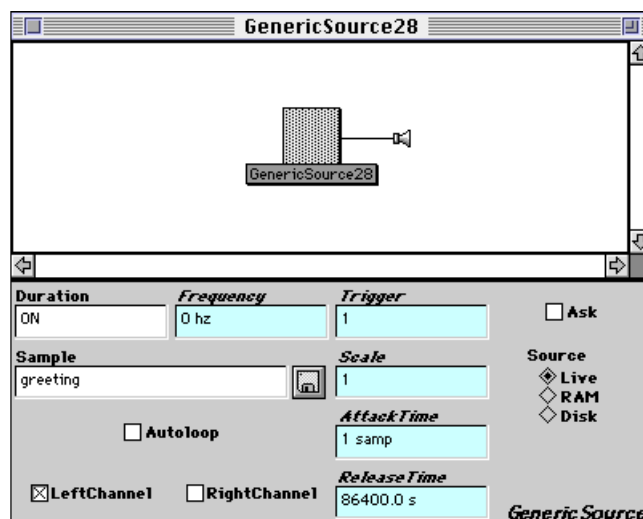
This is an example of a Sound file window, your workspace or “desk” while inside Kyma. Each icon in the workspace represents a Sound.



One way to think of a Sound is that it is a program for the Capybara. To run the program, select the icon, go to the **Action** menu, and choose **Compile, load, start**. True to its name, this menu selection compiles the program for the Capybara, loads it into the Capybara, and then runs the program. Technically, this is what occurs each time you select **Compile, load, start** from the **Action** menu.

The result, though, is that you hear the sound, so from now on, we will refer to this as *playing* the Sound and give you a shortcut method for doing it: holding down the **Control** or **Command** key[§] and pressing the **Space Bar**. Another essential shortcut, perhaps even more urgently required than the shortcut to play a Sound, is the shortcut for *stopping* the sound. Memorize this one so you can instinctively hit it by reflex alone, even if your forebrain activity is being jammed by a full amplitude 8 khz sine wave; remember **Ctrl+K** for “Kill that sound!”.

To edit a Sound, double-click on its icon. This opens a Sound editor and shows the signal flow for this Sound. In this example, there isn't much signal flow to see: the *GenericSource* is feeding into the D/A output. Below the signal flow path, you can see the parameter settings for the Sound.



[§] On the Macintosh, you can use either the **Control** or **Command** key; however, Windows computers do not have a **Command** key, so you must use the **Control** key. To avoid sounding repetitious, we will abbreviate “hold down the **Control** or **Command** key while pressing the **R** key” as “use **Ctrl+R**”.

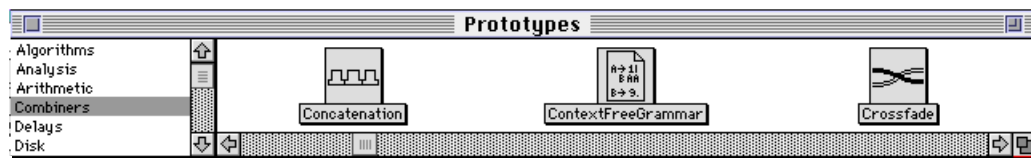
For example, in a *GenericSource*, you can choose whether the source of sound is the **Live** input from the A/D or digital input, whether it is to be read from the **RAM** of the Capybara, or whether it is to be read from the host computer's **Disk**. If you put a check in the **Ask** box, Kyma will ask for the source each time you play this Sound, making it ideal for developing processing algorithms using a sample or disk recording typical of what you want to process and then switching to the live inputs once you are happy with the sound of the sampled input.

Notice that, in addition to the other parameters, you have to supply this Sound with a duration. In this example, the duration is set to **ON**, so it will run forever,[‡] or until you kill it (**Ctrl+K**) or load a different Sound.

As we said before, to play a Sound in the Sound editor (actually to play any Sound anywhere in Kyma), select it and press **Ctrl+Space Bar**.

The Prototype Strip

The prototype strip is the window across the top of the screen. It contains an example of each of the different kinds of Sound modules in Kyma.



Since each Sound in the prototype strip is an example of how to use that particular kind of Sound, you can study a Sound in the prototype strip to get some idea of how to use it.

A Sound in the prototype strip can't be edited (unless you drag a copy of it into a Sound file window and edit that copy), but you *can* listen to it directly in the prototype strip. Press **Ctrl+Space Bar** to play the selected Sound in the prototype strip.

Some of the Sounds in the prototype strip may not make sound when you try them out. This could be because:

- the Sound is triggered by MIDI note events (and will make sound if you play the MIDI keyboard)
- the Sound is subaudio and intended to be used as an envelope (view it with **Full waveform** from the **Info** menu)
- the Sound outputs a spectral control envelope (use it in the **Spectrum** field of an *OscillatorBank*)
- the Sound outputs MIDI (connect the Capybara MIDI output to a synthesizer to hear it)

You can use the Sounds in the prototype strip, along with the examples provided with Kyma, as a source of modules to insert or add to the signal flow diagram in the Sound editor.

Finding Sounds in the Prototype Strip

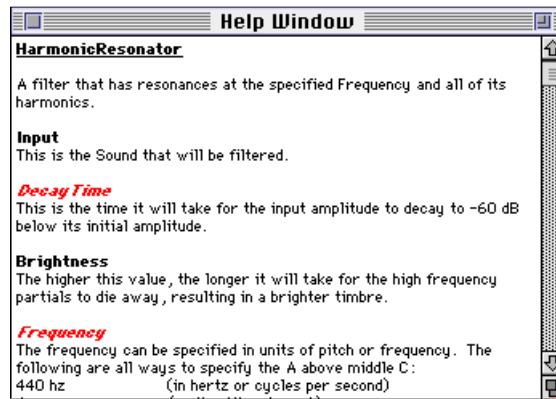
To quickly locate a Sound in the prototype strip, scroll down through the list of categories on the left until you find the category containing the Sound (the categories are in alphabetical order), click on it to select it, and then use the scroll bar across the bottom to browse through the Sound icons (also arranged in alphabetical order by name) in that category until you find the one you are looking for.

Alternatively, you can go up to the **Action** menu and choose **Find prototype...** (or simply use **Ctrl+B** for "befuddled") and type in part of the name of the Sound you are looking for, and then click **OK** or press **Enter**. This will come up with a list of Sounds whose names contain the partial name you entered. To select one of these Sounds, hold down the mouse in the downward arrow button, move the mouse down to the name of the Sound you want, and release the mouse button. Then click **OK** or press **Enter**. This will take you immediately to the right place in the prototypes with the desired icon selected.

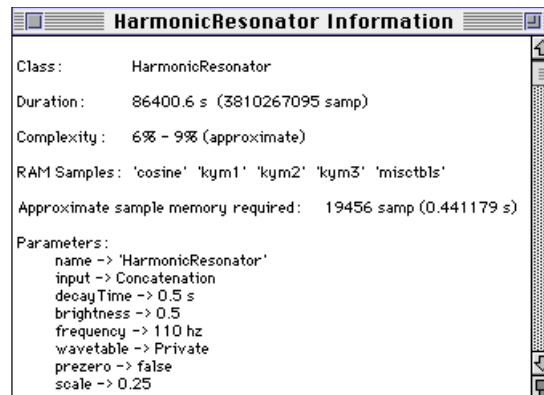
[‡] Well, actually, **ON** is about two years, but in computer years that's *virtually* forever.

Help and Information

For on-line help on a class of Sound modules, select a module of that type, and choose **Describe sound** from the **Info** menu. This gives you a description of how a module of this type behaves and provides a description of each of the module's parameters. These are the characteristics shared by *all* Sounds of this type.



Get info (Ctrl+I) from the **Info** menu provides information on one particular instances of that type of module. It lists the values assigned to each of the Sound's parameters, the names of any inputs, the memory and computational resources it requires, and its duration. These are the parameters of *one* particular Sound.



Both **Describe sound** and **Get info** work on any Sound, not just on the Sounds in the prototypes palette.

Editing the Signal Flow Diagram

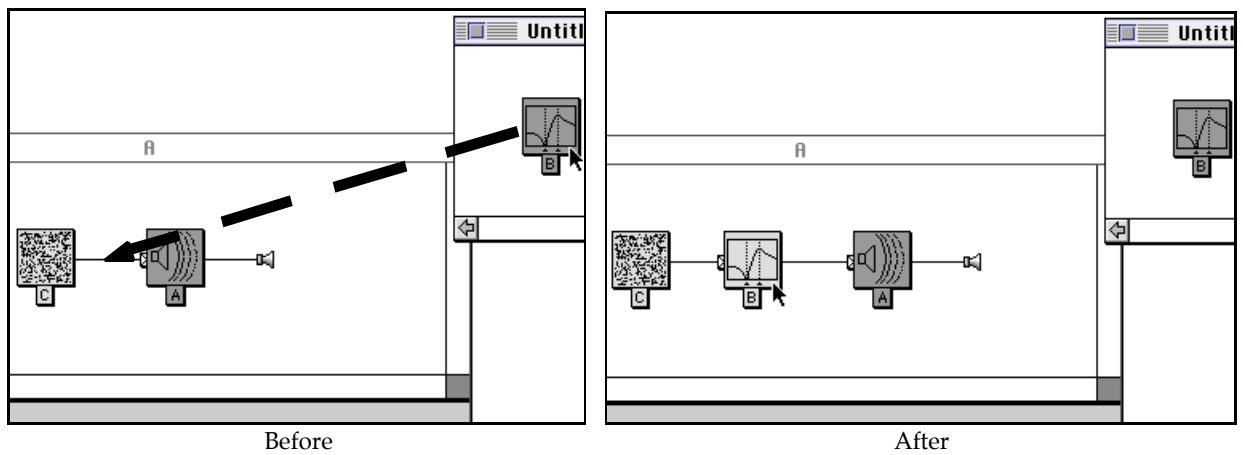
You use the signal flow diagram in the upper half of the Sound editor to connect the different synthesis and processing modules in your sound design.

The signal flow diagram lets you:

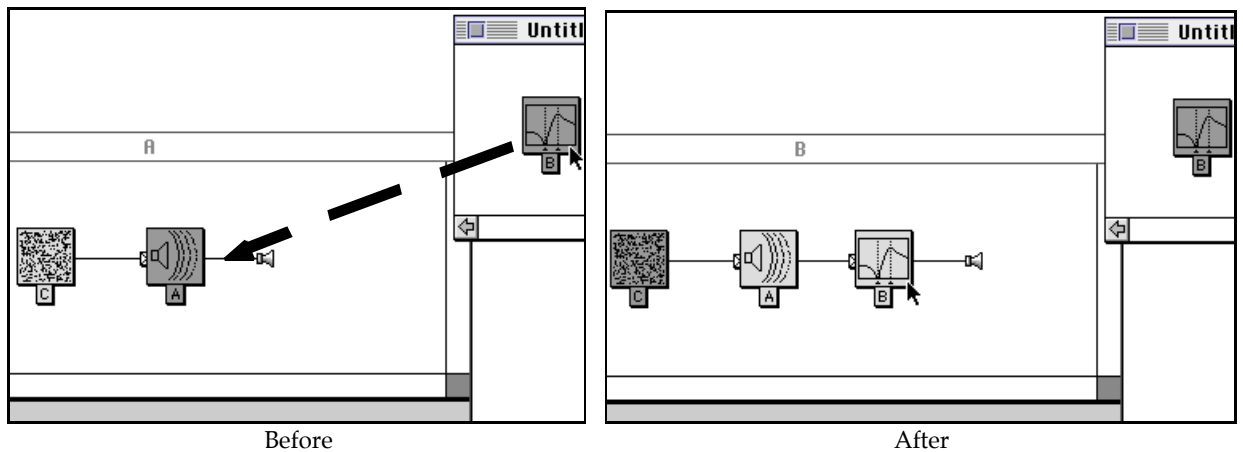
- insert a module between two other modules
- replace a module with a different module
- delete a module from the middle of a chain of modules
- add or remove modules to Sounds that can use multiple inputs
- listen to the audio signal at any point in the signal flow
- change the name of a module

Insert

To insert a new module **B** between two other modules **C** and **A**, drag **B** into the Sound editor, and drop it onto the line between the two modules **C** and **A**

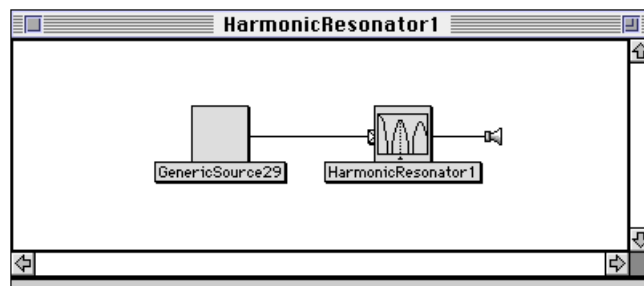


or between the module *A* and the speaker icon.



For example, say you wanted to feed the *GenericSource* through the *HarmonicResonator* filter before sending it to the output. First, find the *HarmonicResonator* filter in the prototypes. Select the **Filters** category in the list on the left and then scroll the icons until *HarmonicResonator* becomes visible (both the categories and the Sounds are arranged in alphabetical order).

Drag the *HarmonicResonator* from the prototypes into the Sound editor and drop it on the line between the *GenericSource* and the output speaker icon (touch the cursor arrow tip to the line connecting the *GenericSource* and the speaker icon).



After the insertion, you should see the *HarmonicResonator* with a little tab attached to its left side. Whenever you see a tab, it indicates that there are more Sounds to the left but that they may not be showing. Click on the tab to hide the input. Click it again to show the hidden input. This ability to show or hide inputs becomes especially useful when you have a complex signal flow diagram and would like to concentrate on one section of it at a time.

While constructing new Sounds, you can typically drag all the modules you need from the prototypes palette into the Sound editor since the prototypes include one of each kind of Sound available in the sys-

tem. However, you can drag a Sound module from *anywhere* in the user interface (the Sound file window, another open Sound editor, a MIDI grid, *etc.*) and place it into the signal flow path.

Probing, Prisms, and History

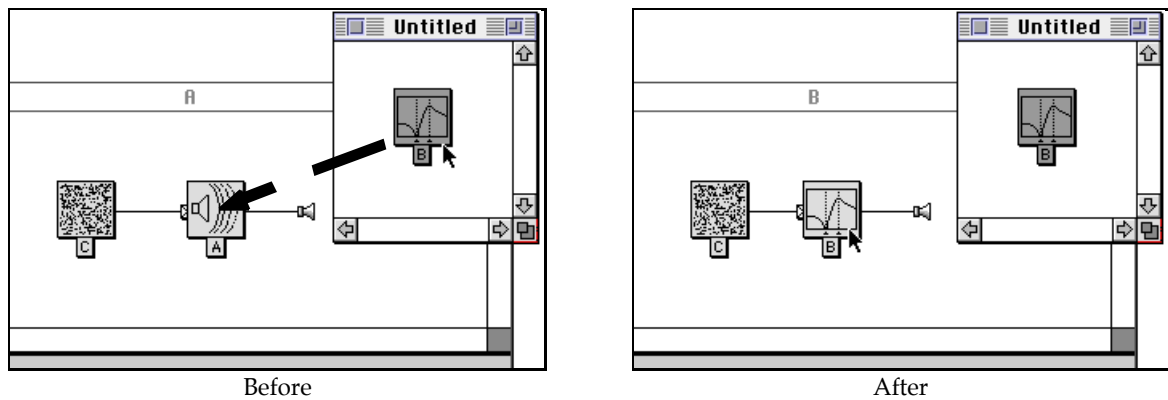
You can listen to the sound at any point along the signal flow path by selecting an icon and using **Ctrl+Space Bar**. For example, if you select and play the *GenericSource* in the example we're constructing, you will hear the unprocessed A/D input. If you select the *HarmonicResonator*, you will hear the *GenericSource* as processed through the resonator. In a sense, when you select and play an icon, you hear that Sound and everything to the left of that Sound. You are listening to the chain of processing up to the point that you have selected.

In other words, the modifiers in Kyma are *nondestructive*. You can always go back and listen to the unmodified sound just prior to where the signal enters a modifier. A modifier is like a sonic analogy to a prism or a lens: when you look at an object through a lens, you observe a systematic distortion of the object that tells you something about the structure of the lens, but the lens does not change the object itself — just the way you see the object when looking through the lens.

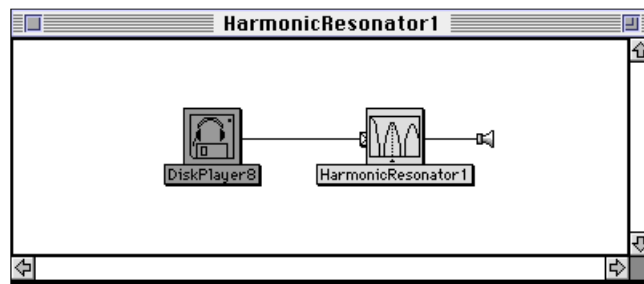
One side-effect of nondestructive modification is that each signal flow diagram becomes a history of the steps you took in order to create it. As you work on a sound, you are leaving a trace of all the operations you applied in order to reach the end result. This means you can apply the same process to another sound source later, simply by substituting the new sound source for the old in the signal flow diagram.

Replace

To replace Sound *A* with Sound *B*, drag *B* into the Sound editor, and drop it on top of Sound *A*. Alternatively, you can select and copy Sound *B*, select Sound *A*, and then paste.



For example, say we wanted to replace the *GenericSource* with a recording read from the disk. Find the *DiskPlayer* module in the **Disk** category of the prototypes, drag it from the prototypes into the Sound editor, and drop it on top of the *GenericSource* icon. (If you are in front of your computer, try this out.)

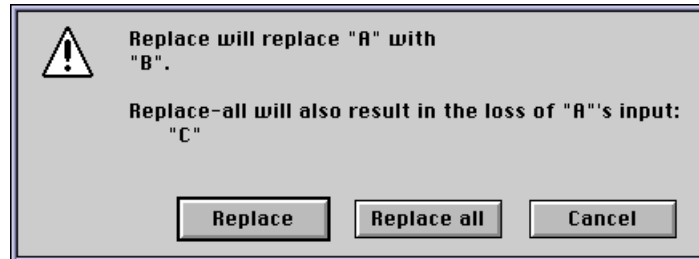


Advisory Dialogs

If you are replacing one Sound source with another, a dialog asks you to confirm that you want to complete the action:

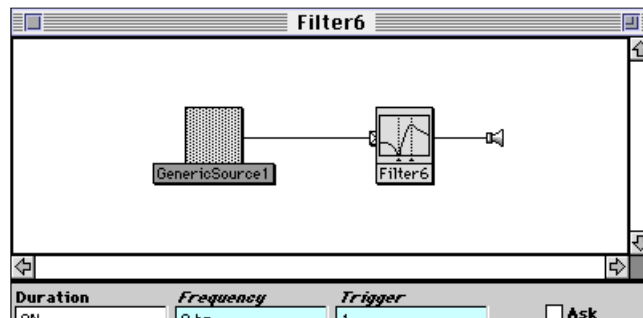


But what if the Sound you are about to replace and the Sound you are about to replace it with both have inputs? In that case, you have a choice of replacing the old Sound but retaining its inputs or of replacing both the old Sound *and* its old inputs.

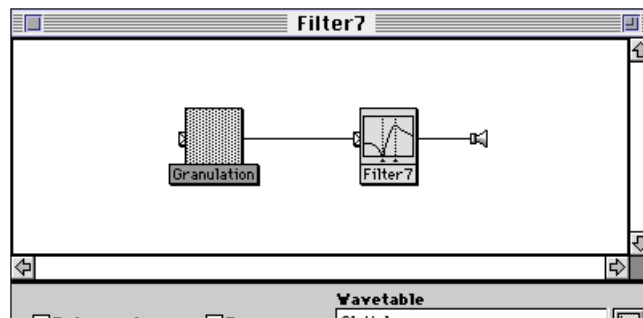


For example, say you have a *HarmonicResonator* filtering a *GenericSource*, and you decide that you would like to try a low-pass filter instead. You would drag the *Filter* module from the prototypes and drop it on the *HarmonicResonator*. The dialog gives you the choice of replacing the *HarmonicResonator* only (**Replace**) or of replacing both the *HarmonicResonator* and the *GenericSource* (**Replace all**).

The result of **Replace** would be the same *GenericSource*, but feeding into a *Filter* rather than the *HarmonicResonator*:



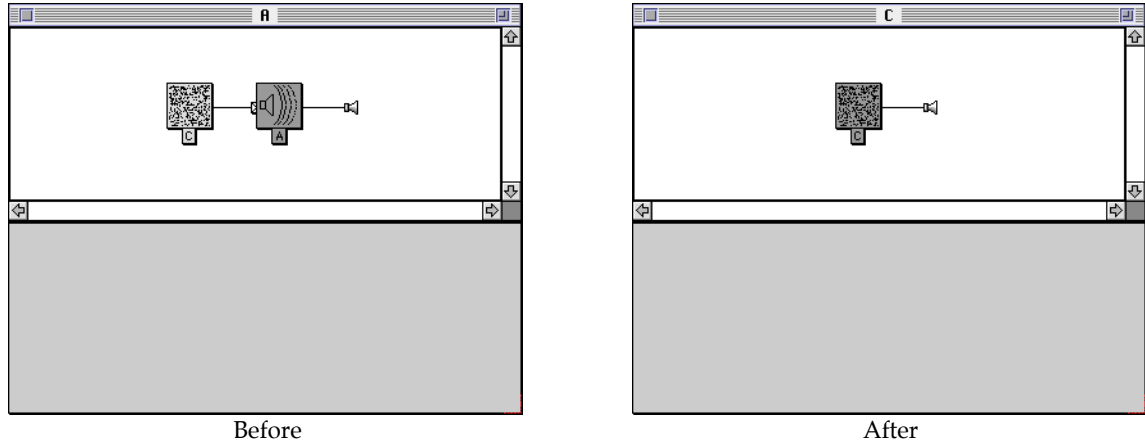
The result of **Replace all** would be a *Filter* with its default input (the *Granulation*). The original *GenericSource* would be lost:



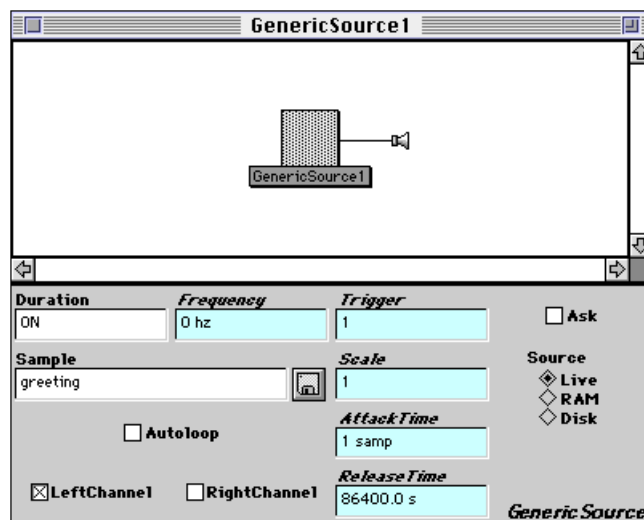
Try these things out on your computer to see more clearly the difference between **Replace** and **Replace all**.

Delete

To remove Sound *A* from the signal path (where Sound *C* is its input), select Sound *A* and press the **Delete** or **Backspace** key. Sound *C* will then replace Sound *A* in the signal flow path.



For example, say we have the *GenericSource* feeding through the *HarmonicResonator* and we want to get rid of the *HarmonicResonator*. Select *HarmonicResonator*, and press the **Delete** or **Backspace** key. Kyma will ask you to verify that you want to replace the *HarmonicResonator* with the *GenericSource* (since this is how the delete operation is actually accomplished — by replacing a Sound with its input).

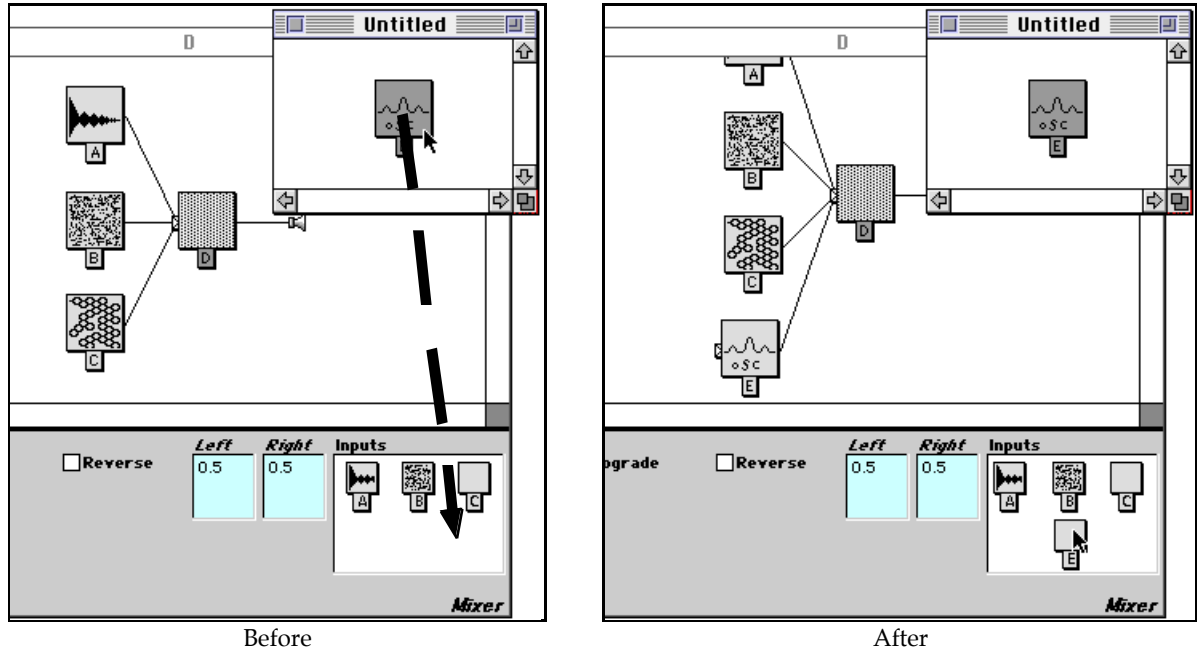


No Dangling Cables...

What if you had decided to delete the *GenericSource*? Sorry, but you can't delete the first Sound in the signal flow chain, because it would leave the *HarmonicResonator* without an input and thus unable to produce sound. You can *substitute* a new Sound source for the old one, but you cannot leave a Sound only partially defined.

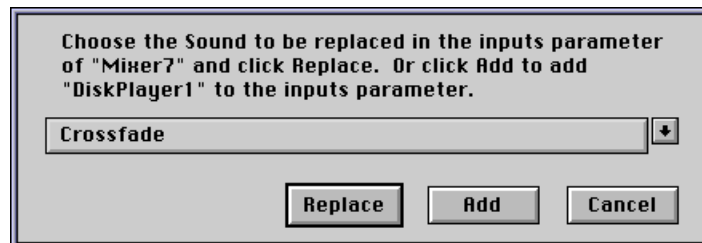
Add

To add an additional input called *E* to a multi-input Sound called *D*, drag Sound *E* into the **Inputs** parameter field of Sound *D*.



Let's say that we wanted to mix the output of a *DiskPlayer* with the output of a *Sample*. Drag a *Mixer* from the **Mixers & Attenuators** category of the prototypes into the Sound editor, and drop it on the line between the *DiskPlayer* and the output icon.

Kyma gives you a list of Sounds and a choice of whether you would like to *add* the *DiskPlayer* to this list of Sounds or use the *DiskPlayer* to *replace* one of the two Sounds already in the *Mixer*.



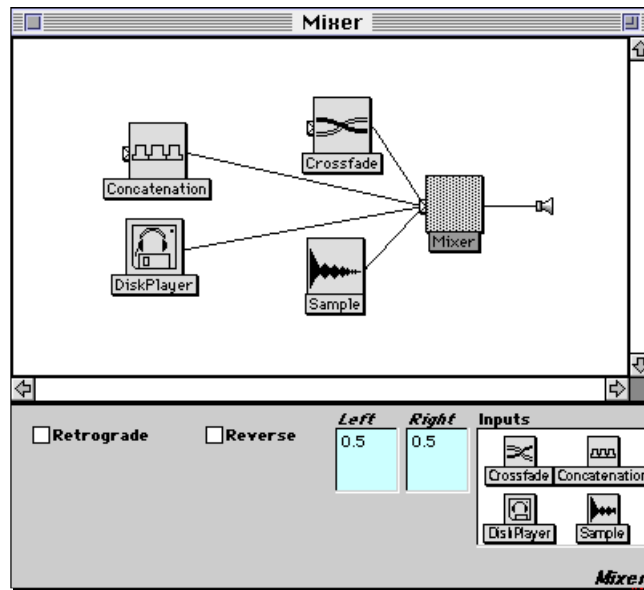
Why does the *Mixer* already have inputs? These are the default inputs. Keep in mind that every prototype is an example Sound, ready to play. In the case of the prototype *Mixer*, it already has two input Sounds. So when you say that you want to use the *DiskPlayer* as an input to the *Mixer*, you also have to decide whether you want to add it to the Sounds that are already there or whether you would like to replace one of the existing inputs.

Click the **Add** button to add the *DiskPlayer* to the collection of *Mixer* inputs. Double-click on the *Mixer* in the Sound editor in order to see its parameters. One of the *Mixer's* parameters is called **Inputs**. The same Sounds that are feeding into the *Mixer* in the signal flow diagram are also shown in the **Inputs** parameter. (If the *Mixer's* inputs are not visible in the signal flow diagram, click the tab on the left side of the *Mixer* icon to show its inputs). Drag the *Sample* from the **Sampling** category of the prototypes and drop it into the **Inputs** field of the *Mixer*.



To update the signal flow diagram, double-click in a white area. Once you do this, you should see the *Sample* as one of the inputs to the *Mixer*.

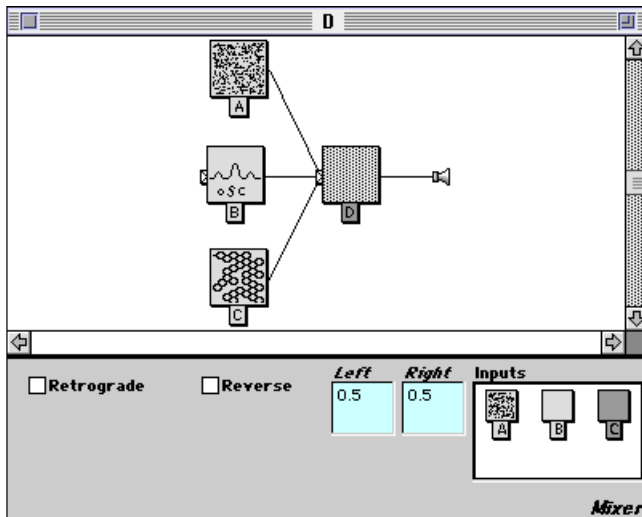
Whenever you want to make sure that a signal flow diagram reflects the most recent changes, double-click in a white area in the diagram and it will display with the latest changes.



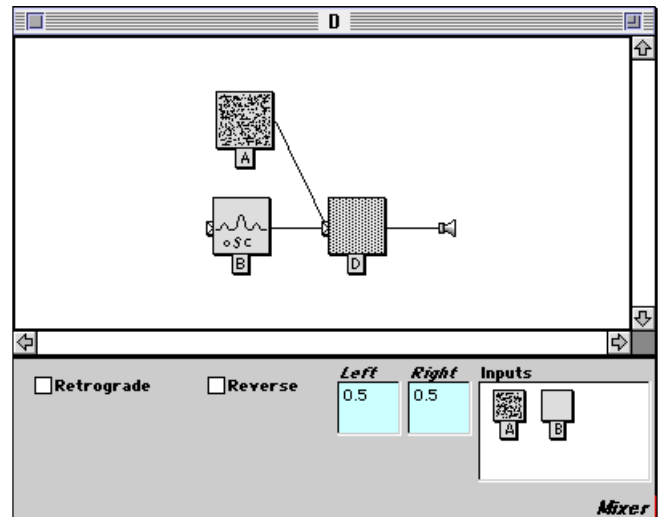
At this point, we have the *DiskPlayer* and the *Sample* as inputs to the *Mixer*, but we also have some other Sounds in there that we *don't* want... which leads us into the next topic, how to *remove* inputs from a multi-input Sound.

Remove

To remove an input *C* from a multi-input Sound called *D*, select the Sound called *C* in the **Inputs** parameter field of Sound *D*, and press **Delete** or **Backspace**.

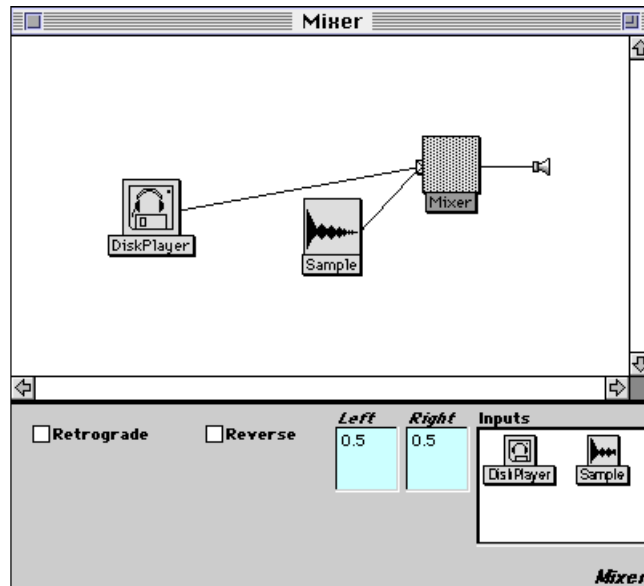


Before



After

In our example case, draw a selection box around *Crossfade* and *Concatenation* (or simply select one and shift-select the other to add it to the selection), and press **Delete** or **Backspace**. Kyma will verify that you really want to get rid of these Sounds. You can click **Yes** to remove one of them at a time with verification on each removal, or click **Remove all** to delete all of the selected Sounds at once.

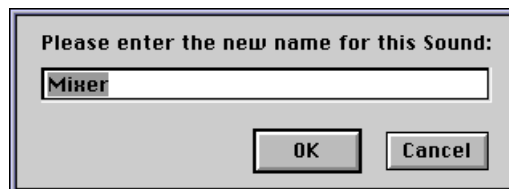


Remember to double-click in the white space to update the signal flow diagram!

Name

As you make changes to the signal flow and the Sound parameters, it can be helpful to rename some of the Sounds from their default names to something more memorable that indicates how this Sound differs from others of the same type.

To rename, select the icon, press **Enter** or **Return**, and type in a new name for the Sound.

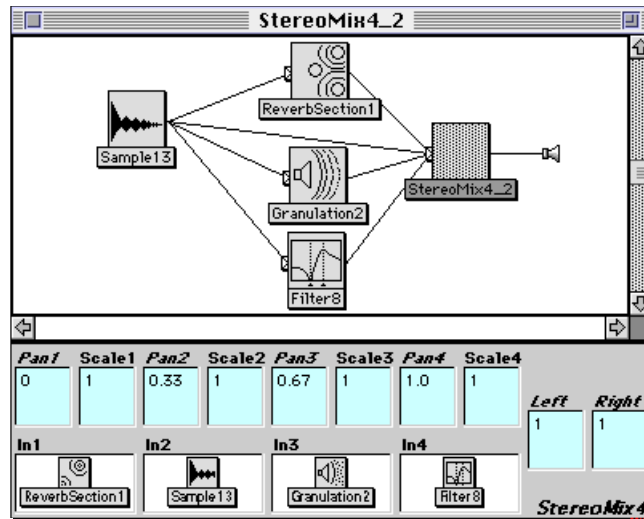


Copies and Identities

You have probably noticed by now that when you drag a Sound from the prototypes, a number is appended to the end of its name. This indicates that the Sound is a *copy* of the prototype and not the same Sound.

Whenever you drag a Sound icon from one window to another, Kyma gives you a copy of the old Sound to drag to the new location. This means that you can freely change the parameters of the dragged Sound without having to worry about altering the old Sound.

Suppose you wanted to drag the *same* Sound into two different input fields. For instance, say you wanted to route one disk recording through three different processing modules, and you didn't want use up disk bandwidth reading the same disk file three times.



Within the Sound editor, you can drag the same Sound (not a copy), by holding down the **Control** or **Option** key when you start dragging. Alternatively, you can copy the Sound, and use **Paste special...** from the **Edit** menu in order to paste the Sound rather than a copy of the Sound.

Notice that you need a *Mixer* as the last module in the signal flow diagram in order to mix the three different processing paths.

Save

When you click in the close box of the Sound editor, Kyma will ask whether you want to keep all the changes you've just made to the Sound. If you choose **Yes**, the changes will be retained. But you have not yet saved the Sound to the disk! This does not happen until you save the entire Sound file window.

There are two ways to save the changes you are making to a Sound in the Sound editor:

The simplest way is to choose **Save (Ctrl+S)** from the **File** menu when in the Sound editor. This will cause the Sound to be saved to the Sound file window, and then the Sound file window to be saved to the disk.

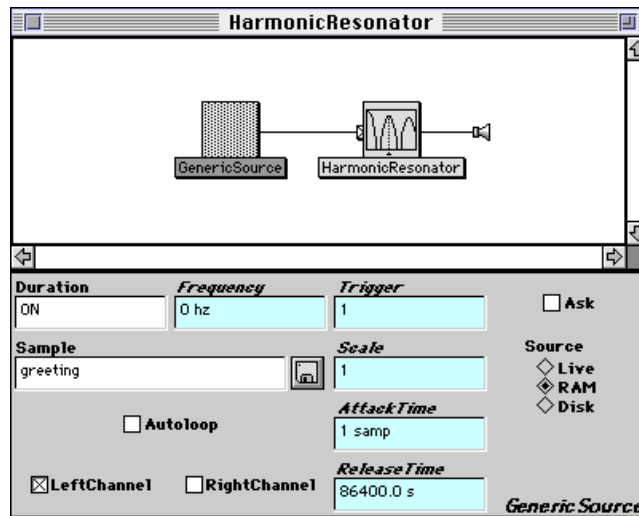
Alternatively, you can close the Sound editor, confirm that you want to save the changes, then choose **Save (Ctrl+S)** or **Save as...** from the **File** menu of the Sound file window.

Parameters

The parameters of a Sound can be constants, hot parameters, functions of time, Sounds, or arithmetic combinations of these things. On-line help on any of the parameters of a specific Sound is available by clicking on the name of the parameter in the Sound editor.

To illustrate each of these parameter types, let's look again at our concrete example Sound: a *Generic-Source* feeding into a *HarmonicResonator*.

To see the parameters of the *GenericSource*, double-click on its icon in the signal flow diagram. Play this Sound to hear what it sounds like before we start modifying its parameters.



To edit a parameter, first place the cursor in that parameter field, then select the current contents of the field, and type in a new value for that field.

To get the cursor into a parameter field, click in that field or use the **Tab** key to cycle through each of the parameter fields one by one until you reach the one you want.

By default, the contents of the field should already be selected when you **Tab** into it or click in it. If not, then position the cursor to the right or left of the current contents and click twice in order to select all (or choose **Select all** from the **Edit** menu).

Once the cursor is positioned in a parameter field and the current contents are selected, any new numbers or text that you type will replace the current contents of the field.



Whenever you change a parameter, you have to use **Ctrl+Space Bar** if you want to hear the results of the change. To put it in terms of Sounds as programs for the Capybara, when you make a change to a program, you have to recompile it before you can hear the results of that change.

Constants

A constant parameter is, true to its name, one that does not change while the Sound is playing. Some parameters must be constant over the course of the Sound (these are indicated by parameter fields with white backgrounds) while others can be either constant or dynamic (indicated by parameter fields that have cyan-colored backgrounds).

To enter a new constant parameter value, select the contents of a parameter field, and type in the new value.

For example, to change the file read by this *GenericSource*, tab into the **Sample** field and type

`nothing2`

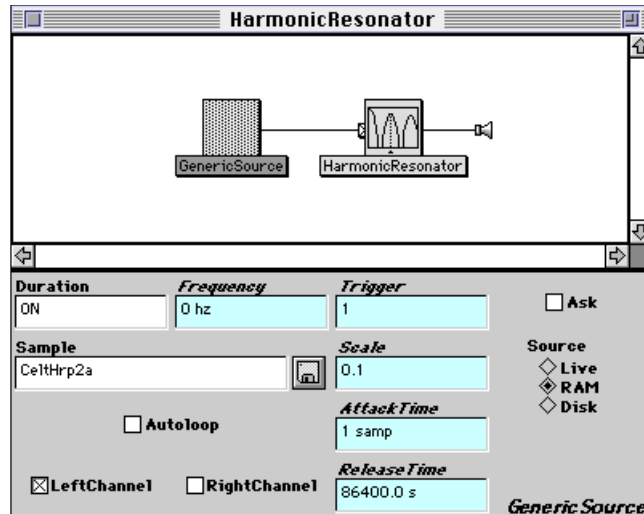
Use **Ctrl+Space Bar** to compile and hear the results of this change.

If you don't remember the name of the file, you can click the disk button to the right of the **Sample** field to choose from a file list. Try clicking on the disk button and find one of the Celtic harp samples like **Celthrp2a** in the **Musical Instruments** folder of the **Samples** folder of the **Wavetables** folder.

Next, try changing the value of **Scale** by tabbing into the **Scale** field and typing the value

`0.1`

to replace the old value.[§]



Then use **Ctrl+Space Bar** to hear the results of your change.

Frequency & Duration Fields

Frequency, **Duration** or **Time** parameter fields require that you specify the *units* of frequency or time. Frequency can be specified in units of `hz` (for hertz), or `nn` (for note number) or specified as an octave number followed by the lettername of the pitch class (for example `4 c` is middle C or MIDI note number 60). Time and duration can be specified in units of `s` (seconds), `ms` (milliseconds), `usec` (microseconds), or longer durations like `days`. You can also type the word `on` (with no units) into a **Duration** field if you want the Sound to stay on forever (actually, just 2 years). See *Specifying Units in Parameter Fields* on page 210 for a complete list of units.

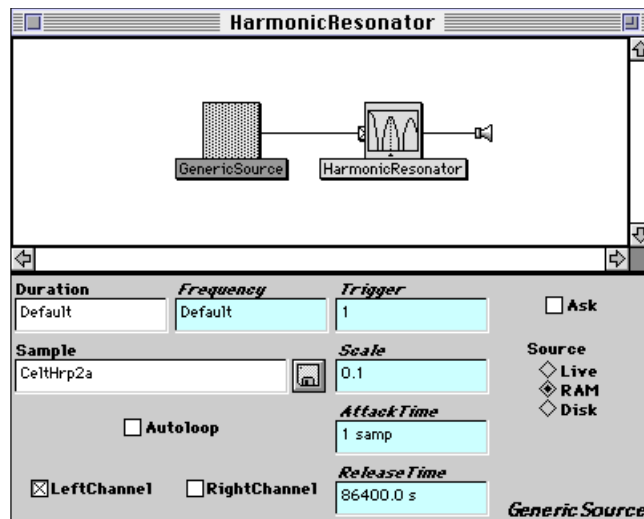
Any Sound that reads a file from disk (for example, a *GenericSource*, *DiskPlayer*, *SumOfSines*) can also read the “natural” or original duration of that file from the file header, so you can use some shortcuts in the **Duration** or **Frequency** fields.

In these Sounds, you can use the word `Default` in either the **Frequency** or **Duration** fields (no units are required because `Default` is automatically in units of seconds for times or durations and units of hertz for frequencies).[†] This specifies that the originally recorded frequency or duration should be used.

[§] It is one of the quirks of Smalltalk that *you must include the leading zero* whenever you type a fractional number. Try getting rid of the leading zero and recompiling the Sound. Kyma will put the error message `Nothing more expected->` into the parameter field. This is because Smalltalk interprets the decimal point as the period at the end of a statement, and it does not expect to see anything beyond the end of the statement.

[†] Alternatively, you can use `0 s` instead of `Default` in **Duration** fields, and `0 hz` instead of `Default` in **Frequency** fields.

In *GenericSource*, for instance, you can get the original duration of the sample file by typing `Default` and the *GenericSource* will play the sample at its original, recorded frequency if you type `Default` into the **Frequency** field.



The advantage of using the word `Default` is that you can do arithmetic on the duration or frequency without having to know or care exactly what it is. For example, if you know you want to lower the frequency by an octave, you can simply use

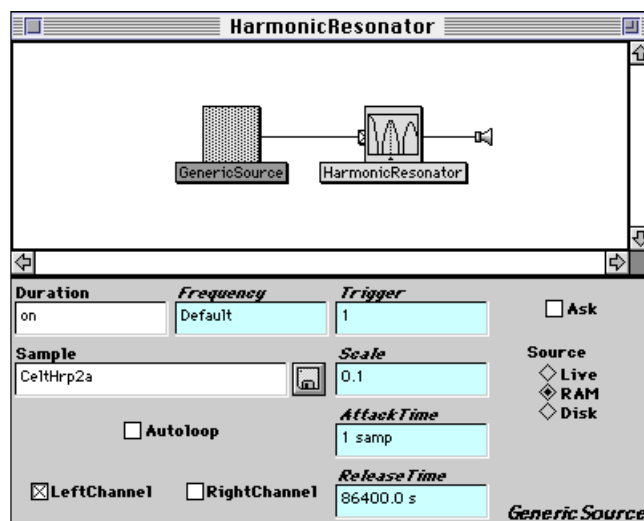
```
Default * 0.5
```

for the **Frequency**.

`Default` is in units of hz for the **Frequency**, so if you want to add to it (to transpose it), you must first convert the frequency to note numbers and then add a value in units of note numbers. For example, to raise the pitch by a perfect fifth (7 half steps), you would type

```
Default nn + 7 nn
```

As it stands now, our example will play through the *GenericSource* once and then it will go out of existence. Let's modify it such that the *GenericSource* stays permanently loaded in the Capybara (until we load a different Sound). To do this, type the word `on` in the **Duration** field, and play the Sound.

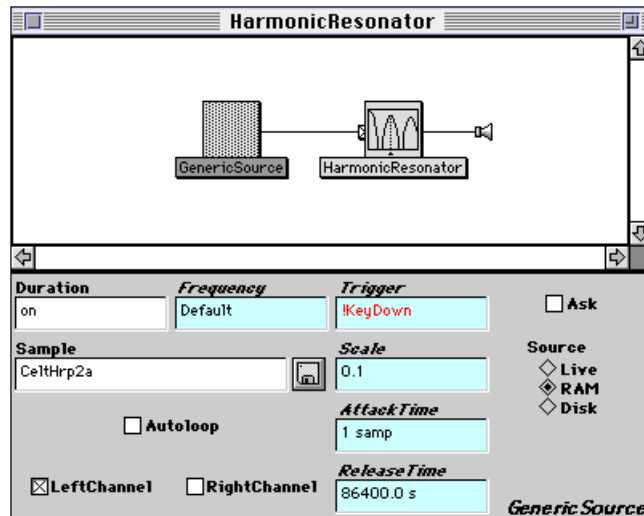


Event Values

Even though the *GenericSource* program is permanently running on the Capybara, we still only hear the sample once because the **Trigger** is an unchanging constant.[§] How can we control the value of **Trigger**? One way would be for Kyma to watch for incoming MIDI keyboard events, setting the value of **Trigger** to 1 when the key is down and 0 when the key goes up.^{*}

To specify that the **Trigger** should come from MIDI key down:

1. Click or **Tab** into the **Trigger** field
2. Press the **Escape** key on your computer keyboard (the cursor changes to a little fader)
3. Play two keys at once on your MIDI keyboard



Play the *GenericSource* and try triggering it with the MIDI keyboard. Try changing the **ReleaseTime** to 50 ms, play the *GenericSource*, and play the keyboard again. **AttackTime** and **ReleaseTime** provide a simple amplitude envelope on the *GenericSource* to avoid on/off clicks.



A red name that starts with an exclamation point (like `!KeyDown`, above) is called an *Event Value*. An Event Value is not a constant, like an ordinary number. Instead, it relies on external events for its value. These external events include: pressing a MIDI keyboard key, moving a MIDI fader, sending out MIDI events from a software sequencer, moving a fader in Kyma's virtual control surface, generating events algorithmically using the *MIDIvoice* or *AnalogSequencer* Sounds, or even writing your own C programs to set these Event Values as if they were variables in your program.

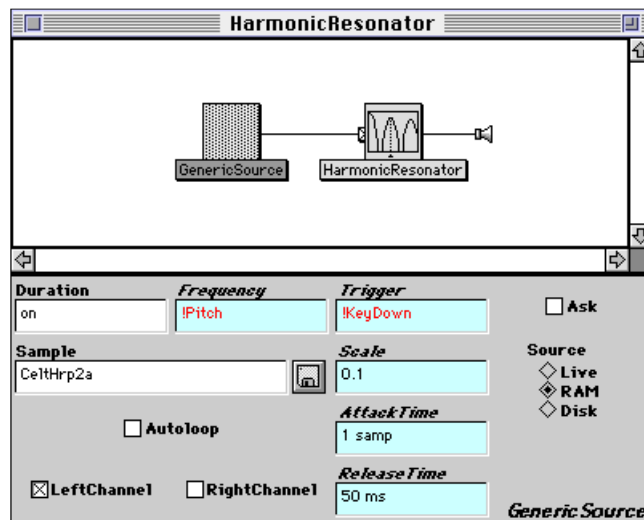
An Event Value can change while the Sound is playing, unlike a constant value which is set once when the Sound first starts up.

Playing this monotone harp *GenericSource* begs the question, how can you control **Frequency** from the keyboard? The steps are almost the same:

1. Click or tab into the **Frequency** field
2. Press **Escape** once (do not hold it down)
3. Play a single key on the MIDI keyboard

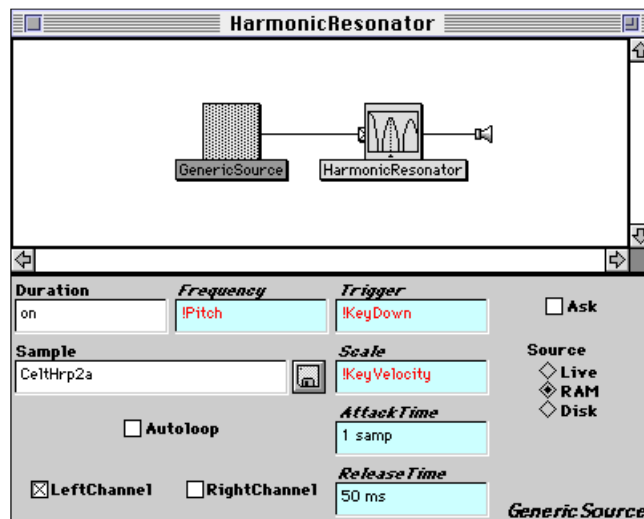
[§] Sounds that have **Trigger** fields are triggered whenever the **Trigger** becomes positive. To trigger the Sound again, the **Trigger** has to become zero or negative, then positive again.

^{*} If you haven't already done so, now is the time to connect the output of your MIDI keyboard and/or other controllers directly to the MIDI in jack on the back of the Capybara. To make sure that you are receiving MIDI, use **Configure MIDI...** (found under the **DSP** menu); it has an option to display all incoming MIDI events.



While we are at it, we might as well **Tab** into the **Scale** field, hit **Escape** and play *three* keys simultaneously in order to control the amplitude of the *GenericSource* with key velocity. Play the *GenericSource* again and play the MIDI keyboard.

At this point, the **Gate** field has the word **!KeyDown**, the **Frequency** field says **!Pitch**, and the **Scale** field contains the word **!KeyVelocity**.

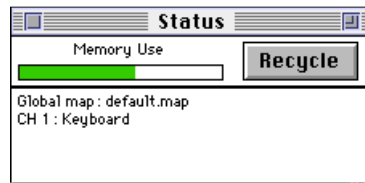


You could also have typed these names directly into parameter fields. Actually *any* word preceded by an exclamation mark will turn red in the user interface and be interpreted as an Event Value, that is, a parameter that can be changed while the Sound is running on the Capybara. If the name that you enter happens to be in your global map, then you can control it from a MIDI device, but even if it is not in the map, you can still control it from the virtual control surface — more about that in the upcoming section on MIDI continuous controllers.

You could, alternatively, have pasted these Event Values into the fields using the **Paste hot...** option in the **Edit** menu (**Ctrl+H**). **Ctrl+H** opens a list of all the Event Value names in your global map (more on global maps coming up on page 47). Scroll down the list, or simply begin typing the first few letters of the desired name to jump to proper position in the list. Then hit **Enter** to paste the name into the field. At the top of the list are some commonly used expressions that you can paste directly into the parameter field in order to save yourself some typing.

Status

Whenever you play a Sound that requires some kind of real time input (such as hitting a MIDI key, moving a virtual fader, or singing into the microphone) the Status window will display a list of the expected sources of input. This is especially helpful information when you play a Sound and don't hear anything because the Sound is waiting for input from the A/D or waiting for a `!KeyDown` trigger.



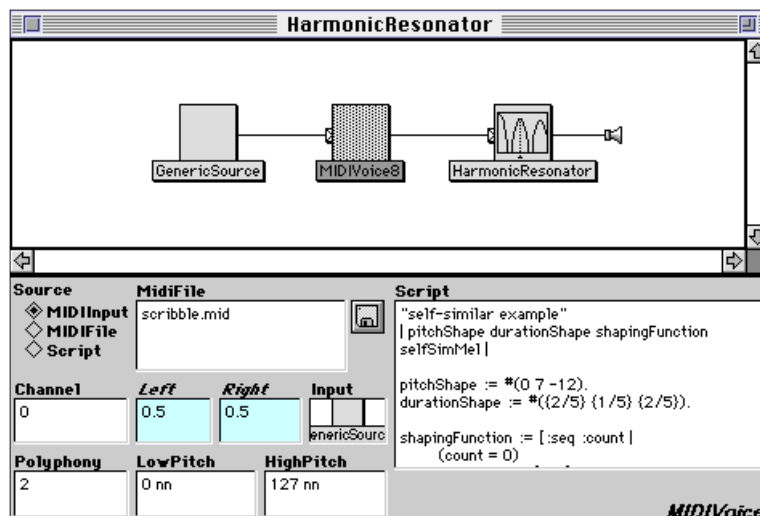
The Status also displays the current global map (the mapping from memorable names in Kyma to actual MIDI controller numbers), the MIDI channel or channels on which it is expecting MIDI input events, the word **Keyboard** (if it expects MIDI key events), and a list of all continuous controller names.

Across the top is a horizontal thermometer display giving an indication of how much memory is available on your host computer (not to be confused with the memory on the Cappybara). As memory is dynamically used up and then recycled,^s you will notice the thermometer growing redder or bluer and also growing further towards the right and shrinking towards the left. Smalltalk does periodic recycling, but you can force it to recycle memory immediately by clicking the **Recycle** button. You should leave this Status window open at all times, because it monitors the available memory and does automatic recycling for you.

Polyphony

Controlling parameters from a MIDI keyboard immediately suggests the idea of polyphony. To make a Kyma Sound polyphonic, make it the input to a *MIDIVoice* module; this defines the Sound as a MIDI voice, assigns it a channel, gives it a polyphony value, and specifies a range of MIDI pitches that it should pay attention to.

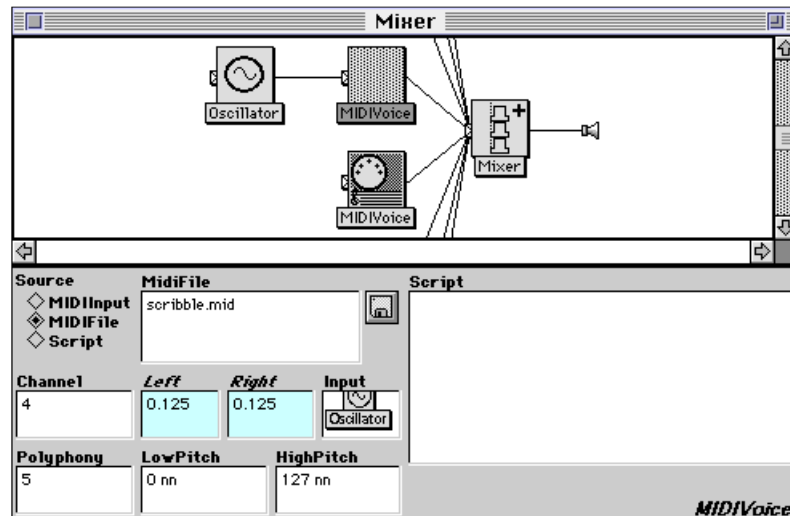
For example, to increase the polyphony on the MIDI-controlled *GenericSource* from the previous section, drag a *MIDIVoice* from the **MIDI In** category of the prototypes, and drop it on the line between the *GenericSource* and the *HarmonicResonator*. Double-click on the *MIDIVoice* to take a look at its default parameter settings.



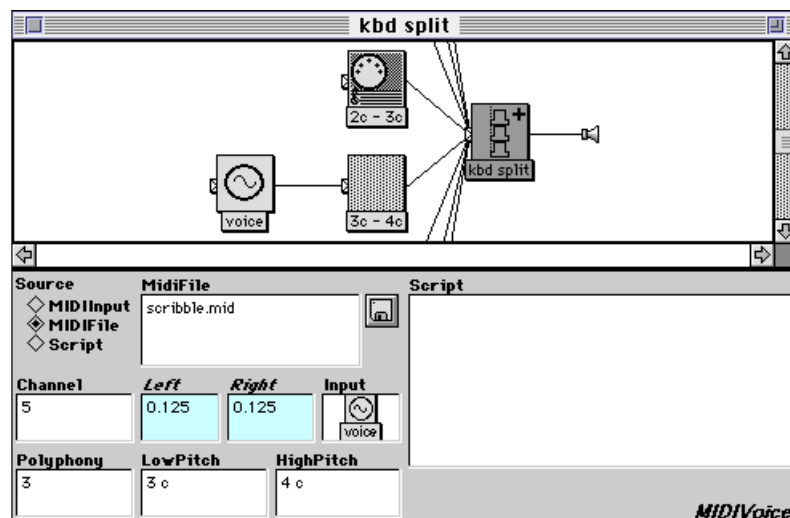
^s Kyma is written in Smalltalk, a programming language that dynamically allocates memory; whenever an object in Kyma is no longer in use, Smalltalk can recycle the memory previously needed by that object.

The MIDI channel is set to 0, so it will default to the MIDI channel specified in **Configure MIDI...** under the **DSP** menu. Change the **Polyphony** to 4, select the **MIDIvoice**, and play it. Now you should be able to hold down four notes at a time on the MIDI keyboard.

To specify that there should be several different timbres, each on its own MIDI channel, feed each of the Sound's (timbres) into its own **MIDIvoice**, assign it its own channel, and then feed all of those **MIDI-Voices** into a **Mixer**.



To implement a “keyboard split”, where one range of note numbers triggers one timbre and another range of note numbers triggers another, create a **Mixer** of two or more **MIDIvoices**. Set the **LowPitch** and **HighPitch** of each **MIDIvoice** to the range of note numbers that should trigger the timbre associated with that particular voice.

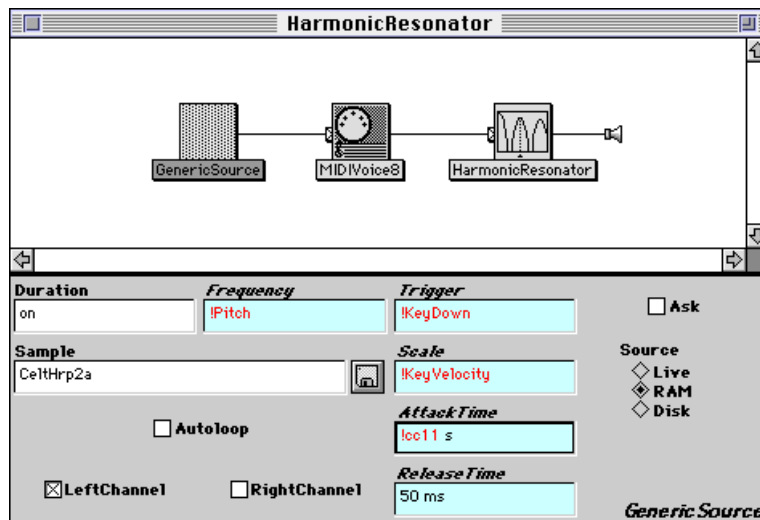


If your timbres are all samples or disk files, take a look at **MultiFileDiskPlayer**, and **KeyMappedMulti-sample**. These provide convenient ways to quickly map samples or disk files to MIDI note numbers or other events.

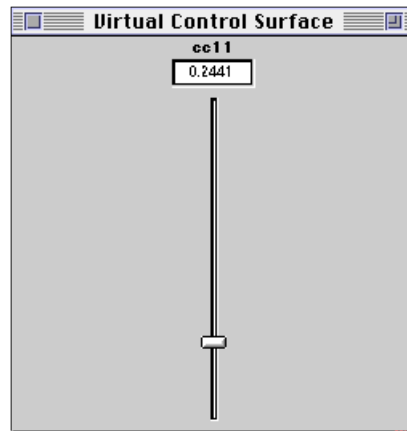
MIDI Continuous Controllers and the Virtual Control Surface

Let's extend this example so that we can control the attack time of the envelope as well.

1. **Tab** into the **AttackTime** field
2. Press **Escape** once (do not hold down the key) and move a MIDI fader
3. Press the **Space Bar** and the letter **s** (to add units of seconds to the value from the MIDI fader)



Kyma will make an association between the fader you just moved and this parameter. When you play the *GenericSource*, the virtual control surface will pop to the front, showing the name of the continuous controller that now controls the attack time. You can adjust the attack time using either the MIDI fader or the virtual fader on the screen.



You can also enter values by typing into the small field at the top of the fader. When the virtual control surface is in front, you can hit the **Tab** key to select the number in the numeric field at the top of the fader. Type a new number between 0 and 1* (remember to include a leading zero before the decimal point), and press **Enter** or **Return**. The fader will jump to the new setting.

* Inside Kyma, the value of a continuous controller is in the range of 0 to 1. If the controller is a MIDI controller, you are probably accustomed to thinking of its range as 0 to 127, but inside Kyma this range is scaled to lie between 0 and 1. This makes it easy to do arithmetic with controllers and to predict the results of multiplying a controller by a constant. For example, if you have a controller called `!Frequency`, and its range is 0 to 1, you can easily change its range to 750 hz to 1000 hz by typing:

```
!Frequency * 250 hz + 750 hz
```

in the **Frequency** field. When the controller value is 0, the value of the expression is 750 hz, and when the controller value is 1, the expression value is 1000 hz. Changing the range of a controller comes up often, so it is worth remembering the general form for this expression:

```
!controller * (maximum - minimum) + minimum
```

If you don't have any MIDI faders, you can still type in a name preceded by an exclamation mark into a parameter field, and it will show up in the virtual control surface where you can control it.[§] Alternatively, you can use **Ctrl+H** or **Paste hot...** from the **Edit** menu to paste a name chosen from a list of names.

Continuing with our example, let's add a continuously variable offset to the `!Pitch` supplied from the MIDI keyboard:

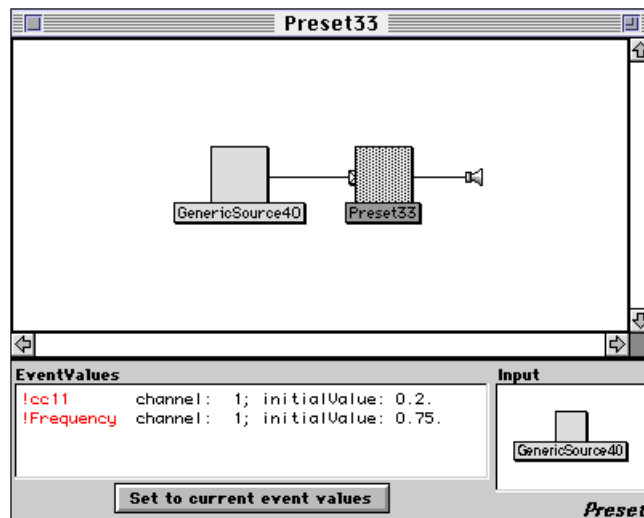
```
!Pitch + !Frequency nn
```

Notice that the continuous controller `!Frequency` must have units of pitch or frequency, in this case, `nn` for note number.

Select the *GenericSource*, use **Ctrl+Space Bar** to compile, load, start the Sound, and try playing the keyboard while moving the `!Frequency` fader on the virtual control surface.

Saving Event Value Settings

After you have developed a Sound that uses Event Values and have fine-tuned the Event Values, it would be nice to save the settings. To do this, drag the *Preset* Sound (from the **MIDI In** category of the prototypes) on to the line connecting the right-most Sound to the speaker icon:



Click on the **Set to current event values** button to capture the current settings of the Event Values. Now, every time you play the *Preset* Sound, all Event Values will jump to those settings at the beginning of the Sound (but you can still adjust them as the Sound is playing).

The Heartbreak of Numeri-phobia?

For many of you, the kind of arithmetic you do in parameter fields comes as naturally as everyday conversation. If you fit this category, then you can skip over this section.

We realize, though, that for some of you, the sight of arithmetic operators *may* bring back memories of that apathetic (or even sadistic) algebra teacher who managed to convince you that math was boring and impenetrable (despite the fact that, as a musician, you have a natural predisposition for mathematical thinking... if only they had not insisted on boring it right out of you). In the event that you were unfortunate enough to have encountered bad math teachers in your earlier years, have no fear. Kyma can bring back your natural love of patterns and algorithms and rediscover your inner mathematician!

Here are a couple of tips for breezing through the arithmetic in Kyma parameter fields.

[§] Kyma defines some built-in Event Values with different appearances in the virtual control surface. `!cc00` through `!cc127` are faders that correspond to the MIDI continuous controllers on the default MIDI channel, `!bt00` through `!bt127` are momentary buttons on the same controllers, and `!sw00` through `!sw127` are check boxes on the same controllers.

You can customize the appearance of the virtual control surface in the global map, see *Virtual Control Surface and Mappings* on page 483.

Scale and Offset

90% of all arithmetic that you will need in parameter fields will fit the same basic pattern. Most of the time, we want to change the range of a fader to be something other than 0 to 1. So most of our expressions will look something like this:

```
!Fader * range + minimum
```

To change the range of a fader that goes from 0 to 1, *multiply* it by the range you really want. For example, if you want your frequencies to have a range of one octave, you could multiply (or scale) the !Frequency fader by 12 half steps:

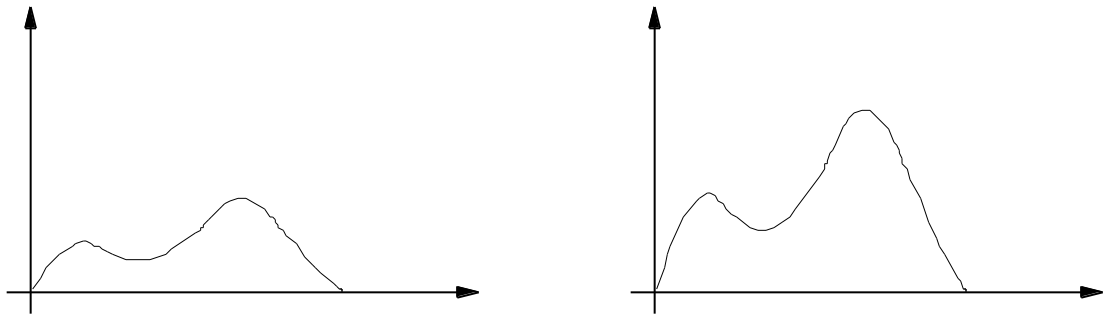
```
!Frequency * 12 nn
```

That would give you a range of note numbers from 0 to 12. Note number 0 is pretty low, so you probably would want to give it a different minimum note number. To set the minimum, you could *add* to (or offset) your value:

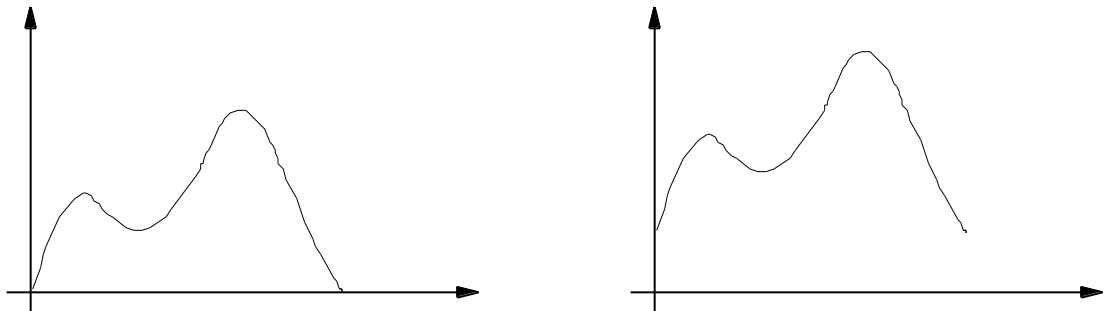
```
!Frequency * 12 nn + 60 nn
```

Now your !Frequency fader covers the range from middle C (note number 60) to one octave higher (note number 72).

Sometimes multiplying is called scaling, because it retains the shape of the function, and just scales its size:



And sometimes adding a constant number is called adding an “offset”, because it doesn’t change the shape or the size of the function, it just shifts the whole thing up or down — setting it off from the original axis. You may have encountered this terminology before in the form of “DC offset”, when some constant voltage is added to your audio signal, shifting it up or down with respect to zero.



In some of the Kyma documentation and nomenclature, you will see the words, scale, offset or *ScaleAndOffset* to mean multiply and add.

Minus 1

One version of the scale and offset that comes up fairly often in Kyma is the one that takes you from the range of (0, 1) to the range (-1, 1). Since the range of a Sound’s output is typically (-1, 1) and the

range of a continuous controller output is $(0, 1)$,[‡] you will encounter situations in which one range must be converted to the other.

To convert from $(0, 1)$ to $(-1, 1)$ you can use

```
!Fader * 2 - 1
```

First, you multiply by 2 because $(-1, 1)$ is twice as large a range as $(0, 1)$. But then you end up with the range $(0, 2)$. So you have to shift it down by subtracting 1 so that you end up with $(-1, 1)$.

If you have pasted a Sound into a parameter field[§] and want to scale its range of $(-1, 1)$ to a range of $(0, 1)$, you can use

```
aSound L * 0.5 + 0.5
```

In other words, you first scale the range to half its size, and then add 0.5 to the result to shift it to the range of $(0, 1)$.

Nyquist

Even more common than the scale and offset is to scale a number without adding an offset. For example, if you have a controller or a Sound that is giving you a function in the range of $(0, 1)$ and you want to use that function to control the frequency of another Sound, you can scale $(0, 1)$ to the range of usable frequencies by using the following expression:

```
!Fader * SignalProcessor sampleRate * 0.5 hz
```

Why? Because in digital audio, the highest frequency you can represent is half the sample rate of the Cappybara. You can get at the current sample rate by sending a message to the `SignalProcessor`. Then you can multiply it by 0.5 to get half the sample rate and append the units of `hz` on the end.

Note Numbers

While all MIDI continuous controllers are scaled in Kyma to the range of $(0, 1)$, Kyma leaves the range of `!Pitch` and `!KeyNumber` to be $(0, 127)$, where 60 is middle C. So if you intend to use a continuous controller as a pitch, you will have to scale it to the desired range of note numbers. For example

```
!Fader1 * 6 nn + 4 c
```

first scales the $(0, 1)$ range of `!Fader1` to the range of $(0, 6)$, then gives it units of `nn` for note number, and finally adds it to the octave-pitch-class designation for middle C.

Flipping Time on its Head

Here is another relationship that comes in handy fairly often: frequency and duration are the inverses of each other. Another way to say this is that the period of a signal (a duration) is the inverse of its frequency. This becomes clearer when you recall that hertz really stands for cycles per second. If you flip cycles per second, you end up with seconds per cycle or the number of seconds it takes to get through one cycle of the waveform.

In other words, if you evaluate

```
441.0 hz inverse
```

in Kyma, you will get the duration of one cycle of a 441 hz signal. If you translate that to samples, you can see how many samples that is; for example if the sample rate is 44.1 khz,

```
441.0 hz inverse samp
```

is 100 samples.

[‡] All continuous controllers have a range of 0 to 1, corresponding to the MIDI controller values of 0 to 127. Pitch bend (`!PitchBend`), however, has a range of -1 to 1, corresponding to the MIDI pitch bend range of 0 to 16383.

[§] We are getting a little ahead of ourselves here. Using Sounds in parameter fields is explained in detail in *Using Sounds as "Control Voltages"* starting on page 52.

So if you see a parameter field that asks for a duration (e.g. **Period**, **HoldTime** or **Delay**) and you would like to specify the duration that corresponds to one cycle of some particular frequency, you can enter the desired frequency and take the inverse of it.

Likewise, in a **Frequency** parameter, you can enter a duration and take *its* inverse in order to get a frequency in hertz. For example, if you are using an *Oscillator* as a repeating envelope and prefer to specify the duration of each pass through the envelope waveform, you can put the inverse of that duration in seconds into the **Frequency** field of the *Oscillator*.

Avoiding Carpal Tunnel

Many of these expressions are available in a list of hot expressions. Choose **Paste hot...** from the **Edit** menu or use **Ctrl+H** to paste one of these expressions into a parameter field, saving yourself some typing.

Fun

You now have the bits and pieces to do most of the arithmetic you will need for Kyma. Given this, you can figure out everything else by puzzling through it, piecing together subexpressions that you have figured out before, drawing a few little sketches and trying a few things out. (Everyone does it like that, so there's no reason to feel inhibited about drawing a sketch or trying out a few test values in order to experiment). And the little secret your high school teachers may just have forgotten to tell you is that it is kind of fun to puzzle this stuff out — especially when the result is sound (and not just numbers scratched in pencil on a blue-ruled sheet of paper).

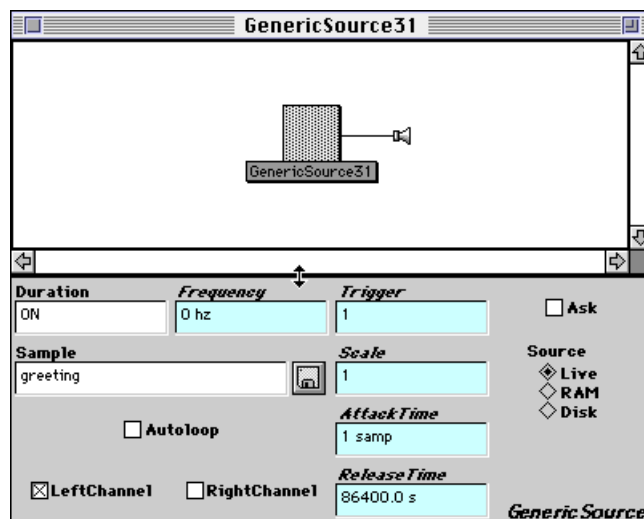
The Real-time Evaluator

The arithmetic and other operations that you perform on Sounds or Event Values (the red names preceded by exclamation points) in parameter fields may *look* like a cross between ordinary arithmetic and the Smalltalk programming language. In fact, these expressions are evaluated in real time by an event-driven evaluator that runs on the Capybara. In other words, the expressions involving Event Values or Sounds that you write in Kyma parameter fields are *not* evaluated on the host PC or Macintosh. They are evaluated in real time on the Capybara; this means that the timing is unaffected by the other tasks being performed by your computer (such as running your sequencer, wave editor, and operating system).

For a list of all functions understood by the real-time evaluator, along with an explanation and example of how to use each one, see *Real-Time Expressions in Parameter Fields* on page 211.

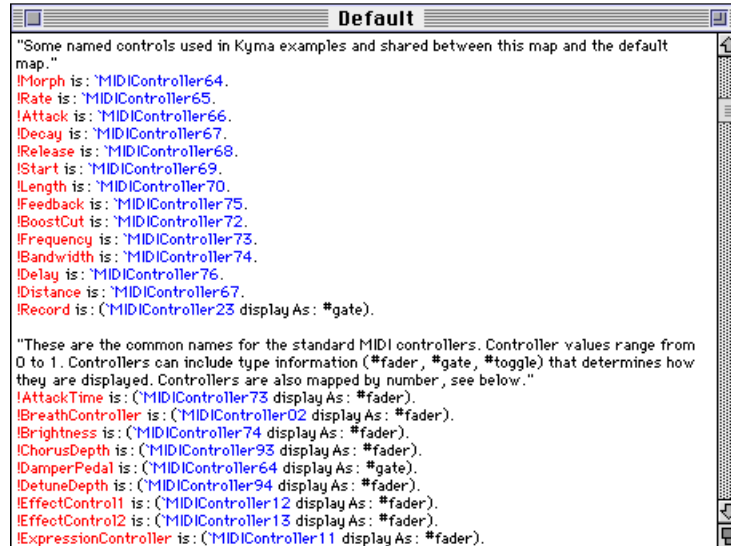
Making More Room in the Parameter Fields

Once you start doing arithmetic in the parameter fields, you can run out of room pretty quickly. To enlarge a parameter field to the full screen size, use **Ctrl+L** (for large window). You can also use the mouse to pull up the center line that separates the signal flow graph from the parameter fields in the Sound editor, thus giving more room to the parameters:



Physical MIDI Faders and Global & Local MIDI Maps

How does Kyma make the connection between a red Event Value in one of the parameter fields and an actual, external MIDI controller? Behind the scenes, there is a global map that takes a set of memorable names (like !Frequency or !Pan) and associates them with (less memorable) MIDI controller numbers (like `MIDIController18 or `MIDIController23). To see the current global map, choose **Open...** from the **File** menu, and select **Global map** as the file type. Then find and open the file named **Default** in the **Global Maps** folder.



The syntax for setting up an association between a (red) memorable name and one of the standard set of MIDI events and controllers (shown in blue) is

```
!MemorableName is: `MIDIControllerNbr.
```

The blue name (the one preceded by a back quote) is called the *Event Source* and the red name (preceded by an exclamation point) is called an *Event Value*.

You can also specify how you would like the controller to display in the virtual control surface. A #gate shows up as a button, a #toggle shows up as check box, a #fader shows up as a fader with a field at the top displaying the numeric value, and a #smallFader shows up as a numeric field (which you can still increment and decrement as if it were a fader by holding down the **Control** or **Option** key while moving the mouse up and down).

Here is an excerpt from the **Default** map showing the syntax for specifying how the controllers should appear in the virtual control surface:

```
!DamperPedal is:          (`MIDIController64 displayAs: #gate).  
!DetuneDepth is:          (`MIDIController94 displayAs: #fader).  
!EffectControl1 is:       (`MIDIController12 displayAs: #fader).  
!EffectControl2 is:       (`MIDIController13 displayAs: #fader).  
!ExpressionController is:  (`MIDIController11 displayAs: #fader).  
!ExternalEffectsDepth is: (`MIDIController91 displayAs: #fader).  
!FootController is:       (`MIDIController04 displayAs: #fader).  
!HarmonicContent is:      (`MIDIController71 displayAs: #fader).  
!Hold2 is:                (`MIDIController69 displayAs: #toggle).
```

The following examples illustrate how you can associate a name with a controller on a specific channel (while that same controller number on a different channel is associated with another name).

```
!Length is:              (`MIDIController07 channel: 16).  
!Depth is:               (`MIDIController07 channel: 15).
```

To prevent a controller from ever showing up in the virtual control surface, tell it to display as `#nothing`:

```
!Harm is:          (`MIDIController71 displayAs: #nothing) .
```

The global map is always there, invisibly associating Event Values with your external MIDI devices. When you first get Kyma, the **Default** map is selected as your global map. To select a different global map, use **Choose global map...** in the **File** menu.*

Whatever your current global map may be, you can always temporarily add to it or override portions of it in a *local* map. The *MIDIMapper* Sound has a **Map** parameter where you can enter the associations that differ from the global map. These associations apply only to those Sounds to the left of the *MIDIMapper* in the signal flow diagram. Anything not overridden in the local map will be specified in the global map.

You can also use a local map to specify controllers that will show up in the virtual control surface *only*, without being connected to *any* external controller. This is especially useful when you have some specific names that you would like to give the controllers for one specific Sound, but when those controller names are not generally useful enough to warrant adding them to the global map.

To specify a named controller that will show up in the virtual control surface only (and not be controllable with physical faders), use the same name for both the Event Value and the Event Source, e.g.

```
!Freq is: `Freq.
```

Local maps are also handy for displaying the true value of a fader. In other words, instead of taking the (0, 1) controller and multiplying it by the desired range in the parameter field, you can change the range of the controller to whatever minimum and maximum value you like. Typically, this is not something you would want to do in the global map, because the desired range is probably going to be different for each Sound. So, although you *can* specify a different range in the global map, it is most often done in a *MIDIMapper*.

To specify a range of values for a MIDI controller, use a statement like the following:

```
!Frequency is: (`MIDIController18 min: 100 max: 1000) .
```

This gives you a fader called `!Frequency` whose minimum value is 100 and whose maximum value is 1000 controlled by MIDI continuous controller number 18 on the default MIDI channel.

The extra tags specifying channel, range, *etc.* are not limited to use in local maps. They can also be used in any new global maps that you design or modify.

Automation

In some situations, you might want to automate parameter changes, rather than controlling them by hand with a MIDI fader or the virtual control surface. There are several ways to accomplish this:

- Use a software sequencer to draw the continuous controller functions, and then control the Cappybara from the sequencer.

- Use the sequencer to draw the time-varying behavior of each controller, export the sequence as a standard MIDI file, and read the MIDI file in Kyma.

- Use the time functions built in to Kyma.

- Use a *Sound* to control the way the parameter changes over time (in the same way that you might use an LFO to control a parameter in a modular analog synthesizer).

- Use the **Script** parameter of a *MIDIvoice* to algorithmically specify how the controller value changes over time.

- Use the *AnalogSequencer* Sound to generate sequences of note events and controller values.

* When you opened the **Default** map, you may have noticed that there were a couple of other maps in the **Global Maps** folder as well. You can make a copy of the **Default** map, and customize it for whatever MIDI devices you have in your studio. For example, the **PC1600 map** is one that we have set up for the Peavey PC 1600 MIDI controller, and the **Lightning map** was designed to map the horizontal and vertical movements of the left and right wands of a Buchla Lightning controller.

Automating Controllers using a Software Sequencer

Type or paste a controller number into a Kyma parameter field that matches the name (or number) of a controller in your sequencing program. For example, you could use the following in a **Frequency** field:

```
!cc120§ * 12 nn + 4 c
```

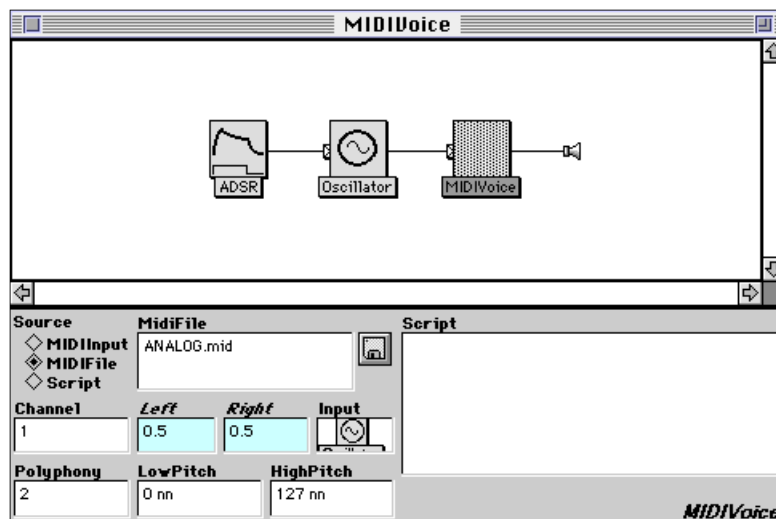
Use **Ctrl+Space Bar** to play the Sound, and then start up (or switch to) your software sequencer. Once in the sequencer, you can record any keyboard events or controller moves. Most sequencing programs also let you edit or draw the controller moves graphically.

Once you have used **Ctrl+Space Bar** to load your Sound into the Capybara, the Kyma software does not have to do much other than updating its status displays, so you can bring the sequencer (or other MIDI-event generating software) to the front. Once the Sound has been loaded into the Capybara, you can think of the Capybara just as you would any other external sound module, at least in terms of how you would control it from the sequencer.[‡]

MIDI Files

Once you have recorded and/or adjusted the time-varying controller functions to your liking in the sequencer, you can (optionally) export the sequence as a standard MIDI file and then use a *MIDIVoice* or *MIDIMapper* to read that file and use it to control a Kyma Sound.

In the Kyma Sounds *MIDIVoice* and *MIDIMapper*, you can choose the source of control for the input Sound. The first choice, **MIDIInput**, indicates that the Sound will be controlled by the live MIDI input, whether that comes from a MIDI controller, a sequencer, or some other software that outputs MIDI to the Capybara. The second choice, **MIDIFile**, indicates that the control will come from the file named in the **MIDIFile** parameter field. The third choice, **Script**, indicates that the events will be generated algorithmically by a program you have written the **Script** field (more on this later).

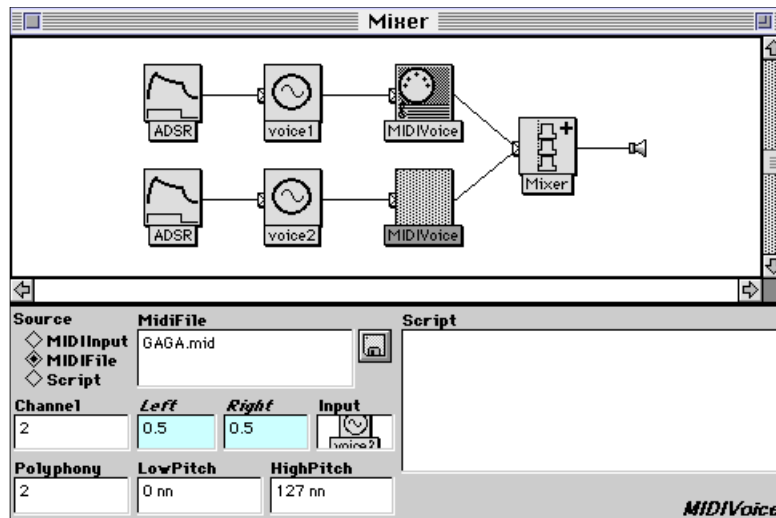


Each time you play the *MIDIVoice* with **MIDIFile** selected as the source, it reads up all of the events on the specified channel of the specified MIDI file and then sends those MIDI events to its input Sound.

[§] The default global map defines !cc120 as MIDI continuous controller 120 on the default MIDI channel.

[‡] For reasons of speed and universal compatibility, Kyma requires that you send MIDI directly to the Capybara via the MIDI input on its back panel. If you are generating the MIDI events in software, you have to send them out of the computer via a MIDI interface and connect the output of that MIDI interface to the Capybara. In other words, treat the Capybara as you would any other external sound module, even if Kyma and your MIDI event-generating software happen to be running on the same computer at the same time.

To read several channels, create several *MIDIvoices*, each assigned to its own channel but reading the same MIDI file, and place all of them into a *Mixer*:



By saving the controller movements in a MIDI file and reading them with the *MIDIvoice*, you can combine the Sound and the means for controlling the Sound into one package — another manifestation of the ubiquitous “sound object”!

Time Functions

Among the functions that the Kyma real-time evaluator can understand are functions that change over time. Examples include `!LocalTime` whose value is simply the current time in seconds since the Sound started playing, or `repeatingRamp:` which generates a ramp or sawtooth-shaped function that changes linearly from 0 up to 1 and then repeats.

Suppose you had a *GenericSource* that was set to read a sample from RAM and that you wanted its frequency to increase gradually over time. You could enter the following expression into the **Frequency** parameter field:

```
4 c + !LocalTime nn
```

This would result in a sample whose pitch would increase continuously at the rate of one half step per second. Alternatively, you might use something like

```
(!KeyDown ramp: 0.1 s) * 3 nn + !Pitch
```

This would give you the pitch from the keyboard plus a pitch bend from zero to three half steps over the course of a tenth of a second. Or, if you didn’t want to rely on any keyboard input at all, you could use something like

```
(1 repeatingRamp: 3 s) * 440 hz + 440 hz
```

which would result in a frequency that would glide from 440 hz up an octave to 880 hz over the course of three seconds, and then repeat the process over and over for the duration of the Sound.

To trigger a Sound periodically, you can use the `bpm:` (beats-per-minute) function. For example, if you put the following into the **Trigger** field of the *GenericSource*:

```
1 bpm: 60
```

it would be triggered once per second. To effectively “loop” a disk file that is 3.5 seconds in duration, you could type the following into the **Trigger** field:

```
1 bpm: (60 / 3.5)
```

This would trigger the disk playback once every 3.5 seconds.

Random Variation

To add randomness to any parameter, you can use the `random` and `nextRandom` functions. Each of these functions generates random numbers between -1 and 1. To generate a new random number every 2 seconds, you would use:

```
2 s random
```

To get a new random number each time there is a trigger like `!KeyDown`, you would use

```
!KeyDown nextRandom
```

or to trigger a new random number 240 beats per minute, you would use

```
(1 bpm: 240) nextRandom
```

In the *GenericSource*, for example, if you choose `celtHrp2a` as the **Sample**, `!KeyDown` as the **Trigger**, and you enter the following expression for the **Frequency** parameter

```
!KeyDown nextRandom * 12 nn + 48 nn
```

you will get a different pitch (between 2 c and 4 c) each time you press a MIDI key.

To limit the choices to a specific list of note numbers, you pick them at random from an array. For example if you type the following into the **Frequency** field

```
(!KeyDown nextRandom abs * 5 of: #(53 50 48 45 43)) nn
```

you will get a different pitch each time you press the key, but the pitches will be chosen from the list of MIDI note numbers `#(53 50 48 45 43)`. In this example we are using the random number as an index into an array of note numbers. Let's take it apart and look at it step by step:

1. First we generate a random number every time a key goes down:

```
!KeyDown nextRandom
```

This gives a number between -1 and 1.

2. But we can't use a negative number to mean a position within the array. So we take the absolute value, forcing all the numbers to be positive:

```
!KeyDown nextRandom abs
```

This gives us a number between 0 and 1.

3. In order to have a chance of selecting any position in the array, we next multiply by the size of the array:

```
!KeyDown nextRandom abs * 5
```

This gives us a number between 0 and 5.[§]

4. The next step is to use this number as an index into the array. To do this, we type `of:` followed by the array

```
!KeyDown nextRandom abs * 5 of: #(53 50 48 45 43)
```

5. And finally, we have to add the units. So we put parentheses around the entire expression and add the units of note number to the end:

```
(!KeyDown nextRandom abs * 5 of: #(53 50 48 45 43)) nn
```

The following expression in the **Frequency** field of our *GenericSource* example adds a random interval to the pitch from the keyboard. A new interval is chosen at random every 0.125 seconds:

```
!Pitch + ((0.125 s random abs * 5 of: #(10 7 5 2 0)) nn)
```

If you hold the key down, you can hear the pitches changing once every eighth of a second, but the sample is not being triggered during that time, so you hear the sequence of pitches trailing off in amplitude as the natural amplitude of the sample dies away.

[§] Note that if an array index has a fractional part, it is always truncated before the index is interpreted as a position within the array. That is why we have to multiply by 5, because numbers like 4.999 will be truncated and give us 4, the last position in the array.

Smoothing Things Over

To filter or smooth out the changes between one Event Value and the next, you can use the `smoothed` or the `smooth:` functions. When you smooth an Event Value, it changes from one value to the next gradually, rather than jumping instantly to the new setting. When applied to pitch, it is something like “portamento” or “slew rate”, but you can apply this smoothing to *any* parameter.

For example, you can add smoothing to the **Frequency** field of the previous example as follows:

```
(!Pitch + (0.125 s random abs * 5 of: #(10 7 5 2 0)) nn) smoothed
```

The default rate of change from one value to the next is 100 milliseconds. To jump more quickly or more slowly between parameter values, specify the time it should take as `smooth: aTime`, for example:

```
(!Pitch + (0.125 s random abs * 5 of: #(10 7 5 2 0)) nn) smooth: 0.5 s
```

SMPTE or MIDI Time Code Synchronization

To trigger Sounds in Kyma from an external source of MIDI time code, use the `triggerAtTimeCode` or the `gateOnAtTimeCodeForDuration:` messages. The Capybara has MIDI inputs only, so you will need to use a device to translate SMPTE time code into MIDI time code.

For example, to generate a trigger at the time code of 1 minute and 12 seconds, you would type

```
00:01:12.00 SMPTE triggerAtTimeCode
```

into the **Trigger** or **Gate** fields of your Sound. To trigger different Sounds at different times, put all of them into a *Mixer*, and set each of their trigger fields to the appropriate start time.

To generate a gate that stays on for a specific duration (rather than a trigger which turns on and stays on indefinitely), use

```
20 s gateOnAtTimeCodeForDuration: 1 s
```

where the first part indicates the time at which the gate should turn on and 1 s indicates how long it should stay on. Notice that you can specify the time in ordinary units, or you can use the SMPTE format showing hours, minutes, seconds, and frames followed by the word SMPTE:

```
hh:mm:ss.ff SMPTE
```

Using Sounds as “Control Voltages”

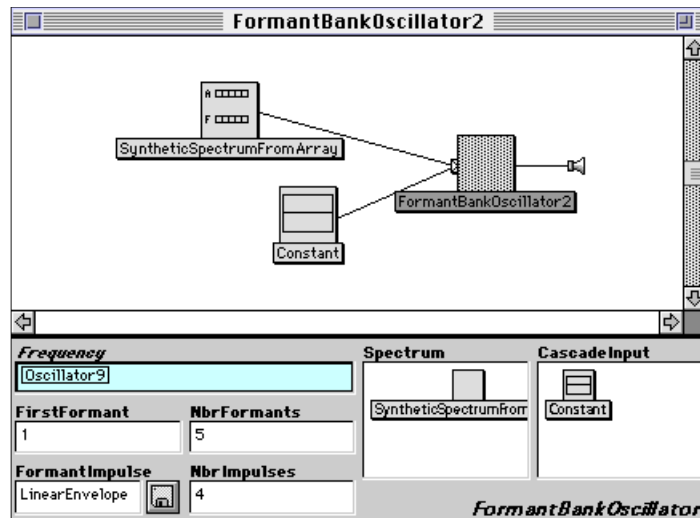
You can also use one Sound to control a parameter of another Sound (similar to controlling one module with the output of another module in a voltage-controlled analog synthesizer). A Sound that is used as a control signal appears in a parameter field as the name of the Sound enclosed in a box.

To place a Sound into another Sound’s parameter field:

1. Select the “control” Sound.
2. Copy it (using **Copy** from the **Edit** menu or **Ctrl+C**).
3. Activate the parameter field by clicking in the field.
4. Paste the Sound into the field (using **Paste** from the **Edit** menu or **Ctrl+V**).

For example, suppose you wanted to slowly shift the frequency of the prototype *FormantBankOscillator*. Drag the prototype *FormantBankOscillator* into your Sound file window and open it. Select the *Oscillator* prototype and copy it using **Ctrl+C**. Then click in the **Frequency** field of the *FormantBankOscillator*, and paste (**Ctrl+V**) the *Oscillator* into the **Frequency** field of the *FormantBankOscillator*.

When you paste a Sound into a parameter field, you see its name enclosed within a box to indicate that it is a control signal.



At this point, the **Frequency** of the *FormantBankOscillator* is the output of the *Oscillator*. The output of a Sound is a kind of “stereo pair” that carries both the left and the right channel output.[‡] To indicate the right channel, use an R (for right) instead. To indicate a mix of the two channels, use an M (for mix).

Sound outputs are in the range from -1 to 1, so we are requesting a frequency that ranges between MIDI note number -1 and MIDI note number 1. This does not make much sense, so let’s add a base pitch and use the variation between -1 and 1 as a deviation from that central pitch. To set the central pitch to second octave C, you could use:

`Oscillator1 L nn + 2 c`

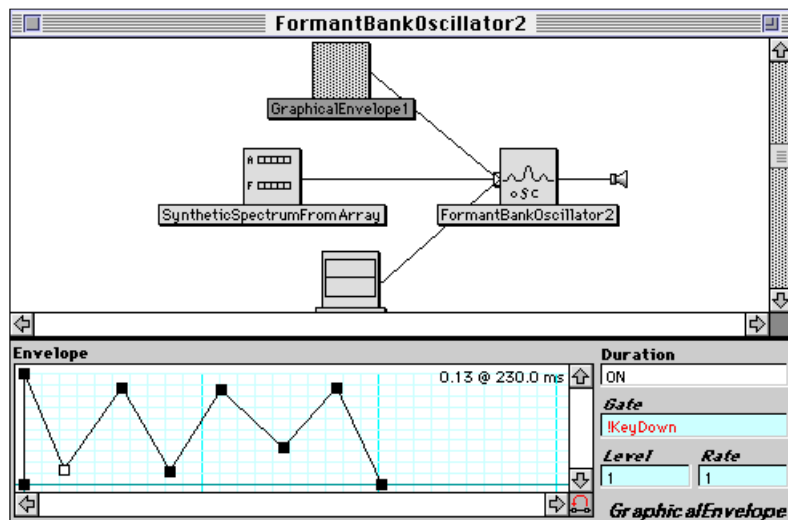
Double-click in a white area of the signal flow editor to force an update. Now double-click the *Oscillator*, and change its frequency to a low value like 1 hz. (Note that the default wavetable for the *Oscillator* is Sine).

Now select and play the *FormantBankOscillator*. You will hear its frequency vary in the shape of a sine wave between a half step below 2 c and a half step above.

Draw

What if you want to draw a shape, rather than rely on the shape of the Sine for your frequency deviation? Replace the *Oscillator* with a *GraphicalEnvelope* (from the **Envelopes** category of the prototypes). Double-click on it to see its parameters. In the **Envelope** field, use shift-click to add several more points and create an envelope that jogs up and down a few times. (Notice that the prototype *GraphicalEnvelope* is triggered by !KeyDown).

[‡] By default, the Sound name is followed by an L (for left), indicating that only the left channel should be used as a control signal.



Now select and play the *FormantBankOscillator*. Each time you press a key on the MIDI keyboard, it will trigger the envelope, and you will hear the frequency vary according to the shape you drew in the *GraphicalEnvelope*.

To make the deviation more extreme, multiply the effect of the *GraphicalEnvelope* by 12 nn. This gives you a frequency deviation of one octave (12 half steps):

`GraphicalEnvelope1 L * 12 nn + 2 c`

I just want to sing

Some of the Sounds in Kyma extract parameters from their inputs, for example, the *AmplitudeFollower* or the *FrequencyTracker* extract the amplitude and frequency, respectively. If you apply one of these modules to the *ADInput*, you can use your own voice (or an instrument) to control parameter values of Kyma Sounds.

Drag an *AmplitudeFollower* from the **Analysis** category of the prototypes, and drop it on top of the *GraphicalEnvelope*.

Select and play the *FormantBankOscillator* and then sing into the microphone. The louder you sing (or yell) into the microphone, the more pitch deviation in the *FormantBankOscillator*. This is one form of “cross synthesis”, controlling the parameter of one Sound with a different parameter extracted from another Sound.

Treating MIDI Events as Audio Signals

By pasting Sounds into parameter fields, you can, in a slightly twisted way, treat MIDI events as if they were audio signals or create shared expressions. The trick is to use the *Constant* Sound.

The *Constant* is used to indicate a value that remains constant instead of varying over time the way most signals do. Paradoxically enough, you can put Event Values into the *Constant's* **Value** field so that the value changes in response to events coming in from MIDI or other sources—in other words, it becomes a “time-varying constant”.

Suppose that you wanted to treat the MIDI pitch as a signal, subjecting it to signal processing operations like delay, attenuation, or filtering before using it to control the pitch of an oscillator. To turn MIDI pitch into a signal, enter

`!KeyNumber / 127`

into the **Value** field of a *Constant*. Why divide by 127? Because a *Constant's* value must be between -1 and 1, and a key number is between 0 and 127.

Now you can feed the *Constant* into a *DelayWithFeedback*, a *HarmonicResonator*, a *PeakDetector*, or some other chain of processing Sounds, and then paste the chain into the **Frequency** field of the *Oscilla-*

tor. Since the value has been scaled to the range of -1 to 1, we have to scale it back to the range of MIDI note numbers and give it some units:

```
FilteredConstant L * 127 nn
```

Another situation in which you might want to use a “time-varying constant” is one in which you have a relatively complicated Event Value expression that is used in several parameter fields, and you would like to treat it as a “global” expression; in other words, if you change it in one place you would like all of the other instances of it to be changed in the same way. To do this, you would enter the expression into the **Value** field of a *Constant*, and then paste the *Constant* into each of the parameter fields to be controlled by the expression.

Update Rate

A parameter that contains a Sound is updated at a rate of once per millisecond (1 khz). If the parameter field contains an arithmetic expression that depends on the value of the pasted Sound, the entire expression will be re-evaluated once every millisecond. If this is faster than necessary, you have the option of slowing down the update rate. After the channel indication, type a colon and the number of milliseconds between parameter value updates. For example to update once every 5 milliseconds, type

```
Oscillator L: 5
```

If you are doing arithmetic with this value, you should parenthesize it to show that the 5 belongs with the *Oscillator*, for example:

```
(Oscillator L: 5) * 12 nn + 4 c
```

Sounds that are used as inputs (rather than controls pasted into parameter fields) are of course updated at the full sample rate (which is 44.1 khz by default, but which can be changed using the DSP Status window, described later in this section).

Algorithmically Generated Events

MIDIMapper and *MIDIvoice* can read Event Values from one of three possible sources:

- real-time MIDI input (from a MIDI controller, sequencer, or other software),
- a MIDI file, or
- a script

If you select the last option, **Script**, then all Event Values to the left of the *MIDIvoice* or *MIDIMapper* will be controlled by events that you generate algorithmically using a program in the **Script** field. This section gives you a little taste of how you can control parameters algorithmically, if you get really interested in this aspect of Kyma, you should study the examples in the file called **MIDIvoice Scripts** and read the section called *MIDI Scripts* beginning on page 522.

Within the **Script** field, you can refer to the *MIDIvoice* or *MIDIMapper* as `self`. Then you can tell it to generate note events or controller value updates.

Virtual Keyboard Events

To generate a note event, specify when the (virtual) key should go down, how long it should be held down, and what its pitch and velocity values are. For example:

```
self keyDownAt: 0.5 s duration: 0.125 s frequency: 60.5 nn velocity: 1.
```

Notice that the pitches do not have to be integer note numbers. You can type in any note number or any frequency you like (which could come in handy if you are interested in alternate tunings).

If you leave off any of the tags, that parameter will take on a default value. So you need only specify the parameters you care about.

Controller Events

There are also methods for setting a controller to a specific value at a specific time and for updating a controller's value over some period of time.

For example, to set controller number 7 to the value 0.5 at time 0, you would type:

```
self controller: !cc07 setTo: 0.5 atTime: 0 s.
```

To cause a gradual increase in its value from 0.5 up to 1 by the time 5 seconds has been reached, you would type:

```
self controller: !cc07 slideTo: 1.0 byTime: 5 s.
```

If you want to make sure that the transition sounds smooth and continuous, you can control the size of the individual steps. For example,

```
self controller: !Frequency slideTo: 1.0 stepSize: 0.1 byTime: 10 s.
```

would use steps of one tenth of a hertz if !Frequency were in units of hertz.

Alternatively, you can specify the number of steps it should take to get from one value to the other. For example,

```
self controller: !Morph slideTo: 1.0 steps: 100 byTime: 3 s.
```

would send 100 updates to the value of !Morph over the course of 3 seconds.

Although it might initially seem that you should always send a huge number of steps or set the `stepSize` to a very small value, there is a tradeoff to consider. If you are using many controllers and each controller is sending many updates, you can start loading down the Capybara with too many events. If the load gets too heavy, it could start interfering with the actual sound generation. So in algorithmic events, as in all things, use moderation.

The Power of Programming

You might well ask yourself at this point, “Why type all that stuff when I can just play the same thing on the keyboard and record it in my MIDI sequencer?” Good question! Because if all you did in the *MIDI-Voice* or *MIDIMapper Script* was to type each individual event, then you *would* be much better off using sequencing software instead.

The real power of the script begins to emerge when you embed these event-generators in some programming control structures. To really see how to do this, you should read about *MIDI Scripts* on page 522. But just to give you a little appetizer, consider this. If you put a single keyboard event into a loop, and change its settings each time through the loop, you can generate *hundreds* of events with just a few lines of code. For example, here is a script that will generate 500 random notes:

```
| r t |

r := Random newForKymaWithSeed: 52.
t := 0.

500 timesRepeat: [
    self
        keyDownAt: t s
        duration: (r next + 1) s
        frequency: 3 c + (r next * 36 nn)
        velocity: r next.
    t := t + r next].
```

and to generate 1000 notes, you would just change the 500 to 1000 (or whatever number you like!)

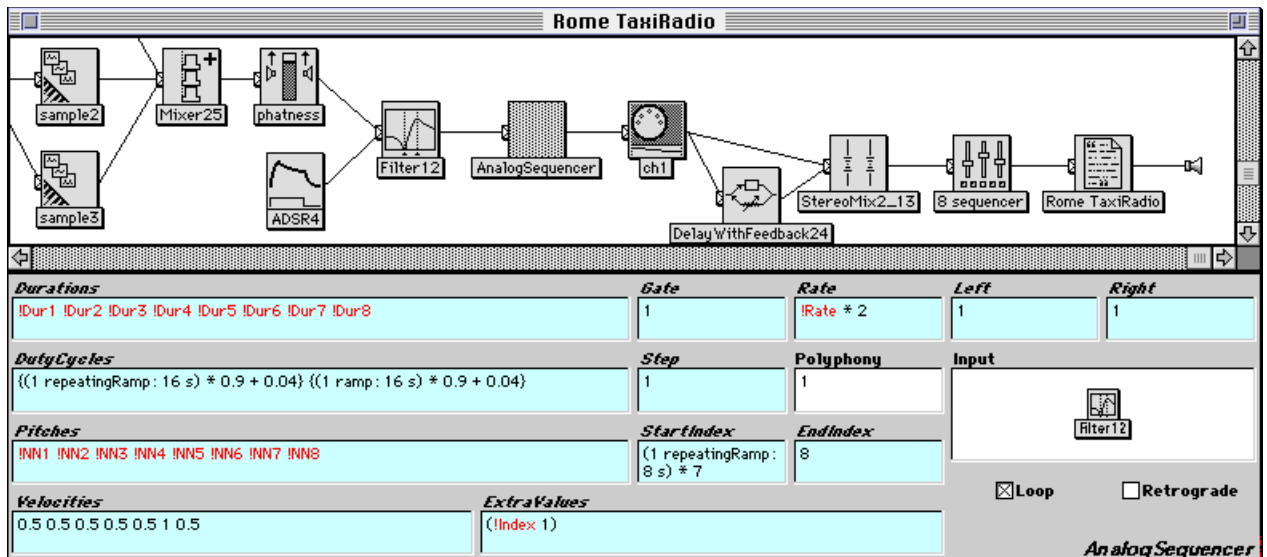
EventCollections

For many kinds of music, it makes more sense to think in terms of notation than in terms of times in seconds. **EventCollections** provide a way to specify notes and rests, rather than start times in seconds. For the full story, see *MIDI Scripts* on page 522.

Analog-style Sequencer

There is more to the “analog sound” than just oscillators, filters and envelopes. Part of that characteristically analog sound emerges from the concept of voltage control and from the kind of sequencers used to send sequences of control voltages to the oscillators, envelopes, and filters.

Kyma’s *AnalogSequencer* Sound generates a sequence of pitches, velocities, durations, duty cycles, plus any number of other sequences of Event Values for controlling the parameters of its input. You can also loop a sequence, go through it backwards, and move the loop points around as the sequence is playing. Any or all of the values in the sequence can be Event Values, meaning that you can alter them from the virtual control surface, MIDI faders, or from another *AnalogSequencer* while the sequence is going.



Like any other Kyma Sound, an *AnalogSequencer* can be mixed with other Sounds or used as an input to another Sound (for example a *MIDIVoice* or another *AnalogSequencer*). Once you start nesting and mixing sequencers, you can quickly end up with complex sequences that, while they do have a pattern to them, also contain a few surprises that can be difficult to predict.

How do I control thee? Let me enumerate the ways...

To summarize, you can control the parameters of a Sound in any of the following ways:

- From the virtual control surface, saving the settings in a *Preset* Sound
- Using MIDI faders, keyboard or other controllers (using the global map and/or *MIDIMapper* Sound to map memorable names to MIDI devices)
- From a MIDI sequencer or other software that outputs MIDI (running on the same computer as Kyma or on another computer connected to the MIDI input of the Cappybara)
- From a MIDI file exported from a sequencer
- From a script in a *MIDIVoice* Sound
- From the *AnalogSequencer* Sound

MIDI Output

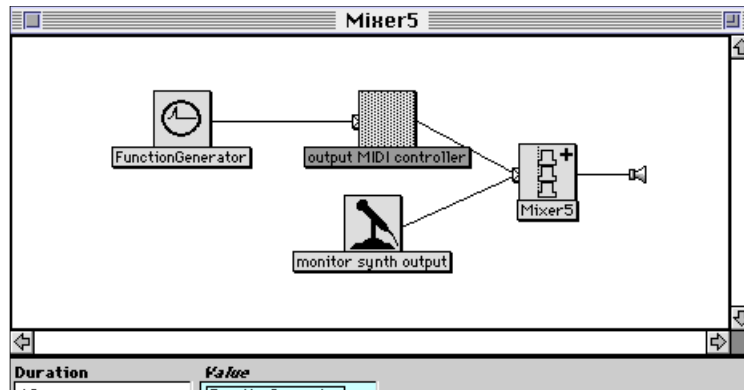
Up to this point, we have been obsessed with control — or at least with how to use MIDI input to control the parameters of Kyma Sounds. Kyma can also be a *source* of MIDI events, and if you hook up your synthesizer or sampler to the MIDI output of the Cappybara, you can control it in some unusual ways using Kyma Sounds.

MIDI Controllers

The output of any Kyma Sound can be used as a MIDI continuous controller output. In other words, its stream of instantaneous amplitudes can be sent out as changes to a continuous controller (at the maximum MIDI rate of 1 khz and with the maximum MIDI controller resolution of 7 bits).

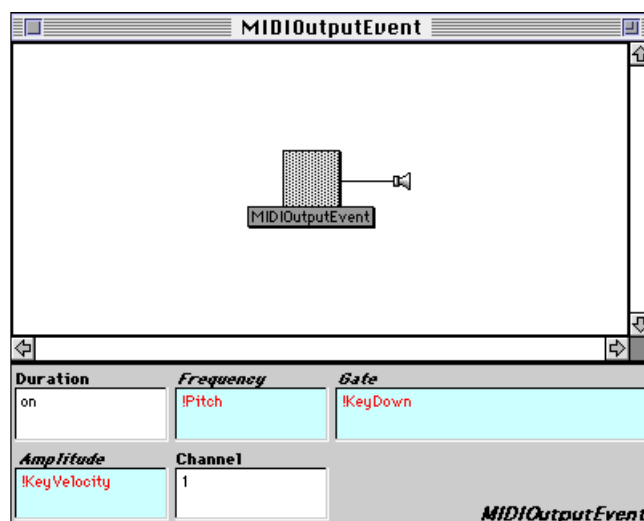
If both the Cappybara and your synthesizer are patched into a mixer, you can listen to both at once. Otherwise, you will need to send the output of the synthesizer into the Cappybara's audio input. Then you use one of the Kyma software *Mixers* to mix the *ADInput* (which will be monitoring the synthesizer) with the Kyma Sound that is sending out the MIDI controller data.[§]

Here is a typical configuration for listening to the output of a synthesizer that you are controlling using a Kyma Sound:



MIDI Keyboard Events

To generate a MIDI keyboard event, you have to supply a pitch, a velocity, and a trigger (which indicates when to send this information as a MIDI note event). As a trivial example, suppose you had a *MIDIOutputEvent* whose **Frequency** was set to !Pitch, whose **Amplitude** was set to !KeyVelocity, and whose **Gate** came from !KeyDown.

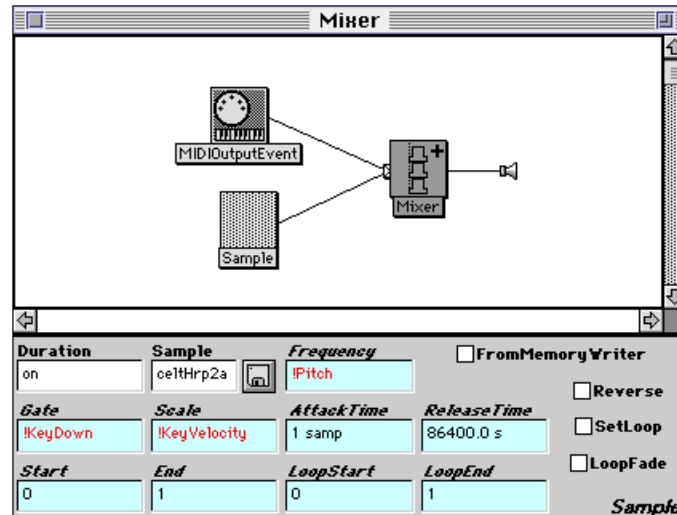


[§] Anytime you want Sounds to play simultaneously in Kyma, you have to put them into a *Mixer* — even if one of the Sounds (in this case the *MIDIOutputController*) is not actually producing an audio signal that you can hear. Even though its output is going to the MIDI output rather than becoming part of the audio stream, its program does have to be loaded in the Cappybara at the same time as the program for reading the *ADInput*. You can read more about this in *Combining Sounds* on page 70.

This would read the pitch and the velocity from the Capybara's MIDI input, and, each time you pressed a key, would output a MIDI note event so it could be read from the Capybara's MIDI output and could control another synthesizer.

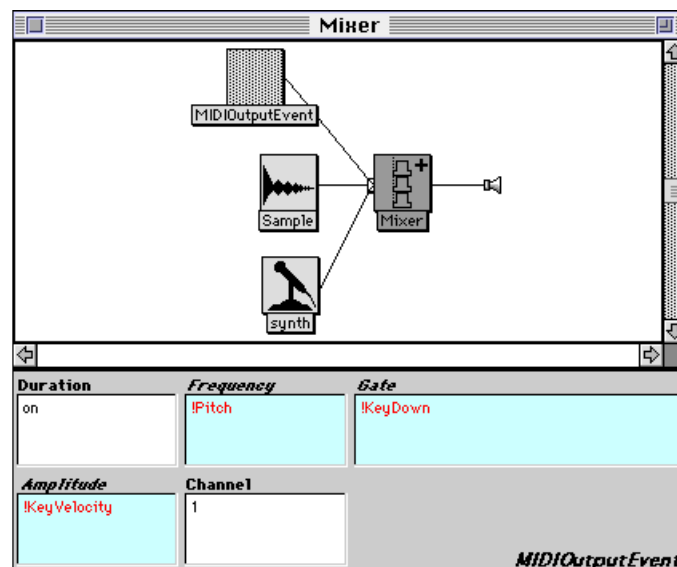
All in all a pretty pointless exercise, since you could have just connected your keyboard directly to the synthesizer in the first place. But let's suppose that you want to control both Kyma and an external synthesizer or sampler at the same time, such that they are doubling each other.

Anytime you want two or more Sounds to happen at the same time in Kyma, you put those Sounds into a *Mixer*. So you could put a *Sample* in a *Mixer* with this *MIDIOutputEvent*, and edit the *Sample* such that it is triggered by !KeyDown and that its **Frequency** comes from !Pitch. Then whatever you play on the Kyma *Sample* will be doubled by any external modules that are connected to the Capybara's MIDI output.



Bringing Audio from an External Module into Kyma

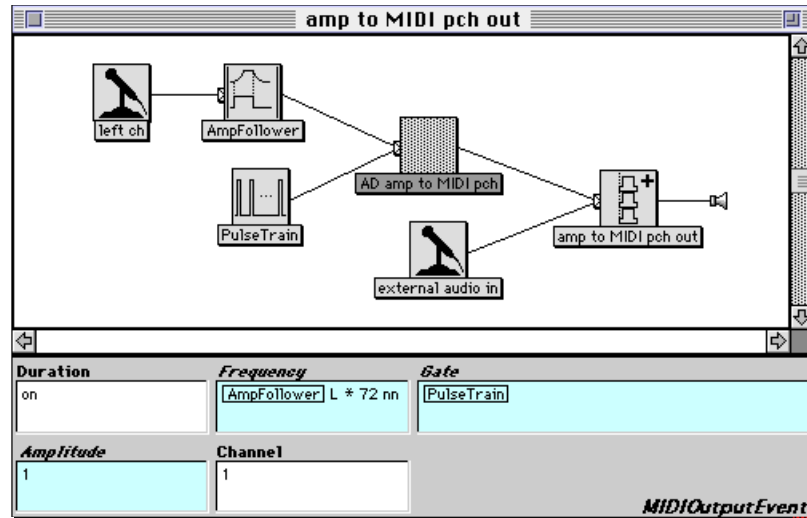
If you have both the Capybara audio output and the external module's output going to your mixing desk, you can listen to both at once. But even if you don't have a mixing desk, you can bring the sound of the external module into Kyma by plugging it into the Capybara's audio input jacks and using a Kyma *Mixer* to mix the sound of the external module with the sounds produced by Kyma. Just drag an *ADInput* module (found under the **Sampling** category in the prototypes) into the *Mixer* that already has the *MIDIOutputEvent* and *Sample* as inputs:



Cross-mapping Parameters

There is nothing to prevent you from using `!KeyVelocity` in the **Frequency** field or `!Morph` in the **Amplitude** field of the *MIDIOutputEvent*. And the fact that you can paste any Kyma Sound into any of the cyan-colored parameter fields means that you can use any Kyma Sound to control MIDI *output* parameters as well.

Here, for example, an *AmplitudeFollower* on the *ADInput* is controlling the *MIDIOutputEvent*'s **Frequency**,[‡] and the *MIDIOutputEvent* is being triggered at a regular rate by a *PulseTrain*.



If you want to send a system-exclusive message to an external MIDI module, you can use the *OutputEventInBytes* Sound.

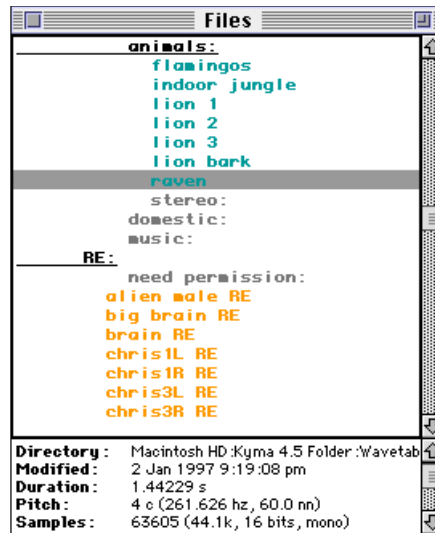
[‡] Values in the **Frequency** field of the *MIDIOutputEvent* are not limited to equal-tempered pitches. Any deviation above or below an equal-tempered pitch will be sent as the closest equal-tempered pitch plus a pitch bend. If you set up your synthesizer or sampler to have a pitch bend range of plus or minus one octave, then the *MIDIOutputEvents* that you send from Kyma can specify the "notes-between-the-keys", allowing for continuous pitch changes and/or alternate tunings.

Other Editors & Info Windows

So far, we have been concentrating on the Sound editor, the prototypes, and the Sound file window. While these are the central editors for sound design, there are some other, ancillary editors that can be of assistance in designing your own Kyma Sounds.

File Organizer

The file organizer helps you use, organize, and keep track of all the sound-producing Kyma files on your disk.



The file organizer is useful for cleaning up your disk, getting rid of redundant or otherwise unwanted files, and getting your Kyma files organized for easy access and searching. You may just want to keep it open all the time, so you can quickly create Sounds based on the files by dragging the files into a Sound file window, rather than starting from a prototype. It is also handy to be able to see the file information and to be able to open an editor by double-clicking on the file name rather than opening an editor from the **File** menu.

Opening and Navigating

Open it by selecting **File Organizer** from the **File** menu. The file organizer gives you a list of all of the files on your disks, color-coded according to whether they are samples, spectra, GA, RE, or MIDI files. The color conventions are:

Item	Color
Unopened disk or folder	gray
Open disk or folder	black underlined
Sample, EX	turquoise
Spectrum	purple
GA	red-brown
RE	yellow-orange
MIDI	green

Double-click on a disk or folder name to hide or show the names of its subfolders or files displayed beneath it, indented according to how deeply nested they are within folders. You can use the arrow keys and the page-up and page-down keys to move around in the list.

Getting Information on the File

To display the information stored in the header of the file, click on the file name in the organizer (or use the up/down arrow keys to select the name of the file). This will display information specific to the file type in the bottom portion of the organizer, including the time and date of last modification and the full path name for the file.

Editing the Files

To edit the selected file, press **Enter** or double-click on the file name. This will open the appropriate Kyma editor or, if you have specified an external editor for this file type,[§] it will open your preferred program for editing files of this type.

Hearing the Files

To hear the selected file, use **Ctrl+Space Bar**. Kyma plays sample files directly. For analysis files (spectrum, GA, or RE), Kyma uses the analysis to synthesize the sound. For MIDI files, Kyma constructs a minimal “orchestra” and plays the notes of the MIDI file on those simple “instruments”.

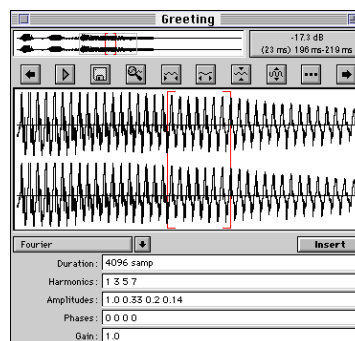
Using the Files in Sounds

To create a Sound based on a file, drag the file name from the file organizer into a Sound file window. For example, if you drag a samples file from the file organizer into a Sound file, it will turn into a *DiskPlayer* whose file name is set to that file. If you drag a spectrum, it will turn into a *SumOfSines* whose **Analysis0** field is set to that file and whose **Duration** and **OnDuration** fields are set to the duration of the spectrum file. Similarly for an RE file, Kyma creates an RE resonators Sound. For a MIDI file, Kyma will create a *Mixer* of several *MIDIvoices*, one for each channel of the MIDI file. Each of these voices has a minimal “instrument” in it, so you can modify it to “orchestrate” the MIDI file score.

To set the value of any parameter field in the Sound editor, you can drag a file name from the file organizer into the parameter field. If it is a parameter that makes sense for the file, the file will paste *its* value for that parameter into the field. For example, if you drag a spectrum file into a **Duration** field, it will set the duration of the edited Sound to be the same as the original duration of that spectrum file.

Sample Editor

Kyma has a basic sample and waveform editor that you can access from the **File** menu using either **New...** or **Open...** (and setting the file type to **Sample file**). In addition to the usual cut, copy, and paste operations on samples, you can use this editor to generate single cycles of periodic waveforms (for use in oscillators), envelope functions (which can be accessed via the *FunctionGenerator* module), and arbitrary tables of numbers (which you can index into via the *Waveshaper* module).



[§] See *File Editors* on page 68 for more information.

Attic

The very top of the sample editor is an overview of the entire file. Drag the mouse across this overview to draw a gray box around any subpart of the file. The region enclosed in the gray box is what you see in the larger waveform display below the line of buttons.

Upstairs

The large waveform view is where you can select, copy, cut, paste, and trim sample files. It shows the familiar representation of a signal as amplitude versus time. To select the entire file, use **Select all** from the **Edit** menu (**Ctrl+A**). Select a sub-section by dragging the mouse across the desired portion of the waveform (selections are shown within large red brackets). If you simply click on one point in time within the waveform, it will mark an insertion point, rather than sweeping out a selection. An insertion point is indicated by a single, vertical red line that blinks like the insertion cursor in a text editor.

Each of the buttons refers to the current, red-bracketed selection or insertion point.



The left and right arrows will jump the display left or right by the size of the selection. The play button plays the selection. The disk button will replace the selection with the contents of a sample file that you select using a file dialog (or, if you have an insertion point rather than a selection, it will insert the entire contents of the other file at that point in this file). The magnifying glass button causes the current selection to fill the window in both the horizontal and vertical direction. All of the buttons with little waveforms and arrows on them either stretch or compress the waveform in the direction of the black arrows. And the ellipsis or *et cetera* button (...) handles some miscellaneous functions. One of the miscellaneous options is to inspect and/or edit the header information in the sample file. The other is a button rather cryptically labeled as **Display Lissajous** which will display the waveform in horizontal direction versus the same waveform delayed by a specified number of samples in the vertical direction.

Downstairs

The lowest portion of the sample editor contains templates for *generating* waveforms or functions. Select a template from the pop up list on the left. When you click the **Insert** button, the selected algorithm will generate a waveform based upon the parameters that you have entered in the template, and if you have a selection in the waveform view it will replace that selection with the newly generated waveform.

The **Fourier** template, for example, has fields for the duration of the waveform, for the number of the harmonics that you would like to add together to create the waveform, for the relative amplitudes and phases of each of the harmonics, and for an overall gain on the entire waveform. To try it, first make sure the entire waveform is selected in the waveform window by clicking in the waveform area and performing **Select all** (**Ctrl+A**), and then click the **Insert** button.

And in the Basement...

By now some of you are probably wondering how to write your own programs to generate arbitrary tables and store them as Kyma wavetables that can be indexed using the *Waveshaper* module. The answer lies deep at the bottom of the list of the templates where it says **Programs**. Here you can find a Smalltalk program corresponding to each of the template waveform generators. Find the program closest to the one you would like to write, and alter it to do exactly what it is that you need. (If you have never programmed in Smalltalk before, you should first read *The Smalltalk-80 Language* on page 513.)

Spectrum Editor

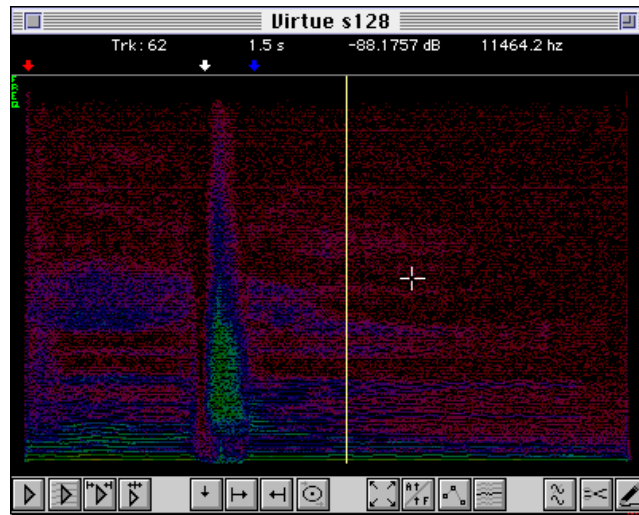
One way to display a sound visually is as amplitude versus time as in the sample editor. Another is to display it as *frequency* and amplitude versus time as in the spectrum editor.

Creating and Opening a Spectrum File

Before you can display a Sound in the frequency domain, you must first create an analysis of the time-domain waveform. The result of this analysis is a spectrum file which you can open and edit in the spec-

trum editor. The analysis can be done in Kyma's Spectral Analysis tool (invoked from the **Tools** menu, see *Tools menu: Spectral Analysis* on page 443), or using the CERL Sound Group's Lemur program.[‡]

Open the spectrum editor on a file from the **File** menu or by double-clicking a spectrum file in the file organizer. (This display looks a lot better in color on the computer screen than it does on the page here.)



The band across the top is reserved for showing marker positions and track information. The large color center area is for displaying the spectrum. The row of buttons across the bottom are for editing and display operations. They are grouped to correspond with the F-keys on your computer keyboard.

This section is intended to be an overview of the spectrum editor. See *Spectrum Editor* on page 487 for a detailed description of all of the features of the spectrum editor.

Spectrum Information

Time is shown from left to right, frequency from bottom to top, and amplitude is mapped to color (the bright colors like yellow and green are the largest amplitude, then the blues and purples, with red (and black) showing the smallest amplitudes.[§] The horizontal lines are referred to as “tracks”. Each track corresponds to the output of a narrow bandpass filter, and you can think of a track as a measure of the amount of energy present in the signal in that one band of frequency over time.

In order to understand how this representation of the spectrum can be used to resynthesize the original sound, think of each of the horizontal lines as representing a “frequency envelope” for a single sine wave oscillator. The change in the color of a track as you trace it from left-to-right corresponds to an amplitude envelope on that oscillator. When you apply each of these frequency and amplitude envelopes to a different oscillator and then put all of these oscillators into a mixer in order to add their results, you will hear something very close to the original signal.

Using the Editor

To display information on a particular track, place the mouse over that track (without clicking down). You will see the track number and the time in seconds corresponding to the mouse location, the amplitude in dB (where 0 dB is the maximum), and the frequency in hertz.

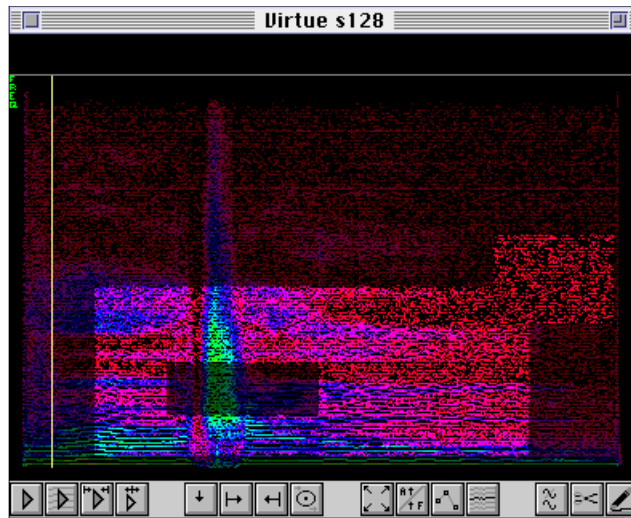
Selecting

To select a single track, click on it. You will hear that track and its color will become brighter to indicate that it is selected. To extend the selection, hold the **Shift** key down while making another selection. To

[‡] Lemur was written by Kelly Fitz, Bryan Holloway, Bill Walker and Lippold Haken, and is available from <http://datura.cerl.uiuc.edu>.

[§] There is a button that lets you switch to showing amplitudes on the vertical axis rather than the frequency envelopes, primarily for editing the amplitude envelopes.

select a region that includes several tracks, use the mouse to draw a box around the region. You can extend boxed selections in the same way that you extend single track selections, by holding the **Shift** key down when you make the new selection. You can also make selections based on a number of amplitude or frequency criteria.



Playing and Scrubbing

Anytime you make a selection, the selected tracks or portions of tracks are immediately played so you can hear what you just selected. There are also buttons to play the entire spectrum file, the part of the file that is selected, and the part of the file between the start and end markers.

To hear different sections of the file, use the mouse to drag the vertical yellow scrub bar over the regions you would like to hear. You can also use MIDI pitch bend to “scrub” (from the old practice of controlling the speed at which analog tape passed by the playback heads by turning both reels with your hands). You can step the scrub bar one frame to the left or one frame to the right (and hear the spectrum at that one frame of time) by using the left and right arrow keys.

Clearing and Deleting

There are two ways to “remove” material from the spectrum. One is to simply set the amplitude of the track to zero for all or part of its duration. This effectively removes it from the mix so it has no effect on the resynthesis during those times when its amplitude is zero. The other is to actually delete it from the analysis file.

In the spectrum editor, zeroing the amplitude is called “clearing”, and removing entire time segments is called “deleting”. Clearing is used to zero the amplitude of all or part of a track’s duration, and deleting is used purely in the same sense that it has in a waveform editor — that of deleting a segment of time from the file.

Extracting Portions of a Spectrum File

You can copy a region and paste it into a Sound file window, where it will show up as a *SumOfSines*. You can then time-stretch and/or frequency-scale that region of the spectrum and end up with a complex timbre that does not necessarily sound anything like the original analysis from which it was extracted. By putting several of these *SumOfSines* into a *Mixer*, you can construct dense textures from bits and pieces of several spectra.

Modifying

The amplitude and frequency envelopes associated with each track can be modified by redrawing them or by applying a “filter” or algorithm to all tracks within the selected region.

Text

There is not that much to say about the text file editor other than it exists and it is a typical text editor, and you can create or edit text files by choosing **New...** or **Open...** from the **File** menu and selecting the file type named **Text**.

You can use the text file editor to create sets of data to be read by the *TextFileInterpreter*, which can map datasets to Sound parameters. You can also use a text file window to test out Smalltalk expressions; type in the expression, select it, and use **Ctrl+Y** to evaluate the expression.

Preferences

You can set some interface preferences by selecting **Preferences** from the **Edit** menu. These are decisions about the appearance or performance of various aspects of the user interface that, once set, will be saved in a file called **Kyma Preferences** or **kyma.pre**. Because they are saved in a file, Kyma will remember them each time you start up the program, so you won't have to reset them each time.[§]

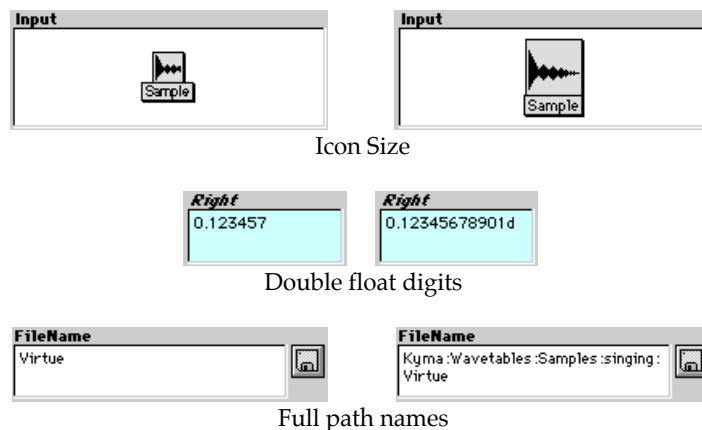
Multi-user Systems

When more than one person is using a single Kyma system, for example in a school, or a large studio or lab, each person can have a unique preferences file.

When you are setting up the system for multiple users, locate the **Kyma Preferences** or **kyma.pre** file (automatically created the first time you quit Kyma). Create a folder for each Kyma user, and drag a copy of the preferences file into each folder. Instruct the users that they should start Kyma by double-clicking the preferences file in their own folders. From then on, any changes that individual users make in their preferences will be saved only in their preferences file and will not alter the system for everyone else.

Appearance

You can decide whether you would like the default icon size in a parameter field to be small or large, whether you would like to display full file path names in parameter fields, and whether you would like to show all the digits of any floating point double-precision numbers.



Performance

Disk Recording/playback and the Graphic Interface

To be able to stop disk playbacks the same way you stop other Sounds (**Ctrl+K**), you should check the box labeled **Update display during disk operations** in the performance preferences.

[§] If your system seems to suddenly start acting strangely, the *first* thing you should try is to throw away your preferences file, and restart Kyma. The preferences file contains information about the basic state of your system, so if that file should happen to have an error in it, it will cause problems in Kyma. Don't worry about throwing away a preferences file, because Kyma will generate a new one for you the next time you quit the program.

When checked this control also means that other “animated” parts of the user interface, like the DSP status window, the virtual control surface, the oscilloscope, and the spectrum analyzer will continuously update even during disk recording and playback.

The only reason you might have for not checking this option is if you experience problems (like clicks or stuttering) in your disk recordings. By not checking the box, you are telling the host computer to devote its full attention to the disk and not to bother with periodic updates to the interface.

Timecode Frames per Second

The **Timecode frames per second** value is used to convert any times you specify in units of SMPTE into seconds. 24 fps is the film industry standard, 25 fps is for PAL (used in most of Europe) and SECAM (used in France and the former Soviet Union) video, and 30 fps is associated with NTSC video (the standard in the Americas and Asia).

This frame rate is used *only* for converting SMPTE to seconds in the user interface. The Cappybara, when it receives MIDI time code, actually keeps track of time in half-frames so it can also do the drop-frame standard associated with color video.

Optimize Sample RAM

If this box is unchecked, each sample requested by your Sounds will be loaded onto *all* the expansion cards. In effect, this is the same as saying that you only have as much sample RAM as you have on a single expansion card.

If you *check* the box, it forces Kyma to load each sample *only* into the memory of the expansion cards that use that sample. So by checking the box, you effectively multiply the amount of sample RAM available by the number of expansion cards in your Cappybara.

The only reason you might have for *not* checking this box is that optimizing the use of the sample RAM can slow down the compile step when you **Compile, load, start** a Sound that has samples in it. If you use samples only rarely, you may not have any need to check this box, but otherwise, you should leave it checked.

Virtual Control Surface

By default, a virtual control surface will automatically open any time you play a Sound that uses Event Values. If you do not check this box, the virtual control surface will open only if you request it from the **File** menu.

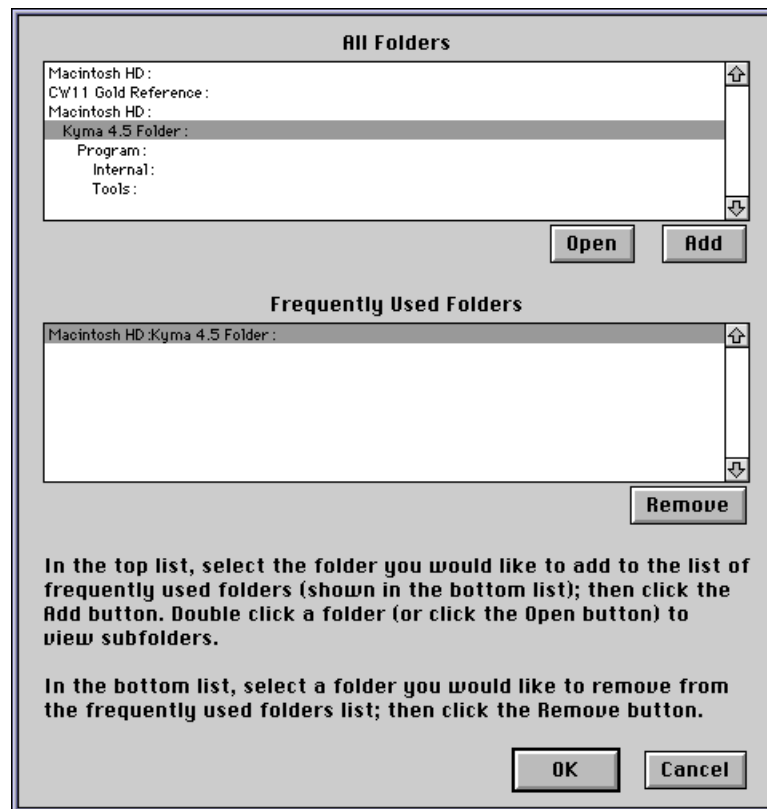
File Editors

If you have a favorite external editor that you would prefer to use for particular file types, check those file types here. For example, if you would prefer to edit SD-II files in your favorite wave editor rather than using the wave editor that comes with Kyma, you would check the SD-II box. The next time you try to edit an SD-II file, Kyma will ask you to find the application. From that point onward, Kyma will remember that it should find and start up your external application every time you edit a file of type SD-II.

Frequently Used Folders

Use this dialog to locate the folder(s) Kyma should search when locating files requested by your Sounds.

Select the folders in the top list (using **Open** to see inside a folder) and add them to the list below using the **Add** button.



If you are frequently seeing the dialog that says Kyma could not yet locate a file, come here and make certain that the samples, MIDI files, and other external files that you frequently use are in folders in this list of frequently used folders. That should speed up the process of searching for these files when you compile Sounds that refer to them.

Spectrum Analyzer

Use this to set the window length and window type for the spectrum analyzer that you can invoke from the **Info** menu. Note that this does *not* have any affect on the *LiveSpectralAnalysis* Sound, the *SpectrumAnalyzerDisplay* Sound, the *FFT* Sound, or the Spectral Analysis Tool.

Miscellaneous

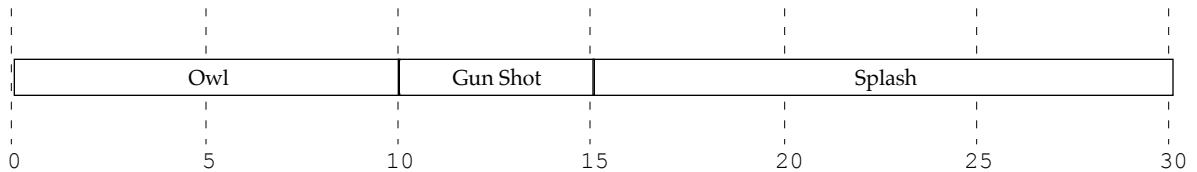
When you start up Kyma as a new user, there is a help window telling you how to get some sound out of the system right away (in case you might have been a little too eager to fire up the system and hadn't read the manual yet). After the third or fourth time of seeing this window, the astute user may decide that it is no longer necessary to read this message. As part of your reward for reading this far into the manual, you now know that you should uncheck the box that says **Show Help on startup**.

By default, Kyma will open an untitled Sound file window each time it starts, giving you a workspace for creating and saving Sounds. If you prefer to use existing Sound files as starting points, and find yourself always closing the untitled window to get it out of the way, you should uncheck **Show untitled window on startup**.

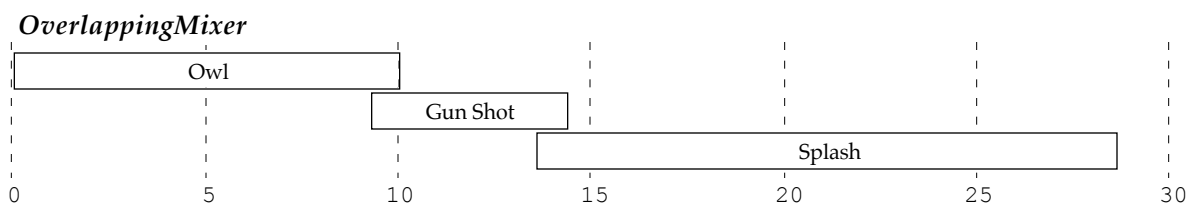
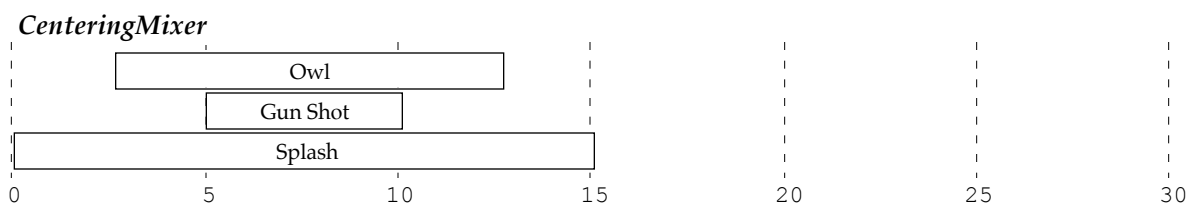
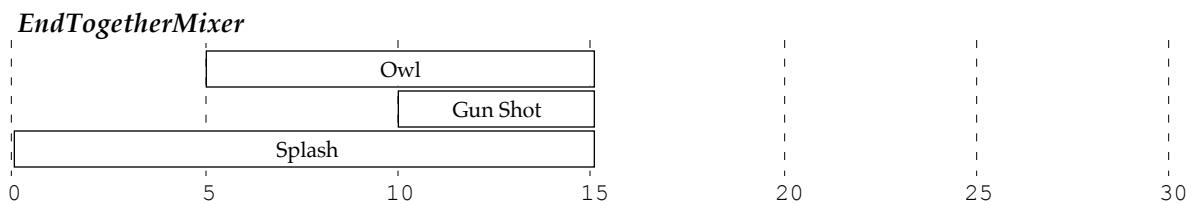
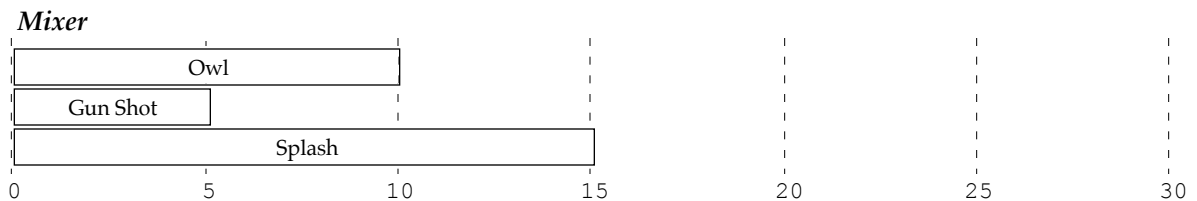
If you run into a problem, and you call the support team at Symbolic Sound, we may ask you to check the box that says **Allow stack traces**. This option gives you more verbose (though most likely impenetrable) error messages that may help us help you track down the problem. Most of the time, though, you really don't want to see this cryptic, low-level stuff, so you should leave it turned off.

Combining Sounds

If you want several different Sounds to occur one after another, assign each of them a finite duration, and put them into a **Concatenation** (concatenate means “chain with”). This is something like splicing together Sound objects on tape or in a hard disk editor.



If you want two or more Sounds to happen simultaneously, put them into a **Mixer**. A **Mixer** adds its inputs together, just like a physical mixing console, and just as the sound waves from different sources add in the air. There are several flavors of **Mixer** in Kyma: **EndTogetherMixer** forces its inputs to end at the same time, **CenteringMixer** lines up the midpoints of all its inputs, and **OverlappingMixer** is like a **Concatenation** except that it overlaps the end of one Sound with the beginning of the next by a specified amount of time.



If you want the Sounds to overlap with each other in time but want some of the Sounds to start later than others, you can use a **TimeOffset** to delay the start time of each Sound. The **TimeOffset** delays the start of a Sound by whatever amount of time you specify in the **SilentTime** field.

Algorithmic Splicing and Mixing

Scripts

To algorithmically specify when Sounds should start and how they should overlap in time, use the *Script* Sound, found in the **Algorithms** category of the prototypes. The *Script* algorithmically constructs a *Mixer* of Sounds within *TimeOffsets* according to a script. Each line of the script has the format

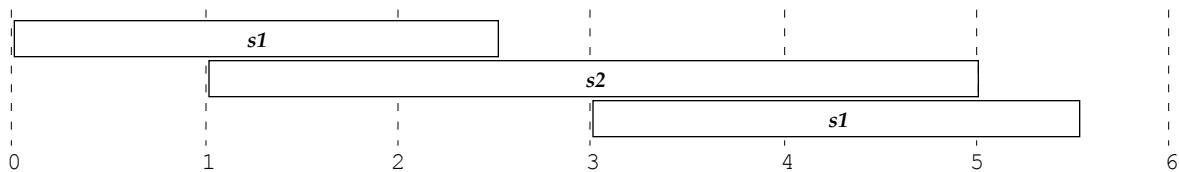
```
soundName start: aTime.
```

In other words, you type the name of an input, the word `start:`, and a time when a copy of the Sound should start. Each of the inputs is used as a kind of model or template for creating a copy, so you can create any number of copies of each input, each one starting at a different time and even overlapping with each other in time.

For example, the script,

```
s1 start: 0 s.  
s2 start: 1 s.  
s1 start: 3 s.
```

would result in the following timeline:



You can extend the idea of templates even further by making some of the parameters of the input Sounds variables instead of fixed constants. A variable is a word preceded by a question mark (and shows up in green in the user interface). Then you can set the values of the variable parameters from the script. The format for setting a variable parameter from the script is:

```
soundName start: aTime varName1: value1 ... varNameN: valueN.
```

In other words, after assigning the Sound a start time, list the name of each variable followed by a colon and the specific value that you would like to assign to that variable. For example, suppose you had a Sound whose **Frequency** field was `?freq` and whose **Gain** field was `?amp`. You could use the following line in a script to create a copy of that Sound whose frequency was 443 hertz, amplitude was one half, and start time was 3.2 seconds:

```
aSound start: 3.2 s freq: 443 hz amp: 0.5.
```

As it turns out, these scripts are actually programs in the Smalltalk programming language. This can come in very handy in cases where you have some systematic description of the Sound you want to create.

For example, say that you wanted to construct some new, strange kind of reverberation based on the *HarmonicResonator*. You decide that you would like to create a *Mixer* of six of these resonators, each with a slightly different resonant frequency. You could get a *Mixer* from the prototypes, open it up and drag six copies of the *HarmonicResonator* one-by-one from the prototypes into the *Mixer*, then alter each of the six **Frequency** fields one-by-one. Then if you wanted to adjust those settings, you would have to open each of the six one-by-one in order to test out a different set of **Frequency** values. Or you could instead leverage the power of programming and create those six copies in a loop:

```
1 to: 6 do: [ :i |  
    res  
        start: 0 s  
        freq: (0.5 * i + 25) hz].
```

Then you could change the number of resonators simply by changing the 6 to a 10 (or whatever). And you could change the relationship between the frequencies, simply by changing the expression following the `freq:` message.

If you are excited by the prospect of creating Sounds algorithmically, see *Scripts and FileInterpreters* on page 525, and *The Smalltalk-80 Language* on page 513.

Sounds vs. Events

It is important, at this point, to make a clear distinction between the *Script* Sound and the script parameter of a *MIDIvoice* or *MIDIMapper*.

In the case of the *MIDIvoice*, you have a single Sound that is loaded on the Capybara, and the script is generating Event Values that will update parameters of that Sound. It is like playing notes on a single instrument.

The *Script* Sound, on the other hand, creates a new Sound — a *Mixer* with several inputs (each of which may be optionally time-offset). Once this mixer Sound has been created, it is then loaded into the Capybara. The *Script* is a way to algorithmically construct Sounds, similar to the kinds of things you do in the graphic user interface: dragging new Sounds into a *Mixer*, setting their values, putting a *TimeOffset* on each one to delay its start time relative to the others, *etc.*

A *Script* in a *MIDIvoice* is like sending MIDI events to a patch on your synthesizer, whereas a *Script* is like actually creating a new synthesizer. In fact, it makes perfect sense to have a *Script* as an input to a *MIDIvoice*. The *Script* would create a Sound and could even place Event Values in some of the parameter fields algorithmically. The *MIDIvoice* would supply events that would change those Event Values over time.

In other words, the idea of the *MIDIvoice* is to supply parameter value updates to an *existing* architecture, while the idea of the *Script* Sound is to algorithmically create a *new* architecture. It would not be a very good use of the *Script* to treat it like a “score”, generating hundreds of note-events for the same “instrument”, because the *Script* would actually create hundreds of copies of the instrument, each set up to play one single note — like a bell choir. The idea of hundreds of notes played on a single instrument fits the *MIDIvoice* model much better — one instrument, lots of notes. (This is not to say that you *cannot* use the *Script* as a score — only that it is not as efficient as doing the same thing with a *MIDIvoice*). On the other hand, if you want to create something that changes from a sampler, to a synthesizer, to a disk playback, or if you wanted to create some music in, say, three large sections each with a different set of “instruments”, then the *Script* would be the most appropriate way to implement it.

Specialized Algorithmic Sounds

In addition to the *Script*, there are several Sounds in the **Algorithmic** category that implement specific algorithms for putting together new Sounds:

CellularAutomata

ContextFreeGrammar

RandomSelection

Repetition

Substitution

Any one of these could have been written instead as a script for the *Script* Sound. The *Script* is the most general solution for constructing Sounds algorithmically, but in some instances it is easier to use one of these more specialized Sounds. See *Prototypes Reference* beginning on page 218 for details on how to use each of these algorithmic “sound constructors”.

Live green, see red, feel blue

At this point, it is probably worth reiterating what the different colors mean when you see them in parameter fields or in global or local maps.

?Green Variable

This is used as a kind of place holder or variable in a parameter field of a Sound. It always begins with a question mark and a letter and can be followed by any number of letters or numbers. It must be associated with an actual value at some point before the Sound is loaded into the Capybara. If you do not set

the value of a variable in a *Script* Sound somewhere to the right of a Sound that uses the variable, then Kyma will ask you to supply a value for the variable when you try to play it. You can also use green variables to indicate which parameters should be visible when you encapsulate a complex Sound. See *The Class Editor* on page 536 for more information on how to create new Sound classes.

Blue Event Source

You will come across the blue Event Sources only in the context of the global map or *MIDIMapper*. These names represent the *sources* of Event Values, not the values themselves. In other words, ``MIDI-Controller07` would represent a source of input; whereas `!Volume` represents the current value of that input.

!Red Event Value

The red Event Values are the memorable names that correspond to the blue Event Sources. You use the Event Values in parameter fields to indicate values that are to be controlled in real time.

The source of the Event Value is defined in global map or local *MIDIMapper*; its value is supplied from MIDI, the virtual control surface, Tools, or any of the Sounds in the **Event Sources** category. When the value of an Event Value changes, it causes updates to any Sound parameters that use it.

Who's on top?

One of the things that makes Kyma different from other languages is that, because Kyma is based on the notion of “sound objects”, rather than on the model of “notes” played on an “instrument”, there is no strictly enforced hierarchy with a master-score or sequencer up at the top controlling everything below it. There are several “score-like” modules, that is, modules that can generate events or that can create Sounds that are offset from each other in time, but these modules are Sounds that can be used in place of any other Sound.

Even the score-like modules can be concatenated or mixed with each other or nested within each other. In other words, Kyma does nothing to prevent you from using one score to set and control the parameters of another score or even of several other scores. So you can create a Sound where the “score” is distributed in modules throughout the signal flow diagram.

This does not mean that there is anything in Kyma to *prevent* you from constructing a straightforward model that perfectly mimics the behavior of more familiar paradigms.

The subtly powerful thing about Kyma is that it encompasses those familiar paradigms while still leaving a door open for you to experiment with new paradigms — with new ways of thinking about sound and structuring sound, ways that could *only* be possible (or at least practical) because you are using a computer.

As an artist and/or researcher you are familiar with the process of dancing on that edge between tradition and innovation, and know that new ideas can be introduced as extensions to familiar or proven concepts. So, we invite you to push at the edges of what is known or what is commonly accepted! And let us know where Kyma helps or hinders you in that regard.

Well, enough of this little philosophical interlude, now let's take a look at the nuts and bolts of how your Kyma system turns graphic signal flow diagrams into actual sound that you can hear.

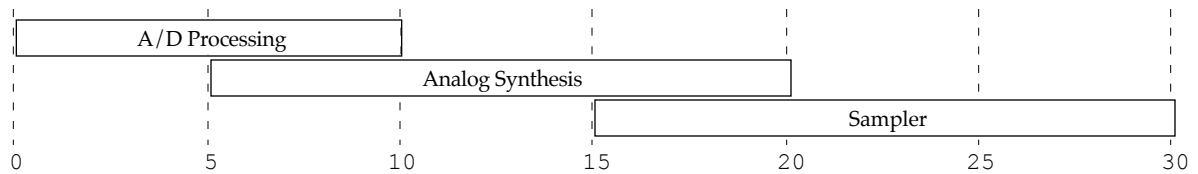
Dynamic Sound Objects

You may have been wondering why so many of the Sounds in Kyma have **Duration** parameters, especially since, in most of the examples, we have been setting the **Duration** to `on`. It turns out that if you know in advance when you want a Sound to be available and when you no longer intend to use it, Kyma can use this information to optimize the use of the processing power of the Cappybara. In other words, you can, in effect, describe a piece of hardware that changes over time.

Suppose for example that you wanted the Cappybara to process your voice for 10 seconds, and that you wanted to begin playing a synthesizer along with the voice during the last 5 seconds. Then you wanted to

continue playing the synthesizer from your MIDI keyboard and have it gradually cross fade with some sampled sounds of flamingos squawking.

In effect, you would like the Capybara to change from an effects processor, to an analog synthesizer, and finally into a sampler at the end, and you would like these hardware phases to overlap in time. Here is a time line sketch of the same thing:



If you specify this as a *Mixer* with *TimeOffsets* on the synthesizer and the sampler, and you give each section a finite duration, then Kyma can schedule the use of resources on the Capybara in a more optimal way:

Time	Sound
0 - 5 s	<i>A/D Processing</i>
5 - 10 s	<i>Mixer (A/D Processing, Analog Synthesis)</i>
10 - 15 s	<i>Analog Synthesis</i>
15 - 20 s	<i>Mixer (Analog Synthesis, Sampler)</i>
20 - 30 s	<i>Sampler</i>

We should reiterate that this is analogous to the hardware itself evolving over time; it is not analogous to sequencing or to the notes of music notation. It is saying that the synthesizer does not exist before it is time for you to begin playing it, that it exists for 15 seconds, and then goes out of existence once you have finished playing it.

This is part of the power of software synthesis — that it allows you to reallocate computing cycles and allocate them toward whatever algorithm you like. You can (literally in the space of one tick of the sample rate clock) switch from using all of the Capybara’s resources as an effects processor to using all of the resources to implement a polyphonic subtractive-synthesis synthesizer. The resources are generic; they are things like how much memory is available and how many multiplications and additions can be done in one sample clock (22 microseconds). Kyma can just as easily use the memory and the arithmetic unit to implement a filter as it can to implement an oscillator or read a sample stored on the hard disk.

Part of the beauty of software is that everything is reconfigurable and modular, and by manipulating symbols and graphic icons, you are actually manipulating electrons flitting across metal pathways at nearly the speed of light, not physical masses that require manufacturing, materials, and space. The fact that it all ends up producing sound in the air is a little bit astounding when you stop to consider it. But we can’t spend too much time feeling astounded by computers and software, since you are probably getting anxious to get to work and there are still a few more things to cover in this overview — like how can we control time?

Time flies like a banana

The Capybara would not be able to implement the time-varying hardware example from the last section if it were not keeping track of time; that is how it knows when it is time to stop one Sound and begin computing the next one.

There are three Kyma modules that allow you to take some control over time on the Capybara: *TimeController*, *TimeStopper*, and *WaitUntil*.

The input to a *TimeStopper* module is loaded into the Capybara, but the time clock is stopped until the value of the **Resume** parameter becomes positive. For example, even if the input to the *TimeStopper* had a duration of 1 *samp* (the shortest duration possible), it would last forever unless the value of **Resume** changed from 0 to 1 at some point.

TimeController is a little less extreme; it merely changes the *rate* at which time passes on the Capybara. If you set the **Rate** to 0, however, it too will stop time.

The input to a *WaitUntil* module will *not* be loaded into the Capybara *until* **Resume** is triggered; then the input plays through at its normal rate and for its normal duration.

These modules affect the time-counter that the Capybara uses in order to determine when to start/stop Sounds and when to issue events.[‡] Thus, they have an effect on things like *Concatenations* and *TimeOffsets*, and MIDI files or scripts in the *MIDIvoice*. They do *not* have any effect on the sampling rate. For example, a *TimeController* would have no effect on the speed of playback of a *DiskPlayer*, but if that *DiskPlayer* were concatenated with an *Oscillator*, a *TimeController* could delay the time at which the *DiskPlayer* ended and the *Oscillator* began.

Suppose you have a situation (say a live, musical performance or a psychoacoustical experiment) where you know the ordering of the Sounds, but you would like to give the performer (or the subject) control over exactly when the Capybara switches from one Sound to the next. If you put each of the Sounds into a *WaitUntil*, and then feed all of them into a *Concatenation*, you can control the exact time at which each Sound is loaded into the Capybara, according to the conditions you place in the **Resume** field. The change from one Sound to the next is instantaneous (to within a single sample-tick), because all of the Sounds have already been compiled and loaded into the memory of the Capybara and are just waiting for their chance to get loaded and start playing!

For example, say you had three sections, each with a different kind of processing on the *ADInput*. You would like the performer to have control over when to start and when to move on to each succeeding section. The first *WaitUntil*'s **Resume** field might have a MIDI switch in it, so the performer could start things by depressing a foot-switch or by pressing a button on a controller. The next *WaitUntil* might have a test on the output of a *FrequencyTracker* on the *ADInput* so that the next section would not begin until the performer sustained a 5 c. The last *WaitUntil* might have a test weighing the strength of the higher partials relative to the strength of the lower ones in order to detect when the performer speaks an "s" (or to detect an attack transient of a musical instrument).

Time (and time again)

When we speak of time in Kyma, there are actually several different time scales that we could be talking about:

- The rate at which the structure is changing.

- The asynchronous Event Values that could be coming from MIDI or generated internally.

- The rate of Sounds pasted into parameter fields as control signals.

- The sampling rate, the rate at which the audio signals are computed.

The structure-changing rate is the rate at which old Sounds may expire and new ones take their place due to *Concatenations* or *TimeOffsets*. These changes take place "between the samples", that is, there is no delay between terminating one Sound and loading/starting the next. In order for this to work, though, all of the changes must be specified in advance.

Event Values are asynchronous updates to the parameter values of the Sound that is currently loaded on the Capybara. These could be coming in live from MIDI devices, a MIDI sequencer, generated by the *AnalogSequencer*, or they could be read from a stored sequence like a MIDI file or the sequence generated by a script in a *MIDIvoice*. In all cases, though, the input is not necessarily periodic and it is not always known in advance when an event might come in. The maximum rate of change for these asynchronous events is once per millisecond, corresponding with the maximum rate of transmission for the MIDI interface.

Sounds pasted into parameter fields are often used in expressions that include Event Values, so the maximum rate of LFO-like control updates is also set at once per millisecond (or 1 khz).

[‡] The Event Values called **!Time** (time in seconds since the start of the entire Sound) and **!LocalTime** (time in seconds since the start of the Sound referencing **!LocalTime**) are also affected by these modules. **!RealTime** (time in seconds since the start of the entire Sound) is not. These Event Values can be read only, not written.

The sampling rate can be set to 48, 44.1, 24, 22.050, or 11.025 khz using the DSP Status window. This is the rate at which audio signals are updated on the Capybara.

Randomly Accessing Sounds

Suppose that you have several different Sounds and that you would like to be able to start and stop them at will, in any order? For example, you might be accompanying a live performer, in which case you would want to be able to synchronize the start and end times of synthesized sounds, samples, and/or processing of the live input. Or you might be setting up a psychoacoustic experiment in which you want to randomize the order in which the stimuli are presented to a subject.

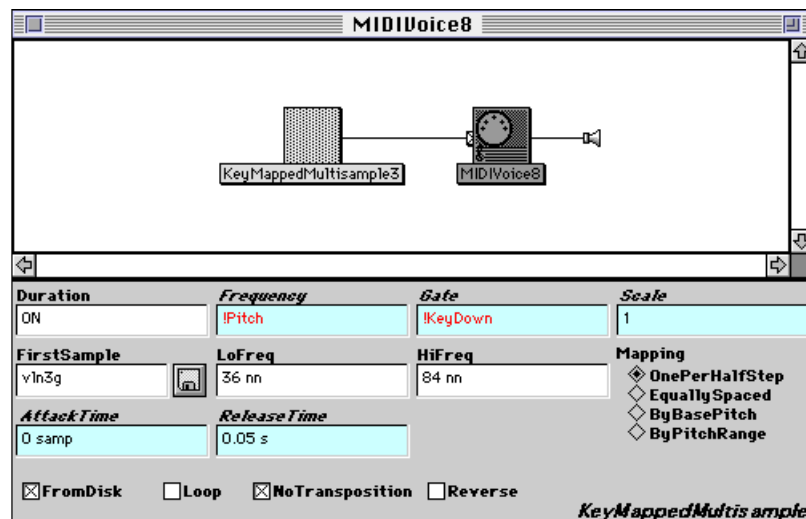
There are two approaches to solving this problem, depending on the amount of processing time you have available, how quickly the transitions must be between Sounds, and whether you require cross-fading between the Sounds.

Mixer model

If you would like any possible succession of Sounds with cross-fading between them, you must put an amplitude envelope on each of the Sounds, and place all of them into a single *Mixer*. If you assign each Sound a unique trigger, you will be able to trigger any Sound at any time and they will overlap by the duration of the decay of the first one and the attack time on the second. This presupposes that you have enough computing power to compute all of these Sounds simultaneously.

The Special Case of Samples and Disk Recordings

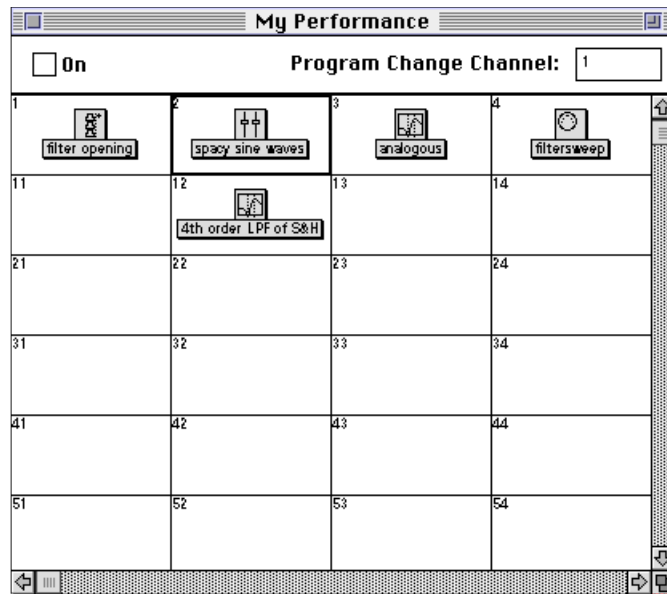
If all of the Sounds that you want to trigger happen to be either disk tracks or samples, there is an alternate solution. Using a *KeyMappedMultisample*, you can trigger any of several disk files (or samples) as long as they are all stored in the same folder on your disk. Using a *MIDIvoice* or *MIDIMapper* to define polyphony, you can specify the number of disk files that could be sounding at one time. If all you want is a simple crossfade between disk files, for example, you can get by with a polyphony of 2.



Compiled Sound Grid

If you do not have to crossfade between Sounds, you can use a compiled Sound grid for the purpose of randomly accessing Sounds. The idea behind the compiled Sound grid is that you do the first step of the **Compile, load, start** process — the compile step — to all Sounds in the grid at once, and save the results on disk. Then you can do the final two steps — loading and starting — at the moment you want to actually play the Sound. Because you have done the most time consuming part of the processing ahead of time, you can load and start arbitrary Sounds in an arbitrary order much more quickly.

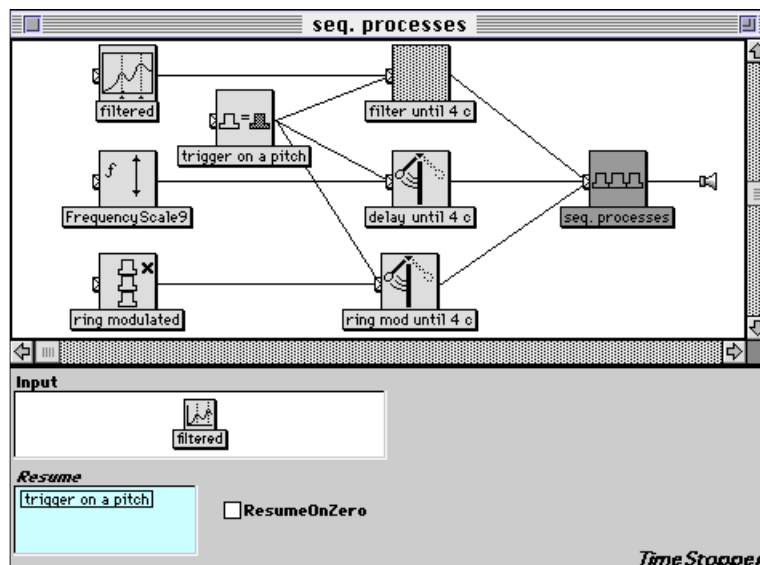
To play a compiled Sound, tab into or click in its box. Alternatively, you can send the MIDI program change number corresponding to the number in the upper left corner of the Sound's box.



The compiled Sound grid is useful when you can afford a little bit of silence between downloading different Sounds. For example, if each Sound corresponds to one section of a musical performance or one set of psychoacoustic stimuli. Downloading is fast but it is not instantaneous, and you cannot crossfade between the currently loaded Sound and the new one you are loading.

The Special Case of Sequential Ordering

If you know the order of Sounds ahead of time and would like to control the time at which one Sound ends and the next one begins, you can use a *Concatenation* in which each Sound is nested within a *WaitUntil* or *TimeStopper* Sound. In this configuration, each new Sound is loaded instantly, with no delay, since the Sounds in the *Concatenation* are stored in the RAM of the Capybara.



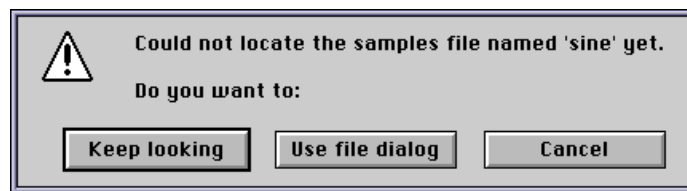
Compile, load, start

How does Kyma take the graphical representation of the signal flow (a Sound with an uppercase “S”) and turn that into actual sound (lowercase “s”) that you can hear? When you select **Compile, load, start** from the **Action** menu (or, more expediently, use the key equivalent of **Ctrl+Space Bar**), Kyma compiles the Sound, loads it into the Capybara, and tells the Capybara to start its time-counter.

Compile

First, Kyma rewrites the graphical representation of signal flow into a sequence of time-tagged instructions to the Capybara. At the same time, it also compiles any parameter field expressions, so that they can be evaluated in real-time by the real-time expression evaluator on the Capybara. Also at the same time, Kyma is deciding how it should split up your signal flow diagram into subparts that can be assigned to different expansion cards on your Capybara (remember that you have at least two expansion cards in your Capybara and may have as many as eight of them). In other words, Kyma keeps track of how much sample RAM is left on each card and how much processing each card is being asked to do and uses this information to decide where to schedule each part of your Sound.

- During the compile step, you will see cursor change to a watch, and you may be asked to help locate samples, MIDI files, or other external files needed by your Sound. Kyma will search for a file for some amount of time and then put up a dialog asking whether you prefer to locate it yourself or whether Kyma should keep looking; if you know exactly where the file is, it can be quicker to find it yourself; otherwise, just hit **Enter** and Kyma will eventually find it.*



Load

- Once the time-tagged sequence of instructions has been divided up among the processors, Kyma sends the information to the Capybara. During this step, the cursor changes to an animated icon of a host computer downloading information to the Capybara.
- During this time, Kyma will also download any samples or wavetables required by the Sound from the hard disk of your host computer into the sample memory on your Capybara. Whenever a sample is being loaded into the Capybara, the cursor changes to a picture of a waveform being downloaded into the Capybara.

Start

- Once enough of this information has been loaded, Kyma tells the Capybara to start the time-counter. You can always tell when the Sound is playing (even if it is silent) because the cursor turns hollow; the Capybara will tell the host computer when the duration of that Sound has expired, so the cursor will be filled in again at the termination of that Sound.

* Your Sounds often require “external” files such as samples, wavetables, MIDI files, text files, analysis files, *etc.* To speed up the process of compiling you should specify the folders or directories containing these files as your **Frequently Used Folders** in the **Preferences** (accessed from the **Edit** menu). For example, you might keep all of your waveforms in one folder, all of your samples in subfolders of a samples folder, all of your MIDI files in another. Or you might prefer to keep all of the files used in a particular project in subdirectories of a single directory associated with that project. However you choose to organize your files, specifying them in the **Frequently Used Folders** will allow Kyma to locate them more quickly as it compiles your Sounds.

You may notice that sometimes the Sound will start to play before Kyma has finished downloading it to the Capybara. In these cases, Kyma has calculated that enough time-tagged events have been loaded that it can afford to let the Capybara start playing while Kyma continues to load the remaining events.

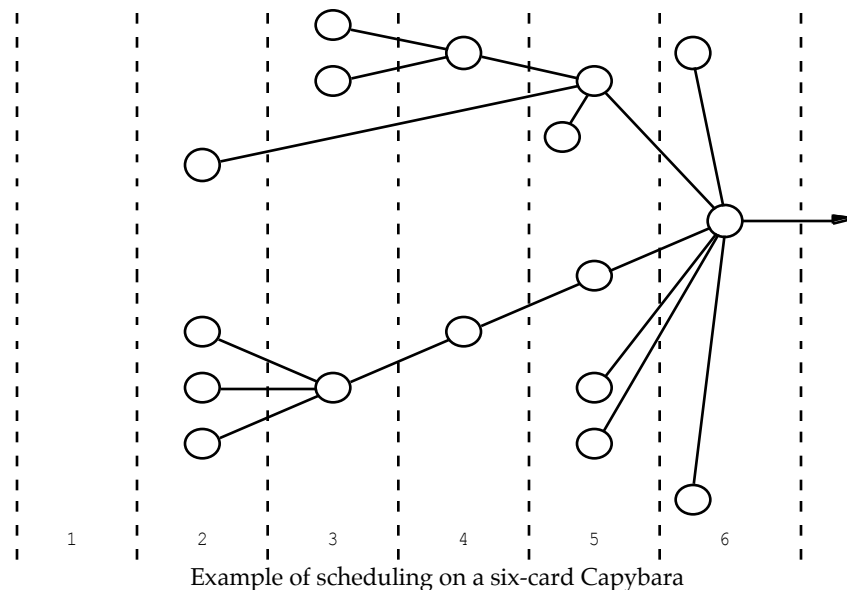
Compile, load

To compile and load a Sound without actually starting it, use **Compile & load** from the **Action** menu. Then you can control when the Sound starts by using **Ctrl+R** or **Restart** from the **DSP** menu (which always restarts whatever Sound is currently loaded in the Capybara).

This can be useful in a performance or in any other situation where you want to start a Sound without waiting for it to first compile and load.

Scheduling on Multiple Processors

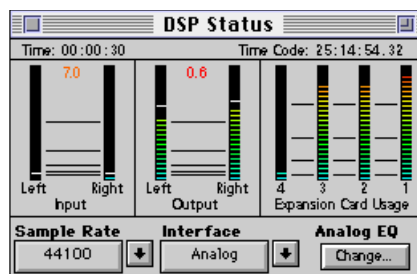
There are two constraints that Kyma has to satisfy when dividing up your Sound among the multiple processors: one is that the processor has enough memory and computing time available to handle that subpart, and the other is that a Sound's input must be scheduled on the same processor as the Sound it is feeding into or on a processor to the "left" of the Sound it feeds into.



To get a graphical indication of how much computing time your Sound is taking up on each of the processors in your Capybara, use the DSP status window (choose **Status** from the **DSP** menu).

DSP Status Panel

This window gives you a picture of the state-of-the-Capybara. In the upper left you can see the time that has elapsed since the start of the currently loaded Sound. This time is reset each time you play the Sound or use **Ctrl+R** to restart the currently loaded Sound.



Sample Rate

To change the sampling rate, choose from one of the rates listed below the **Sample Rate** label. The input and output low pass filters will be adjusted accordingly (so as to block out any frequencies above half the sampling rate).

Interface

If you have digital I/O installed in your Cappybara, you can use this list to switch between using the Analog inputs (the built-in converters), or the Digital inputs, in which case the Cappybara expects a digital signal in either the AES/EBU or S/PDIF formats. You can have digital inputs and outputs or analog inputs and outputs, but you cannot have a mixture of both (*e.g.* digital in and analog out). Also, make sure that the digital signal path does not form a loop; in other words, the digital output of the Cappybara should not be fed through several boxes and fed back in to the Cappybara at the end, because the Cappybara will not be able to tell where it should be finding its clock — whether it should pay attention to the incoming clock or whether it should pay attention to its own internally generated clock.

When you switch to digital I/O, it will appear as if one entire expansion card is always fully loaded, but don't worry about this; although it does take up a little extra processing on your last card to use the digital I/O, it is not really taking up as much time on that processor as the status display indicates.

VU meters

From left to right, the first two meters are VU meters showing the left and right input levels. The next two give the left and right output levels.

Computation Used on Each Expansion Card

The next grouping has a meter for each expansion card in your Cappybara. Although they are numbered from high to low, Kyma will begin scheduling Sounds on the leftmost (highest numbered) expansion card, then, if it runs out of time on that one, will spill over onto the next expansion card and so on. For Sounds that have *Concatenations* or *TimeOffsets*, you may be able to see how the amount of processing required on each card changes over time as some Sounds stop and different ones start up.

Strategies for Non-real-time

In some situations, after the Sound has been loaded, you may see a warning informing you that your Capybara is close to the limit on computation time. If this happens, go ahead and listen to the Sound anyway. Kyma is set to be slightly conservative on this warning, so it could be that the Capybara is still able to keep up with real time. If you have given the Capybara more than it can consistently do in real time, you will hear digital clicks and breakup in the sound. In general, the further it is from being able to compute in real time, the more clicks and pops you will hear; you may also hear something that sounds like granular time-stretching. But don't despair! There are still quite a few tricks you can employ in order to get around this.

What Influences the Real-time Capabilities?

Whether a particular Sound can be realized in real time or not depends on several factors:

- What kinds of modules are in the Sound (some of them require more computing time than others)

- How many modules are in the chain (the more modules, the more computing time required)

- The polyphony setting on any *MIDI Voices*, *MIDI Mappers*, or *Analog Sequencers* in the Sound (the number of modules in the chain to the left of a *MIDI Voice* will be multiplied by the amount of polyphony that you request)

- The number of expansion cards in your Capybara (if Kyma can split the computation among several cards rather than having to schedule all of the modules on a single card, then the Capybara can do more real-time computation).

Using DSP Status

Take a look at the DSP status window while you are playing your Sound. If the usage bar for the last (rightmost) card is fully in the red,^{*} it is usually a sign that you are requesting something beyond what your Capybara can compute in real time; it indicates that Kyma tried to find resources for the Sound on the first card, second card, *etc.*, and when it couldn't find enough resources anywhere else, it just tried to schedule it on the last card because there were no other choices left.

Record to Disk

The quickest solution is to select the Sound, go to the **Action** menu, and select **Record to disk....** This will record your Sound to the disk (don't worry if it still sounds like it is sputtering; it will really record it onto the disk without the breaks). From then on, you can play the digital recording using a *DiskPlayer*, *Sample*, or *GenericSource*.

This is the quick and easy solution for Sounds that do not require any live interaction from MIDI or the audio inputs. Even if your Sound is being driven from a sequencer, you can export the sequence as a standard MIDI file, check the box that says to read MIDI from a file rather than from the live inputs, and then record the Sound to disk.

But what if you are performing one of the parts live from a MIDI keyboard, controlling some of the faders in real time, or using the microphone inputs as sound sources? Or, barring all of that, what if you don't really want to record the entire Sound to disk just because it would take up too much disk space? There are some alternate solutions!

Disk Caching

One approach is to record sub-branches of your Sound on the hard disk, and read the samples from the disk from then on, thus freeing up time on the Capybara to compute the rest of the Sound in real time. Place a *DiskCache* just to the right of the branch that you have decided to record to disk. Then play the

^{*} When the digital interface is selected for audio input and output, the rightmost bar in the DSP status window will always be in the red, regardless of the actual usage on that card.

DiskCache. It will record everything to its left into a disk file. When it has finished recording, remove the check from the **Record** box in the *DiskCache* parameters. From this point forward, that entire branch will be read from the disk rather than computed in real time.

The advantage of using a *DiskCache* over simply recording to disk is that, with the *DiskCache*, you never lose track of what you did to create that disk recording. Everything that you used to generate the recording is still there, to the left of the *DiskCache* in the signal flow graph. This makes it easy to make changes to the original Sound and re-cache the results. No more trying to remember exactly how you created a particular track or where the disk recording of a particular Sound might be. The Sound and its recording are linked together in the Sound structure.

In some cases, disk-caching is something like multi-tracking, where different “layers” of sound are recorded, one at a time, in a hard disk recorder. In the case of Kyma Sounds, though, the “tracks” might occur anywhere within the Sound structure; they are not necessarily all inputs to one *Mixer*.

There are several different strategies for choosing which part of the Sound to cache on the disk. You might choose the most computationally expensive branch in order to free up the most time. Or you might be forced to choose only those parts that do not rely on live input — the fixed or sequenced parts of your Sound that do not change from playing to playing. It does not make sense to have nested *DiskCaches*, because that is like storing the same thing on the disk twice.

Optimizing your Sounds

Sometimes you can optimize your Sounds in such a way that the sonic result is exactly the same but the entire Sound requires less processing time.

Get a Baseline

Use **Get info** from the **Info** menu to get a measure of the complexity of your Sound in terms of a rough estimate of the percentage of your Cappybara that it requires in order to run in real time. Jot down this percentage before you start simplifying so that you can see which things have an impact.

Look for Common Inputs

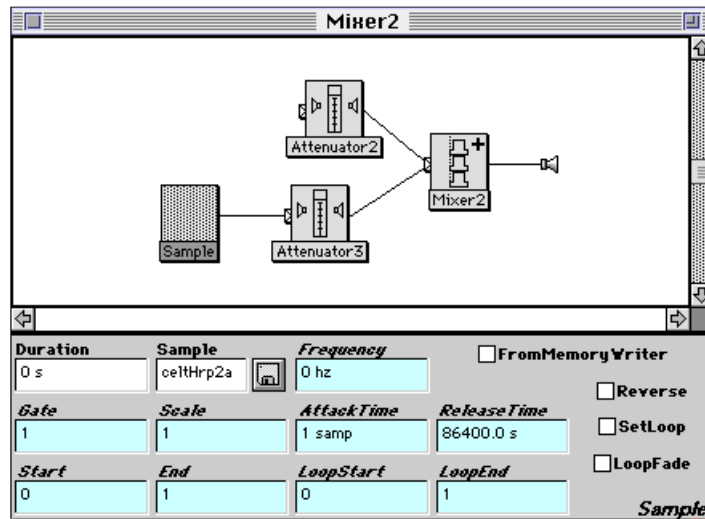
If you can identify two identical Sounds that are being processed in different ways, you can copy one of them, select the other one, and choose **Paste special...** from the **Edit** menu. This will cause them to be identical. That way, the Cappybara can compute this Sound once and feed the result to two different places (rather than recomputing it for each branch).

Check Polyphony

If you have any *MIDI Voices*, *MIDI Mappers* and/or *Analog Sequencers*, check that you haven’t requested more polyphony than you had intended. When you request 10 voices of polyphony, Kyma makes 10 copies of the Sound, so you should request exactly the polyphony that you need, not more.

Remove Redundancy

The most common source of redundancy is in amplitude scaling. For example, if you have a *Sample* feeding into an *Attenuator*, feeding into a *Mixer*, there are potentially three places where you could be scaling the amplitude.



Often you can combine these amplitude scales to simplify your Sound. In this example, you could specify the level in the **Scale** field of the *Sample*, and delete the *Attenuator*. Then you could take the scale factor in the *Mixer* and multiply it by the value in the **Scale** field of the *Sample*, making it possible to change the *Mixer* scale factors to 1.0.

Other Sounds that have potentially redundant scale factors include the *MIDIVoice*, *MIDIMapper*, and *Oscillator* (which you can scale by putting a constant into its **Envelope** field or by setting the **Scale** parameter in its AR or ADSR envelope).

Anything that you could simplify in an algebraic expression, you can probably also simplify in a Kyma Sound. For example, if you have several inputs to a *Mixer*, and each of them is going through its own *Attenuator*, and all *Attenuators* are set to the same value, you can eliminate the *Attenuators* and do all the scaling in the *Mixer*. That is like saying

$$a \times x + a \times y + a \times z = a \times (x + y + z)$$

Downsample Control Sounds

If you have some Sounds pasted into parameter fields, you can experiment with updating them at a slower rate to see if it has any negative impact on the sound. Recall that you can specify the number of milliseconds between updates, as for example

Oscillator L: 5

would mean to read the value of the Oscillator every 5 milliseconds and use that to set the value of the parameter. For many parameters, you can get by with update rates slower than once per millisecond and this will reduce the amount of computation required by your Sound.

Use Equivalent Sounds that are Less Complex

There may be a module that will do exactly what you need but which is less complex than a module you are using to do the same things. For example, you might be using an *Oscillator* to play once through a function where a *FunctionGenerator* could be used to do that more efficiently. You might be using an exponential envelope in a situation where a linear one would do as well. You may be using a *FunctionGenerator* with a Ramp waveform where you could use the expression `1 ramp: 2 s` to accomplish the same results using less computing power. Use **Get info** to compare the relative complexity of different Sounds that can be used to accomplish the same results.

Poor Scheduling

If it appears that only one of your expansion cards is being overloaded and that there are cards to the right of that one that are completely free, you should probably contact us and send us a copy of the Sound (either by email, via our *FTP* site, or on a floppy), because we would like to have a look at it so we can improve the scheduling algorithm.

Sample Rate

If all else fails, you *could* lower the sample rate. This is not a satisfactory solution in the long term, because it will also cut off the high frequencies of your output. However, it can be a good way to work with your Sound in real time until you get it adjusted the way you like it, at which point you can set the sample rate back up again and record the Sound to disk.

Throwing More Hardware at the Problem

If you find yourself running out of real-time computation power all the time, you might consider adding some more expansion cards to your Cappybara. In general, the more expansion cards in your Cappybara, the more polyphony and complexity you can squeeze out of it in real time.

Never Give Up

Never fear, between all of these strategies, you will be able to find some combination that will get around any temporary “out of real time” messages.

Learning Kyma

When people tell us their experiences in learning Kyma they describe three stages: first, the excitement of looking around, trying out all the examples, hearing the sounds, and being able to manipulate them in real time with MIDI, plus an exhilarating sense of learning many new things at a very fast pace.

At some point they report reaching a plateau where they may feel frustrated, as though they can't seem to make Kyma do what they want anymore. But what is really going on is that they are starting to move from the first layer (using and tweaking the existing Sound examples) into a deeper layer where they are starting to create their own Sounds and getting into programming. It's not that Kyma has suddenly gotten harder, it's that they are trying to do more difficult and complex things with Kyma.

However, once they have worked their way through this plateau, they have reached a point when they have suddenly felt truly fluent in Kyma. Once you make it through this plateau period, Kyma becomes your own environment, your own language that you can customize and use to do anything you like.

You may experience these same three stages or you may experience something entirely different. In any case, we would like to hear from you about your experiences in learning Kyma, both because we can use your experiences to make Kyma more intuitive and because we may be able to use your experience to help other people learn Kyma more effectively.

Congratulations!

If you have read this far, you have what it takes to become a Kyma guru, sought after and admired by your friends who will be constantly bothering you for your expert advice and assistance.

But that is not why you're reading this book; you're reading the book because you want to use Kyma in your own work. So, it is time to press on; do the rest of the tutorials and soon you will be fluently creating and tweaking sounds in an intense feedback loop between you and your machine (and feeling something along the lines of what a gambler feels at the handle of a slot machine).

Your interest in Kyma indicates that you have an intellect and a curiosity beyond the ordinary, that you are still driven by the same insatiable and joyfully persistent drive to learn and understand the world that you had as a child.

The easy way out would have been to take the standard, off-the-shelf sounds, the ones that everyone else has, the ones that would identify you as one of the crowd. But you have a need to get inside the sound, to control it, to truly understand what you are doing, and to break new ground, to create new sounds that have not been done before. Since you've been this way all of your life, you already know that it's not the easiest way to be. But it *is* the most satisfying. So congratulations for refusing to lower your standards. And we hope you will stay in touch with us, because it is stimulating for us to get to talk with people like you.

OK, time to order out for a pizza, unplug the telephone, turn the page and start in on the tutorials. (But don't hesitate to plug the phone back in and call Symbolic Sound if you run into any questions!)

Tutorials

Learn Kyma in 24 Hours!

One month, one hour per day, taking every 7th day off *or*

Two hours a day for two 6-day weeks *or*

Four hours a day for six days *or*

In 24 hours straight! (not recommended)

Part I: A Tour through the Examples

Building your own Sound library

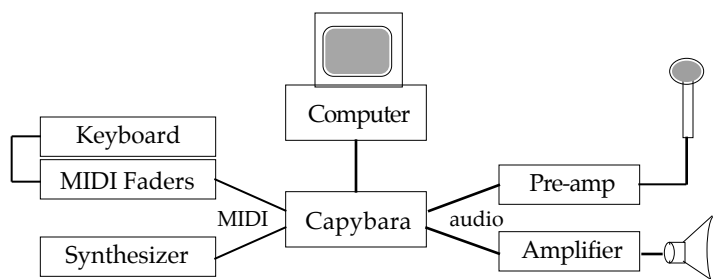
One of the best ways of learning how to design your own Sounds in Kyma is to start by examining and modifying the Sounds that other people have made. Whenever you make a modification that sounds interesting, you can rename the Sound, add a short annotation to it (so that later you remember what it does), and save it in a new Sound file. This is a great way to start gradually building up your own customized Sound library that you will be able to draw upon in the heat of creative passion. Once you become familiar with the examples that came with Kyma and have extended that set with your own examples, it will be a rare situation in which you would not be able to recall a Sound that is close to what you need and begin modifying *that* (rather than having to start from scratch each time).

The Itinerary

This “tour” through the examples is a tutorial is intended to give you an overall sense for what kinds of things are possible in Kyma and to help you get started building your own Sound library. We’ll do this by looking at each of the example Sound files, getting an idea of what kinds of Sounds it contains, dissecting some of the Sounds from that file in detail, and experimenting with modifying parameters of some Sounds to get a feel for which parameters have what effects.

By the way, this tutorial is not intended as an introduction to sound synthesis, just an introduction to the example Sound library in Kyma. If you find that you need an introduction to sound synthesis, we suggest that you check the local library or bookstore for a copy of one of the books listed in the Background Materials Appendix, call Symbolic Sound to find out about attending one of the intensive Kyma Immersion weekends, and/or check to see whether an educational institution near you might offer an introductory course on electronic or computer music (you may even be reading this book as part of a course right now).

In general though, you should continue to plunge forward through this tutorial even if you happen to encounter a few terms or concepts along the way that are not completely familiar. Most of the concepts and terminology will become clear once you start listening and experimenting, and any remaining questions can be easily cleared up by a text book or a course. And if you do take a course, the intuitive understanding that you will have acquired by working with these concepts in Kyma will put you far ahead of all of your classmates (so please be kind to them and don’t gloat *too* much when you discover how much better you understand everything).



You will get the most out of this “tour” through the examples if you take a moment right now to make sure that you have everything connected and set up as follows:

The Capybara analog outputs should be connected to an amplifier and speakers or to a headphone amplifier and headphones. Alternatively, if you are using the digital I/O, the digital out should be feeding into a digital-to-analog converter such as a DAT, and the DAT outputs should be connected to the amplifier or headphone amplifier.

If you have a microphone, feed it into a preamplifier to bring it up to line level (your mixer may have an input that does this), and connect it to the left channel input of the Capybara. If you are using digital I/O, then the microphone must be connected to an analog to digital converter (say, another DAT machine), and the output of that DAT should be connected to the digital input of the Capybara. If you do not have a microphone or if you prefer to use a musical instrument, synthesizer, sampler, DAT or CD as a source, simply connect the line-level (or digital) outputs of your source to the audio inputs of the Capybara.

If you have a MIDI keyboard and/or MIDI controller, connect the MIDI output of the controller to the MIDI input on the back of the Capybara. If you would like to use both a keyboard and a controller, connect the MIDI output of the keyboard to the MIDI input of the controller. Then connect the MIDI output of the controller to the MIDI input of the Capybara.

If you would prefer to use your software sequencer as a source of MIDI note events and MIDI controllers, connect the MIDI output from your computer’s MIDI interface to the MIDI input on the back of the Capybara. Then, connect the MIDI output from the Capybara to one of the MIDI inputs on your computer’s MIDI interface.

If you have an external sound module that you would like to control from Kyma, connect the MIDI output from the Capybara to the MIDI input on the back of the external sound module.

To check that you are getting audio input, find *ADInput* in the system prototypes under the **Sampling** category. Select it and use **Ctrl+Space Bar** to play it. Choose **Status** from the **DSP** menu. Start making noise into the microphone and make sure that one or both of the input level bars are jumping. If not, then it means that the audio input is not properly connected to the Capybara.

To check that you are getting MIDI input, choose **Configure MIDI...** from the **DSP** menu. Click the **Show MIDI messages** button. Play a few keys on the MIDI keyboard and move a few faders on your MIDI controller. The values in the window should change while you do this. It will also tell you what MIDI channel(s) it is receiving MIDI messages on. If you do not see the values changing in this window, it means that the Capybara is not receiving MIDI input. Click the mouse to exit this window, and check your MIDI connections again and make sure that your keyboard and controller are both switched on. If you are using something like MIDI patchbay or other MIDI routing program, you may have to make sure that the input is routed directly to the output.

The examples provided with Kyma are set up to use MIDI channel 1 for input. You should set your MIDI devices to output on MIDI channel 1. If this is not possible, may need to change the default MIDI channel (see *DSP menu: Configure MIDI...* on page 435) or modify the global map (see *Global and Local Maps* on page 482).

Once you have everything set up and verified, it is time to start having fun!

In each of the Sound files in the **Examples** folder, you will find examples of different kinds of sound synthesis and processing. The examples in each file are described in the section in this chapter with the same name as the Sound file.

For many examples, there are two alternate versions provided: the different versions achieve similar (although not exact) sound synthesis or processing results, but use different amounts of Capybara processing and memory resources. These example Sounds are labeled with the minimum number of Capybara expansion cards that are needed for real-time operation at 44.1 khz sample rate. For example, there are two versions of *glottal chopped*: *glottal chopped (5)* for Capybaras with five or more expansion cards, and *glottal chopped (2)* for Capybaras with two or more expansion cards.

Start by choosing **Open...** from the **File** menu, make sure the file type is **Sound file**, and open the file called **analog** from the **Examples** folder which you should find in the **Kyma** folder.

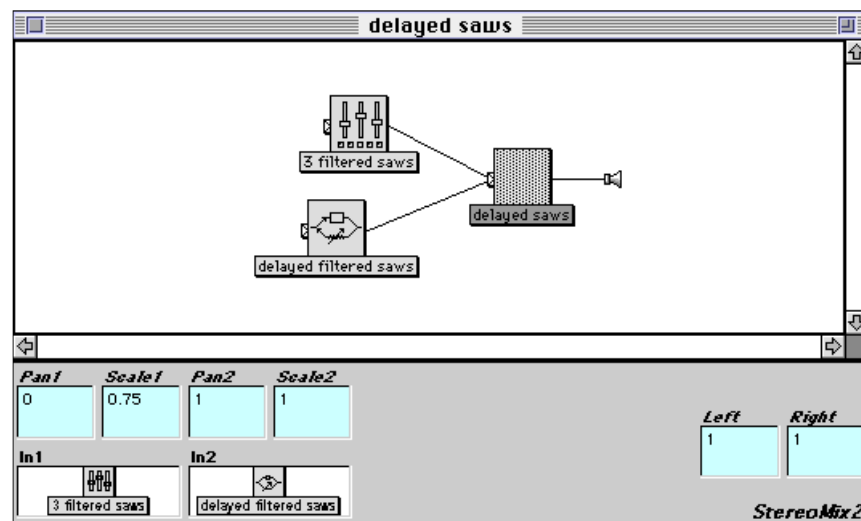
Analog

Select and play *RomeTaxiRadio* to get an idea of the kinds of Sounds contained in this file. These are all emulations of the kinds of sounds you could get from the old modular analog synthesizers. Use the virtual control surface to change `!Rate` from 1 to 0 and back to 1 again. Change `!NN8` and `!NN7` to modify the last two pitches in the sequence. Experiment with the other faders and take note of what effect each of them has on the sound. To set a fader to a precise value, click in the small text box at the top of the fader, type in the value you want, and press **Enter** or **Return**. The fader will jump to that new value. When you have had enough, use **Ctrl+K** to stop the Sound.

Dissection

Select and play the Sound named *delayed saws*. Play the MIDI keyboard and experiment with each of the faders in the virtual control surface, listening to how each affects the sound.

Double-click on *delayed saws* to take a look at how the Sound is put together. Let's dissect this Sound, starting with the leftmost module (the output) and moving rightwards.



The rightmost module is a *StereoMix2*. Double-click it so you can see its parameters displayed in the lower half of the Sound editor. There are two inputs to the mixer: *3 filtered saws* and *delayed filtered saws*. **Pan1** is set to zero, meaning that it is panned hard left, and **Pan2** is set to one meaning that **In2** is entirely in the right channel. **In1** is attenuated a little bit, since **Scale1** is set to 0.75. **In2** is at full amplitude since **Scale2** is set to one.

Following the lower branch first, double-click on *delayed filtered saws* to view its parameters. This branch takes *3 filtered saws* as its input and delays it by one second.

Now double-click *3 filtered saws*. This is called a *Preset* Sound. It gives you a way to save the current settings of faders in the virtual control surface so that the next time you play the Sound, they will start out at those settings. Notice that the **EventValues** parameter has a setting for each of the faders you saw in the virtual control surface:

```
!Atk channel: 1; initialValue: 1.0.  
!CutOff channel: 1; initialValue: 0.75.  
!Detune1 channel: 1; initialValue: 0.0750853.  
!Detune2 channel: 1; initialValue: 0.0511945.  
!Feedback channel: 1; initialValue: 0.665529.  
!Rel channel: 1; initialValue: 1.0.  
!Volume channel: 1; initialValue: 0.156997.
```

You can read these settings, but you can't modify the text, at least not by typing. The only way to alter this parameter is to change the settings of the faders in the virtual control surface, and to then press the button labeled **Set to current event values** found just below the **EventValues** parameter field.

Proceeding ever leftward, let's next double-click the module called *LPF*. If you don't see a module to the left of *3 filtered saws*, click on the small tab attached to the center of the left edge of the *3 filtered saws* icon. This little tab with a right-pointing arrow in it indicates that a Sound has hidden. Click on the tab to show the Sound's immediate inputs. Click again to hide the inputs of a Sound. To display all of the inputs of a Sound (*i.e.* all of its immediate inputs, the inputs to *those* inputs, *etc.*), click the tab while holding down the **Control** or **Command** key. You can use these tabs to keep your signal flow graphs uncluttered and easy to read, while still being able to edit any part of the graph when necessary. (See *Editing the Signal Flow Diagram* beginning on page 26 for more information about using the signal flow diagram.)

Here we get our first glimpse of some of the red Event Values we were controlling from the virtual control surface. This is a generic filter that you can set to lowpass, highpass, or allpass. This particular filter is set to lowpass, so it will tend to attenuate all frequencies in its input that are above the value in the **Frequency** field. The other parameters tell us that this is a fourth order filter and that an Event Value called **!Feedback** controls a parameter similar to the "resonance" control of analog filters. When you were experimenting with the faders in the virtual control surface you probably noticed that you could make the filter ring or even blow up by increasing the value of **!Feedback**.

The expression in the **Frequency** field

```
filter cutoff L * (!Cutoff * 11050 hz + 200 hz)
```

indicates that, depending on the setting of **!Cutoff**, the maximum **Frequency** value will be between 200 hz and 11250 hz. This maximum value is in turn multiplied by a Sound called *filter cutoff* that generates an envelope shape.

To see how the envelope shape controlling the filter cutoff is generated, double-click the icon named *filter cutoff*. This is an **ADSR** (Attack time, initial Decay time, Sustain level, Release time) envelope generator. From the parameters we can tell that when the envelope generator is triggered by a MIDI key-down event, it goes from 0 to maximum amplitude with 10 milliseconds, drops to 75% of its maximum amplitude in the next 10 milliseconds, where it remains for as long as the key is held down, after which it drops back to zero in 5 seconds starting from when the key is released. So, as you already heard when you were experimenting earlier, each time you press a MIDI key, the filter opens up to allow more high frequencies to pass through, and when you release the key, the filter closes back down again.

What input is *LPF* filtering? Double-click the icon called *phatness*. This is a **Gain** module that is set to multiply the amplitude of its input by ten. In this case, since the input is already a full-amplitude signal, the effect of the **Gain** will be to distort the signal by clipping it, turning it from a sawtooth-like waveform into more of a square-shaped waveform, and as a result, changing the timbre from something with all harmonics to something with only the odd harmonics.

To see and hear this for yourself, select *phatness* and choose **Oscilloscope** from the **Info** menu. Leave all the other faders at zero, and type 0.01 into the field above the **!Volume** fader (and hit **Enter**). Then play some low pitches on the keyboard. Gradually increase **!Volume** using the fader until you see and hear the waveform flatten out at the top and bottom because it hits the maximum amplitude. Notice that changing **!Volume** changes not just the loudness, but the timbre as well.

To see how the spectrum is affected, make sure *phatness* is still selected, and choose **Spectrum analyzer** from the **Info** menu.* Hold down C 6 on your keyboard (two octaves above middle C), and alternate the position of the **!Volume** fader between almost zero and almost one by clicking with the mouse towards the bottom and then towards the top of the fader. Notice how every other spectral line disappears when the signal is clipped? Try this a few times, listening to the change in timbre that accompanies this change in the spectrum. There is another, more subtle effect you can observe on the spectrum: as soon as the waveform is clipped, it has an infinitely sharp corner on it, and infinitely sharp corners have an infinite number of harmonics, so some of the short lines you see clustered around the harmonics are actually ali-

* By the way, this explanation is not essential to understanding and using Kyma — it is just an interesting aside.

ases of those higher harmonics, artifacts of one of the basic facts of life in the digital domain: you cannot represent any frequencies above half of the sampling frequency.

Next let's see how the sawtooth is created before it is distorted by the *Gain* module. Double-click the icon named *3 key-mapped saws*. This is a *Mixer* with three inputs: *saw1*, *saw2*, and *saw3*. The *Mixer* adds the outputs of all three of these Sounds and then multiplies the sum by the current value of `!Volume`. Since the *Gain* subsequently multiplies that result by ten and the value of `!Volume` can range from zero to one, the end result is that the sum can be multiplied by a number in the range of zero to ten, depending on where you set the value of `!Volume`.

So far all we've seen is adding, filtering, enveloping, and distortion. When do actually get to the part that generates sound? Funny you should ask (did you?), because that is what we finally get to with this next and final level; double-click on *saw1* to see its parameters. A *KeyMappedMultisample* uses an entire folder full of samples as its source material and uses the policy described in the **Mapping** parameter to map different samples to different MIDI key numbers or ranges of key numbers. In *saw1*, each sample from the folder is mapped to the range of pitches specified in the header of the samples file. Why use different samples for different pitch ranges? Because of that fundamental law of the digital domain: you can't represent any frequencies above half the sampling rate. So the higher the fundamental pitch, the fewer harmonics its waveform can have, because those harmonics, being at frequencies that are multiples of the fundamental, can quickly get high enough to exceed the half sample rate limit. That's why *saw1* uses waveforms with many harmonics for the bass notes, and waveforms with only a few harmonics for the really high frequencies.

Take a look at the **Frequency** of *saw1* and compare it to the same parameter field in *saw2* and in *saw3*. If it is difficult to read everything that is in the **Frequency** field, position the cursor on the center line dividing the signal flow graph from the parameter fields until the cursor turns into a double-arrow pointing up and down. Then use the mouse to drag that center line upwards, making all the parameter fields larger so you can read their contents.

Notice that the **Frequency** values differ by the value of `!Detune1` and `!Detune2`. By making these three "oscillators" slightly out of tune with each other, you can get them to sometimes reinforce each other, sometimes cancel each other, and generally get a more dynamic, evolving timbre than you would be able to get out of fixed frequency oscillators. Try this out right now. Select *3 key-mapped saws* (the mix of the three sawtooth oscillators) and choose **Oscilloscope** from the **Info** menu. Experiment with different values for `!Detune1` and `!Detune2`.

Now, what about this expression in the **Frequency** field?

```
!KeyNumber smoothed nn + !Detune1 nn
```

This is saying that the frequency is the MIDI key number in units of note number (`nn`) plus some fraction of a half step that depends on the value of `!Detune1`. The *smoothed* means that when you change from one note number to another, it will take 100 milliseconds to make the transition, rather than making it instantaneously.

Modifying the Example

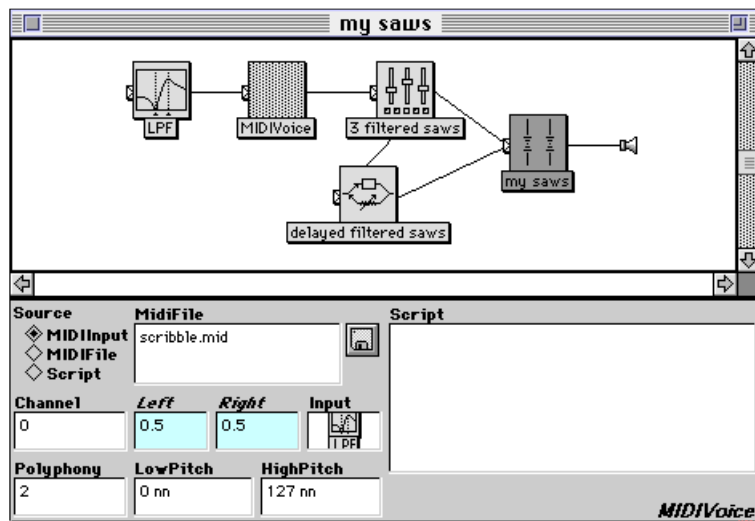
Now it's time to start mutating the Sound in order to both gain a deeper understanding of how it is put together and to start building your own Sound library. First, close the Sound editor, and make a duplicate of *delayed saws* so you can retain the original Sound and make your changes to the duplicate. (Do this by selecting the icon and using **Ctrl+D** for duplicate). To change the name of your new Sound, make sure *DuplicateOfDelayed saws* is still selected, and hit **Enter**. This gives you a dialog where you can enter a new name for the Sound; you might as well call it 'my saws', just to be stunningly original.

In order to keep track of which Sounds came with the system and which ones you design yourself, you can create a new Sound file (by typing **Ctrl+N** and selecting **Sound file** as the file type) and drag the *my saws* icon into this new window. Save it on the disk, creating a new folder for the purpose called **my examples** and giving the file itself a memorable name like **my analog**.

At this early stage, this may seem like just an exercise, but in fact, it is a good idea to start using Sound files to categorize and organize your new Sound designs from the very beginning. As soon as you start tweaking Sounds, you start hearing things you've never heard before, and even if you are diligently fol-

lowing the tutorials step-by-step, you might make a fortuitous mistake that leads to some fantastic result that you will want to save for future reference.[§] The idea is to save your Sound in a file along with other, similar Sounds and to give the Sound file a name that will lead you back to that file weeks or months later when you might be searching for examples of analog-like sounds.

Double-click *my saws* so you can start editing and making it your own. As a first step, increase the polyphony using a *MIDIvoice* (found in the *MIDI In* category of the prototypes, or by searching the prototypes using **Ctrl+B** and typing a partial word like *midi* into the dialog box). Where should the *MIDIvoice* go in the signal flow graph? If we want all the voices to share the same delay, we should put the *MIDIvoice* before the *DelayLine*. It also makes sense that the *Preset* should apply to all the voices at once, so let's place it on the line between *3 filtered saws* and *LPF*.



Leave all the *MIDIvoice* parameters at their default values, and select *my saws* again so you can use **Ctrl+Space Bar** to compile and load the new Sound. Once the Sound is loaded, experiment with playing two-voice polyphony on the MIDI keyboard.

The constant, 1 second delay is getting a little tedious, so let's control the delay with a MIDI fader or on-screen controller. Double-click *delayed filtered saws*, hit the **Tab** key until you get to the **DelayScale** field, hit **Escape** on the computer keyboard, and move one of your MIDI faders. (If you don't have any MIDI faders, use **Ctrl+H** to get a list of hot parameters; start typing the word *delay*, and then hit **Return** so that the Event Value *!Delay* is pasted into the **DelayScale** field). Select and play *my saws* again. Hold down some MIDI keys and move *!Delay* from zero up to its maximum value (using either a MIDI fader or the virtual control surface). You can't fail to notice that the delay has the interesting side effect of changing frequency as you change the length of the delay. After letting the extreme change die down, experiment with very subtle changes to *!Delay*.

Experiment with all the Event Values in the virtual control surface until you have the settings to your taste. Then double-click *3 filtered saws*, and press the button named **Set to current event values**. This will save your current settings along with the Sound.

Add an *Annotation* as the rightmost module in the signal flow (found under **Variables and Annotation** in the prototypes). Select the *Annotation* and press **Enter**, so you can give your Sound a descriptive name that will help you locate it later. Double-click the *Annotation*, and type a brief description of this Sound in the **Text** field. Sure, right now this seems like a lot of annoying extra work, but in a few weeks, when you revisit this Sound and try to remember exactly how it works, you will be glad you left yourself a little reminder note. Once you have finished with your annotation, close the editor, and close your new Sound file, saving the changes you made.

[§] Kyma was specifically designed to tempt you off the path of diligently following directions and lead you down the path of experimentation, so don't expect that you won't be fooling around with developing some additional Sounds on the side, even as you diligently and systematically work your way through the tutorials!

Take a few moments now to listen to some of the other Sounds in the **analog** file, so you will know what is available here and can come back to draw upon these as you need them.

Audio Demonstrations

These examples are set up as mini-laboratories to demonstrate different synthesis techniques or psychoacoustic phenomena. Use them to demonstrate these concepts to your students and/or to yourself!

Using the Additive Synthesis Lab

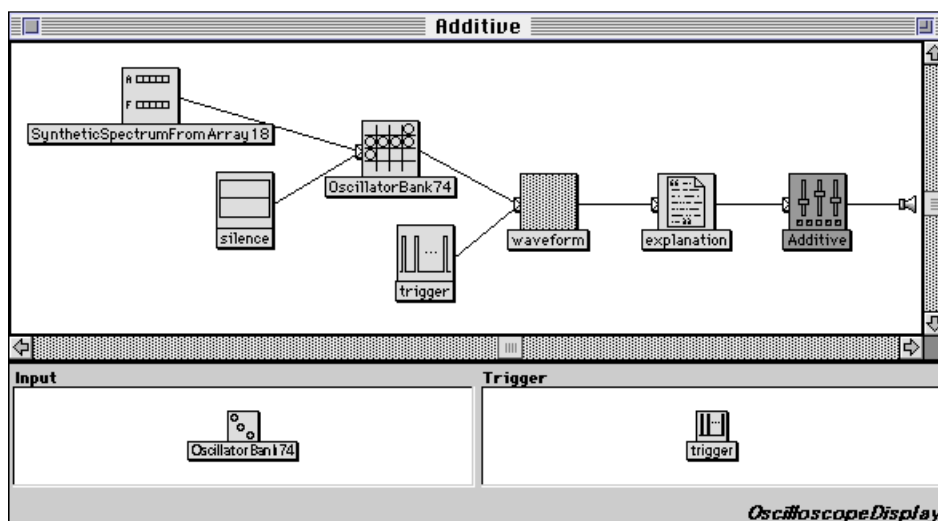
For example, use **Ctrl+Space Bar** to compile, load, start the Sound called *Additive*. In the virtual control surface, you can increase or decrease the amplitude of 16 sine wave harmonics, and see the effect this has on the waveform in the small oscilloscope display.

Click in the numeric field above the **F02** fader; this should select the contents of that field. Type in a 0 to set the fader value to its smallest value. Then press the **Tab** key. This selects the next fader to the right. Type in a 0 here as well. Continue to tab over and enter zeroes, observing the change in the waveform as you remove each harmonic up through **F16** and you only hear (and see) the first harmonic or the fundamental.

Now go back and add a very small amount of each harmonic back in. If you see the waveform clipping or hear it distorting, you may have to reduce the !Volume.

How the Additive Synthesis Lab Works

Close the virtual control surface, and double-click on the icon called *Additive* to see how this Sound was put together.



The rightmost sound is a *Preset*. Double-click on it so that its parameters are displayed in the lower half of the Sound editor. The *Preset* is used to set all the Event Values to reasonable values when you first play the Sound. For example, you would not want the initial value of !Volume to be zero, because it might confuse the person using the virtual control surface if they don't hear any sound.

Position your cursor over the line dividing the signal flow graph from the parameter fields. When you see it change to an up/down arrow, click down and drag the dividing line up, until you can see all of the settings in the Event Values parameter field. You cannot edit these values, but you can reset them to the current settings of your virtual control surface by clicking on the button labeled **Set to current event values**. Drag the dividing line back down so you can see the signal flow graph again.

Double-click on the next module, the one called *explanation*. An *Annotation* does not affect the sound at all; it is more like a comment in a program — you can read it, but it doesn't affect the result (other than to make you feel extremely happy to find it there if you are trying to understand the program!) Whatever is written in an *Annotation* anywhere within the signal flow graph will be displayed in the virtual control surface when you play the Sound. This comes in handy when you want to include a word or two of explanation for the person using the virtual control surface.

Next is a module named *waveform*. This is a *OscilloscopeDisplay*, and it is another type of Sound that does not have any affect on what you hear — just on what you *see* in the virtual control surface. It shows the waveform of its **Input** as if you were feeding the **Input** into an oscilloscope, and uses the name of the Sound (in this example *waveform*) as the name of the oscilloscope display in the virtual control surface. Imagine a kind of virtual oscilloscope screen inside the Capybara. The *OscilloscopeDisplay* writes its **Input** on this virtual screen from left to right, until it receives a trigger. Each time it receives a trigger it starts over again at the left and continues writing the waveform that it gets from its input. As quickly as it can (but still on trigger boundaries), the host computer reads the virtual display and plots the entire thing on the computer screen all at once.*

So the **Trigger** input to the *OscilloscopeDisplay* should be something periodic, something that puts out a one at the beginning of each cycle in the **Input** waveform; that way you will always get a picture of complete cycles of the input waveform and the waveform will not appear to “drift” across the screen to the right or left because it is out of synchronization with the trigger. The best Sound to do this job is the *PulseTrain*, because it puts out a one once per period and is zero at all other times.

Double-click the Sound called *trigger* to see its parameters. The period of repetition is set to 256 samp. You can optionally vary the “duty-cycle”, or the amount time during each cycle when the waveform value is above zero. Try this out right now by checking the **VariableDutyCycle** box, typing !Duty into the **DutyCycle** field, selecting the *trigger* Sound and choosing **Oscilloscope** from the **Info** menu. As you slowly move !Duty from 0 up to 1, you can see that the width of the “pulse” gets wider and wider. At the two extremes of 0 and 1, you don’t hear anything, because the value isn’t changing, and sound does not exist without *change*. Remove the check from the **VariableDutyCycle** box before proceeding.

Double-click *OscillatorBank74*.[§] This generates a bank of 16 sine wave oscillators. The frequencies and amplitudes of those sine wave oscillators are supplied from *SyntheticSpectrumFromArray18*. Double-click on that Sound to see how the amplitudes and frequencies are specified.

A *SyntheticSpectrumFromArray* takes an array of frequencies and an array of associated amplitudes and puts them together to create a spectrum for controlling an *OscillatorBank* or a *FormantBankOscillator*. You can optionally associate an array of bandwidths with the spectrum for Sounds like *FormantBankOscillator* which make use of bandwidth; in this case, though, we have left **SendBandwidths** unchecked because an *OscillatorBank* does not use bandwidth information.

Envelope puts an overall amplitude envelope on all partials; in this case, the overall amplitude is controlled by !Volume. The value of **NbrPartials** is arbitrary, but you should generally use a number at least as large as the value of **NbrOscillators** in the *OscillatorBank*; in this example, we are generating amplitude and frequency values for 16 sine wave oscillators.

Amplitudes, **Frequencies**, **Bandwidths**, and **Envelope** are all hot parameters (indicated by the light cyan background color in the parameter fields), meaning that you can use Event Values and Sounds as elements of the array. For example, the value of **Amplitudes** is:

```
!F01 !F02 !F03 !F04 !F05 !F06 !F07 !F08 !F09 !F10 !F11 !F12 !F13 !F14 !F15 !F16
```

an array of amplitudes, each controlled by a fader in the virtual control surface.

* Note that although both the right and the left channels of the input are written to the virtual oscilloscope display on the Capybara, only the left channel is read and displayed on the host computer. You will be able to display both channels in a future update of the software.

§ By the way, if you are wondering how it got a name like that, here’s the explanation: whenever you drag a Sound from the prototypes into a Sound editor or Sound file window, Kyma makes a *copy* of the prototype. When Kyma makes a copy of the prototype, it appends a number onto the end of the prototype’s name; for example, the third time you drag an *OscillatorBank* from the prototype strip, it will be named *OscillatorBank3*. The current number for each prototype is stored in the preferences, so it will continue giving you consecutive numbers for as long as you keep the same preferences file. So, in this particular case, this was the 74th *OscillatorBank* used by the person who created this Sound. Kyma tacks the number onto the end of the name to help you distinguish between this Sound and others of the same class without having to immediately rename it as you are constructing the Sound. However, you should select the Sound and hit **Enter** to give it a more mnemonic name as soon as possible.

Now examine the value of **Frequencies**:

```
{256 samp inverse} {256 samp inverse * 2} {256 samp inverse * 3} {256 samp inverse * 4} {256 samp inverse * 5} {256 samp inverse * 6} {256 samp inverse * 7} {256 samp inverse * 8} {256 samp inverse * 9} {256 samp inverse * 10} {256 samp inverse * 11} {256 samp inverse * 12} {256 samp inverse * 13} {256 samp inverse * 14} {256 samp inverse * 15} {256 samp inverse * 16}
```

The first value

```
256 samp inverse
```

is a frequency whose period is 256 samples long.^{*} The next element in the array is twice the first, the next is three times the first, the next is four times the first and so on. In other words, these are integer multiples of the first frequency, otherwise known as harmonics of the first frequency which is otherwise known as the fundamental.

But why the curly braces? Because you have to give Kyma some indication of what constitutes a single element of the array. If you had not enclosed each expression within curly braces, then Kyma would have interpreted each subpart of the expression as one element in the array; in other words, 256, samp, and inverse would have been interpreted as individual elements of the array. But when you enclose it within curly braces, the expression is evaluated first and *then* used as a value in the array.

To evaluate {256 samp inverse} or for that matter, any other expression in a parameter field, place the mouse between the open brace and 256 and click twice; this should select the entire expression within the curly braces. Then use **Ctrl+Y** (or **Evaluate** from the **Edit** menu) to evaluate the expression and print out its value:

```
172.265625d hz
```

or approximately 172.3 hz.[§] Once you have seen the result, hit **Delete** to erase the still-selected number.[‡]

All of the prototypes in the **Spectral Sources** category (including the *SyntheticSpectrumFromArray*) generate spectra in the same format. Do the following to get an idea of this format:

Turn down the gain of your amplifier (because the sound you are about to hear is not pleasant!).

Select the *SyntheticSpectrumFromArray*, and choose **Oscilloscope** from the **Info** menu. You should see the virtual control surface with 16 faders on it and a flat-line waveform display.

Set the !Volume fader up to its maximum value of 1.

^{*} Recall that frequency is the inverse of period. The easy way to remember this is to recall that frequency is cycles-per-second (cycles/second) and that the period of a signal is the number of seconds-per-cycle (seconds/cycle).

[§] The “d” at the end of the number indicates that this is a double-precision floating point number, where double-precision gives you 12 significant digits instead of the normal 6 significant digits you get with a regular floating point number.

[‡] Since you can evaluate expressions in Kyma, you can use it as a calculator. Create a new text file using **New...** from the **File** menu, and select **Text** as the file type. Save the file on your disk as `kymacalc.txt` or `kyma calculator`. Now type in the expression:

```
51.75 * 0.07
```

select it, and use **Ctrl+Y** to evaluate it. If you have some formulae that you use a lot, enclose them within double-quotes so you can keep them around in this file and be able to evaluate just one of them at a time. For example, enter

```
"(9.0/5.0) * 100.0 + 32.0"
```

and click twice between the open-quote and the open parenthesis to select everything within the double-quotes. Then use **Ctrl+Y** to evaluate the expression. By substituting the current temperature in Celsius for 100.0, you can use this expression to tell us how cold or hot it is where you are in terms of Fahrenheit.

So now you have a little Kyma-calculator to do quick arithmetic calculations on the side as you need them, do your accounting, compute your income taxes, compute the phase of the moon, or whatever...

Now, try the following:

Bring up !F1, the amplitude of the first harmonic to about half way. (You should see periodically spaced vertical lines)

Bring up the amplitude of !F8, the 8th harmonic to about 3/4 of the way up.

Set !F4 to about 1/4 of the way up.

Set !F12 to its full amplitude of 1.

What you see is a picture of the spectrum repeated 32 times across the width of the oscilloscope display. In fact, you should be able to create a kind of “shape” using the fader settings and see it repeated in the 32 spectra. Try this right now. Set !F1 to its maximum value, !F2 to just slightly less than that, !F3 to slightly less, and so on until !F15 is just above zero and !F16 is set to zero. You should see a kind of sawtooth shape repeated 32 times in the oscilloscope display.

Cautiously bring up the gain on your amplifier (not too fast or your ears will be very sorry). You should hear a very high pitch. How high? Well, the period is 16 samples long. So this seems like the perfect opportunity to employ your Kyma-calculator text file. Enter the expression:

```
16 samp inverse
```

select it, and use **Ctrl+Y** to evaluate it and see what the frequency is in hertz.

The output of the left channel of the *SyntheticSpectrumFromArray* is:

```
amplitude1, amplitude2, ... amplitude16
```

repeated every 16 samples. The right channel (not shown on the oscilloscope) is:

```
frequency1, frequency2, ... frequency16
```

We take a closer look at this kind of spectrum format later on, in the section called *Live Analysis, Resynthesis* on page 131.

But for now, take a moment to try out the other audio demonstrations (or make a note to come back to try these later, once you have finished the rest of these tutorials). For some of the examples, you may need to click the grow box on the virtual control surface in order to be able to read all of the text and have more detailed control of the virtual faders.

Backgrounds, Textures

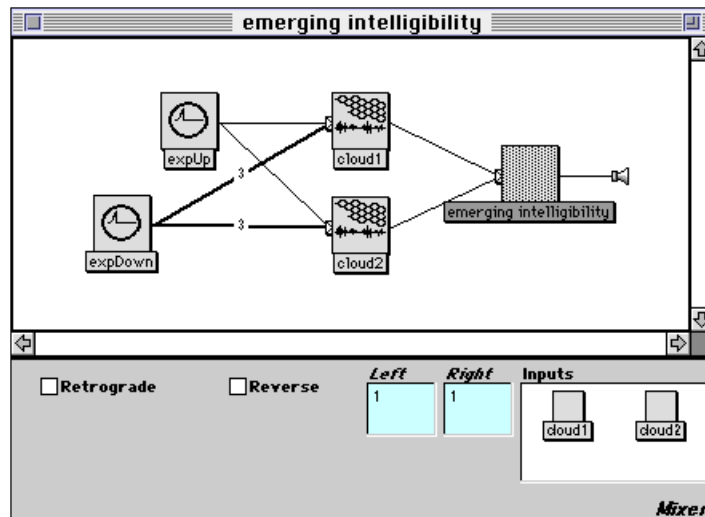
Sounds in this file are useful for generating dense, evolving, and spectrally rich textures.

For example, select and play the Sound called *tornado sirens*. Using the virtual control surface, try making gradual adjustments to the following parameters, and jot down what effect changes to that parameter have. Return each parameter to its approximate original position before proceeding to adjust the next parameter, so you can hear their individual effects:

```
!Pan
!Density
!Frequency
!Cycles
!FrJitter
!PanJitter
!DurJitter
```

SampleCloud Example

Play the Sound called *emerging intelligibility* and double-click on it to open an editor. Open the entire signal flow graph by holding down the **Command** or **Control** key and clicking the tab on the left side of the rightmost Sound:



If you double-click the rightmost Sound, you'll see that this is a *Mixer* of two *SampleClouds* bearing the strikingly original names *cloud1* and *cloud2*.

Double-click *cloud1* to have a look at its parameters. This *SampleCloud* is going to operate on a sample called `virtual DEF` granulating it by applying the sample called `gaussian` as the envelope on each individual grain.

The *SampleCloud* picks short segments of **GrainDur** duration from the sample file indicated in the **Sample** field at the position indicated by **TimeIndex**. To each of these grains, it applies an amplitude envelope whose shape is indicated by **GrainEnv**. The number of grains that can be playing at any one time is controlled by **Density** up to the maximum indicated in **MaxGrains**. **Pan** controls the stereo position of the grains, and **Amplitude** gives an overall amplitude for the entire cloud of grains. **TimeIndex**, **GrainDur**, and **Pan** can each be assigned some amount of random jitter in their values, so there can be some variation in where each grain is taken from the sample, how long each grain lasts, and where it is placed in the stereo field.

TimeIndex controls the location where grains will be chosen within the sample called `virtual DEF`, with `-1` marking the beginning of the sample, `0` marking the middle, and `1` marking the end. In this example, **TimeIndex** is set to:

```
1 repeatingFullRamp: (10 s)
```

which means that it changes from `-1` to `1` over the course of 10 seconds and then repeats. The effect of this is to move in normal time order through the sample from beginning to end and then to wrap around to the beginning again.

However, this nice orderly progression through the sample is interfered with by the **TimeIndexJitter** parameter, controlled here by another Sound called *expDown*. **TimeIndexJitter** is the amount of random jitter added to the position given by **TimeIndex**. In other words, its value is an indication of how much variation there should be in exactly where the grains are chosen from the sample.

Let's take a look at *expDown*, but first, in the interest of saving your speakers, turn your gain on you amplifier all the way down. Then select *expDown* in the signal flow graph, and plot its shape by choosing **Full waveform** from the **Info** menu. Unfortunately *expDown* lasts for 60 seconds, so you can run and get yourself something to drink while it plays and plots its full waveform on the screen. When you get back from your break, you can see from the full waveform of *expDown* that the amount of random jitter in the **TimeIndex** starts out high and decreases exponentially over the course of 60 seconds until it is almost zero. In other words, at the beginning of this Sound, grains could be taken from anywhere within the sample (because the jitter amount is large), but by the time 60 seconds have gone by all the grains are being chosen from around the current value of **TimeIndex**.

expDown also controls the random variation in grain duration, so initially, there will be a lot of different duration grains, but by the end, all the grains should be about 0.1 seconds long. Similarly for **Pan**: the grains could start out anywhere in the stereo field, but by the end, they are all being placed about a quarter of the way towards the left speaker.

Density, on the other hand, is controlled by a Sound called *expUp*, so we can make a reasoned guess that the density of grains per second is going to exponentially increase over the duration of this Sound, with the density starting out very sparse and ending up almost continuous.

Flip back and forth between *cloud1* and *cloud2* to see if you can detect how the parameters differ. (Hint: one of them is a parameter we haven't discussed yet...)

Seed is used to "seed" or start the random number generator. By giving each of these two *SampleClouds* a different seed, we can ensure that they will not be exact duplicates of each other, because the stream of random numbers used for parameter jitter will be unique for each cloud. Thus, by adding them together in the *Mixer*, we can get a higher maximum density of grains at any one time (in this case 50 at a time).

The other parameter that is different between the two clouds is **Pan**, meaning that one of the clouds will tend to be toward the left and the other will tend to be towards the right in the stereo field.

Modifying the SampleCloud

Close this Sound, create a new Sound file window, drag a copy of *emerging intelligibility* into the new window, and double-click on it to edit. Any time you drag a Sound from one window to another (including from the prototypes or the Sound editor) you are automatically dragging a copy of that Sound, not the original. So you can modify this new one without concern that you might also be changing the original.

Edit *cloud1*, setting the parameters as follows:

Parameter	Value
Duration	on
Amplitude	!Amp
TimeIndex	1 repeatingFullRamp: (!Rate * 100) s)
TimeIndexJitter	!TJitter
GrainDur	!GrainDur s
Density	!Density

Play *cloud1* and adjust the parameters in the virtual control surface. Try the following initial settings:

Fader	Value
!Amp	0.3
!Density	0.4
!GrainDur	0.1
!Rate	0.6
!Tjitter	0.0

Experiment with adjusting each of these parameters, one at a time. !Rate controls the speed at which the **TimeIndex** moves from -1 to 1 using the repeatingFullRamp function. The maximum amount of time it takes is 100 seconds, and the minimum is 0 seconds (*i.e.* as fast as possible).

Double-click on *cloud2* and set all of its parameters to the same values as the parameter values in *cloud1*. The quick way to do this is to drag the *cloud1* icon into each of the parameter fields of *cloud2*, one by one. (You can use this little trick with any two Sounds that have parameter names in common).

Change the **Sample** parameter of both *cloud1* and *cloud2* to Violin (or to one of your own samples).

Play the mix of *cloud1* and *cloud2* and adjust the parameters to your liking. To save these settings, drag a **Preset** from the **MIDI In** category of the prototypes onto the line between the **Mixer** and the output speaker icon. Double-click the **Preset** and press the button called **Set to current event values**. From now on, when you play the **Preset**, it will set all of your parameters to those initial values.

With **Preset** selected, press **Enter**, and enter a descriptive name for the modified Sound. Then close the Sound editor, and save the Sound file window in the folder with your other Sound file.

Compression, Expansion

The Sounds in this file are examples of using the *DynamicRangeController* (DRC) module to compress or expand the dynamic range of another Sound. The DRC performs a time-varying attenuation on its **Input** based upon the amplitude envelope of the Sound used as its side chain input.

Compression

The idea of compression is to take the full range of amplitudes and compress them into a narrower range by attenuating the larger amplitudes, and then optionally scaling all of the amplitudes up so that the low amplitude parts of the signal become much louder relative to the maximum.

By way of analogy, if you were listening to music through speakers and kept your hand on the volume knob of the amplifier, and turned it down whenever it got loud and turned it back up again whenever the music got soft, you would be acting as a compressor. Of course since there would be some amount of delay between when you heard the music get loud and when your reflexes actually managed to turn down the volume, this would work better if you could listen to the music directly over headphones and could delay the output through the speakers by some small amount. For the same reason, there is a delay parameter on the DRC, so that there is time for the DRC to attenuate sudden changes in the amplitude envelope.

You can use a compressor to prevent an input from ever getting too loud; for an example select *orig, limiter (Info-Full waveform)*, and choose **Full waveform** from the **Info** menu. Another application of compressors is removing unwanted amplitude variations, caused, for example, by performers moving around with respect to the microphone (plot the full waveform of *orig, smoothed out (Info-Full waveform)* as an example).

There is also a characteristic timbre to compressed sounds, and it can make percussive and plucked string sounds seem, psychoacoustically, louder or “punchier” by reducing the difference between their sharp attacks and quick decays — in effect stretching out the decay time so that the sound stays louder for a longer period of time.

For an explanation of the DRC parameters, find the *DynamicRangeController* in the **Dynamics** category of the prototypes. Select it, and then choose **Describe Sound** from the **Info** menu. Leave that window open, and play the Sound called *Compressor w/hot controls* in the *compression, expansion* Sound file window. Then you can experiment with different parameter settings.

Expansion

Expansion (not too surprisingly) is like the opposite of compression; it takes amplitudes that might have been very close together in the original amplitude envelope, and spreads them further apart in the amplitude envelope of the output. For example, you might want to take all amplitudes below some noise floor and specify that they should be mapped to zero amplitude (or close to zero amplitude), creating a kind of “noise gate” to remove unwanted background noise when it is softer than the desired signal. Examine the Sound called *noise gate* to see an example of this.

Experiment with the Sound called *Expander w/hot controls* to hear the effect of different parameter settings.

Other Effects

By using different Sounds at the side chain and the input, you can create other effects like ducking, gating, or de-essing.

Take a look at the Sound called *Ducking*. It is set up as a compressor with a speaking voice as the side chain and a violin sample as the **Input**. Whenever the speaking voice is triggered, the violin sample “ducks” behind it into the background.

use voice to gate another sound is similar except that it is set to expand instead of compress, so when the voice is triggered, it opens the gate on the *Noise* input, and when the voice is silent, the gate is closed.

In *pick out unvoiced* the EX version of a sample is used as the side chain and the original sample is used as the **Input** to an expander. Since EX files emphasize the high end of the spectrum, only the highs open the gate. As a result, only the consonants make it through the gate, the vowels are blocked out. In *virtual lisp*, *pick out unvoiced* is used as the side chain, and the original sample is used as the **Input** to a *compressor*. As a result, the consonants are suppressed and the vowels pass through unattenuated.

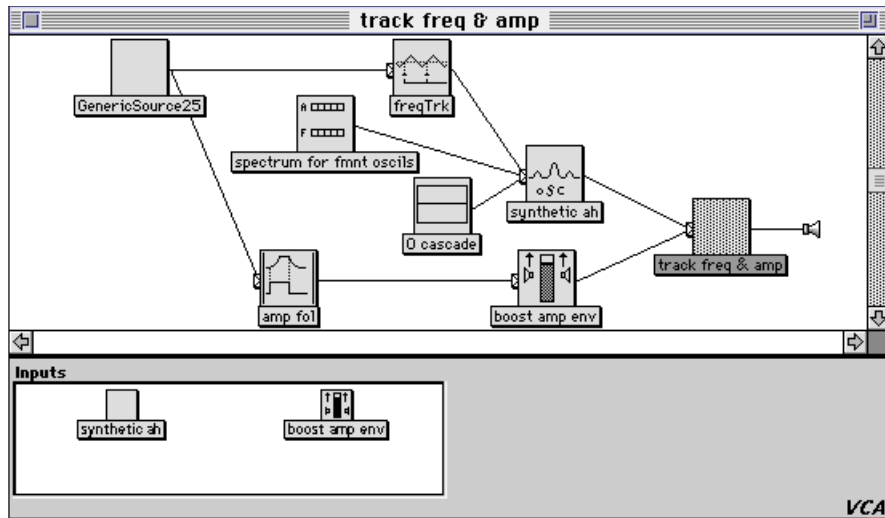
Live lisper uses a *LiveSpectralAnalysis* to pick out the unvoiced parts of a *GenericSource* and uses that as the side chain to a compressor with the same *GenericSource* as **Input**. Try this out on the default speech sample, and then try playing it again, this time choosing the live input, so you can try removing the consonants from your own voice.

Cross Synthesis

The idea of cross synthesis is to track a parameter of one sound and use it to control that same parameter or a different parameter in another sound. The result is the superimposition of some of the characteristics of one sound onto another.

Simple Cross Synthesis

Double-click *track freq & amp* and open the signal flow graph completely by holding down the **Command** or **Control** key while clicking on the arrow tab on the left edge of the rightmost Sound. To make it clearer, drag the *GenericSource* further to the left, and position *0 cascade* and *spectrum for fmnt oscils* so they are not on top of each other.



Select *GenericSource25* and play it. Then select *track freq & amp* and play it. Can you tell what is going on (especially given that preface in the first paragraph)?

The rightmost Sound is a *VCA* since it imitates a “voltage-controlled amplifier” (actually more like a 4-quadrant multiplier). In actuality, all it does is multiply its two inputs — *synthetic ah* and *boost amp env*.

Follow the *boost amp env* thread. *boost amp env* itself is a *Gain* that boosts its input by eight times. Its input, *amp fol* is an *AmplitudeFollower*. Double-click it to see its parameters. To read a description of how this kind of Sound works, click on the italicized label *Amplitude Follower* in the lower right corner of the parameter fields. This opens a help window containing a description of the Sound and each of its parameters.

After you have read the description of what the *AmplitudeFollower* does, select the *GenericSource25*, and use **Full waveform** from the **Info** menu to see how the amplitude changes over time. Leave that window open, select *amp fol*, and plot its full waveform. Line up the two windows, one below the other to compare them.

Because of the absolute value, the output of the *AmplitudeFollower* is all above zero. Because of the averaging, it is a little sluggish in its reaction time. For example, at around 0.7 seconds, there is a big spike in the unprocessed waveform (the “tch” in “virtue”), but in the amplitude envelope, this spike never gets very large, because the *AmplitudeFollower* is averaging over 0.1 seconds. Try changing the **TimeConstant** to one tenth of its value: 0.01 s. Then plot the output of the *AmplitudeFollower* and compare the size of the spike. By decreasing the **TimeConstant**, you can make the envelope respond to shorter features in the waveform. Take note of the minimum and maximum values written at the top of the waveform window. That should explain why we needed to feed the output of the *AmplitudeFollower* into a *Gain* before using it as an amplitude envelope on *synthetic ah*.

Go back and double-click on *synthetic ah* now to follow *that* thread. This is a *FormantBankOscillator* whose **Frequency** is

```
freqTrk L * SignalProcessor sampleRate * 0.5 hz
```

The frequency seems to be coming from the Sound called *freqTrk* ... but what does the rest of that line mean? Recall that the output of a Sound is always in the range of $(-1, 1)$, and in the case of the *FrequencyTracker* it is within an even narrower range of $(0, 1)$. But frequencies are in a larger range: 0 hz to half of the sampling rate. (Remember, this is a digital system, so the highest representable frequency is half of the sampling rate). By multiplying by half of the sampling rate, we assure that whenever *freqTrk* is at its maximum value of 1, the frequency will be half of the sampling rate:

```
SignalProcessor sampleRate * 0.5
```

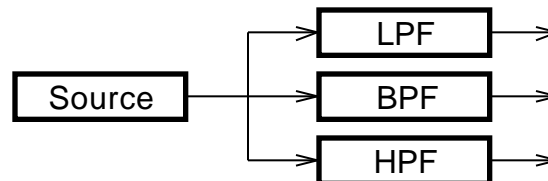
This little expression is so handy that we put it in the list of hot parameters, so you can simply paste it into a parameter field rather than having to type it. Try it now by doing a **Ctrl+H** (hot paste). The second item on the list is **Half the sampling rate**.

Double-click *freqTrk* now. This Sound is trying to track the fundamental frequency of its input. The more you know about the frequency range of the **Input**, the better it can do at tracking the fundamental frequency. In this case, we know the **Input** is within the range of 3a to 4c, so entering those as the **Min-Frequency** and **MaxFrequency** respectively, helps the frequency tracker avoid octave errors. As far as all the other parameters, it is best to leave those at their default values.

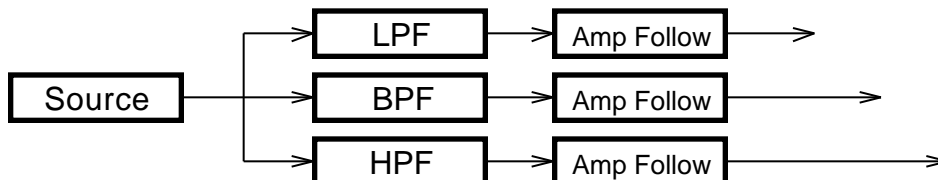
This is a synthesized sound whose frequency and amplitude envelopes come from a sampled sound — a simple example of cross synthesis.

More Amplitude Followers!

Imagine taking the *GenericSource* and feeding it into a bank of 3 filters, so that one filter covers the low frequencies, one passes the midrange, and the last one's covers the high frequencies.

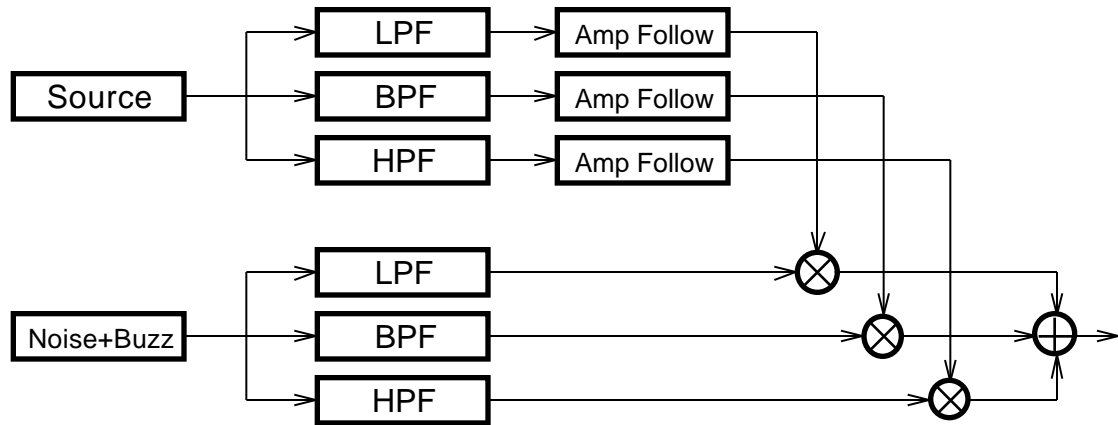


Now imagine putting an amplitude follower on the output of each of those filters. That way, the first envelope would show how much energy was in the lower frequencies, the second would show how much energy was in the midrange, and the last one would show how much energy was in the high frequencies, each one showing how that energy was changing over time.



And finally, imagine multiplying each of those envelopes by a synthesized signal, just as we did with the *synthetic ah*. Instead of a *FormantBankOscillator* though, imagine multiplying by the output of three different filters (independent of the first three filters, but with the same settings) with a mix of white noise and a harmonically-rich buzz oscillator as their input.

As a final step, imagine adding the outputs from all three filters together using a mixer:



Now play *Traditional vocoder-22 bands*, which is pretty much what we have just described except that there are 22 pairs of *bandpass* filters, not just three filters as shown above.

Classic Vocoder

Use `!Frequency` to change the frequency of the buzz oscillator. `!Noise` crossfades between the noise and the buzz oscillator inputs so that it is purely noise at the top and purely the buzz oscillator when it is at zero.

`!TimeConst` is the same as the `TimeConstant` in the *AmplitudeFollower* of the previous example. It controls how quickly the 22 amplitude followers will respond to changes in the filters' outputs. Higher values will give a reverberated quality, while very small values will make the speech more intelligible.

`!Bandwidth` is a control on the bandwidths of all 22 bandpass filters. Set the `!Noise` fader all the way to the top, and then gradually decrease the value of `!Bandwidth`, making the filters ring more and more until they are almost pure sine waves (because that is all that can fit in such a narrow band). You may have to boost `!InLevel`, the level of the noise input, for very narrow bandwidths because less of the signal can get through when you make the pass band so narrow.

Finally, set the `!Live` fader all the way to one, and try speaking into the microphone. At this setting, you control the filters with your voice.

This is an example of imposing the amplitude and spectral content of human speech onto some synthetic sources like noise or an oscillator. You can use the same concept to impose characteristics of human speech onto the sound made by an animal or machine. For example, try out *AD-mr dophin*, selecting `Live` when the *GenericSource* asks for the source. Now is your chance to tell off the smart dolphin from the first example.

Now Add Frequency Deviation

You, the ever astute reader are no doubt asking "Aha, but what about the frequency tracking?" remembering that the original example had a frequency tracker along with the amplitude follower. It turns out that if you add frequency tracking within the band of each of the bandpass filters in the previous section, you end up with the algorithm used by the *LiveSpectralAnalysis* Sound and the Spectral Analysis Tool (which we will cover in depth in subsequent tutorials). The result is a set of amplitude envelopes just as before, *along with* a set of corresponding frequency envelopes. Each amplitude and frequency envelope is used to control the amplitude and frequency of a single oscillator. Then all of the oscillators are added together for the final output.

Try playing *piano man*. This is an example of resynthesis using the frequency envelopes from one analysis and the amplitude envelopes from another. More accurately, the amplitude envelopes are gradually cross-faded from one analysis to the other.

FFT cross synth

Try playing *talking forced air heater*. This Sound takes the spectral envelope of the *GenericSource* called *modulator (take spect env of this)* and, in the frequency domain, multiplies it by the spectrum of *input (apply spect env *to* this)*. Then it applies the FFT again to bring the product back into the time domain where you can hear it as the basic sound of the input (in this case the sound of a forced-air heater) with the formants of the modulator (in this case the sound of speech). *fft cross voice and harp* is another example of this particular flavor of cross synthesis.

RE Analysis Tool

RE, or resonator-exciter analysis, is another variation on the theme of crossing the formant characteristics of one sound with the basic spectral content of another. The idea is to start with a sample or digital recording, and to break it down into two parts: a time-varying filter and an input signal for that filter. Once you have broken the original sound into those two components, you can begin mixing and matching components — feeding the input signal from one into the time-varying signal of another, or feeding samples or even synthetic signals into the filter.

“Oh really, Mr Dolphin?”

To hear an example of this kind of synthesis, play *smart dolphin*, and smoothly change !Index from 0 to 1. Keep trying until you hear what the dolphin is saying (the trick is to move the !Index fader smoothly and to take about 1.5 to 2 seconds to get from 0 up to 1).

This example is a looped dolphin sample being fed into an RE filter. As you move !Index you are changing the coefficients for the filter — controlling the rate at which this filter changes over time (somewhat analogous to controlling the rate at which you change the shape of your mouth and position of your tongue when you speak).

Grain Man

granulation into RE filter is an example of feeding a synthetic signal through an RE filter. Trigger the time-varying filter using the MIDI keyboard. Control the rate of the grains using !Rate on the virtual control surface. Try setting !Rate to 1, triggering the filter, and then moving !Rate to a smaller number while the filter is playing. This sounds like the audio counterpart to the graphic special effect that looks like someone turning into lots of tiny spheres and disintegrating.

Wielding your Tools

Let's try creating one of these filters using the RE analysis tool:

1. From the **Tools** menu, select **RE Analysis**. It opens an analysis window with several parameters.
2. Use the **Browse...** button to choose the sample *Virtual DEF* from the *Speech* folder of the *Samples* folder of the *Wavetables* folder. It will play back once for verification.
3. Leave the settings for **Filter order**, **Update rate**, and **Averaging time** at their default settings. The filter order corresponds to twice the maximum possible number of resonances in the final filter (some of the resonances may simply reinforce each other rather than being at different frequencies). Later, you can try analyzing again with 64 as the order number to hear if it makes a difference.
4. Click the button named **Create RE only**, indicate that you want to save the file in the RE folder of the *Wavetables* folder, and *wait* until the thermometer display indicates that the analysis is complete.

At the completion of the analysis step, Kyma will create and play an example Sound based on the RE file just created. This is the RE filter with noise as the input.

Double-click on the Sound to edit it. Change the parameters of *time index* Sound as follows:

Parameter	Value
Duration	on
Trigger	!KeyDown
OnDuration	6.02 s * (1 - !Rate) * 2

Next, edit the *Noise* module, changing its **Duration** to on.

Play this edited version of the Sound, adjusting !Rate to go through the sequence of filter coefficients more slowly or more quickly.

Find the Sound called *alternate input to RE filter* in the Sound file window. Substitute this Sound for *noise input* in the Sound editor on *Virtual DEF RE*. You can do this as follows:

Select *alternate input to RE filter*

Copy it

Return to the Sound editor and select *noise input*

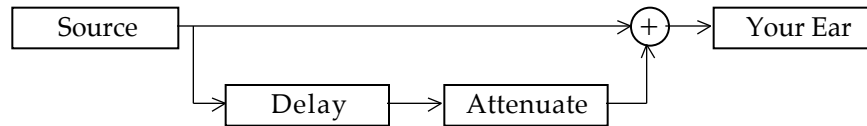
Paste

Play the newly altered version of *Virtual DEF RE*. This example imposes the formants of the speech onto the basic material of the harp gliss.

Experiment with substituting some of your own, broadband samples for the harp gliss in the *Generic-Source* when it asks. In general, the best kinds of inputs are those which are continuous, have a narrow dynamic range, and a rich spectral content (*i.e.* the closer they are to continuous white noise, the better the results).

Delays, Chorusing, Reverb

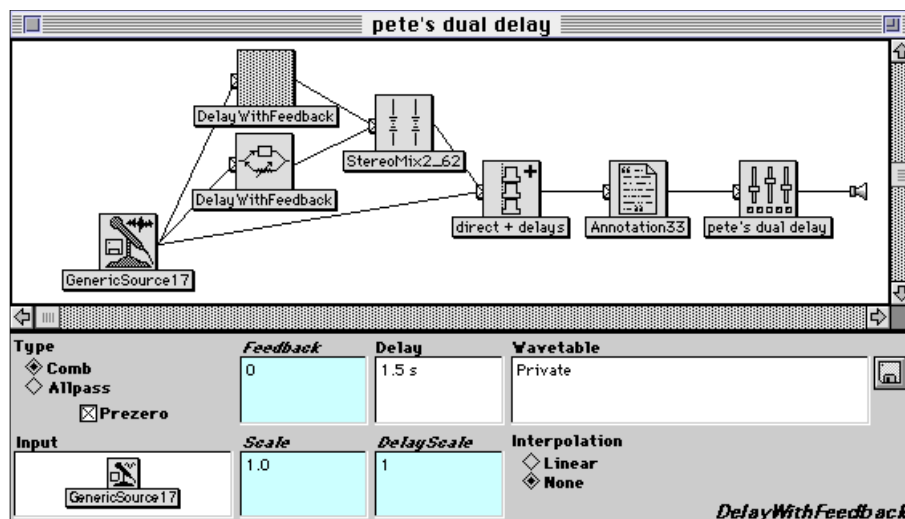
Think for a moment about how an echo occurs. You shout or clap your hands, sending out a sound wave that propagates out in all directions, a growing sphere emanating from your hands or mouth. It takes some time for the sound to travel (around 600 meters per second) until it reaches a cliff wall or the flat side of a building, where, like light on a mirror, it reflects back to you (having lost a little bit of its energy to friction with air molecules on its way there and back and also when it ran into the wall which absorbed some of the energy). To say the same thing diagrammatically:



In other words, you would hear the clap directly at your ear (after a negligible delay), and then again with attenuation after it had bounced off the building.

Delays

Double-click on *pete's dual delay*. To see the entire structure, click the tab on the left edge of the *Annotation* while holding down the **Control** or **Command** key. Now, to make the structure more clear, drag the *GenericSource* downwards and to the left until it is clear that the *GenericSource* flows directly into the *Mixer* called *direct + delays* as well as into two different *DelayWithFeedback* modules.



If you double-click each delay, you can see that the upper delay (feeding into the left channel) is 3 seconds and the lower delay (feeding into the right channel) is half that long. Try playing *pete's dual delay* with the default source. First you hear the direct source, then the left echo, and then the right echo. Play it again, this time choosing **Live** as the source. Sing your favorite round (e.g. "Row, Row, Row Your Boat", your favorite Bach canon, or whatever). By fitting each phrase of the round into 3 seconds, you can get the canon effect with direct sound and the two delays. Experiment with singing (or playing) arpeggiated chords into the microphone.

Chorusing

Open up *chorusing using random*, and use **Ctrl+click** to display the entire signal flow diagram. This Sound, like *pete's dual delay* has the direct source added to some delayed versions of the source. But in this case, the delays are much shorter than the 3 second and 1.5 second delays of the previous example.

Select the *GenericSource*, and play it by itself. This is a recording of Pete Johnston singing his own favorite canon into *pete's dual delay*.[§] Now compare it to the chorused version by selecting and playing *chorusing using random*. In the virtual control surface, type in a value of 0 for the `!Delay`, and hit **Enter** or **Return**. This is the source with no chorusing. Then type in a 1 and hit **Enter** or **Return**. This is the maximum chorusing.

How is it that we can almost always tell the difference between one person singing and several people singing in unison? One factor is that different sound sources are different distances from your ear, so there should be slightly different delays on each performer. Another is that the performers may be slightly out of tune with each other. Both of these effects can be modeled using a *DelayWithFeedback* with a variable delay time.

Look at the parameters of *10-25 ms*. The **Delay** is set to 25 ms and the **DelayScale** (the proportion of **Delay** that is actually used) is

```
0.7 + ( randLFO1 L abs * !Delay * 0.3)
```

In other words, the maximum delay time of 25 ms is multiplied by a number that varies between 1 (0.7 + 0.3) and 0.4 (0.7 - 0.3), depending on the value of `!Delay` and of the low frequency oscillator.

The low frequency oscillator is a sine wave oscillator with a subaudio, random frequency. Take a look at *randLFO1*, in particular, its **Frequency** parameter:

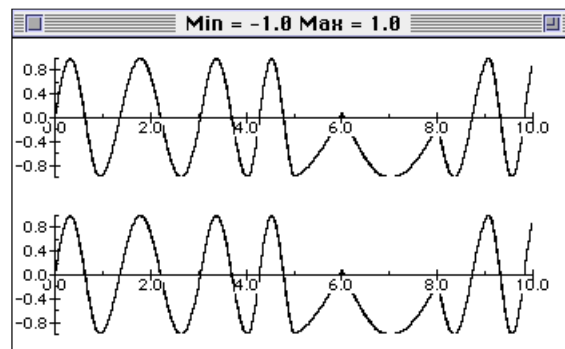
```
0.37 hz + 1 s random hz
```

This looks a little confusing because there is a time value in a frequency field, but the time just refers to how often a new random number should be generated. The expression

```
1 s random hz
```

generates a random frequency between -1 hz and 1 hz and changes to a new random frequency once per second. When added to 0.37 hz, it gives the LFO a frequency range of -0.63 to 1.37 hz. A negative frequency tells the *Oscillator* to read through its wavetable backwards. In the sine wave case, a negative frequency sounds the same but is 180 degrees out of phase from the same frequency without the minus sign.

Take a look at some of the output of *randLFO1* by selecting it, choosing **Full waveform** from the **Info** menu, and asking it to plot 10 seconds of the output. (Because the *Oscillator's* duration is `on`, you must specify the amount of time to plot). This signal is controlling the delay, so the delay times will vary smoothly above and below some fixed delay but at a random rate.

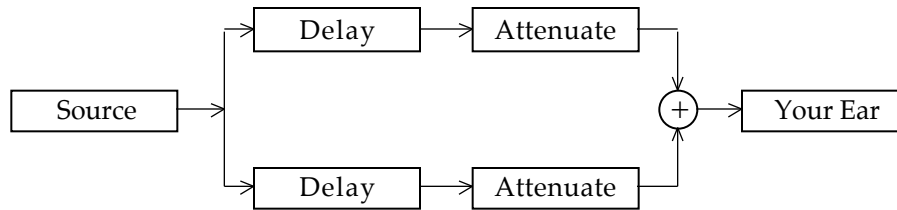


Reverberation

When a sound wave propagates within an enclosed space, it takes some amount of time to get from the sound source out to the edges of the enclosure where it runs into a wall; some of the energy is absorbed by the wall, and some of it gets reflected back into the room, taking some additional time to get to your ear.

[§] Besides being the Technical Director and Kyma guru at Tape Gallery in London, Pete does a pretty good imitation of Freddie Mercury in this recording (though he says he can do even better when he is not out of practice).

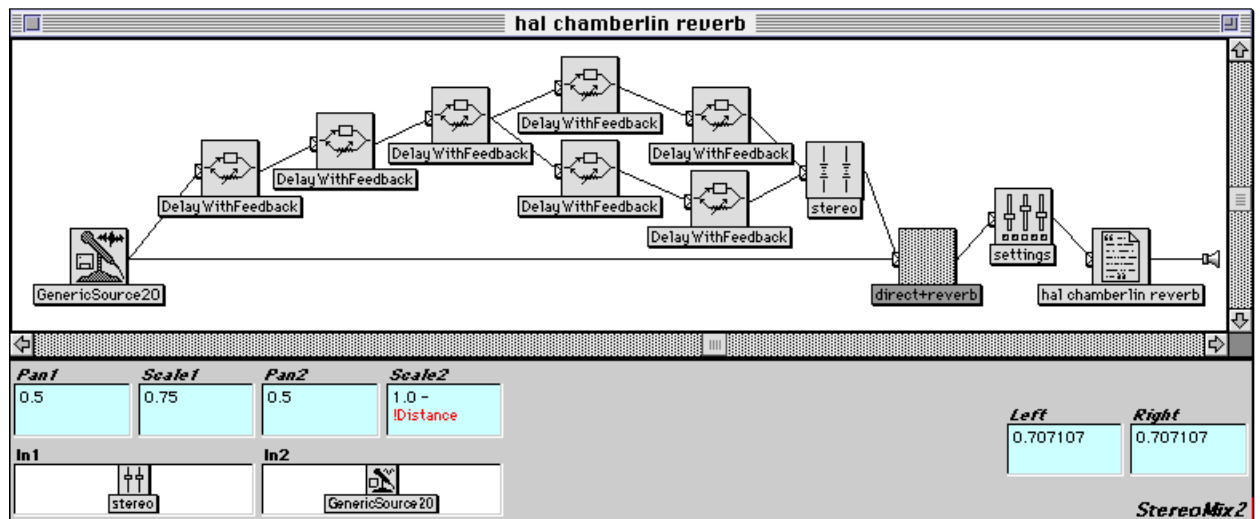
So you could model a sound source in an enclosed space as:



The delays model the amount of time it takes for the sound to get from the source to your ear, or from the source to the wall, bounce off the wall and get back to your ear. The attenuators represent how much of the sound is absorbed by wall or lost to friction in the air before it gets to your ear. To model a real room, you would need to add the effects of many of these delays with attenuation, and essentially that is what reverberation models are — combinations of delays and attenuators. Of course, finding just the *right* combination of delays and attenuators is both a science and an art.

Now you, too, can dabble in the art of reverberation using some basic elements found in Kyma, especially the *DelayWithFeedback*, the *HarmonicResonator*, the *ReverbSection*, the filters, and one or more mixers.

Several classic reverberation models are based on allpass and comb filters combined in parallel (in a mixer) or in series (one feeding into the next). For example, open the Sound example named *hal chamberlin reverb*, and **Ctrl+click** the tab on the left side of *settings*, so you can see the entire structure. Drag the *GenericSource* down and to the left until it is clear that the *GenericSource* feeds directly into *direct+reverb* and through two different networks of delays to the nested allpass filters, with the final stages of the left and the right channels going through slightly different chains.



Play *hal chamberlin reverb* and adjust the faders in the virtual control surface. Double-click on one of the *DelayWithFeedback* modules to look at its parameters. You can use the *DelayWithFeedback* to implement either an allpass or a comb filter by selecting the appropriate filter type. Check the **Prezero** box so that the delay line is filled with zero when it starts up, so that you don't hear whatever happens to be left in the delay lines from the last time. By setting **Feedback** to some nonzero value, you can generate many attenuated echoes in a single module, rather than having to add each delay and attenuation one-by-one.

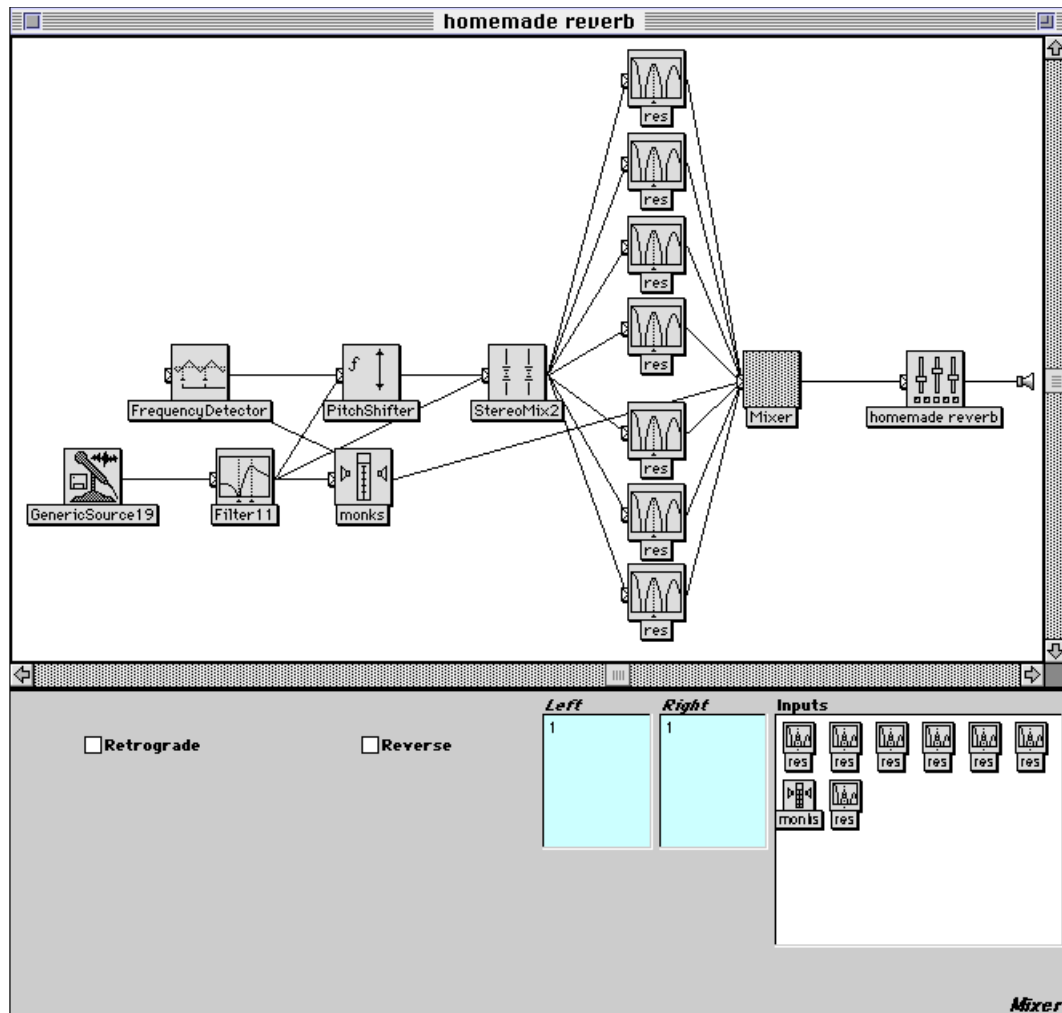
Next, take a look at *homemade reverb*. This is an example of *HarmonicResonators* used in parallel with each other. Play it first and experiment with different settings in the virtual control surface. Then double-click it to have a look inside. This is an example of creating a Sound structure algorithmically, using a *Script*.

Double click *DIY reverb*, and look at its script:

```

1 to: 7 do: [ :i | res start: 0 s freq: (25 + (0.5 * i)) hz].
monks start: 0 s.
  
```

This script creates seven copies of the *HarmonicResonator* called *res*, each with a slightly different value for the **Frequency** parameter, and one copy of *monks*. It feeds all eight of these Sounds (the seven reverberated and one direct copy of *monks*) into a mixer. To see the result of the *Script*, select *DIY reverb*, and choose **Expand** from the **Action** menu. You will see a warning that you cannot recover the original Sound if you replace it with its expansion, but just ignore that for now, because we don't have to save the changed version. You should see a *Mixer* with lots of *res* inputs:



Take a look at the parameters of the *res* modules. Each one has a slightly different value for **Frequency**. You have probably also noticed that the red Event Values have changed to blue Event Sources. These are system names for the actual MIDI controllers that the red Event Values are mapped to in the global map. In other words, the red names are just mnemonic (*i.e.* easier to remember) names for the system MIDI controller names. Close the Sound editor but do not keep the changes to *homemade reverb*.

Play *Euverb* and experiment with the parameters in the virtual control surface. This reverberation model, contributed by Eugenio Giordani, is constructed from comb filters, allpass filters, lowpass filters, and harmonic resonators combined in series and in parallel. Open it up to study how it was created. You may want to open up one signal path at a time, because it is fairly complex. On the other hand, if you're feeling courageous, it is a pretty impressive sight when you use **Ctrl+click** to open the whole thing at once!

Disk Recording, Playback

In Kyma, you can record to the disk and play back from the disk under the control of MIDI triggers or other triggers that depend on the audio signal (such as exceeding a threshold amplitude).

Playback

For example, select *8 disk files, 3 at a time* and use **Ctrl+Space Bar** to play it. Once it is downloaded, try playing the MIDI keyboard.[§] This Sound maps each of several sound effects stored on disk to a key on the MIDI keyboard. Since there are more keys than there are recordings, the key-assignment pattern repeats every 8 half steps on the keyboard. The *MIDIvoice* is used to specify how many of the disk files can be sounding at the same time. Unlike *Samples*, which must fit in the Cappybara's sample RAM, the only limitation on the duration of disk recordings is the amount of free space you have on your hard disk(s). On the other hand, you can get more polyphony using samples, and you can play through samples in reverse.

Looping

In theory, you cannot loop the files that you play directly from the hard disk. But *you* are a power user (we know because you have read this far in the tutorials)! So here is the secret: put a metronome trigger in the **Trigger** field. For example, take a look at *pseudo looping a disk file*. Its **Trigger** field is set to

1 bpm: (60.0/2.57211)

Where did this strange tempo come from? The metronome expects to see the number of beats per minute. So a setting of 60 would give us one trigger per second. If you divide that number by the duration of the recording in seconds, then the tempo will slow down for durations longer than one second and speed up for durations shorter than one second.

But how can you know the duration of the recording? One way is to open the file organizer (choose **File organizer** from the **File** menu), and select the name of the file in the list. All the information on the file is displayed at the bottom of the file organizer. Another quick way to get the duration is to set the **Duration** parameter of the *DiskPlayer* to 0 s. Then select the icon and choose **Get info** from the **Info** menu. Among other things, this will list the duration of the recording. In both the file organizer and the info window, you can select the duration (without the units), copy it, and paste it directly into the **Trigger** field as the divisor.

To see an example of how to loop a section from the middle of a disk file, open *looping 3 s from middle of Disk file* (don't you just love these names?).

Recording

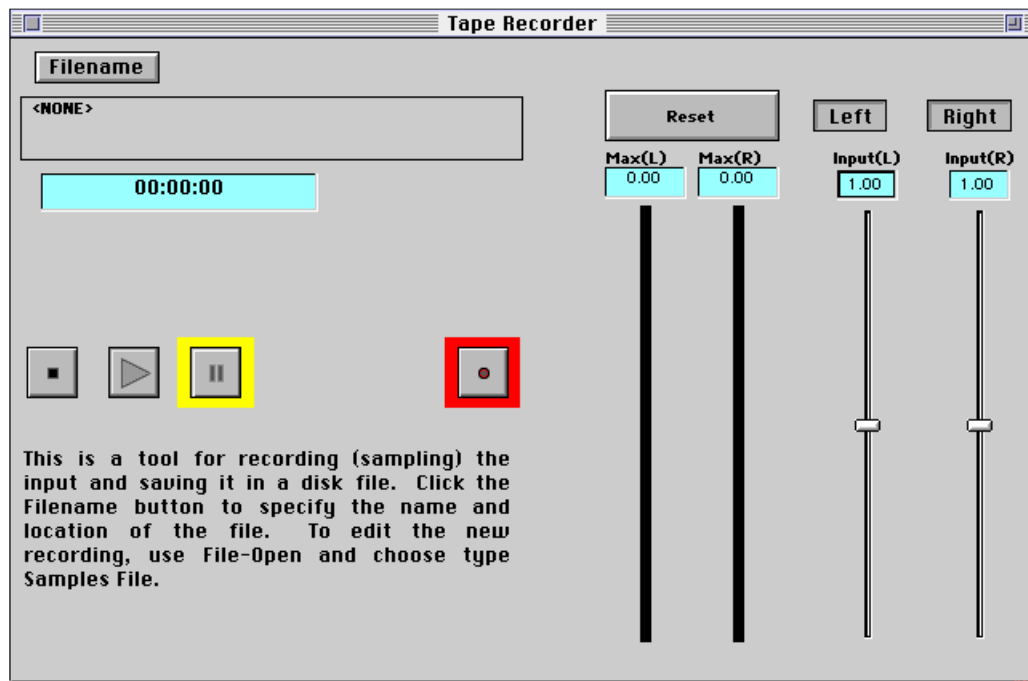
You can record any Kyma Sound to disk, including the *ADInput*, other *DiskPlayers*, Sounds that are too complex to be rendered in real time, and individual Sounds buried within a signal flow graph.

Tape Recorder Tool

The quickest way to record an external input to the hard disk to use the Tape Recorder Tool.

Try using it now to record your voice through the microphone. Choose **Tape Recorder** from the **Tools** menu. This opens a graphic interface mimicking the control panel of a typical tape recorder.

[§] You can stop a disk file playback or recording the same way you interrupt the playback of any other Sound: using **Ctrl+K**. If you ever find yourself in a situation where **Ctrl+K** fails to stop the Sound, the most likely reason is that the preferences have been set to disable user interface operations whenever a disk operation is taking place. Even when this preference item has been set, there is still one other way to stop the disk playback: by holding down the **Shift** key and clicking the mouse. Under most circumstances, you should use **Preferences** from the **Edit** menu to set the appearance preferences to **Update display during disk operations**. If you are doing a critical recording and find that user interface updates are interfering with recording, then you can switch off this preference item temporarily, and remember to use shift-click if you want to interrupt the recording. Once the recording is done, remember to switch the preference back so that Sounds that access disk recordings will not behave differently from other Sounds.



Click **Filename** to choose the format of the recording (set it to AIFF, 16-bit, mono, for example), and to choose where it should be saved on the disk.

Press the red record button (symbolized by a red circle on the button face) to put the tape recorder into monitor mode. This automatically presses the pause button and monitors the input. It won't start recording until you press the play button or unclick the pause button.

In monitor mode, test the input levels. **Max(L)** and **Max(R)** show the maximum levels seen so far at the input. Click **Reset** to reset those to zero. Attenuate the input using the two faders on the right.

When you are ready to record, press the play button (symbolized by the large rightwards pointing arrow), and start speaking.

To stop recording, press the stop button on the left (symbolized by a black square at the center of the button).

To listen to the recording, press the play button.

When you are done, click in the close box of the window.

Action-Record to disk... and File-Play

This Sound file also contains examples of other variants on the record-to-disk theme. For example, you can record *any* Kyma Sound to the disk; you are not limited to recording the *ADInput* only.

Try recording the *Concatenation* example in the prototypes (**Algorithms** category). To record any Sound to disk, select its icon, and choose **Record to disk...** from the **Action** menu. Give this a try with the *Concatenation* in the prototypes. Record a 16-bit mono AIFF format file to your desktop, giving it a name like *deleteMe* so you can get rid of later and keep from cluttering up your disk. To play the recording, select **Play...** from the **File** menu, and locate *deleteMe* in the file dialog. This is a quick way to play disk files when you don't need to edit them.

Listen Wet, Record Dry

The **Action** menu option is fine for recording a complete Sound to the disk, but what if you want to have the disk recording operation occur as one sub-part of a larger Sound structure?

Then your best bet is to use the Sound called *DiskRecorder* (found in the **Disk** category of the prototypes). Double-click on *record dry, listen processed* to see how it is put together. This Sound takes a *Generic-Source*, feeds it into a *DiskRecorder*, and then processes the output of the *DiskRecorder*. A structure like

this could be used to let a performer listen and adapt to the way the sound is being processed, while capturing the dry, unprocessed performance on disk. This leaves *you* with more flexibility for editing the dry performance or changing something about the processing later.

Double-click *record dry input* to see its parameters. A **CaptureDuration** of 0 s indicates that the entire duration of the **Input** should be captured. If you want to capture a specific amount of time, then enter that amount of time in this field. You may be tempted to type `on` here, but if you do, Kyma will warn you that your disk is not large enough to store a 2 year recording (unless you have a very, very, *very* large hard disk).

FileName is the name of the file that you want to record into. If you don't provide a full pathname for the file name, Kyma will automatically place the file into the `Program` folder of the `Kyma` folder. To record the sound elsewhere on your disk, change the appearances preferences (click on the **Appearances** button after choosing **Preferences** from the **Edit** menu) to display the full path name of files. Then go back to the *RecordToDisk*, use the disk button next to the **File** parameter field to select a different file in the target folder, and replace the file name part of the full pathname, leaving the rest of the pathname unchanged.

Trigger is set to 1, meaning that the recording is triggered immediately, as soon as you load this Sound. If you are recording from the microphone, though, there is one more level of protection before it actually starts recording to disk. Kyma will give you a metronome count-off just before the recording starts.

The remaining parameters have to do with the format of the disk recording: **WordSize** is the number of bits-per-sample (the resolution), **Format** is one of the standard sample formats, and you can choose whether you want to record two independent channels or a mono mix of the two channels of the **Input**.

Leave this Sound editor open, so you can modify the Sound in the next section.

Triggered Recordings

You can probably guess from looking at the parameter names how one might go about triggering a recording rather than always having the recording start as soon as the Sound is loaded.

Modify the *DiskRecorder* in the *record dry, listen processed* example as follows:

Set **Trigger** to `!KeyDown` (by pressing **Escape** and then two simultaneous keys on the keyboard)

Set **CaptureDuration** to 3 s

Once the Sound starts playing, you can monitor the processed input continuously, but it will not start recording to disk until you press a MIDI key.

Select and play *record dry, listen processed*. When it asks for a source, choose the sample called `virtual DEF`. Kyma may ask if you want to record over the file named `deleteMe` which, of course, you do.

Once you hear the processed voice, hit a MIDI key. At that point it will record for 3 seconds and then stop.

Check the results by holding down the **Control** or **Command** key, and clicking on the disk button next to the **FileName**. This is a handy way to open the waveform editor on the file named in that field.

Click in the waveform part of the editor, select-all (**Ctrl+A**), and press the play button to hear what you have recorded. It should be three seconds of the unprocessed voice.

Close *record dry, listen processed* without saving the modifications that you have just made.

Multi-tracking

Recording a New Track in Synchronization to an Old One

Suppose you would like to synchronize a new track to one that you recorded previously. Double-click on *listen Trk1, record Trk2* to see an example of how to do this.

This is a *Mixer* with a *DiskRecorder* and a *DiskPlayer* as its inputs, so it plays back one disk file while recording a new one. You can use this kind of structure to make a new recording while listening to the old one over headphones, for example.

Mixing Two Disk Files Down to One

Double-click *mix 2 files down to 1* to see how it works. The rightmost Sound is a *DiskRecorder*, so it does the recording of the two other files.

Next is a *Preset*. This saves the last position of the faders, so they don't start out at zero.

Feeding into that is a *StereoMix2* with hot parameters controlling the level and the pan position of each input. Each of the inputs is a *DiskPlayer*. So the output of the *StereoMix2* is a mix of the two input files as controlled in real time by the faders in the virtual control surface.

Select *mix 2 files down to 1*, and instead of playing it normally using **Compile, load, start...**, select the next option lower in the **Action** menu: **Compile & load**. This will compile the diagram and load it in the Capybara but will not start playing the Sound until you are ready.

Next, click in the virtual control surface so that it comes to the front; this way you will be able to control the faders while the Sound is being recorded.

Once you are ready to start recording, use **Ctrl+R** to "replay" the Sound that is currently loaded in the Capybara. Click the mouse at different fader levels during the recording.

To audition the result, control-click on the disk button next to the **FileName** field in the *DiskRecorder*. Select-all, and use the play button to listen to the recording.

Reading an Old Track, Processing it, Recording into a New Track

Double-click *oldtrk->processing->newtrk*. Control-click the tab of the rightmost Sound to completely open up the signal flow diagram. This is an example of a *DiskPlayer* that is fed into some processing, and then fed into a *DiskRecorder* to record the processed track as a new track.

Try playing this Sound, recording into our old standby, the *deleteMe* file. Have you been noticing that the cursor has a black dot inside during disk operations? As soon as the cursor changes back to normal, use control-click on the disk button if you would like to check the results.

Caching

Caching is related to multi-tracking, but with a few useful twists. A cache is a place for storing something where you can pick it up quickly at some later point. In this case, the idea is to compute some sound ahead of time and store it somewhere on the disk where you can read it exactly when you need it. You can use *DiskCaches* to record the fixed, unchanging parts of your Sound to disk and free up additional processing time and sample RAM resources for the parts of the Sound that are performed in real time.

A *DiskCache* module can be placed anywhere within a Sound structure, and it serves as a storage place for everything to its left in the signal flow diagram. Double-click *disk caching example* to see an example.

Double-click *cache the crystals*. If you play the Sound when **Record** is checked, it will record everything to its left into its cache. If you play it with **Record** unchecked, it will play back from its cache. Play it now, in order to store *crystals & aliens hum tunes* into the cache.

Now, remove the check from **Record**, and play it again. From now on, it will read from the recorded cache. If you ever want to make a change to *crystals & aliens hum tunes*, just check the **Record** box to record the changed version into the cache. By using a *DiskCache* to save the recording, you never lose track of exactly which Sound is recorded in which file; it is always the Sound directly to the left of the *DiskCache* that is recorded into the file named in the **FileName** parameter field of the *DiskCache*.

Play *disk caching example* and play along with the ice crystals on the MIDI keyboard. Any Sound that requires live input *can* be cached, but keep in mind that once you cache it, it is just like making a recording of your performance. Every time you play it back from the cache it will be the same. Sometimes this is exactly what you want — to be able to record a performance so that you can add other layers to it in real-time when you play it back.

Distortion

By applying some nonlinear processing to your Sounds, you can add extra harmonics to the output, making it spectrally richer. This is the technique behind things like frequency modulation, ring modulation, waveshaping, and even cranking your guitar amp beyond the range where it has a linear response — all in an effort to broaden the spectrum and avoid those pure, yet boring, sine waves.

Waveshaping

The key to producing distortion in a signal is to process it in a *nonlinear* way. Demonstrate this to yourself right now by opening up *WS distortion*. A *Waveshaper* uses the value of its **Input** as the index into the wavetable named in **ShapingFunction**.

Try out a linear shaping function by setting the **ShapingFunction** parameter to **FullRamp**. **FullRamp** is a straight line from -1 up to 1. To see **FullRamp**, hold down the **Command** or **Control** key and click the disk button next to the **ShapingFunction** field; **FullRamp** looks like a straight line. Now select *WS distortion*, and choose **Oscilloscope** from the **Info** menu. Use a MIDI keyboard to play the harp sample and look at the waveform.

Next, try changing the **ShapingFunction** to **WinSine4**. Hold down the **Command** or **Control** key and click the disk button in order to see what the **WinSine4** wavetable looks like; this is definitely *not* a straight line. Then select *WS distortion* and choose **Oscilloscope** from the **Info** menu. Play some low notes on the MIDI keyboard and look at the difference in the shape of the waveform being displayed on the oscilloscope.

Clipping

Waveshapers are not the only way to apply a nonlinear process to a signal. One really simple way is to simply turn it up too loud. This clips the signal, and introduces sharp edges (and thus new harmonics) in the waveform where before there were only smooth curves.

Take a look at *sub-fundamental distortion*. Notice that *your sound here* is fed into a **Gain** and that the **Gain** is set to 10. Select the **Gain**, choose **Oscilloscope** from the **Info** menu, and watch what happens to the shape of the waveform as you increase the !Volume fader. This clipped signal is fed into **LPF** which filters out all but the lowest frequencies. Try playing *sub-fundamental distortion* and adjusting !Volume. For complex signals, clipping introduces distortion products that are below the fundamental as well as above the fundamental of the signal.

Frequency Modulation

The reason that frequency modulation (FM) can be used to synthesize complex tones is that it, too, is a nonlinear process and adds extra harmonics where there were none before. In Kyma, you can use any Sound as an FM modulator, including samples or the live input. Try playing *Harp FM*; this uses a Celtic harp sample as the modulator on a sine wave oscillator. The higher the modulation index, the more harmonic distortion is introduced.

In general, the idea of EQ (equalization) is to use filters and dynamic range control to attenuate or emphasize different ranges of a Sound's spectrum. The original idea was to emphasize a different frequency band on each track so that when you mixed them all together, they would fall into different critical bands for the listener and would not mask each other or lose apparent loudness. It goes beyond that though, because some of the side-effects and artifacts of these techniques have become a desirable part of the overall timbre. This processing has become identified with certain styles in recorded music — additional evidence in support of one of the basic concepts behind Kyma: that sound is sound; there really is no hard and fast division between synthesis, recording, effects, mixing, EQ. At every step along the way, you are generating, shaping and sculpting the sound.

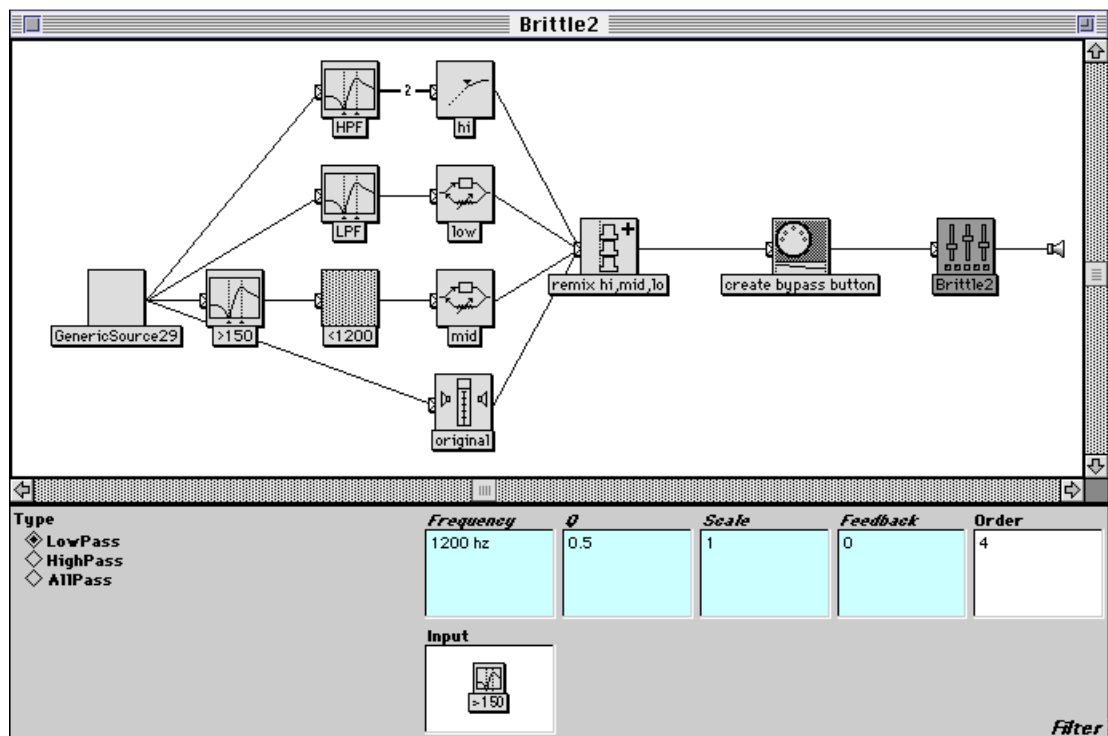
All of the EQ examples have one thing in common: they split the spectrum up into separate bands and then do something different to each frequency band.

GraphicEQ

It could be as simple as giving a different amplitude scale to each of the bands, for an example, play the Sound called *graphic eq*. This is noise being fed into the *GraphicEQ* Sound. Use the virtual control surface or your MIDI faders to adjust each of the faders in turn so you can hear what each frequency band sounds like.

Emphasizing the Highs

Now try the Sound called *brittle2*. After experimenting with the parameters, double-click on it to see how it works.



A *GenericSource* is fed into a highpass filter, a bandpass filter, and a lowpass filter. The higher frequencies are compressed using the *DynamicRangeController*. The mid-range frequencies are delayed by $!DelayMid * 10 \text{ ms}$, and the low end is delayed by $!DelayLow * 10 \text{ ms}$. Then the three frequency bands are all added back together for the result.

Notice that the *GenericSource* is also fed straight into an *Attenuator* called *original*. Take a look at the parameters of *original*. The *GenericSource* is at full amplitude whenever the $!Bypass$ button is held

down. At all other times, the amplitude of the unprocessed original is set to zero, so it won't contribute to the result at all.

Double-click on *mid*, and look at its **Scale** parameter:

```
!Mid * (1 - !Bypass)
```

This is what allows you to control the contribution of the midrange to the overall mix using the !Mid fader. Whenever the !Bypass button is held down, its value is 1; otherwise it is 0. So when !Bypass is held down, the amplitude of *mid* is zero, so it contributes nothing to the final result. This is a little trick that you can probably find other uses for in other circumstances. If you want an either-or condition, scale one Sound's amplitude by !aGate and scale the other one's amplitude by (1 - !aGate).

create bypass button is a *MIDIMapper* used, in this case, to make !Bypass show up as a momentary button on the virtual control surface (rather than as the default fader controller). The syntax is:

```
!Bypass is: (`Bypass type: #gate)
```

It's called a gate, because it stays at a value of 1 for as long as you hold it open; it doesn't just send out a single 1 as a trigger and then return immediately to 0.

Double-click the module called *hi* to see its parameters. This is a *DynamicRangeController* set up to act as a compressor. The idea is to compress the dynamic range of the signal so that there is less of a difference between the smallest and the largest parts of the amplitude envelope. It does this by tracking the amplitude envelope of the **SideChain**, and attenuating the **Input** whenever the amplitude envelope exceeds the **Threshold**. It is not a uniform attenuation, however. It uses the **Ratio** to determine how much it should attenuate the **Input**.

This particular **Ratio** value of 4/1, means that for every 4 dB increase in the amplitude envelope, attenuate the **Input** so that it will only increase by 1 dB. **AttackTime** and **ReleaseTime** control the responsiveness of the envelope follower on the **SideChain**: **AttackTime** the responsiveness to increases in the amplitude of the **SideChain**, and **ReleaseTime**, the speed with which the envelope follower reacts to decreases in the amplitude of the **Input**. The envelope follower introduces a small delay in the **SideChain** relative to the **Input**. You compensate for this by delaying the **Input** by the time specified in the **Delay** field; that way the attenuation will kick in at the right time, and not just *after* a change in the **Input** amplitude.

Although it may seem counterintuitive that attenuating the strongest parts of the signal can make the signal sound louder, that is just because we haven't gotten to the last parameter yet: **Gain**. Once you have compressed the *range* of the amplitudes, you can then apply a gain to all of the amplitudes. You can put a lot more gain on the signal without having to worry about clipping, because the largest amplitudes have been attenuated. The quieter parts of the signal are now louder relative to what they could have been before, which explains why the compressor emphasizes background noise, breathing, mouth clicks, *etc.* For percussive sounds, a compressor can effectively *lengthen* the amount of time that the sound is at a high amplitude before it decays into the background, giving the ear and the auditory system more time to register it as being loud, hence the "punchier" sound of drums or plucked strings when you put them through a compressor.

Experiment with different settings on the *DynamicRangeController* by setting the following parameters to the following values:

Parameter	Value
Threshold	!Thresh * 0.1
AttackTime	!Attack * 0.1 s
ReleaseTime	!Rel * 0.1 s
Delay	!Del * 0.1 s

Play the *DynamicRangeController*, and experiment by adjusting the faders on the virtual control surface. If you find a setting that you like, double-click on *Brittle2* (the *Preset* module), and click on the button labeled **Set to current event values**. This will recall the current settings each time you play *Brittle2*.

Now take a look at the input to *mid*, the two cascaded filters called **>150** and **<1200**. You can already guess from their names what they do. The first is a high-pass filter that allows only frequencies above 150 hz to pass through. That feeds into a low-pass filter that blocks frequencies above 1200 hz. The two filters together act as a bandpass filter, allowing frequencies between 150 hz and 1200 hz to pass through unattenuated.

Once again, double-click on *mid*. Try setting the **Feedback** parameter to !Feedback. Play *brittle2*, and use the virtual control surface to set !Feedback up around 0.9. Then try different fader settings for !MidDelay. If you like this altered version, drag a copy of *brittle2* from the Sound editor into your own Sound file window and save it (after having renamed it so you can remember what it does). Then close this Sound editor without saving the changes so you can also keep the original version around.

Experiment with the other examples in this Sound file for more ideas on how to shape the spectrum of Kyma Sounds using filters to selectively process subranges of the Sound's total bandwidth. As an exercise, think about how you might go about splitting the input into even *more* frequency bands and what different kinds of processing you could apply to each of those bands.

Envelopes

In general usage, the word “envelope” is used to mean some kind of wrapper or enclosing cover around something else. In some sense, you can think of an amplitude envelope as enclosing or defining a sonic “event” or discrete “object”.

First, in the interest of preserving your speakers for a little while longer, turn the gain on your amplifier down. Once your speakers are protected, play the Sound called *endless sine*. This is the kind of sound that is quickly relegated to the background. It sounds as if it could be the hum of a machine or a distant motor; it isn’t changing, so it seems safe to ignore it, and it almost disappears after a while. Putting an amplitude envelope on the endless sine will give it “closure”, and make it seem like a single “event” or “object”.

FunctionGenerator

To apply an amplitude envelope to any Sound, the method is the same: multiply the Sound by the envelope. To see an example of this, double-click on *apply shape of env to oscil amp*. Select *FunctionGenerator17*, and choose **Full waveform** from the **Info** menu. A *FunctionGenerator* reads through its wavetable exactly once each time it is triggered, so what you see in the full waveform display is a picture of the Gaussian wavetable.

Leave that full waveform window open, and return to *apply shape of env to oscil amp*; this is a *Product* module. Select it, and choose **Full waveform** from the **Info** menu. What you see is the shape of the Gaussian wavetable applied to the amplitude of the sine wave oscillator. (Actually, it looks more like the Gaussian shape mirrored across the zero axis, because the sine wave is both positive and negative, and when you multiply the positive value from the Gaussian curve by a negative value from the sine wave, you end up with a negative number).

AR and ADSR

Double-click on *a different shape*. As you can see, this is the same configuration: a *Product* with two inputs: *sine wave* and *ADSR env*. Select *ADSR env*, and choose **Full waveform** from the **Info** menu. Then double-click *ADSR env* to see its parameters.

An ADSR envelope is based on the idea of an envelope that will be controlled by a trigger. You specify how long it should take to reach the maximum amplitude when the trigger is first “attacked”, how far it should “decay” after that initial attack and how long it should take to decay that far, what its amplitude should be while the trigger is “sustained”, and what the envelope behavior should be when the trigger is finally “released”. You can see the different parts of the envelope clearly in the full waveform display: the attack of 0.2 seconds, an initial decay down to 0.75 of the full amplitude over the course of 0.1 seconds, a flat sustain portion, and finally a release that takes 0.5 seconds.

In this example, the envelope is triggered once and has a finite duration, so the sustain is computed by Kyma as the total duration minus the sum of the attack, decay, and release durations.

GraphicalEnvelope

Play the Sound called *graphic env chopper*, and use the MIDI keyboard to select different pitches and get a feel for how it behaves. Double-click on it to see how it is put together. Notice that there is a configuration that looks a lot like the last Sound we looked at; it’s a *Product* with 3 *detuned GA saws* and *GraphicalEnvelope14* as its **Inputs**.

Double-click *GraphicalEnvelope14* to see how it got its name. This Sound has a rubber-band style editor for drawing the desired envelope shape. When you click in the **Envelope** field, you should see two little red arrows. These mark the beginning and ending of a loop. Count the number of spikes or ridges between the red arrows. Make sure the original Sound is still playing, and hold down a key on the MIDI keyboard. When you press a key, it goes through the entire envelope up to the left arrow once, then loops between the arrows as long as you hold the key down, and finally when you release the key, it will go through the segment after the end-loop arrow. What it stores is not the absolute amplitudes, but the slope of each

segment, so it will take whatever amplitude it happens to be when you release it and use the slopes of the segments following the end loop to do the “release”.

Generalizing the Idea of Envelope...

In Kyma, an envelope is a time-varying function that can control any of the parameters of another Sound. To control a parameter by an envelope, copy the envelope Sound (**Ctrl+C**), click in the parameter field, and paste the envelope Sound into the parameter field (using **Ctrl+V**). Actually, any Sound can be used as an envelope, because any Sound can be pasted into the parameter field to control that parameter.

Look, for example, in the **Frequency** field of the low pass filter in *graphic env chopper*:

```
lfo L * 5000 hz + 5200 hz
```

This is using an oscillator as a kind of “frequency envelope”. Double-click on *lfo* to see its parameters. Its **Wavetable** is Buzz and its **Frequency** is

```
9 s inverse
```

If you ever want a repeating envelope, you can set the **Wavetable** of an *Oscillator* to the envelope function, and then set its **Frequency** to the inverse of how long it should take to go through the envelope once. Why? Because that is the number of seconds per cycle. And frequency is the number of cycles per second.

Formants

Picture a sine wave added to a delayed version of itself. If the delay time is equal to the amount of time it takes to complete one cycle of the sine wave, the delayed sine wave will reinforce the original sine wave. If the delay time is one-half of one cycle, adding them together will completely cancel them out. To prove this to yourself, play *a delay is a filter*, and experiment with different delay times. Then play *different delays cancel different freqs* to see how different frequencies are affected by the same delay time.

Now picture lots of sine waves bouncing around inside a violin or inside your vocal cavity. Because the reflections are delayed and then added back to the original source, the same thing happens inside your head as happens in the delayed sine waves experiment we just did: some frequencies are boosted and others are attenuated. These characteristic boosts and attenuations in the spectrum are called *formants*, and they tend to be independent of the fundamental frequency of the sound, acting more as a fixed filter on the basic glottal pulse (which *can* change wildly in frequency).

In general, someone with a large head (an alien giant, for example) will have lower frequency formants, and someone with a tiny head (say, a munchkin) will have higher frequency formants. So, if you move the formants without changing the fundamental frequency, you can make a voice more menacing and dangerous-sounding by lowering its formants or make it more cute and less dangerous by raising its formant frequencies. Prove this to yourself by playing *low menacing (SID6.7)*[§] and *munchkin* and talking into the microphone. You feel like saying different things through the low menacing voice than you do through the munchkin voice.

Play *shift formants* and experiment with the !Formants fader. The Sound called *female shift formants* does a similar thing but to a live input. In both cases, this technique performs a spectral analysis of the voice, uses the *SpectrumFrequencyScale* to raise or lower the frequencies while leaving the formants in the same place, and then uses a *ScaleAndOffset* to scale both the frequencies and the formants back up to the original fundamental frequency.

Next try *ring mod voice*. This one also works on live input and uses single sideband ring modulation to accomplish a similar result.

Yet another way to accomplish a similar result is use the *Vocoder*. Try *shift formants (vocoder)*. Like the other examples, this one shifts the formants without changing the fundamental pitch, but in this case the pitch is a monotone, because it is supplied by a sawtooth oscillator.

Synthesizing Formants

You can also *synthesize* formants (rather than shifting around the ones that are already there) using the *TwoFormantElement* or the *FormantBankOscillator*. Try playing *crossfade formant filter params*, and experiment with the !Vowel fader. Double-click on *crossfade formant filter params* to see how it is constructed. The idea behind it is that a spectrally rich signal is fed into two filters which are combined in parallel. More specifically, in this Sound, a sawtooth oscillator feeds into two *TwoFormantElements* named *formants 1 & 2* and *formants 3 & 4*. The outputs from these two formant filters are added together in a *Mixer* called *4 formants*. Then the *Mixer* is fed into a *MIDIMapper* in which !Vowel is mapped to the range of 0 to 7 and a grid size or step size of 1.

Double-click on *formants 3 & 4* to see how its parameters are controlled. The frequency of **Formant1** is

```
((!Vowel of: #(2450 2300 2450 2550 2400 2200 2150 2200)) smooth: 0.25 s) hz
```

and its amplitude is

```
((!Vowel of: #(-12 -8 -16 -26 -29 -16 -12 -19)) smooth: 0.25 s) dB
```

[§] This example was used by Francois Blaignan to process the voice of SID 6.7 in Paramount Picture's *Virtuosity* while Francois was still at Serafine Sound Design. Francois has since gone on to start his own sound design studio and has become the cross-synthesis guru of Hollywood.

In both of these expressions, `!Vowel` is used as an index into an array of possible frequencies and amplitudes.* For example, if `!Vowel` is 0, the formant frequency is 2450 hz and its amplitude is -12 dB (the values for the vowel A). If `!Vowel` is 7, the formant frequency will be 2200 hz and its amplitude will be -19 dB.

When a new frequency and amplitude are selected, they are approached smoothly over the course of 0.25 seconds, rather than jumping immediately to their new values (this is due to the

`smooth: 0.25 s`

following each array.

Next, play the Sound called *wow*, and play some low notes on a MIDI keyboard. This is a **FormantBankOscillator** synthesizing the vowels *OO* and *A*. If you open *wow*, you will see that it is a **FormantBankOscillator** whose **Spectrum** comes from two, interpolated **SyntheticSpectrumFromArrays**: one containing the frequencies, amplitudes and bandwidths for an *A*, and the other containing the same information for an *OO*. A **FunctionGenerator** on a Gaussian wavetable, triggered by `!KeyDown`, controls the interpolation between the parameters for *OO* and *A*, making a smooth transition between the two.

* The formant amplitudes and frequencies for these examples were found in *Computer Music*, written by Charles Dodge and Thomas Jerse, published by Shirmer Books.

Frequency Scaling

One of the most common modifications to apply to a sound is to change its pitch or frequency. There are several techniques for frequency scaling, and, depending on the model you are using to synthesize the sound, each has its own set of side-effects or artifacts.

DiskPlayers and Samples

One of the simplest and most direct techniques for modeling an acoustic sound is to put a microphone near the sound source and to record how the air pressure changes over time. This stream of air pressure measurements (known as a *waveform*, a *sample*, a *digital recording*, or a *disk track*) is recorded in a disk file where it can later be read, value-by-value to recreate the original sound (at least the original sound as it was recorded at the one point in space where you placed the microphone).

If we define the frequency of a sound (*very* loosely) as the rate at which the waveform is wiggling up and down, then it follows that the faster we read the waveform, the faster the wiggling, and thus the higher the frequency. The slower the rate at which we read the waveform, the slower the changes in the waveform and the lower the frequency. It also follows that the faster we read through a recording, the less time it takes to get through the whole thing; in other words, using the simple “sampling” model, you cannot change the frequency of a sound without also changing its duration.

Try playing the Sound called *frequency scaled disk* and adjust the `!Rate` parameter. When you lower the **Rate** parameter of a *DiskPlayer*, you lower *all* the frequencies of that sound. If the original sound source had some formants (resonant characteristics that boost the amplitudes of some frequencies and attenuate the amplitudes of others) then the formants will effectively be lowered in frequency too. Through years of experience in listening, we associate low frequency formants with large resonant cavities and high frequency formants with tiny resonant cavities. That’s why a sped-up recording tends to sound like a chipmunk or some other tiny animal and a slowed-down recording tends to sound like a giant or some threateningly large animal. You can use this effect to play upon ancient instincts and associations, for example, by lowering the frequency when you want to create an ominous atmosphere or a dangerous and powerful voice, and by raising the frequency to create a sense of lightness or a “cute”, childlike voice.

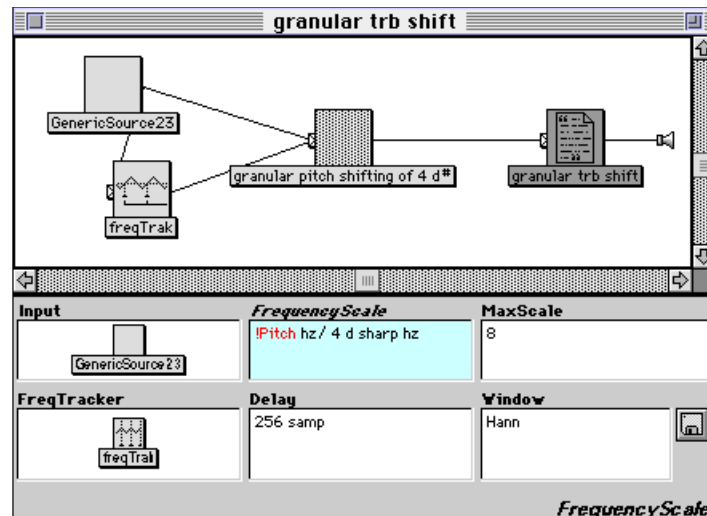
There are other situations, a sampling keyboard imitating an acoustic instrument for example, in which changing the duration of the sound and changing the apparent size of the sound source are undesirable side-effects. Try the Sound called *freq scaled sample*. This is a single recording of a harp tone whose frequency is controlled using the MIDI keyboard pitch. Try playing the lowest note on your keyboard and then the highest. The difference in duration is pretty noticeable. One way to minimize the duration differences and timbre changes is to use several different samples, taken from different frequency ranges of the original instrument, and map them to keys or ranges of keys on the MIDI keyboard. This doesn’t get rid of the side-effects, but since any single sample is not being distorted very much, it reduces the effects.

Select *trb from RAM* and do a **Compile, load, start**. Then play a scale from the bottom to the top of your MIDI keyboard. You can hear where the different trombone samples have been mapped to different key ranges. (For more detail on the *KeyMappedMultiSample*, see *Sampling* on page 161).

Granular

Compare the last Sound you played to *granular trb shift*. **Compile, load, start granular trb shift** and then play a scale across the full range of the MIDI keyboard.

Now double-click on the Sound to see how it is put together.



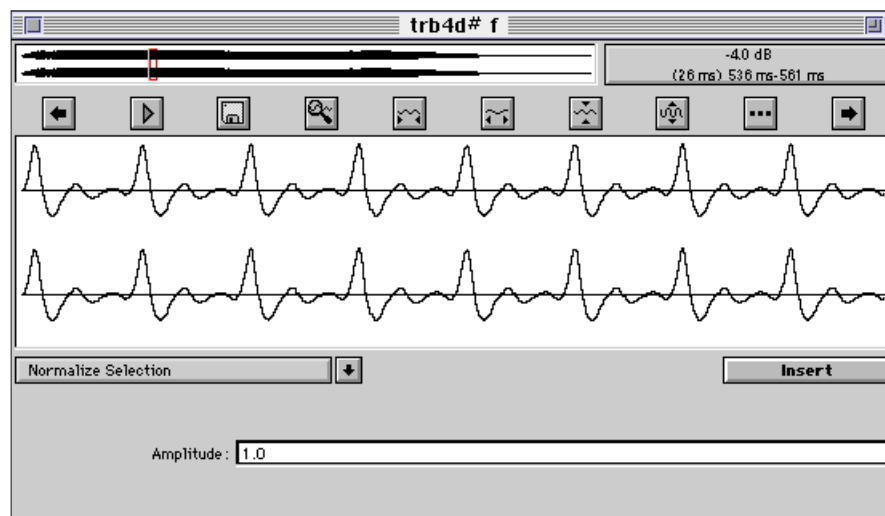
A *GenericSource* is fed into a *FrequencyScale* Sound and a *FrequencyTracker*. Look at the parameters of the *FrequencyScale*. It takes the *GenericSource* and an estimate of the frequency of the *GenericSource* as inputs. Then it scales the frequency of the *GenericSource* by

!Pitch hz / 4 d sharp hz

which is the ratio of the frequency as supplied from the MIDI keyboard to the original frequency of the sample. Notice that both !Pitch and 4 d sharp are not frequencies but pitches, so they have to be converted into units of frequency in hertz. The ratio between two MIDI note numbers is quite different from the ratio of the two note numbers expressed in hertz; for example the ratio between 4 a and 3 a would be 69/57 (about 1.2) in note numbers but 440/220 (exactly 2) in hertz. You can convert between most units of frequency or duration by typing the name of the desired units after the value in the original units. In **MaxScale**, enter the largest frequency ratio required. In this case it is 8/1 or a maximum shift of 3 octaves up.

For **Window**, select a wavetable that can serve as a nice, smooth granular amplitude envelope. In this case we have chosen Hann for a Hanning window. To see the shape of the grain envelope, hold down the **Control** or **Command** key and click on the disk button to the right of the **Window** parameter field. Any smooth up and down shape like this can be used as the **Window**, for example, you can try Gaussian, and even LinearEnvelope out to hear how they might affect the sound.

How is **Window** function used? Take a look at the waveform of the trombone.



It looks very much like a series of impulse responses (as we saw in the earlier tutorial on filtering).

The *FrequencyScale* works best on sounds that can be modeled as a source of impulses hitting a resonant filter, for example the trombone (flapping lips generate an impulse that is fed into the filter of the body of the instrument) or the human voice (glottal pulses passing through the resonant cavity of the mouth and lips). The *FrequencyScaler* looks for what it thinks are individual impulse responses in the waveform and creates little grains by putting an amplitude envelope (shaped like the **Window** function) on each one. Then, to lower the frequency, it stretches these grains further apart so they occur less often. And to raise the frequency, it pushes the grains closer together so that they occur at a faster rate. This is the same as decreasing or increasing the rate of the impulses *without* changing the impulse response of the filter. If the impulse-into-filter model is a good match for the way the sound was originally produced, then the *FrequencyScaler* will do a good job of changing the frequency without changing the duration or moving the formants around. Why? Because the formant structure is inherent in the filter or the resonator and can be inferred from the way it responds to a single impulse. Since this method does not change the shape of the impulse response, it does not change the formants.

The **FreqTracker** input is used to get an idea of where to find the impulses in the **Input**. The **Delay** is to compensate for the delay through the *FrequencyTracker*; it delays the **Input** by the same delay introduced by the *FrequencyTracker*, so that the frequency estimate lines up with the current **Input**. The minimum delay you should use is 256 samp and the maximum is 20 ms. Within those boundaries you should use something close to the period of the lowest frequency you expect to see in the input. Remember that to convert a frequency to a period, type the frequency in hertz followed by the word “inverse”, for example

```
4 d sharp hz inverse
```

The quality of the frequency scaled sound depends on the quality of the frequency estimate, so it is important to start with the best possible frequency estimate. Double-click the *FrequencyTracker* to see its parameters. The most important parameters to adjust are **MinFrequency** (the lowest frequency you expect to see in the **Input**) and **MaxFrequency** (the highest frequency). The narrower you can make this range, the better your frequency tracking will be. In this case, we know that the original frequency of the single trombone tone was 4 d sharp, so the range of 4 d to 4 f is pretty safe!

All of the other parameters are pretty nonlinear (in that tiny changes in these parameters can destroy the frequency tracking), so it is recommended that you do not change these. You can experiment with more or fewer **Detectors**, with the caveat that more is not necessarily better in terms of the effect this will have on the tracking.

This technique has one more advantage (even beyond the fact that it does not affect the duration or the formant structure), because it can work in real time on live input. For example, play the Sound called *scale w/Frequency* and try various settings of the !Frequency fader. If you put !Frequency low enough, you will hear the individual grains. Play it again, this time choosing live input in the *Generic-Source* dialog, and try frequency scaling your own voice in real time. You may have to edit the Sound to adjust the **MinFrequency** and **MaxFrequency** parameters of the *FrequencyTracker* to more closely match your own range.

Wavetable Synthesis

If you model a sound as oscillators reading from wavetables, then you can select durations and frequencies independently of one another; frequency is controlled by the size of the increment you use in stepping through the wavetable, and duration is simply how long the oscillator is left on.

For example, *wavetable frequency scale* uses a GA resynthesis of three different trombone tones mapped to two different ranges of keys. The lower range has 2 c at its low end, 3 c sharp at its high end, and fills in the intermediate timbres by continuously morphing between those two endpoints. The higher range has 3 c sharp as its low endpoint, 4 d sharp as its high end and, similarly, morphs between the two to get the intermediate waveforms and envelopes.

See *Wavetable Synthesis* on page 185 to see how to create a GA analysis/resynthesis from the **Tools** menu.

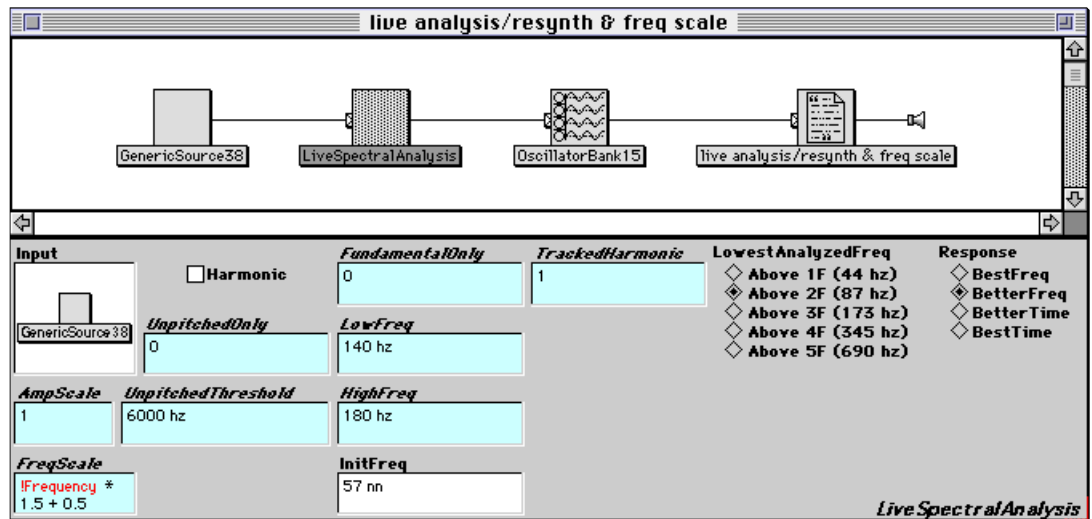
Additive Synthesis

If you resynthesize from an analysis file by adding together many sine wave oscillators, then you can scale the frequency by simply scaling the oscillator frequencies up or down. The rate at which you read through the amplitude and pitch deviation envelopes is independent of the frequency of the oscillators, so you can scale the frequency without affecting the duration.

Try *additive synth freq scale* as an example of reading an analysis from RAM and using it to control sine wave oscillators whose frequency you can scale up and down.

The analysis does not have to be read from the disk, it can be generated in real time from a live input. Try, for example, *live analysis/resynth & freq scale* first with the default Virtue sample, and the second time, choosing live input for the *GenericSource* and trying it on your own voice.

Double-click *live analysis/resynth & freq scale* to see how it is put together.



A *GenericSource* feeds into a *LiveSpectralAnalysis* which controls an *OscillatorBank*. Double-click the *LiveSpectralAnalysis* to view its parameters. It is set for fundamental frequencies above 2 f (or 87 hertz) and *BetterFreq*. The *LiveSpectralAnalysis* uses a bank of bandpass filters to analyze the input, and **Response** is a control on the bandwidth of those filters: *BestFreq* is the narrowest bandwidth, then *BetterFreq*, etc., with *BestTime* being the widest bandwidth. The **FrequencyScale** is set to

$$!Frequency * 1.5 + 0.5$$

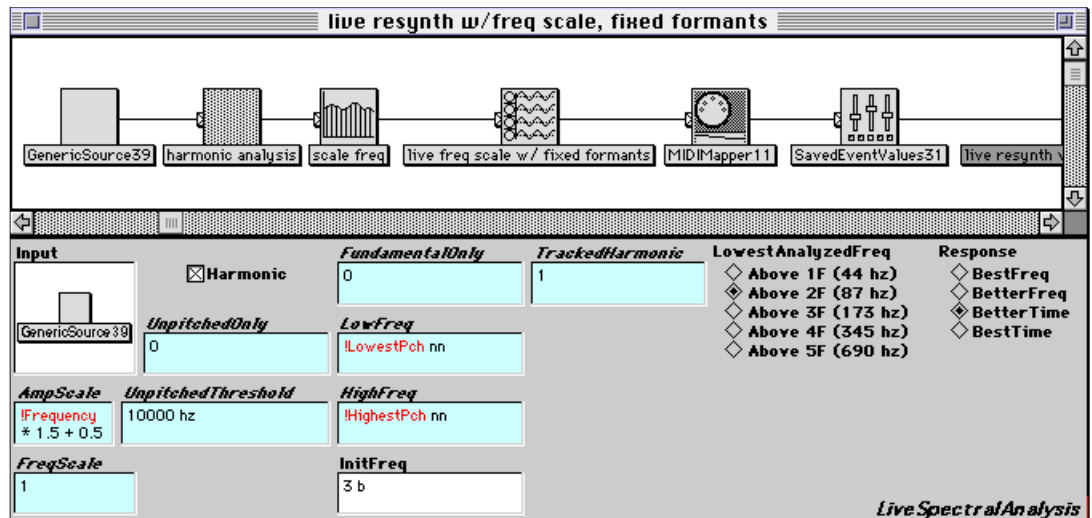
which means it ranges from one octave below the original pitch (0.5) to one octave above (2), depending upon the value of *!Frequency*. Since **Harmonic** is not checked, this is not doing an harmonic analysis, so none of the rest of the parameters are used.

As you can hear, this method does not affect the duration, but it still suffers from the “chipmunk” syndrome, since it scales all oscillators without changing their amplitude (and thus effectively shifts the apparent formants up or down along with the fundamental frequency).

Fixed Formants

If you have a high voice, play *live resynth w/freq scale, fixed formants* next. If you have a low voice, try *low freq live input, fixed form* instead. The first time you play it, use the default input, and then play it again, this time choosing the live input for the *GenericSource* and then singing or speaking into the microphone to frequency scale your own voice. Adjust *!HighestPch* and *!LowestPch* to the range of MIDI note numbers covered by your singing or speaking voice (where 60 is middle C).

Double-click the Sound to see how it differs from the moving formants version.



The signal flow is similar to the last Sound we looked at except that the *LiveSpectralAnalysis* feeds into a *SpectrumScaler* before it goes to the *OscillatorBank*. If you take a look at the *SpectrumScaler* parameters, you will see that the frequency scaling expression has moved from the *LiveSpectralAnalysis* to here. The *SpectrumScaler* scales the frequencies of the oscillators but keeps the apparent formants in the same place.

In order for the *SpectrumScaler* to work, you must feed it an *harmonic* analysis. Take a look at the parameters of the *LiveSpectralAnalysis*, and you will see that **Harmonic** is, indeed, checked. The **LowFreq** and **HighFreq** parameters are the lowest and highest fundamental frequencies that you expect from the **Input**. To be completely truthful, they specify the range of frequencies you expect from the harmonic number given in the field called **TrackedHarmonic**. In this example, the **TrackedHarmonic** is set to 1, so the **LowFreq** and **HighFreq** refer to the first harmonic (the fundamental). The **AmpScale** is set to

`!Frequency * 1.5 + 0.5`

only because scaling down the frequency ordinarily makes the resynthesis louder, so this is here to try to compensate by scaling down the amplitude as you scale down the frequency — no deep hidden meaning.

RE Synthesis

If you slow down the EX file feeding into an RE filter, you get something that sounds lower in frequency and has a longer duration, but which keeps some of the original formant structure. Try the Sound called *RE w/ slower & lower EX input*. You will, most likely, have to adjust `!Atten` as you adjust `!Rate`, since some rates will cause the filter to overflow.

Open this Sound for editing, and take a look at the parameters of the *FunctionGenerator* called *timeIndex*. Notice that it is triggered by `!KeyDown` and that its **OnDuration** is

`6.01422 s * (1.0/!Rate)`

6.01422 seconds is the duration of the original sample. Whenever `!Rate` is increased, we want to go through the RE filter faster, and, because we are scaling the duration by the inverse of `!Rate`, faster rates mean smaller durations. Likewise, `!Rate` less than one will give us longer durations (which is exactly what we want, since going through the EX file at a slower rate should give it a longer duration).

For more details on how to create an RE analysis using the **Tools** menu, see *Cross Synthesis* on page 103.

Surrealism

In the Sound collection called *surrealistic scaling* you can find some great alien voices as well as some additional techniques for scaling the frequency.

For example, open and try out the Sound called *timewarp alien*, which sounds like an alien having problems with echo-suppression on a trans-galactic phone call. In the signal flow graph, find the module named *SingleSideBandRM7* and play it. Like straight ring modulation, single sideband ring modulation gives you the sum and the difference frequencies of its input and a sine wave — except that, in single-sideband modulation, one of the sidebands is canceled out, leaving only the sum frequency. Click on the name **SingleSideBandRM** (in the lower right corner of the parameter fields) to read a description of how this works. Then click on the parameter names for **Frequency** and **FreqScale**. This Sound is set up so that, at 1000 hz, it does single sideband modulation, and the further the input frequencies are from 1000 hz, the more they are like ordinary ring modulation — so different parts of the input spectrum are being modulated differently. Frequency components of the input that are close to 1000 hz will be scaled down by the **FreqScale** ratio 0.75.

Try replacing **FreqScale** with

```
!Scale * 2
```

and the **Frequency** parameter with

```
!Frequency * 2000 hz
```

and experiment with different combinations of the two parameters. As you can hear, while this method does not give as predictable and clean results as some of the frequency scaling methods discussed earlier, it does have some unique properties that you can use in designing new kinds of processing.

Next, play *watery harp* and then double-click on it to see how it works. It turns out that when you change the length of the delay in a *DelayWithFeedback* (by changing **DelayScale**) you will, as a side-effect, change the frequency of whatever sound is currently in the delay line. So, although you might not guess it from its name, the *DelayWithFeedback* is yet another Sound you can use to scale the frequency of its input.

Monotony

So far we have spoken only of scaling the frequency while still maintaining the original contour of the melody or the prosody. But how about removing all that annoying melodic content in order to achieve a completely monotonous melody or phrase? Yes, with Kyma you can do it!

Try playing *you are monotonous* and sing into the microphone. (If you're at a loss for what to sing, try your national anthem). If you have a low voice, you might find that *Low voice monotony* will work better for you. Next try playing along with what you are singing on the MIDI keyboard, and listen to what happens when what you play on the keyboard diverges from what you are singing. Try to sing flat, and then use the keyboard to correct the pitch of your voice.

There are several different techniques for monotonizing but the basic idea is this: Whenever the frequency of the input goes *up*, scale it *down* with the frequency scaler. Whenever the frequency of the input goes *down*, scale it *up* by the same amount in the frequency scaler. The result is a melody or prosody that never varies from one pitch.

In order to do this, you need some way to scale the frequency up or down, and you need some way of tracking the input's frequency so you know how to cancel it out. For example, look at *you are monotonous (kbd)*. This is an *ADInput* going into a *FrequencyScaler* and a *FrequencyTracker*. Take a look at the parameters of the *FrequencyScaler*. Note that its **FrequencyScale** parameter is set to:

```
!Pitch hz / ( [freqTrak] L * 22050.0 hz)
```

Let's "deconstruct" this expression in order to understand its effect. First of all, notice that it is a *ratio of two frequencies* — the frequency from the keyboard and the frequency of the input signal.

The frequency of the input is in the denominator, so whenever the frequency of the input gets larger, the ratio of the keyboard frequency to the input frequency gets smaller and vice versa. (For example, the ratio 1/6 is smaller than the ratio 1/3). This is exactly the behavior we were looking for in order to cancel out the changes in the input frequency.

Now to explain a few of the details. The *FrequencyTracker* is multiplied by 22050 hz. Why? Because the output of the *FrequencyTracker* is in the range of (0,1). In order to scale that to the range of possible

frequencies, we have to multiply by the largest possible frequency value, which, in digital audio, is always one half of the sampling frequency. So this expression is assuming a sampling rate of 44.1 khz. To modify this expression to work at any sampling rate, change it to:

```
freqTrak L * SignalProcessor sampleRate * 0.5 hz
```

The other detail to notice is that !Pitch from the keyboard must be converted to frequency in hertz before taking the ratio of the two frequencies. Ratios of note numbers are not the same as the ratios of the equivalent values in hertz! As a quick proof by example, consider the ratio of 4 a to 3 a. In note numbers, this would be 69 / 57 or about 1.2, whereas, in hertz, the ratio would be 440.0 / 220.0 or exactly 2.

Try out *you are monotonous (kbd)*, singing into the microphone, and playing either the same pitches or different pitches on the keyboard. If you have a particularly low voice, you may want to try *Low voice monotony* instead.

The other examples in this collection are all examples of live “monotonization” using various different techniques. Try each of them out, so you can get an idea of the strengths and weaknesses of each approach.



F13

If you don’t have to monotonize an audio signal in real time, you can monotonize an analysis of the signal using the Spectrum editor. First analyze the recording using the spectral analysis tool. Then open the analysis in the Spectrum editor. Select the lowest 8 tracks, and push the filter button (third from the right in the row of buttons across the bottom of the editor) for a list of modifications that can be performed on the selected tracks. Choose **replaceFrequencyWithAverage**, and wait for it to compute and replot. Experiment with selecting and monotonizing different combinations of harmonics.

Live Analysis, Resynthesis

Using the *LiveSpectralAnalysis* module, you can analyze any Kyma Sound in real time and resynthesize the sound at the same time, using an *OscillatorBank*.

Resynthesis with Frequency Scaling

For example, play the Sound called *live analysis/resynthesis*, and hit **Enter** when the *GenericSource* asks what input to use. What you are hearing is not the sample but a bank of sine wave oscillators. The oscillator frequencies are tied to the MIDI keyboard (try playing it right now to verify this).

Compile, load, start *live analysis/resynthesis* again, but this time choose the live input as the *GenericSource*. Speak, sing, or otherwise perform into the microphone while playing the keyboard to create changes in the frequency of the output.

Intercepting and Modifying Selected Tracks

We didn't come up with a way to separate the live input into hundreds of sine wave oscillators just so we could add them all back up together again and reproduce the original sound. No, like all good sound designers, we want to warp, mangle, mutate, transmogrify, and otherwise mess around with the sound before resynthesizing it.

One way to do that is with *SpectrumModifier*. Despite its deceptively simple-sounding name, this is the mangler/transmogrifier you've been looking for. The Sound has two stages: selecting (or rejecting) tracks based on some criteria, and then modifying the amplitudes or frequencies of the selected tracks.

Frequency Domain Filters

Let's start with something simple: creating a pseudo-bandpass-filter effect by rejecting tracks (oscillators) whose frequencies fall outside a given range. Play *pseudo bandpass* and try adjusting `!HiCutoff` and `!LoCutoff` in the virtual control surface.

Double-click on *pseudo bandpass*. Notice that it has a *LiveSpectralAnalysis* feeding into a *SpectrumModifier*. Double-click the *SpectrumModifier* to look at its parameter settings, in particular, the following:

Parameter	Setting
LoFreq	<code>!LoCutoff</code> hz
HiFreq	<code>!HiCutoff</code> hz
FreqHysteresis	1 hz
HearAll	<i>unchecked</i>

By having **HearAll** unchecked, we are specifying that we want to hear only those tracks which meet the criteria. Tracks satisfying the criteria in this case are those whose frequencies are above `!LoCutoff` hz and below `!HiCutoff` hz. By setting the **FreqHysteresis** to 1 hz, we can avoid tracks popping in and out because some small deviation (like vibrato) is taking it in and out of the specified range.

Track-dependent Frequency Scale & Offset

Next try playing *subwoofer-ize*, and try different settings for `!LoTrack`, `!InHarm`, and `!Scale`.

Open it up and take a look at the parameter settings for the *SpectrumModifier* named *scale freqs of tracks below LoTrack*, in particular:

Parameter	Setting
Select	<i>unchecked</i>
LoTrack	!LoTrack
HiTrack	1000
FreqScale	!Scale
FreqOffset	!InHarm
LoAmp	0
HiAmp	0.2
AmpHysteresis	0.1
HearAll	<i>checked</i>

In this example, **Select** is not checked. This indicates that the criteria describe the tracks that should be *rejected*, not selected. In other words, whatever the list of conditions are for selection, you should add a “NOT” to the end of the list. So in this case, it is saying “select only the tracks above !LoTrack, NOT!” In other words, select only the tracks *below* !LoTrack.

But notice that it also has the **HearAll** box checked. What does it mean to select some tracks if you are still going to let all of them through? This is where **FreqScale** and **FreqOffset** come into the picture. The selected tracks will have their frequencies multiplied by !Scale and then have !InHarm added to them.

In short, this is a strange kind of “filter” where the “pass band” is defined by track (or oscillator) number rather than frequency and where the tracks within the pass band have their frequencies scaled and offset.

Reducing Background Noise

Play *reduce background hiss*, and experiment with different settings for !LowAmp. Open it up to examine the parameter settings of *reject amps below !LowAmp*, in particular:

Parameter	Setting
Select	<i>checked</i>
LoAmp	!LowAmp
HiAmp	0.2
AmpHysteresis	!LowAmp * 0.5
HearAll	<i>unchecked</i>

In this example, there is some background hiss that was present in the original recording and is faithfully reproduced in the resynthesis. The *SpectrumModifier* is being used to filter out tracks based on their amplitudes. If the noise in the original recording is consistently of lower amplitude than the desired signal, then you can use this method to reduce or remove the noise from the resynthesis.

Probabilistic Resynthesis

Next try the Sound called *spectral granulation*, experimenting with different settings for the very-scientifically named !Burbliness parameter.

The important parameter settings in this *SpectrumModifier* are:

Parameter	Setting
Select	<i>checked</i>
Probability	1 - !Burbliness
Seed	-0.2
HearAll	<i>unchecked</i>

Once a track has passed through all the other criteria, it still has one more test to pass before it is selected, and, in a case of art-imitating-life, this last test is completely arbitrary, and, in fact, random. As a final

criterion for selection, you can assign a likelihood that any given track will be selected on any given frame. If you set the **Probability** to 1, then this last test will have no effect, because any tracks that have satisfied the other criteria have a likelihood of one — in other words, a 100% chance of being selected. If you set **Probability** to 0, then you won't hear anything, because all tracks have zero chance of being selected. In this case, we have set it to $1 - !\text{Burbliness}$, so the larger the value of **!Burbliness**, the less often a track will be selected when it is tested. This explains why the sound breaks up more and more as you increase the value of **!Burbliness**.

Try changing the setting of **Probability** to

```
TrackNumber / 512
```

This weights the higher tracks more than the lower numbered tracks, giving a kind of probabilistic high pass filter effect, since the higher tracks are selected so much more often than the lower ones. (**!Burbliness** still shows up in the virtual control surface because it also has an affect on the **AmpScale** in the *LiveSpectralAnalysis*. Don't worry about it, because it doesn't have too much of an effect other than to change the overall amplitude slightly).

On occasion, though, a low bloop still gets through, because the lower tracks still have a nonzero likelihood of occurring. To remedy that, try changing the **Probability** to

```
TrackNumber - 32 / 512
```

This makes the probability of the first 32 tracks negative (effectively zero), and weights the remaining tracks according to track number.

Skiping Harmonics

The final example in this file is called *split spectrum*.

Notice that the *LiveSpectralAnalysis* feeding into the *SpectrumModifier* in this example has the **Harmonic** box checked. That means that, in this case, track number is synonymous with harmonic number. (This was not true for the previous examples which did not have **Harmonic** checked).

Take a look at the parameters of the *SpectrumModifier* in this example, in particular:

Parameter	Setting
FreqScale	!Frequency
Probability	<code>TrackNumber rem: 3</code>
Seed	<code>-0.2</code>
HearAll	<i>checked</i>

The message `rem:` is short for "remainder," so this is another way of saying the remainder left over after dividing `TrackNumber` by three.

In other words, the probability of the first harmonic being on is 1, for the second 2 (but the maximum probability is 1, so anything greater than 1 is just the same as 1), for the third harmonic 0 (because the remainder is zero), and so on. In other words, the pattern is two harmonics on, next harmonic off, two harmonics on, next harmonic off.

The selected harmonics are frequency scaled, so you can split apart the spectrum of the input as it is playing, with some harmonics going down in frequency and others continuing as they were.

Uncharted Territory

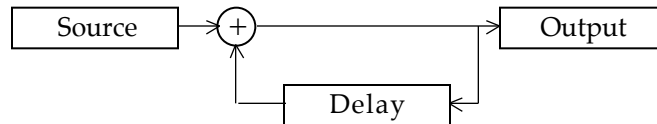
It hasn't been possible to do this sort of thing in real time until fairly recently, so anything you develop with these Sounds has a high probability of being something that no one has heard before. So experiment and have fun!

Looping, Feedback

Several Kyma Sounds have internal feedback loops, but you can also create your own feedback loop by writing into memory with one module and reading out of it (after an arbitrary delay) with another. In the signal flow diagram, the writer must occur before the reader.

MemoryWriter/Reader

Take a look at the first Sound in this file, the one called *write memory, read memory*. This is an implementation of a basic feedback loop which you might draw as:



Notice that in the Kyma Sound the last module in the diagram is a *MemoryWriter*. It writes the sum of the source plus the delayed source into a delay line named, in this example, *recording*. Notice that the Sound called *read the memory* is a *Sample*, one of several Kyma Sounds that can read from the sample RAM of the Capybara. Double-click on *read the memory* because there is one very important parameter setting that is easy to overlook. Any time you are reading something from memory that is being written into memory elsewhere in the Sound structure, you must check the **FromMemoryWriter** box. Otherwise, Kyma will search your hard disk looking for the sample named *recording*. If you check **FromMemoryWriter**, then Kyma knows that it has to put the *MemoryWriter* on the same expansion card as the *Sample* that reads the memory. Try playing *write memory, read memory* to verify that it does indeed sound like a simple feedback loop.

Actually you could have done exactly the same thing in a much easier way — by feeding the source into a *DelayWithFeedback* and controlling the feedback with a fader. But what if you want to insert some kind of processing in the feedback loop? Open up *process in fdbk loop* to see an example of this. The feedback portion of the signal goes through a single side band ring modulator before it is added in with the source and fed back into the delay line — so the processing is compounded each time around the loop. Try playing *process in fdbk loop* to hear the effect.

In *write memory, granulate memory*, you see an example of using a *GrainCloud*, rather than a *Sample*, to read out of the delay line. The result is a kind of granulated delay line. Try playing it once with the default source chosen and then switch it over to **Live** source so you can try singing some long tones into it.

As you probably noticed in this example, the delay line might already have some leftover sound in it from a previous example. Look at *granular read w/zero mem* for an example of how to clear the memory in the delay line before using it. Notice that the input to the *MemoryWriter* is a *Concatenation* of a *Constant* (with **value** zero) followed by the feedback loop. This will write zeroes into the memory on all the cards first, so the delay lines will be silent at the beginning.

FeedbackLoopInput/FeedbackLoopOutput

What if the modifications you put into the feedback loop are so extensive that they cannot all be scheduled on the same expansion card as the memory writer and memory reader? To implement a feedback loop that crosses expansion cards, use the modules called *FeedbackLoopInput* (writes into the delay line) and *FeedbackLoopOutput* (reads from the delay line). Note that these modules can only implement delay lines of from 12 to 2048 samples long, so if you can get away with using the previous design for feedback, you should preferentially do it that way, because that gives you a larger range of delay times (down to a single sample).

Open up *basic Feedback in/out example* to see a stripped-down illustration of how to set one of these up. Reading from left to right in the signal flow diagram, the first module is a *FeedbackLoopOutput* called *read from loop*. Notice that its parameters are set so that it reads from a “connection” called *feedback* after 128 samples of delay. This is fed into an *Attenuator* so you can control the amount of feedback, then it is fed into a *DelayWithFeedback* to add additional delay if desired. From that point on, it should look

pretty familiar. The delayed feedback is added to the source in a *Mixer*, and the *Mixer* is fed into the module that writes into the “connection” — the *FeedbackLoopInput*. This is a simplified example, because you could implement the same thing using a single *DelayWithFeedback* module.

For something a little more interesting, play *slithering sibilance*. This is the same basic configuration but with a 12-pole *AllPass* filter in the feedback loop.

Next, try *thirds rising*. This one has a spectral analysis and resynthesis in the loop. The resynthesis oscillators can be scaled up or down by the interval you set in the virtual control surface.

MIDIVoice Scripts

The *MIDIVoice* and *MIDIMapper* modules give you three choices for the source of MIDI events: the live MIDI input stream (originating with external controllers or a sequencer or other software running in parallel with Kyma), a MIDI file stored on disk, or a MIDI script (which generates MIDI events algorithmically and then emits them).

These examples illustrate some of the tools for algorithmically generating MIDI events using the **Script** parameter of a *MIDIVoice* or *MIDIMapper*. For more information about MIDI event scripts, see *MIDI Scripts* beginning on page 522.

Notes and Controllers

Notes

The syntax for a basic note-on event is:

```
self keyDownAt: aTime duration: aDur frequency: aFreq velocity: 0To1.
```

When you drop the velocity tag, the event defaults to a velocity of 1.

Transcription of Short Sequences

Take a look at (and compile, load, start) *machautKyrie* to see a simple application of these note events: direct transcription of written notation by hand into MIDI script events. For short note sequences, entering the events by hand is often the quickest and most straightforward way to do it. Here is an excerpt from the Machaut example:

```
MM := 90.  
  
"-----"  
self keyDownAt: 0 beats duration: 3 beats frequency: 4 a.  
self keyDownAt: 3 beats duration: 1 beat frequency: 4 a.  
self keyDownAt: 4 beats duration: 1 beat frequency: 4 b.  
self keyDownAt: 5 beats duration: 1 beat frequency: 4 g.  
self keyDownAt: 6 beats duration: 2 beats frequency: 4 a.  
self keyDownAt: 8 beats duration: 0.5 beats frequency: 4 f.  
self keyDownAt: 8.5 beats duration: 0.5 beats frequency: 4 e.  
(and so on...)
```

The first line of this script informs Kyma of the metronome that should be used to define the beat for the rest of the script. The remaining lines are a transcription of each note from the piece.

Modulo Arithmetic

The script of a *MIDIVoice* or *MIDIMapper* is actually a program written in Smalltalk; this means that you can generate events according to an algorithm. The script in *Kurt's new rhythm method* illustrates the use of modulo arithmetic in an algorithm that creates a kind of twisted drum machine.

In this script, a subset of seven different fixed-duration fixed-frequency events is algorithmically selected for scheduling on each beat. The following is an outline of the script:

```
MM := 300.  
0 to: 200 do: [ :i |  
    (a series of omitted event selection decisions)  
].
```

The first line of the script sets the metronome. The rest of the script is a loop that counts the variable *i* up from 0 to 200. Inside the loop, a series of decisions are made to determine whether each of the seven possible events should be scheduled for this beat.

The first of the seven event decisions is:

```
i \% 4 = 0 ifTrue: [
```



```

self
    keyDownAt: i beats
    duration: 25 / 100 hz
    frequency: 100 hz].

```

This part of the script causes an event (of the given frequency and duration) to be scheduled whenever the beat counter *i* has no remainder when divided by four (*i* \ 4 = 0). This is the same as saying that this particular event will be scheduled on every fourth beat. The other decisions are based on whether the current beat is a multiple of 2, of 3, of 5 (within the first 50 beats), of 13, of 7, and finally when it is not a multiple of 7, 5, or 3 after the first 50 beats.

Here is the entire script:

```

MM := 300.
0 to: 200 do: [ :i |
    i \ 4 = 0 ifTrue: [
        self
            keyDownAt: i beats
            duration: 25 / 100 hz
            frequency: 100 hz].

    i \ 2 = 0 ifTrue: [
        self
            keyDownAt: i beats
            duration: 25 / 250 hz
            frequency: 250 hz].

    i \ 3 = 0 ifTrue: [
        self
            keyDownAt: i beats
            duration: 25 / 150 hz
            frequency: 150 hz].

    (i < 50 and: [i \ 5 = 0]) ifTrue: [
        self
            keyDownAt: i beats
            duration: 25 / 400 hz
            frequency: 400 hz].

    i \ 13 = 0 ifTrue: [
        self
            keyDownAt: i beats
            duration: 25 / 700 hz
            frequency: 700 hz].

    i \ 7 = 0 ifTrue: [
        self
            keyDownAt: i beats
            duration: 50 / 50 hz
            frequency: 50 hz].

    (i > 50 and: [i \ 7 ~= 0 and: [i \ 5 ~= 0 and: [i \ 3 ~= 0]]]) ifTrue: [
        self
            keyDownAt: i beats
            duration: 100 / 1000 hz
            frequency: 1000 hz].
].

```

Random Events

The next example generates one hundred random pitches with random durations and random placement in the stereo field. Edit the Sound called **100 uniform random** to follow along.

To generate one hundred of these events, we first create a random number generator, and then repeat the code for the generating single event one hundred times:

```
| r t |
r := Random newForKymaWithSeed: 52.
t := 0.

100 timesRepeat: [
    self
        keyDownAt: t s
        duration: (r next + 1) s
        frequency: 3 c + (r next * 36 nn)
        velocity: r next.
    t := t + r next].
```

You may recognize this as a Smalltalk program (the implication being that you can use any Smalltalk expressions and control structures to algorithmically generate your MIDI events)!

Local variables are declared at the top by placing them between vertical lines. Then a random number generator is created and stored in the variable *r*. Another variable, *t*, which will be keeping track of time, is initially set to zero:

```
| r t |
r := Random newForKymaWithSeed: 52.
t := 0.
```

Next we repeat the statement within the square brackets one hundred times. Each time through, the duration is set to some random number between 1 and 2, the frequency is set to a pitch between 3 c and 6 c (not limited to integer note numbers), the velocity is set to a random number between 0 and 1, and the start time of the next event is set to a random time up to 1 second after this event:

```
100 timesRepeat: [
    self
        keyDownAt: t s
        duration: (r next + 1) s
        frequency: 3 c + (r next * 36 nn)
        velocity: r next.
    t := t + r next].
```

To hear the results, play **100 uniform random**. Take a look inside the *Attenuator* to see how `!KeyVelocity` is really being used to control stereo placement, rather than amplitude.

Random generates random numbers that are evenly distributed between zero and one. Let's modify our program slightly, replacing **Random** as the random number generator with several **OneOverF** generators, one for pitch, one for duration, one for velocity, and one for the next start time:

```
| rPch rDur rVel rStart t |

rPch := OneOverF newForKymaWithSeed: 52 states: 128.
rDur := OneOverF newForKymaWithSeed: 52 states: 128.
rVel := OneOverF newForKymaWithSeed: 52 states: 128.
rStart := OneOverF newForKymaWithSeed: 52 states: 128.

t := 0.
200 timesRepeat: [
    self
        keyDownAt: t s
        duration: rDur next s
        frequency: 3 c + ((rPch next * 36) roundTo: 1) nn
        velocity: rVel next.
    t := t + (rStart next * 0.5)].
```

Like **Random**, **OneOverF** generates random numbers between zero and one, but it tends to generate numbers that are close to one another for awhile, then make a large jump to a new number, then generate numbers close to that one for awhile, *etc.* It gets its name from the shape of the distribution in the frequency domain: $1/f$ or the inverse of the frequency; in other words, there tend to be large, slow changes and small, fast changes. To hear what this sounds like, play **200 oneOverF**. Try changing the value of the seeds to hear a different set of events.

Controllers

Play **100 uniform + ctrl** and then open it to see an example of how to specify continuous controller values in a MIDI script. Here the controller **!Morph** is set to 0 at the beginning and is told to slide from its previous value to a new random value on each note taking 10 steps to get there:

```
| r t |

r := Random newForKymaWithSeed: 2.
t := 0.
self controller: !Morph setTo: 0 atTime: 0 s.
100 timesRepeat: [
    self
        keyDownAt: t s
        duration: (r next + 1) s
        frequency: 3 c + (r next * 36 nn)
        velocity: r next.
    t := t + r next.
    self controller: !Morph slideTo: r next steps: 10 byTime: t s].
```

Next, play **200 oneOverF + ctrl** and then open it to see how it works. In this script, **!Morph** is set to zero initially, and then slides to a value of one over the course of 40 seconds.

Event Collections

Sometimes it is more convenient to specify the MIDI events as collections of notes and rests, without having to specify start times for each event. The actual start times can be inferred from the duration of the **Note** or **Rest** and where it occurs in the collection of events. This corresponds more closely to written music notation, where note and rest symbols arranged horizontally are interpreted as a sequence of events in time (where each event's start time occurs right after the previous event's duration has expired), and notes or rests arranged vertically are interpreted as all starting at the same time.

In the MIDI script language, an **EventSequence** is a collection of **Notes**, **Rests**, or other **EventCollections** that occur one after another in time (corresponding to horizontal placement in music notation). Since you can also construct sequences of other collections, you can create higher level structures as well. For example, sequences of **Notes** and **Rests** could be collected in a measure; sequences of measures could be collected into a phrase; sequences of phrases could be collected into sections; sequences of sections could be collected into movements, etc. until you run out of traditional musical names for the structures(!)

An **EventMix** is a collection of **Notes**, **Rests**, or other **EventCollections** that occur all at the same time (corresponding to vertical placement in music notation). Like an **EventSequence**, the **EventMix** is recursively defined (i.e. you could have an **EventMix** of **EventMixes**), allowing you to define hierarchical structures, somewhat analogously to the way you can define Sounds.

You can also create a generic **EventCollection** of **Notes**, **Rests**, or other **EventCollections**, specifying that you haven't yet decided whether the events should be simultaneous or sequential but will send a message to the object later to specify actual start times and turn it into a sequence or a mix of other events.

EventSequence

Play the Sound called *re-turn of phrase*, and then open it so you can study the MIDI script (by the way, in case you missed this earlier, you can use **Ctrl+L** to make any parameter field larger — an essential little feature for viewing and editing these scripts).

The first line is just a declaration of all the variables: `measure`, `group`, `phrase`, and `section`. Next, a new sequence is created: a half note on 2 g, followed by a quarter note on 3 d, followed by a half note 2 d:

```
| measure group phrase section |

measure :=
  EventSequence events: #(
    {Note durationInBeats: 2 frequency: 2 g}
    {Note durationInBeats: 1 frequency: 3 d}
    {Note durationInBeats: 2 frequency: 2 d}).
```

Next, `group` is defined as a sequence of three measures — but not just the same measure repeated three times; it is the original measure, followed by the measure transposed up a perfect fifth (7 half steps) and with all its durations scaled to half their original values, followed by the original measure transposed down a perfect fourth:

```
group :=
  EventSequence events:
    (Array
      with: measure
      with: ((measure dim: 0.5) trsp: 7)
      with: (measure trsp: -5)).
```

Then, `phrase` is defined as a sequence of three groups — with each copy of `group` modified in the same way as each copy of `measure` was modified to form the original `group`. Things are starting to look pretty self-similar here:

```
phrase :=
  EventSequence events:
    (Array
      with: group
      with: ((group dim: 0.5) trsp: 7)
      with: (group trsp: -5)).
```

By now, you can guess that `section` is going to be defined as a sequence of three phrases, and that each copy of `phrase` is subjected to the same transformations:

```
section :=
  EventSequence events:
    (Array
      with: phrase
      with: (phrase inv: 3 d)
      with: (phrase trsp: -5) retrograde).
```

The same set of transformations can be applied to **Notes**, **Rests**, **EventSequences**, and **EventMixes**, and each will be affected in the appropriate manner; for example, if you apply `trsp:` to a **Rest**, it just remains as is; if you apply `trsp:` to an **EventSequence**, each one of its component events is transposed. In this example, we could have gotten by with defining a single note as our starting point and applied transformations to it to derive the remaining notes in the measure:

```
n := Note durationInBeats: 2 frequency: 2 g.
measure :=
  EventSequence events:
    (Array
      with: n
      with: ((n trsp: 7) dim: 0.5)
      with: (n trsp: -5)).
```

The final statement in the script tells the **EventSequence** in `section` to play on this *MIDI Voice* (i.e. the Sound for which this **Script** is a parameter), and sets the metronome to MM = 160.

```
section playOnVoice: self bpm: 160.
```

Another Way

Re-re-turn of phrase shows a variant on this same algorithm. Rather than building up the sequences of sequences and storing them in different variables, *Re-re-turn of phrase* uses the Smalltalk control structure `timesRepeat:` to repeatedly replace the old phrase with a new phrase:

```
| thing |

thing :=
    EventSequence events: #(
        {Note durationInBeats: 2 frequency: 3 c}
        {Note durationInBeats: 1 frequency: 3 g}
        {Note durationInBeats: 2 frequency: 2 g}).

6 timesRepeat: [
    thing :=
        EventSequence events:
            (Array
                with: thing
                with: ((thing dim: 0.5) trsp: 7)
                with: (thing trsp: -5)).

    thing playOnVoice: self bpm: 160.
```

Embedded Event-like Behavior

Sometimes the “instrument” itself has some event-like behavior buried within it. Play the Sound called *PnoToHrp* and then open it so you can study the MIDI script.

After all the variable declarations:

```
| pat1 dblPat group |
```

the first step is to create an **EventSequence** and save it in the variable called `pat1`.

```
pat1 :=
    EventSequence events: #(
        {Note durationInBeats: 4 frequency: 4 g}
        {Note durationInBeats: (8/5) frequency: 4 b flat}
        {Note durationInBeats: (12/5) frequency: 5 c}).
```

`pat1` is a sequence that lasts for 8 beats. You could think of it as a whole note in a measure by itself, followed by a measure containing a half note followed by a dotted half note with both notes under a bracket labeled 5:4 or “five in the time of four”.

“But,” you are thinking, “it doesn’t sound like that; it sounds more like a group of five, a group of two, and a group of three.” The secret lies buried within the **TimeIndex** parameter of *PNO*, a *FunctionGenerator* called *time index*. Take a look at the **Gate** parameter of *time index*. It is saying that for as long as the key is held down, the gate should be triggered at the rate of 160 beats per minute — scaled to be a little bit faster so we can get five beats in the time of four. So whatever pitches this “instrument” plays, it will play quarter note quintuplets at MM = 160 (something like the notation equivalent of a slash above the note heads to indicate a 5:4 quarter note tremolo on each note):

```
!KeyDown bpm: 160 * (5/4)
```

The changing value of the **Morph** parameter in *pno* is not controlled by the MIDI script either. Instead, it is embedded directly in the parameter field as a function that goes from zero to one over the course of 12 seconds (the first 1 just indicates that the ramp should start immediately and continue for 12 seconds rather than restarting each time a key goes down; otherwise, you would replace the 1 with `!KeyDown`):

```
1 ramp: 12 s
```

EventMix

Play the example called *out-o-tune*, and then edit the Sound to examine its script. This is an example of creating and using an **EventMix**.

First we declare the variables within vertical lines. Then we set the variable `chord` to an **EventMix**. In this case, the chord is constructed out of perfect fourths above 3 d — and these are really perfect fourths, because we constructed them by scaling the frequency of 3 d (obtained by sending the message `hz` to 3 d) by the ratio 4/3.

```
| chord seq |

chord :=
  EventMix events: (Array
    with: (Note durationInBeats: 2 frequency: 3 d)
    with: (Note durationInBeats: 2 frequency: 3 d hz * 4/3)
    with: (Note durationInBeats: 2 frequency: 3 d hz * (4/3 ** 2))
    with: (Note durationInBeats: 2 frequency: 3 d hz * (4/3 ** 3))
  ).
```

Next, we construct a sequence of these chords. Each chord has all of its frequencies scaled by 4/3 above the frequencies of the previous chord. In order to do frequency scaling (multiplying the frequencies by a value) rather than pitch transposition (adding a value to the note number), we use the transformation `freqScale:` rather than the transformation `trsp:`.

```
seq :=
  EventSequence events: (Array
    with: chord
    with: ((chord freqScale: 4/3) dim: 0.125)
    with: (chord freqScale: 16/9)).

seq playOnVoice: self bpm: 60.
```

Finally, the controller called `!Frequency` is set to start out at 0 and, at the time 3 seconds, to drop down to -36 in 60 steps over the course of one second. Notice that you can assign any values to controllers in the MIDI script; they are not limited to a range of (0,1) or (0,127):

```
self controller: !Frequency setTo: 0 atTime: 0 s.
self controller: !Frequency setTo: 0 atTime: 3 s.
self controller: !Frequency slideTo: -36 steps: 60 byTime: 4 s.
```

TimedEventCollections

Take a look at (and have a listen to) *serial time points*. This script contains an example of a **TimedEventCollection**: a set of events associated with a set of starting beats. First, we declare the variables, define a set of intervals, and a collection of events based on those intervals:

```
| set evs beats prime t |

set := #(0 1 11 5 2 4 3 9 10 6 8 7).

evs :=
  set collect: [ :int |
    Note durationInBeats: int frequency: 3 a + int nn].
```

Next, we derive a set of starting beats from the set. The distance from one beat to the next comes from the set:

```
t := 0.
beats := set collect: [ :int | t := t + int].
```

Here is where we create the **TimedEventCollection**:

```
prime := TimedEventCollection startingBeats: beats events: evs.
```

Then, starting on beat 0, we play the events (after scaling all the `!KeyVelocity` values to one quarter of their original values; look at the *Pan* to see how we have twisted the meaning of `!KeyVelocity` to really control stereo placement):

```
(prime velScale: 0.25) playOnVoice: self onBeat: 0 bpm: 400.
```

On beat 11, we start up another copy of the **TimedEventSequence**, this one inverted about 3 a and placed in the opposite channel:

```
((prime inv: 3 a) velScale: 0.75) playOnVoice: self onBeat: 11 bpm: 400.
```

TimedEventCollections Read from MIDI Files

You can also create a **TimedEventCollection** by reading in the events and starting beats from a MIDI file. If you want to play exactly what's in the MIDI file, it would be quicker and more straightforward to simply select MIDI file as the **Source** in your *MIDIVoice*. However, once you have read the MIDI file into a script you can transform it, twist it, distort it, in short, do all the kinds of things composers love to do!

For example, take a look at the **Script** for the Sound called *transform events from MIDI file*:

```
| pat |

pat :=
    TimedEventCollection
        timesAndEventsFromMIDIFile: 'omino.mid'
        channel: 1.

(pat inv: 60 nn) playOnVoice: self onBeat: 5 bpm: 120.
(pat trsp: 31) retrograde playOnVoice: self bpm: 120.
```

In this script, `pat` is a **TimedEventCollection** whose start times and events are read from channel 1 of MIDI file `omino.mid`. To hear the original MIDI file, select **MIDI file** as the **Source**, and play. Then select **Script** as the **Source** and play again to hear the transformed file. Type the name of one of your own MIDI files in place of `omino.mid` and try it again.

Generic EventCollections

In randomized times on event collection, you can see an example of a generic **EventCollection**, one that is not yet ready to make a commitment to full seriality or simultaneity:

```
| pat pat2 pat1 |

pat :=
    EventCollection events: (Array
        with: (Note frequency: 3 d durationInBeats: 1)
        with: (Note frequency: 3 f durationInBeats: 1)
        with: (Note frequency: 3 a sharp durationInBeats: 1)
        with: (Note frequency: 3 b sharp durationInBeats: 1)).
```

But the message `randomizeUsing:totalBeats:quantizeTo:maxSpacing:` changes the **EventCollection** into a **TimedEventCollection** by generating a set of random starting beats and randomly picking a set of events from the events supplied. You specify the total duration of the result, the smallest distance between beats (with `quantizeTo:`) and the longest time between beats (`maxSpacing:`):

```
pat1 :=
    pat
        randomizeUsing: (Random new seed: 92)
        totalBeats: 16
        quantizeTo: 0.5
        maxSpacing: 1.
```

Markov Chain

In Kyma, the **MarkovChain** is a generic object, not necessarily tied to musical applications. It is a kind of sparse, fast-food version of the classic Markov chain: instead of weighted transitions, you give it a “training sequence” in the form of an **Array**, and it constructs weighted transitions based on what transitions occur and how many times they occur in the training sequence. Once you have “trained” the object this way, you can ask it to generate any number of sequences, each of which will be statistically related to the original training sequence you gave it.

For example, if you evaluate the following, short program,

```
| chain seq |

chain :=
  MarkovChain
    trainingSequence: #(
      what is your name
      is it an animal name
      or is your name one that would belong to a human
      like my name is a name that would
      belong to a very human being
    )
    chainLength: 2.
  seq := chain newSequenceOfLength: 100 seed: 9008.
```

you might get something like this as the result:

```
OrderedCollection(your name is a name that would belong to a very human being)
```

Since the chain length is 2, every pair of words in the generated sequence is one that occurs in the original training sequence. However, the entire chain is something new, not found anywhere in the sequence. Each time you ask the **MarkovChain** for a new sequence, giving it a different seed, the result will be a new sequence. Notice that we asked for a sequence that was 100 words long; when the **MarkovChain** backs itself into a corner and can’t find any more legal transitions, it returns what it has been able to come up with so far.

Now let’s look at a musical example. Here’s the script from the example called *modified Markov chains*:

```
| t chain dur pitchOffset seq |

MM := 160.

chain :=
  MarkovChain
    trainingSequence: #(
      #(0 4) #(3 2) #(-5 2) #(2 4) #(3 2) #(-5 2) #(2 4)
      #(3 2) #(-5 4) #(2 2) #(3 2) #(-5 4) #(2 2) #(-5 2)
      #(5 2) #(-7 4) #(1 2) #(-2 2) #(1 4) #(0 0.5) #(12 4)
    )
    chainLength: 2.

t := 0.
1 to: 10 do: [ :i |
  seq := chain newSequenceOfLength: 100 seed: i * 26.

  seq do: [ :offDur |
    dur := (offDur at: 2) / i.
    pitchOffset := offDur at: 1.
    self
      keyDownAt: t beats
      duration: dur beats
      frequency: 1 g + pitchOffset nn + i nn
      velocity: i/10.
    t := t + dur]].
```


The training sequence is just an **Array of Arrays**, each subarray containing two numbers — nothing specifically musical yet:

```
chain :=
  MarkovChain
    trainingSequence: #(
      #(0 4) #(3 2) #(-5 2) #(2 4) #(3 2) #(-5 2) #(2 4)
      #(3 2) #(-5 4) #(2 2) #(3 2) #(-5 4) #(2 2) #(-5 2)
      #(5 2) #(-7 4) #(1 2) #(-2 2) #(1 4) #(0 0.5) #(12 4)
    )
    chainLength: 2.
```

Next, there is a loop in which the index, *i*, takes on the values from 1 to 10. Each time through the loop a new sequence is generated, using a different seed each time, and saved in the variable called *seq*:

```
1 to: 10 do: [ :i |
  seq := chain newSequenceOfLength: 100 seed: i * 26.
```

In Smalltalk, you can also create a loop by sending the *do:* message to an **Array**. The following is a loop in which the variable *offDur* takes on each of the values stored in the **Array** called *seq*. Each value in *seq* is a subarray containing two numbers. Take a look through this code to see if every step makes sense to you. (If not, don't worry, it is all explained in more verbose style in the following paragraph!)

```
seq do: [ :offDur |
  dur := (offDur at: 2) / i.
  pitchOffset := offDur at: 1.
  self
    keyDownAt: t beats
    duration: dur beats
    frequency: 1 g + pitchOffset nn + i nn
    velocity: i/10.
  t := t + dur].
```

Each time through the loop, the second element of *offDur* is divided by *i* (the index of the outer loop) and stored in *dur*. The first element of *offDur* is stored in the variable *pitchOffset*. Then a MIDI event is created that starts on beat *t*, lasts for *dur* beats, and whose frequency is 1 g plus the number of half steps stored in *pitchOffset* plus a number of half steps equal to the index *i*. Then the time counter *t* is incremented by the duration of the event just created and the loop repeats. The overall effect is that we hear several, related sequences, each one faster and at a higher pitch than the one before it.

Using MIDI Files with the MarkovChain

The longer the training sequence, the more likely the generated sequences are to sound related to the original without being exact copies of the original. But entering long arrays by hand can be tedious, so here is a little trick for obtaining long sequences of MIDI events for use in a **MarkovChain**: Create a **TimedEventSequence** by reading it from a MIDI file, make sure it is monophonic, and then create a **MarkovChain** using just the events from the **TimedEventSequence**, ignoring the starting beats.

Play *fantasia bill* and continue reading while it churns away on this long sequence of MIDI events. Here is the script:

```
| pat chain seq |

pat :=
  TimedEventCollection
    timesAndEventsFromMIDIFile: 'DESAFIN1.MID'
    channel: 3.

pat := pat asMonophonicEventSequence.
chain := MarkovChain trainingSequence: pat events chainLength: 3.
seq := chain newSequenceOfLength: 200 seed: 3.
(EventSequence events: seq) playOnVoice: self onBeat: 0 bpm: 90.
```

First we create a **TimedEventSequence** from a MIDI file.[§] Then we get rid of any polyphony using the `asMonophonicEventSequence` message. Then we create a **MarkovChain** (length 3), using the events from the **TimedEventCollection**. We then ask the **MarkovChain** to create a new sequence. Finally, we create a new **EventSequence**, supplying the Markov-generated sequence as the events. When the example plays, you will hear one part of the original sequence, followed by the Markov-generated sequence.

Recursion

Take a look at the Sound called *self similar melody*. This script is an example of how you can create a Smalltalk “block” and then call it recursively:

```
| pitchShape durationShape shapingFunction selfSimMel |

pitchShape := #(0 7 -12).
durationShape := #({2/5} {1/5} {2/5}).
shapingFunction := [ :seq :count |
    count = 0
    ifTrue: [seq]
    ifFalse: [
        EventSequence events:
            ((1 to: pitchShape size) collect: [ :i |
                shapingFunction
                    value:
                        ((seq trsp: (pitchShape at: i))
                            dim: (durationShape at: i))
                    value: count - 1])]].
selfSimMel :=
    shapingFunction
        value: (Note frequency: 4 c durationInBeats: 16)
        value: 4.
selfSimMel playOnVoice: self onBeat: 0 bpm: 45.
```

As usual, we start by declaring the variables. Then we set `pitchShape` to an **Array** of intervals specified in half steps, and we set `durationShape` to an **Array** of ratios, each of which corresponds to an amount by which to scale a duration:

```
| pitchShape durationShape shapingFunction selfSimMel |

pitchShape := #(0 7 -12).
durationShape := #({2/5} {1/5} {2/5}).
```

Next we store an entire block of code (*i.e.*, everything within the square brackets) in a variable called `shapingFunction`. This block has two arguments (`:seq` and `:count`), and they are listed at the very beginning of the block, separated from the rest of the block by a vertical bar. The code within the block is not executed at this point; it is just stored in a variable so it can be executed later:

```
shapingFunction := [ :seq :count |
    count = 0
    ifTrue: [seq]
    ifFalse: [
        EventSequence events:
            ((1 to: pitchShape size) collect: [ :i |
                shapingFunction
                    value:
                        ((seq trsp: (pitchShape at: i))
                            dim: (durationShape at: i))
                    value: count - 1])]].
```

[§] Thanks to Bill Walker (walker@taurus.apple.com) for the MIDI file **DESAFIN1.MID**.

To actually execute the block of code, we have to supply values for each of its arguments. The following line executes the code stored in `shapingFunction` and stores it in the variable `selfSimMel`. It sets the value of the first argument to a **Note** event on middle C and the second argument to the number 4:

```
selfSimMel :=
  shapingFunction
    value: (Note frequency: 4 c durationInBeats: 16)
    value: 4.
```

Now let's examine what the block of code will do given these values. First, it checks whether `count` is equal to 0. It is not, so it chooses to execute the `ifFalse:` block:

```
count = 0
  ifTrue: [seq]
  ifFalse: [
    EventSequence events:
    ((1 to: pitchShape size) collect: [ :i |
      shapingFunction
        value:
          ((seq trsp: (pitchShape at: i))
            dim: (durationShape at: i))
        value: count - 1]])]
```

Let's look at what the `ifFalse:` block does. It uses the `collect:` message to create an **Array** of events for a new **EventSequence**. For each entry in `pitchShape` and `durationShape`, it executes the block stored in `shapingFunction` with the event in `seq` transposed by the current entry in the `pitchShape` array and its duration scaled by entry in the `durationShape` array, and one less than the value of `count`. The result is that `shapingFunction` is evaluated three times (because the `pitchShape` array is three long) with a `count` value of 3. Eventually, `count` will be 0, and the function returns the value of `seq`.

To summarize, the first time it calls the function with a single event and `count` is 4; then inside there it calls itself three more times with `count` set to 3, and each of those calls itself three more times with `count` set to 2, each one of those calls itself three more times with `count` set to 1, and finally it calls itself three times with `count` set to 0, so each branch returns at that point and as it returns back up through all the calls to the function, the **EventSequences** are collected at each step, until it gets back up to the top with a recursive collection of **EventSequences**.

As the last step, the new **EventSequence** plays itself.

```
selfSimMel playOnVoice: self onBeat: 0 bpm: 45.
```

Of course, you could get the same result using iteration:

```
| c1 |

c1 := Note frequency: 4 c durationInBeats: 16.
4 timesRepeat: [
  c1 :=
    ((1 to: 3) collect: [ :i |
      c1 trsp: (pitchShape at: i))
      dim: (durationShape at: i))].
```

but doing the same thing with recursion is just a little bit more thrilling for your mind (which must be kept entertained if you are not to lose it).

Morphing

Imagine a scene from your favorite science fiction movie where the character changes from flesh to molten metal or from human to saurian-faced alien. Now imagine that visual morph supported by an analogous morph of the sound — a voice changing from human speech to metallic clanging or pitched-down porcine screeches. Or imagine a sweetly chirping bird morphing into a wailing siren, or a lounge singer morphing into a werewolf howling. Sound tracks have lagged behind visual morphing effects for long enough! Let's all get out there and create the sound morphs to match or surpass those spectacular visual effects!

The Venerable Cross-fade (and why it is not enough)

Play *Crossfade amplitudes*, and open it up to take a look at it; this is a *Crossfade* between two *Samples*, controlled by a ramp function. At the beginning you hear the female voice as the loudest and at the end you hear the male voice at the loudest, but at all times you hear them as two separate voices.

Let's try refining this a little, and "pitch down" the female voice to the same frequency as the male voice while cross fading the amplitudes. You can hear the results in *cross amp & freq*. That's a little more interesting, though it still sounds like two voices.

Now try *cross spectra*, a *SumOfSines* Sound that cross fades between harmonic, spectral analyses of the male and female voices. This sounds like something entirely different from crossfading. While the voice does sound a little odd in the intermediate stages, it always sounds like one single voice that is changing in character, not like two separate people.

Finally, try *dominic-carla*, to hear an example of what sounds almost like a child's voice going through different stages of growth.

Select *cross spectra* in the Sound file window, and expand it by typing **Ctrl+E**. Double-click on the expanded version to see how the *SumOfSines* is put together: the Sound called *interpSpect* does a crossfade between its two inputs, *analysis0* and *analysis1*. Notice that the left channels of the two inputs can be controlled independently of the right channels of the two inputs. The left channel of a spectral source contains all the amplitudes and the right channel contains all the frequencies.

Variations on a Gender Bending Theme

You already know how many ways there are to do crossfades — variations on the speed of the fade, the shape of the crossfade function, all the combinations of fading functions on the starting and ending sounds, *etc.* There are even *more* ways to do morphing. In the first place, in crossfading you have control over only one parameter — amplitude; in morphing, you have control over two parameters — amplitude and frequency.[§]

Play, for example, *synchronize & crossfade* to hear a resynthesis of the two original samples used in the subsequent morphing examples. You have control over the crossfade function using **!Morph** in the virtual control surface (or the corresponding MIDI fader). Experiment with morphing at different points in the phrase and at different speeds.

Now compare the following Sounds: *freq first*, *then amp*, *linear morph*, and *morph on vowel* (time will slow down on "you", giving you time to complete the entire morph during a vowel).

Independent Frequency and Amplitude

In *freq first, then amp*, the frequencies morph before the amplitudes. In other words, it gradually cross-fades between the frequencies of the female voice to the frequencies of the male voice, until it reaches a point when there are all male frequencies but all female amplitude envelopes. Then it quickly crossfades

[§] In actuality, you have control over hundreds of pairs of amplitudes and frequencies should you choose to morph each of the analysis tracks independently of each other. While this is impractical, it may make sense in some instances to morph subsets of the harmonics independently of one another.

between the amplitude envelopes of the female and male voices. The left channel (the amplitude) is controlled by

$$2 * !Morph - 1$$

while the right channel (the frequency) is controlled by

$$2 * !Morph$$

Because the values are limited to the range of 0 to 1, these functions clip at their low and high extremes. In other words, $2 * !Morph$ will reach its maximum value of 1 when $!Morph$ is 0.5 and will stay at its maximum value of 1 for all larger values of $!Morph$. On the other hand, $2 * !Morph - 1$, will remain 0 for $!Morph$ values of 0 through 0.5, and will then climb from 0 up to 1 as $!Morph$ increases from 0.5 up to 1.

The result sounds like a woman growing larger in size because of the lower fundamental frequency and the shift to lower formant regions or resonances. At the last moment, it seems to “come into focus” as the normal formant regions of the male voice take over.

Amplitude and Frequency Together

The next example, *linear morph*, controls both amplitudes and frequencies with a single morph parameter. This gives a different impression than the last example; it sounds more continuous (although this approach is the most sensitive to artifacts in the analysis).

I'd like to buy a vowel

In listening to the previous examples, you probably noticed that most of the artifacts occur during the consonants, so another approach might be to morph only during vowels. *morph on vowel* uses *WarpedTimeIndex* to stretch time during the [u] vowel in “you”, giving you enough time to complete the morph during a vowel. This was done by adding

$$!Stretch\ s$$

or $!Stretch$ number of seconds to each of the **IdealTimePoints** following the time point at which the vowel occurs. (You can determine the time at which the vowel starts by locating it in the graphical spectrum editor).

Sound Effects

The next example, *harley to lion*,[†] uses a *GraphicEnvelope* to control the shape of the morph; double-click on the Sound called *env* to see the function. This is a handy way to design your morph functions in an interactive, yet repeatable way.

Try playing *cat baby* and then double-click on it to take a look at how it was constructed. This is a *Concatenation* of a resynthesized cat (to establish an unmodified, familiar sound in the listener’s mind first), a cat-to-baby morph, a baby-to-cat morph, and finally another resynthesis of the cat. The *singing were-wolf* example is similar.[§]

Musical Phrases

didgeri-mouth uses an *Oscillator* to control a morph back and forth between a didgeridoo and a mouth harp using the Gaussian wavetable as the shape of the morph.

In *drum to tubular*, a *TimeOffset* is used to delay the ramp function that controls the amplitude morph, so the pitch is completely morphed before the beginning of the amplitude morph.

[†] Special thanks to Frank Serafine for permission to include these analyses of sound effects samples from his CD, *Platinum Sounds for the 21st Century* from L2 Sound FX. For more information on this sound library, call +1-800-779-L2FX.

[§] Special thanks to Scott Whitney at Hollywood Edge for permission to include the analyses we did of the cat, the baby, and the dog, which are derived from samples in their CD-ROM sound library *Hollywood Edge: The Edge Edition*. For more information on these sound libraries, visit their website at <http://www.hollywoodedge.com>.

Single Musical Tones

You can also morph between analyses of single tones from musical instruments. For example, *harp to ah (kbd)* uses the function

```
1 - (!KeyDown ramp: 1.5 s)
```

to morph from an analysis of a harp tone to an analysis of someone singing “ah” each time a MIDI key goes down.

You can also create more subtle morphs between different tones played on the same instrument but in different frequency ranges. As an example, take a look at *KBD morph TRB GA*. This examples uses GA synthesis to morph between three spectra; the morph is controlled by the MIDI key being played. For example, the morph between 3 c sharp and 4 d sharp is:

```
(!KeyNumber - 3 c sharp removeUnits) / (4 d sharp - 3 c sharp) removeUnits
```

or, to describe it another way,

```
<distance between !KeyNumber and 3 c sharp> / <full pitch range for this morph>
```

or the ratio of where the current key number is relative to the low end of the pitch range to the full pitch range.

How to do your own Morphs

But enough of these examples; what you really want is to try morphing your own samples, so here is a step-by-step example.*

I. Selecting and Preparing the Samples

The more characteristics the samples share, the smoother (although less dramatic) the morph will be. Generally speaking, it is best to choose samples that are about the same length, that are normalized and compressed to remove extreme amplitude changes, and, particularly if you are trying to morph speech, that are extremely well synchronized with each other in time. In fact, if you have the option, you should record one performer first and have the second performer listen to that track while recording the second track, trying to “lip synch” (ear synch?) as closely as they can to the first performer.

In the current version of Kyma, you will be much happier if you limit your samples to about six seconds or less in duration. If you intend to morph a long stretch of speech or music, it is best to look for the natural breaks in the speaking or singing (where the performer took a breath or paused, or the silence just before a consonant or new attack) and to split the sample up into sections bounded by these natural breaks. Then you can either do a string of partial morphs or morph during one of the sections in the middle.

While you would be well-advised to follow these guidelines for your first few morphs, once you have gotten a feel for how it works, you can break with some of the guidelines and start experimenting and discovering your *own* morphing techniques (the concept of morphing is still new and there is room for innovation and invention!)

II. Analyzing the Samples

For this example, let’s use the recordings of Kurt and Carla speaking the word “morphing”. To analyze, you can use the spectral analysis from Kyma’s **Tool** menu, or, if you are on the Macintosh, you can use a third party shareware program called Lemur[§] which is capable of exporting analysis files for Kyma.

1. Open the spectral analysis tool by choosing **Spectral Analysis** from the **Tools** menu.

* While the Spectral Analysis tool works on two-card systems, only a few sine wave oscillators can be used to audition the real-time analysis-resynthesis to test the parameters. You can still do the analysis and listen to the full resynthesis after producing the spectrum file, or you can use Lemur to do the analyses in non-real-time. If you plan to do a lot of analysis/resynthesis, we recommend that you use a 3-card system as the minimum configuration.

[§] Lemur was written by Kelly Fitz, Bryan Holloway, Bill Walker, and Lippold Haken and is available for downloading from their web site at <http://datura.cerl.uiuc.edu>. At this web site, you can also find Lime, a music notation program for both Macintosh and Windows written by Lippold Haken and Dorothea Blostein.

2. Select the first sample by clicking the **Select** button and using the file dialog to locate the sample `Cmorph` from the `speech` folder of the `Samples` folder of the `Wavetables` folder. Kyma will start playing the sample over and over with a 1 second silence between repetitions. To change the duration of the silence between repetitions of the sample to two seconds, type in a 2 in place of the 1, and press **Enter**.

Listen to the sample and estimate the pitch range of the speech. Then click **Next** to continue.

3. Choose the highest analysis frequency that is still lower than the lowest frequency you hear in the sample. In this example, the voice seems to go as low as 3 d or 3 c, so try **2 F** as the setting.

The analysis is done by a bank of band-pass filters equally spaced in frequency from 0 hz up to half of the sampling rate; the lower the frequency you choose here, the more filters there will be, and the closer those filters will be spaced in frequency. For **1 F**, it uses 512 filters, for **2 F** it uses 256 filters and so on (divide the number of filters in half each time you go up an octave in the analysis frequency).

Press the **Audition** button to hear a real-time analysis and resynthesis with the analysis parameters as you have them currently set. Just below the **Audition** button, Kyma will print out the number of sine wave oscillators being used to resynthesize the sound from the real-time analysis. This number will vary depending on the analysis frequency you have chosen and the number of expansion cards in your Cappybara.

Try out a different analysis frequency by choosing **5 F** and then clicking the **Audition** button again. Clearly this setting does not give you enough filters to adequately cover what is going on in the signal, so change it back to **2 F**.

4. In the last step, you effectively chose the number of bandpass filters to be used in the analysis. This next step controls the bandwidth of those filters. In the everpresent tradeoff between time and frequency, you can choose **BestFreq** and get the narrowest bandwidths (and the most time smearing), or you can choose **BestTime** to get the best time response (but the widest bandwidth and thus the worst frequency resolution), or you can choose any of the intermediate settings to get intermediate bandwidths and intermediate results with respect to time smearing and frequency resolution.

Leaving the frequency set at **2 F**, try out each of **BestFreq**, **BestTime**, **BetterFreq**, and **BetterTime**, clicking the **Audition** button after each choice to hear the differences. Generally speaking, you can tell when the bandwidth is too narrow if the result sounds like it has reverb or a delay on it, and you can tell when the bandwidth is too wide by a kind of roughness or distorted sound in the resynthesis. When you find the best sounding analysis (or the least-strange-sounding analysis if you are feeling negative today), click **Next** to continue.

5. You are ready to try the first analysis! Click the button labeled **Create Spectrum File**. The name of the analysis file will be the name of the original sample followed by an `s` (for spectrum) and the number of analysis filters used (or, to put it another way, the number of analysis “tracks” that can be read by an *OscillatorBank* and resynthesized).

Create your own wavetables folder called `MyWavetables`. In it, create a folder `Spectra`, and within `Spectra` create a folder called `speech`. Save the analysis file in this `speech` folder. The analysis tool automatically creates a *SumOfSines* Sound and places it into an untitled Sound file window for you. Just for fun, double-click on this Sound, change the **Duration** to `On`, and multiply the **OnDuration** by 10 to time-stretch the resynthesis.

A straight spectrum is best for time-stretching or frequency-scaling, or when the two files you are morphing have no definite pitch to them. However, if the original samples have identifiable pitches in them, the morph will probably work better if you proceed to the next step: the quasi-harmonic analysis.

If your original sample has an identifiable pitch to it, you should continue with the analysis by clicking on **Set Quasi-Harmonic Parameters**. Since `Cmorph` (and indeed human speech in general) *does* have an identifiable pitch, click on that button now to continue.

6. In this next section of the analysis, Kyma asks you to help identify the fundamental frequency of the original sample. The first step is to identify parts of the sound that do *not* have any identifiable pitch to them.

Adjust the fader upwards until you remove all of the pitched parts of the resynthesis. You'll know when the fader is in the proper position when all you can hear is the "ph" in "morphing" and a little gasp for breath at the end. You do *not* want to hear any of the "ing" and you do not want to hear the pitched parts of the sound breaking up.

It may seem a little counterintuitive to remove the pitched parts of the signal when you are trying to analyze just the pitched parts. The trick is that once you have removed everything except the noisy bits, then, by process of elimination Kyma knows which parts of the sound *do* have a pitch to them.

Once all you hear are clicks, air bursts or gasps, you can click **Next** to continue.

7. *Please wait* while Kyma prepares the next screen; watch the thermometer, or get yourself a glass of water, but don't start hitting keys until Kyma has displayed a color graph of the lower end of the spectrum.
8. The goal for this section is to look at the spectrum, decide which track is the fundamental frequency, and to trace the fundamental frequency track using the white line you see (now drawn straight across the lower part of the spectrum).

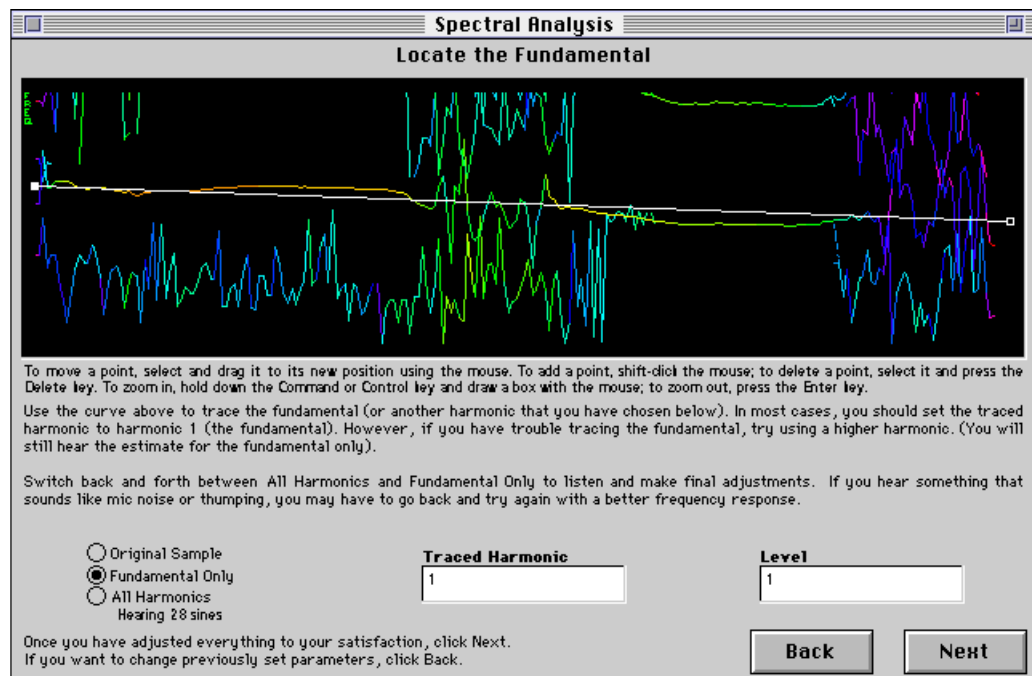
First, zoom in on what seems to be the fundamental (the orange colored line at the bottom of the screen):

Click in the black spectrum display to select it. You can tell when it is selected because you will see the square endpoints on the white line.

Hold down the **Control** or **Command** key until you see the cursor change to a magnifying glass.

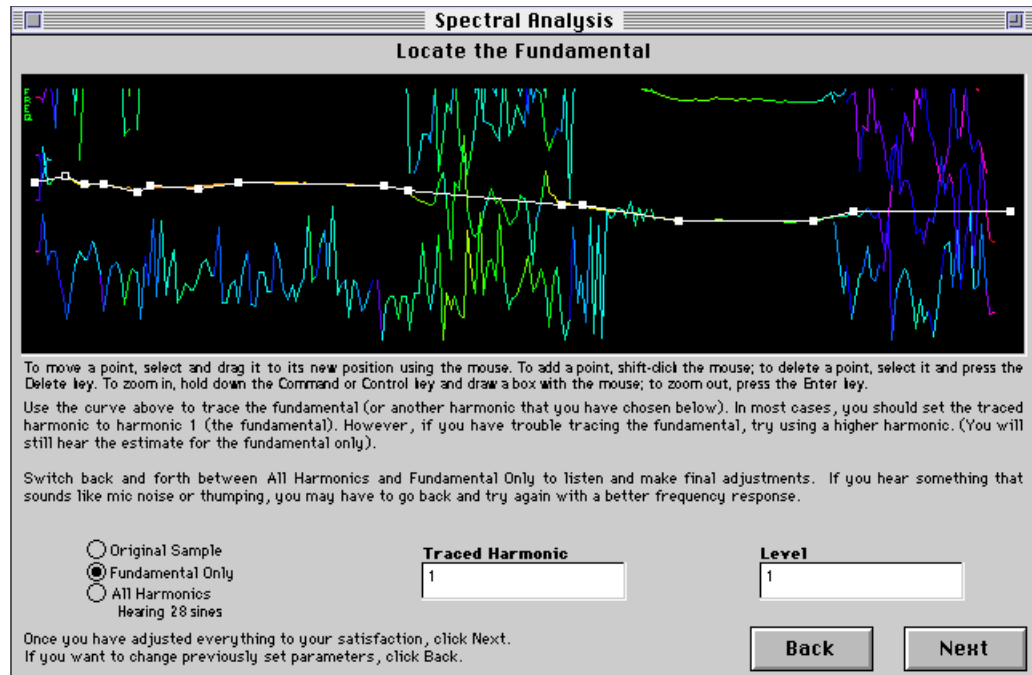
Still holding down the key, draw a box around the lower fourth of the spectrum. (The plus sign inside the magnifying glass is transparent, and the center point of the plus is where the cursor will draw the box).

Line up the first endpoint of the white line with the beginning of the fundamental track by clicking on the white box and dragging it upwards. Adjust the last endpoint similarly so that the white line draws a straight line closest to the orange fundamental.



Now add some additional breakpoints to the white line so that it traces the fundamental more closely. To add a new point, click the mouse while holding down the **Shift** key.

When you get to the “ph”, the “g”, and breath at the end of the word, the fundamental seems to get lost for a short time, and it just looks like crazy random green dots during that segment. Just put a breakpoint where the fundamental picks up again following the “ph”; in other words, always draw a straight line right through the ambiguous segments.



Once you have traced the fundamental with the white line, let things settle for one repetition of the sample and then listen. This is an oscillator whose frequency follows the estimated fundamental frequency curve you have drawn with the white line.

Now click where it says **Original Sample** to remind yourself of what it sounds like, and then click on **All Harmonics** to compare the resynthesized version with the original. If it sounds OK, click **Next**. Otherwise, you may have to trace the fundamental a little better or go back and try different analysis parameters. If you have been following along though, everything should sound pretty good at this point.

9. Click the button for **Create Quasi-Harmonic Analysis**, and leave the pitch set to its default 60 nn. 60 nn is the default pitch for any analysis that has more than one pitch in it or that has no definite pitch. The only time it makes sense to set this to anything other than 60 nn is for single instrument tones or isolated sung tones.
10. Click **New Sample** to go back through the entire process, this time selecting **KMorph** as the sample to analyze.

Follow the directions on the pages of the tool itself this time (referring back through steps 1-9 if necessary to refresh your memory). There are a few things that should be done differently for **KMorph**:

Set the analysis frequency to **1F** (because his voice is about an octave lower)

Set the time/bandwidth response to **BetterFreq**

Set the level to about 0.75 to avoid clipping

When you get to the part where you trace the fundamental, you can draw the zooming box around the white line, because it is pretty close to the real fundamental. Once you have zoomed in, it should be easy to see where the fundamental is and trace it out as you did before.

Once you have saved the four analysis files on disk (**CMorph h**, **CMorph s256**, **KMorph h**, and **KMorph s512**) close the spectral analysis tool window.

III. Synchronizing the Analyses

At this point, you could do a morph, but because this is a speech example, there is one more step that can make the morph work much better — that is to refine the time synchronization by looking at features in the analysis itself. You can probably skip this step if you are analyzing single instrument tones or if the two original samples are so different from each other that they don't really have any corresponding features that could be lined up. But in instances like this one, where you have two people speaking or singing the same word or words, the synchronization step is essential.

1. Choose **Synchronizing Spectra** from the **Tools** menu. Read the instructions, and then click the **Continue** button to start.
2. First, pick the guide spectrum, the one whose timing will not be warped. Click the **Guide** button, and locate the analysis file called `CMorph h`.
3. Once the spectrum is displayed, scrub through it to get an idea of the time at which each phoneme occurs. To scrub, you can either drag the yellow vertical line across the display with the mouse or use pitch bend on your MIDI keyboard to move the yellow line. To make fine adjustments in the position of the yellow line, use the left and right arrow keys on your computer keyboard. To remove a marker, select it and press **Delete**.
4. The next step is to put a marker at the time point at which each “feature” occurs in the guide spectrum. To put in a marker, position the scrub bar at the time when the feature occurs, and press the **m** key on your computer keyboard.

First scrub all the way to the left edge of the spectrum, and press **m** to put in a marker at the very beginning of the file

Scrub until you hear where the “m” first opens up into an “o”. Get close to the point, and then use the right-arrow key to move right, frame-by-frame until you hear the “m” change. Then press the **m** key.

Scrub over to the “r”, which looks like the point at which the spectrum gets a little wider. Hit **m**.

Scrub over to the “ph”, and use the arrow key to find the point at which the pitch seems to disappear. Put a marker here to mark the “ph”.

Scrub to the end of the “ph” visually. Use the right-arrow to determine which frame makes the transition from “unvoiced” to “voiced”. Mark it.

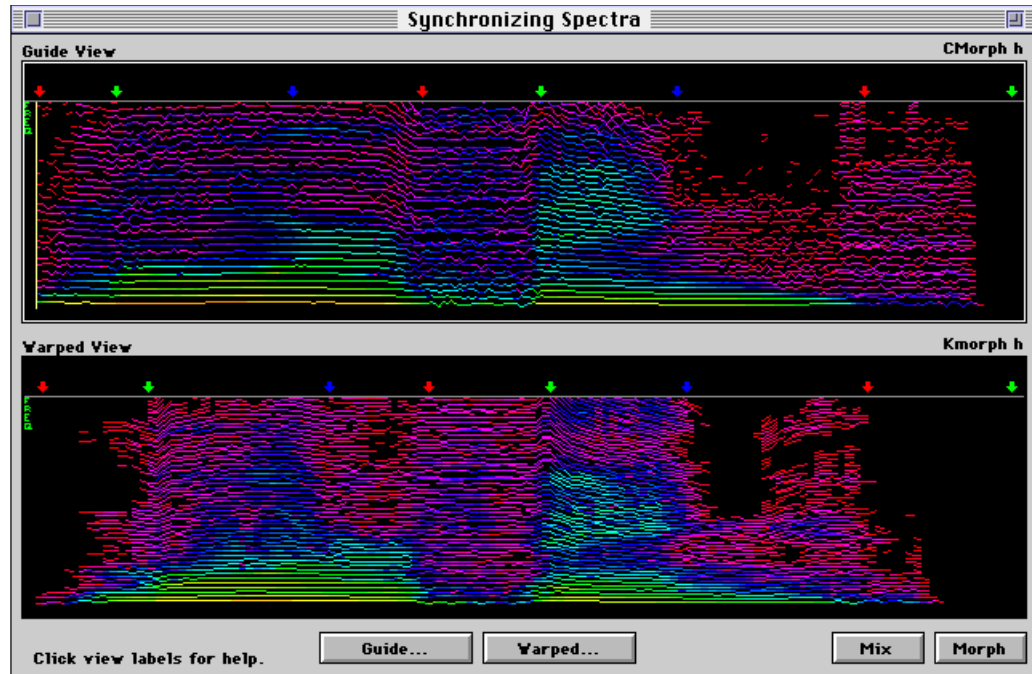
Scrub to the end of the “i” and beginning of “ng”. Mark it.

Mark the end of “ng”. You can see this one visually and also use the arrow key to find where the voice disappears.

Scrub all the way to the right edge, and mark the end of the file.

5. To save your work, press the **s** key. This will save the markers in the spectrum file. The cursor will change to the pen to show that it is writing to the file, and the resynthesis will play at the end to show that it has finished.
6. Now click the **Warped** button. Select the spectrum file whose timing will be warped such that all its marked features line up with the marked features in the guide file. In this instance, we want the file called `KMorph h`.
7. Mark all the same phonemes in `KMorph` that you marked in `CMorph` in step 4. Once you have finished, double check that the markers on corresponding phonemes are of the same color in both the guide spectrum and the warped spectrum. In other words, if the last marker is red in the guide, it should also be red in the warped.

8. Once all the markers are in *KMorph*, press the **s** key to save the markers into the spectrum file.



9. Click the **Mix** button. This produces a *Mixer* of the two resynthesized spectra. Select the *Mixer* and play it. Pan between the left and right outputs on your amp or mixer to assess how well the signals in the right and left channels line up in time. Double-click on the *Mix* and look at the *WarpedTimeIndex* that feeds into *KMorph*. Notice how the ideal time points (upper parameter field) are different from the actual time points (lower parameter field). The *WarpedTimeIndex* takes the real time points (where you placed the markers) and stretches or compresses time between those markers such that they line up with the ideal time points.
10. Click the **Morph** button. This creates an *OscillatorBank* that is ready to morph. Play it, and use the !Morph fader on the virtual control surface or MIDI faders to morph from one voice to the other.
11. Edit *crossSpects*, setting **LeftInterp** to !Volume and **RightInterp** to !Frequency. Play *morph*, and experiment with different settings of !Volume and !Frequency. Try morphing with !Frequency first, and then !Volume. Then try !Volume first, followed by !Frequency, and all the other combinations. This will give you a feel for what will make good morphing functions.
12. Copy and paste a *GraphicalEnvelope* into the !LeftInterp field. Experiment with drawing different amplitude morph shapes in the *GraphicalEnvelope* editor.

Alternatives to the Synchronizing Tool

You can, if you prefer, try to synchronize the two samples in a wave editor *before* doing the spectral analyses (for instance, you could cut out silences or cut/repeat cycles of vowels to make the phonemes of the two samples line up in time), or you can try using a TDM Plug-In called VocAlign™ from Synchro Arts Ltd in Surrey UK. Or you can use these techniques to get fairly closely synchronized samples in the time domain, and do the final synchronization tweaks in the synchronize spectra tool.

IV. Invent your own Morph

Every pair of samples will suggest a different kind of morphing technique. Some will work best with straight spectra, some work best as quasi-harmonic spectra, some work best synchronized, some work best without synchronization. Experiment, get a feel for what the tools do, and then go wild!

Output MIDI

These examples illustrate how you can generate MIDI events in Kyma and send them out to control external MIDI devices like synthesizers and samplers. You can use this to synchronize external devices to Kyma Sounds, to extract parameters from audio signals and send them out as MIDI events, or to algorithmically generate and process MIDI events in Kyma before sending them to your other sound modules.

Getting Wired

If you don't have any MIDI-controlled sound modules other than Kyma, then you can skip this section. If you do have a synthesizer/sampler, then hook everything up as follows:

Capybara MIDI output connected to synthesizer MIDI input

If you have a mixer, mix the outputs of the Capybara and the synthesizer

If you do *not* have a mixer, then patch synthesizer audio output into Capybara audio input

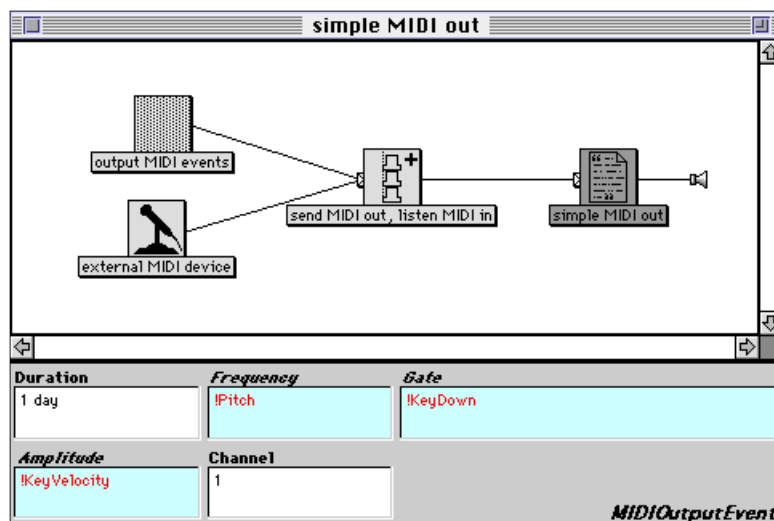
If your synthesizer/sampler has pitch bend, set it to ± 12 semitones

MIDI Notes Off

The quickest way to send an "all MIDI notes off" command in Kyma is to use **Ctrl+M** or else to choose **MIDI notes off** from the **DSP** menu.

Sending Out MIDI Note Events

Double-click *simple MIDI out* to see its structure and parameters:



Then double-click *output MIDI events*. This is a *MIDIOutputEvent*. You supply the three elements of a MIDI note event: a frequency, a gate, and an amplitude, and this Sound puts together a MIDI note event and sends it to the MIDI output of the Capybara. In this instance, we have set these parameters to come from the corresponding values from the MIDI input: **Frequency** to !Pitch, **Gate** to !KeyDown, and **Amplitude** to !KeyVelocity. So this Sound is performing the redundant but illustrative task of reading note events sent to the Capybara's MIDI input, splitting them into !Pitch, !KeyDown and !KeyVelocity, and then giving them to the *MIDIOutputEvent* Sound which puts them back together into a MIDI note-on event which it sends to the Capybara's MIDI output. Obviously you didn't need Kyma to do that; you could have just connected the keyboard to the synthesizer directly. But wait, there's more...

But first just a quick note about the structure of this Sound: notice that *external MIDI device* and *output MIDI events* are both feeding into a *Mixer*. This is not a *Mixer* in the usual sense, but in the broader,

Kyma sense of “simultaneity”. Unlike most Kyma Sounds, a *MIDIOutputEvent* does not produce an audio signal; it simply sends note events to the MIDI output port. Nevertheless, you can schedule it to occur at the same time as the *ADInput* (which, in this case, is the audio signal from an external synthesizer) by placing both Sounds in the same *Mixer*. (If you are mixing the external synthesizer with Kyma through your actual physical mixer, you can delete the *Mixer* Sound, replacing it with the *MIDIOutputEvent*, since you can hear the synthesizer through your mixer and do not need to bring it into Kyma in order to be able to hear it).

What happens if you send out a MIDI note whose pitch is not in 12-tone equal-tempered tuning? Try it out by entering the following into the **Frequency** field of the *MIDIOutputEvent*:

```
!Pitch * 0.5 + 48 nn
```

Select and load *simple MIDI out* and try playing half steps on the MIDI keyboard. Because the note numbers are being scaled by one half, you get a quarter-tone scale. Kyma computes the difference between the nearest equal-tempered note number and the note number you are requesting and sends the difference as pitch bend. In order for this to work correctly, you must have your synthesizer’s pitch bend set for ± 12 half steps.

Synchronizing and Doubling

Select *synchronizing* and **Compile, load, start** it. When you play the MIDI keyboard, you hear both a Kyma Sound and whatever patch is loaded into your external synthesizer, both controlled by the same MIDI keyboard events. This is one way to synchronize and blend timbres from Kyma with timbres from other sources.

Open up *synchronizing* to take a look at the signal flow diagram. *filt noise+atk* is a *Mixer* whose inputs are: *VCA* (a MIDI-controlled subtractive synthesis Sound), *FM ATK* (an attack synthesized using frequency modulation), and *send MIDI out* (a *MIDIOutputEvent* that forwards all your incoming MIDI keyboards events to the MIDI output on the Capybara).

Instead of mirroring the Kyma Sound, let’s change the *MIDIOutputEvent* so that the synthesizer doubles the Kyma Sound in fourths. Double-click *send MIDI out*, and change the **Frequency** parameter to read

```
!Pitch + 5 nn
```

Compile, load, start synchrony again and play some nice quartal harmonies on the MIDI keyboard.

Now let’s further modify the *MIDIOutputEvent* so as to trigger a little burst of repeated notes that die away over two seconds each time you hit a MIDI key. Change the **Gate** to

```
!KeyDown bpm: 250
```

to send MIDI notes out (repeating the same pitch) at a rate of 250 beats per minute. Change the **Amplitude** parameter to

```
1 - (!KeyDown ramp: 2 s)
```

so that each time you hit a key, it will start out at full amplitude and then linearly fade out to zero over the course of 1 second. **Compile, load, start synchrony** using **Ctrl+Space Bar**, and try playing the MIDI keyboard.

Processing MIDI

Take a look at the signal flow graph for the Sound called *MIDI as Sound*. This sound illustrates a little trick by which you can change MIDI data into a fake signal and then treat it as part of the signal flow path. If you take a *Constant*, and paste a MIDI event into the **Value** field, you can then feed that *Constant* into a chain of processing modules — thus treating MIDI as if it were an audio signal.

For instance, find the module called *KeyDown as Sound* in the open example. Its **Value** is set to *!KeyDown*. Then the *Constant* is fed into a *DelayWithFeedback* (called *keydown del&fdbk*). Look at the parameter fields of *send MIDI out*. Notice that *keydown del&fdbk* has been pasted into the **Gate** field. (To use a Sound to control a parameter, select the Sound, copy it, and paste it into the parameter field).

Compile, load, start MIDI as Sound and play the MIDI keyboard. Once you have a feel for what it is doing, try cautiously adjusting *!Delay* and *!Feedback* to hear how this affects the results.

If you want to turn MIDI key number into a signal, you have to divide it by 127 in the **Value** field of the *Constant* (because the *Constant* takes values between -1 and 1), and then multiply by 127 `nn` in the parameter field where you actually use the result. For example, look at the way MIDI pitch is turned into a signal and then mapped to a pentatonic scale using a *Waveshaper* in the Sound called *double pentatonic*.

Generating MIDI

Try playing the Sound called **output random MIDI events** and adjust the faders in the virtual control surface. Double-click on it to edit and look at the parameters of **events**. The **Gate** parameter is:

```
1 bpm: (!Rate * 1000)
```

which generates a trigger at a rate of `(!Rate * 1000)` beats per minute. The **Frequency** parameter

```
!Pitch + (((1 bpm: (!Rate * 1000)) nextRandom * !Jitter * 7)) of: #(0 1 2 3 5 8 13)) nn
```

is a bit more complicated, so let's break it down into sub-parts. We can paraphrase it as

```
!Pitch + <Something> nn
```

and then break down `<Something>` as

```
<anIndex> of: #(0 1 2 3 5 8 13)
```

In other words, we compute an index, and then use it to look up the value stored at that position in the array: `#(0 1 2 3 5 8 13)`.

The result is that we will add 0, 1, 2, ... or 13 half steps to whatever pitch is being played on the keyboard.

Now, how do we come up with `<anIndex>`? We can look at `<anIndex>` as

```
<aMetronome> nextRandom * !Jitter * 7
```

In other words, at some number of beats per minute, generate random numbers between -1 and 1 and multiply them by `!Jitter * 7`. We multiply by 7 because there are seven entries in the array. Multiplying by `!Jitter` allows us to control how much variation there is in the random index generator.

Finally, `<aMetronome>` is

```
1 bpm: (!Rate * 1000)
```

The first number is the trigger or gate for turning on and off the metronome. Since it is a constant 1, the metronome will stay on all the time. The rate of the metronome is going to be between 0 and 1000 beats per minute, depending upon how you have set `!Rate` in the virtual control surface.

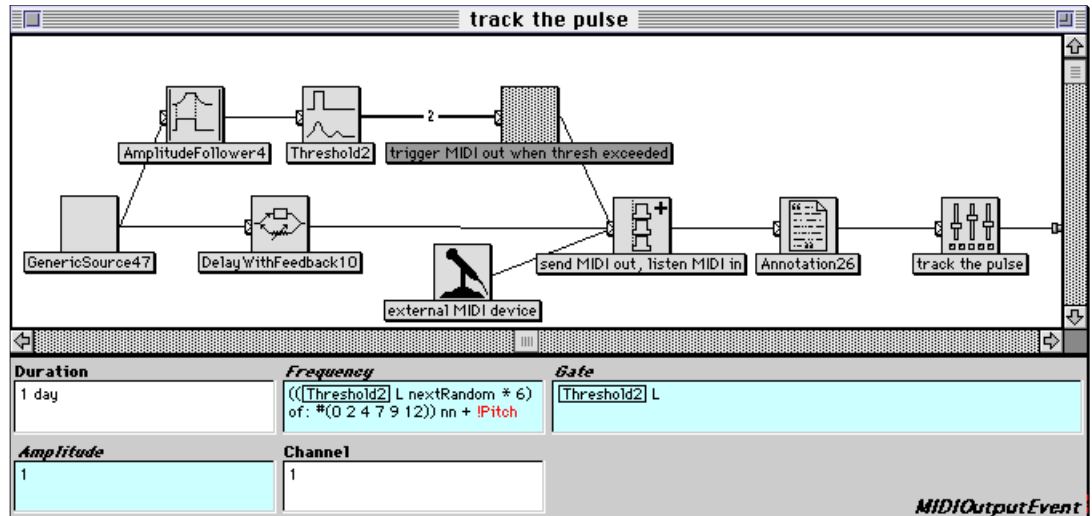
Extracting Parameters from Audio Signals

Once you have started down the path of pasting Sounds into the parameter fields of the *MIDIOutput-Event*, another idea immediately suggests itself: what about taking some of the "analysis" Sounds, the ones that track various parameters of audio input, and converting those to MIDI output events?

Tracking Amplitude Envelope

For example, play the Sound called *track the pulse*, and listen to the effect of different settings for `!Thresh` in the virtual control surface. (This works best with a staccato or percussive patch setting on your external synthesizer).

Double-click *track the pulse* to see how it works:



Look at the parameter settings for *trigger MIDI out when thresh exceeded*. Notice that **Gate** is set so that it triggers the MIDI output event each time a threshold is exceeded (we will take a close look at the *Threshold* Sound in a moment), and that **Frequency** is set to:

```
(( [Threshold2] L nextRandom * 6) of: #(0 2 4 7 9 12)) nn + !Pitch
```

In other words, each time the threshold is exceeded, it picks a new random index into an array of intervals and adds that number of half steps to the current pitch from the keyboard.

Now let's take a closer look at *Threshold2* (double-click on it to show its parameters). The output of this Sound is zero except when its input exceeds the **Threshold** at which point the output becomes one. Hysteresis is like inertia or a tendency for the Sound to prefer to stay in its current state of either one or zero. To switch from zero to one, the input actually has to exceed the **Threshold** *plus* the **Hysteresis**; once it is at one, it has to dip below the **Threshold** *minus* the **Hysteresis** before it gives up and goes back to zero. This protects the Sound against clicking on and off rapidly if the **Input** is wavering just around the **Threshold**.

The input to the *Threshold* is an *AmplitudeFollower* on a *GenericSource* which, in its default setting, is playing back a drum loop from a sample CD called *Sample Material* from Sounds Good.[‡] Notice that the *AmplitudeEnvelope* Sound has a parameter called **TimeConstant** which we have set to $!Tc * 0.1 s$, so that it will be one tenth of its setting in the virtual control surface. This is a kind of responsiveness factor or averaging time. If it is too short, the output will react to every minute change in the input and you will just hear the original waveform; on the other hand, if it is too long, then you will lose some of the fast transients in the input that should be reflected in the amplitude envelope.

Notice that the *GenericSource* is also fed through a *DelayWithFeedback* and into a *Mixer* with the output of the synthesizer and the *MIDIOutputEvent*. If you look at the delay parameters, you can verify that the *GenericSource* is delayed by the same amount of time as the **TimeConstant** in the *AmplitudeEnvelope*, because an amplitude envelope follower is, by definition, a little "sluggish", because it is ignoring instantaneous changes in amplitude in the waveform in favor of getting an overall picture of how the amplitude is changing over time. Delaying the original audio signal by about the same amount as the delay introduced by the envelope follower makes the audio line up more exactly with the MIDI events that are being triggered by the original's amplitude envelope.

[‡] Special thanks to Thomas Tibert for permission to include some of the samples from his Sounds Good collection. Thomas (a fellow Kyma user) has an interesting ear and has gathered together an intriguing selection of sounds on several CDs including samples of instruments from all over the world, hilarious outtakes from old documentaries, analog synths, and excellent performances and recordings of more traditional instrument tones and loops. See his web site for more information <http://www.ivo.se/sounds.good/>.

Note that if you decide to switch the *GenericSource* from reading a sample to reading the live input, remember that you may have your external synthesizer plugged into the Capybara's audio inputs. If so, you will have to unplug at least one of the inputs and replace it with the output of the microphone pre-amplifier. You should also edit *external MIDI device* and set the right or left check boxes so you will hear only the external synthesizer and not the microphone input.

Tracking Frequency

Find a sustaining, less percussive patch on your synthesizer and try playing *pch to MIDI* ($> 3F$). It starts out by controlling a sine wave oscillator with the fundamental frequency of the input. To listen to the external MIDI synthesizer, bring the !OscilOrMIDI fader all the way down to zero. This example is designed for tracking inputs whose fundamental pitch is third octave F or higher. Next try *Pch to MIDI* ($> 2F$). This is set up to track inputs whose fundamental pitch is 2 F or higher.

In the virtual control surface, you can adjust !High and !Low to the highest and lowest MIDI note numbers you expect in the input, so you can modify either of these Sounds to track the frequency of different samples or of the live input.

The technique behind these two Sounds is likely to change by the time of the next release, so it does not make sense to describe it in too much detail now. In the meantime, though, these may be useful Sounds if you ever have need to track the pitch of live inputs or recorded samples.

Sampling

Kyma is not a “sampler” *per se*, but it does provide several ways for you to use samples and digital audio tracks as raw material for further processing and modification.

Sampling to Disk

If you haven’t already read *Disk Recording, Playback* on page 112, you should read through it at this point, because it details several ways in which you can record your own samples to disk. It also covers some techniques for playing back recordings from the disk, including how you can trigger disk playbacks with MIDI events.

Playing Back Samples from Disk

Suppose you have a large number of sound effects and you need a quick way to synchronize them to picture. Or suppose that you have a piece for live performers and “tape”, but you would like one of the performers to have control over when to trigger different sections of the pre-recorded sounds so the performers are not slaved to the tape. The *KeyMappedMultiSample* could be used in either of these situations to make it possible for you to trigger long disk tracks or a large number of smaller disk recordings directly from your MIDI keyboard.

For example, try playing *sound fx from disk*, and triggering the animal sounds from a MIDI keyboard. Open it up to see what it is — a *MIDIVoice* on a *KeyMappedMultiSample*. The *KeyMappedMultiSample* takes the samples within a folder and maps them to MIDI key numbers or ranges of key numbers.

Notice that in this example, the **FirstSample** is *flamingos* — meaning that all samples in the same folder as *flamingos* are to be mapped to MIDI key numbers. Notice also that **FromDisk** is checked, meaning that these samples will be triggered directly from disk and *not* read into the Capybara RAM first.

The list of options under the **Mapping** are the different policies you can choose for mapping samples to key numbers. In this case, the mapping policy is **OnePerHalfStep**, meaning that the first sample is mapped to key number 1, the next sample is mapped to key number 2, and so on. The pattern repeats if there are fewer samples than there are MIDI keys.

Finally, notice that the **NoTransposition** box is also checked. This lets you use different keys to trigger different recordings without changing the pitch of those recordings.

Click on the disk button next to the **FirstSample** field and find another folder containing samples files (for example, the *Water* folder in the *Samples* folder of the *More wavetables* folder). Play *sound fx from disk* again, and trigger this different set of samples from your MIDI keyboard.

For an example of a different mapping policy, take a look at *trb from disk*. This one maps samples[§] to ranges of key numbers based upon the base pitch stored in the header of an AIFF sample. **Compile, load, start trb from disk** and try playing some idiomatically trombone-like passages on the MIDI keyboard. You can hear where it switches over from one sample to the next.

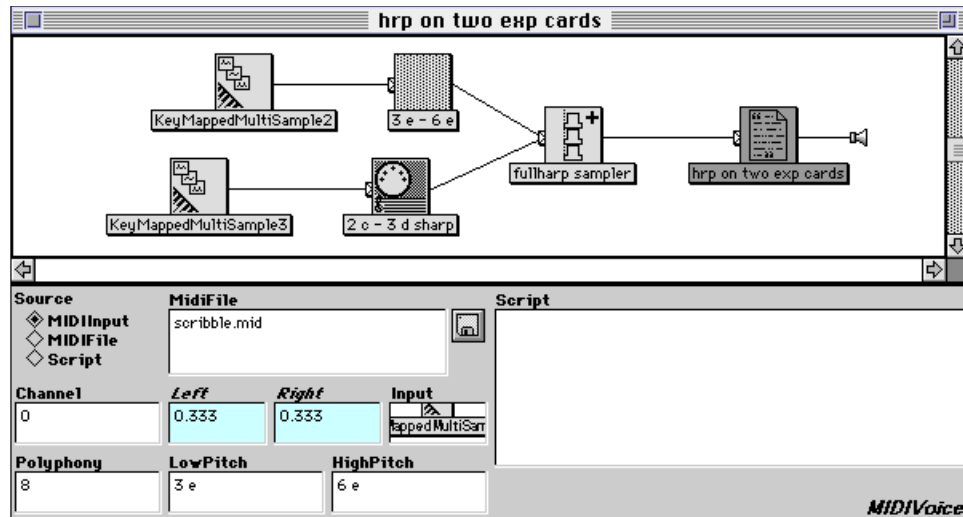
Playing samples directly from disk is especially useful when any one of your samples is too long to fit into the sample RAM of one Capybara card (about 20 seconds at 44.1 khz if you have 3 megabytes per card, and about 90 seconds if you have 12 megabytes per card), and when you do not require more than four-voice polyphony (the current limit on simultaneous, random-access disk tracks).

[§] These samples were contributed by fellow Kyma user Lippold Haken, who is working with trombonist and electrical engineering student Patrick Wolf to do spectral analyses of trombone and cello tones using Lemur. These analyses are then resynthesized in real time using Kyma and controlled and performed using the Continuum — a multidimensional controller that Lippold is developing. Patrick Wolf played the trombone and the recordings were made at Pogo Records in Champaign, Illinois.

Playing Back Samples from the Capybara RAM

If you read the samples from Capybara RAM, rather than from disk, you can get greater polyphony at the expense of a slightly longer wait during the compile step while Kyma loads the samples from disk into the Capybara RAM. For an instrumental-sounding example, try out *20-harp subset in RAM*.

If there are too many samples to fit into the RAM of one expansion card, you can split the samples up into separate folders and assign a different *KeyMappedMultisample* to each folder. For example, open the Sound called *hrp on two exp cards*:



Feeding into the *Mixer* called *fullharp sampler* are two *MIDIvoices*: *3 e - 6 e* and *2 c - 3 d sharp*. Double-click *3 e - 6 e* and notice how **LowPitch** and **HighPitch** are set in the parameter fields. This restricts the range of MIDI key numbers that can trigger the *KeyMappedMultisample*. The *KeyMappedMultisample* has *celthrp 3 e* as its **FirstSample**.

Compare this to the lower branch of the signal flow graph. The lower *MIDIvoice* has a different range of key numbers and a different **FirstSample** (from a different folder) for its *KeyMappedMultisample*.

Choosing Samples other than by MIDI Key Number

velocity violin is an example of a *MultiSample*. The *MultiSample* takes a list of samples file names and an **Index**. The index selects which of the samples should be read the next time the *MultiSample* is gated. This allows you to select samples using something other than MIDI key numbers. This example uses

```
!KeyVelocity * 7
```

to index into the array of seven samples files.

But why stop with MIDI events? Double-click on *live input selects samples* and look at the parameter settings of *MultiSample9*. This *MultiSample* is triggered when the input from the *GenericSource* exceeds an amplitude threshold. Its **Frequency** is derived from a frequency tracker on the *GenericSource*, and it is scaled by an *AmplitudeFollower* on the *GenericSource*. The **Index** selecting which sample to trigger is determined by the expression:

```
( [freq] L * SignalProcessor sampleRate * 0.5 hz) nn removeUnits mod: 5
```

In other words, it is the frequency of the input in hertz, converted into units of half steps (nn stands for note number). The units of half steps are removed, because this is just an index, not a frequency. Then the number is taken modulo 5,[†] so that each half step should trigger a different sample up until the 5th half step when it starts to repeat the pattern.

[†] Just as a refresher, modulo arithmetic is a kind of circular arithmetic similar to arithmetic you perform every day when you are figuring out time. Time on a circular clock is usually numbered modulo 12. For example, if it is 9 o'clock and you want to know what the time will be in two hours, you can think (9 + 2 = 11). At 11 o'clock if you were asked what the time would be in two hours, you would think (11 + 2 = 13) if you were in the military, but as a

Sampling On-the-fly

You can record live sounds in to the Capybara RAM with one Sound and play them back with another. For example, play the Sound called *live scrambling*. When the *GenericSource* asks for the source, press **Enter** to use the default source (a recording stored on the disk). This Sound writes the recording into Capybara RAM and then reads out of Capybara RAM using four Samples with random loop starts. Try different settings for `!Length` — the length of the *Sample* loops.

Now play *live scrambling* again, but this time, set the *GenericSource* to use the live input and try singing, speaking, whistling, and making other mouth noises into the microphone. Whenever you exceed the `!Threshold` amplitude, six seconds of the microphone input is captured in RAM. You can see how long you have to record by watching the `!TimeLeft` fader. Then if you remain quiet enough not to trigger another recording, the *Samples* will continue to loop on whatever was captured the first time. Each time you exceed the threshold, it will record over the previous six second recording. Play with this for a while, trying different length settings and different kinds of vocal noises.

Then double-click *live scrambling* to see how it is put together: After the *Annotation* and *Preset* (the *sine qua non* of any well-documented Kyma example!) there is a *Mixer* called *record live input & chop*. True to its name, one branch of this *Mixer* records the input into RAM, and the other branch of the *Mixer* reads out of that same RAM with random loop points.

Double-click on *record when threshold exceeded* to see its parameters. Its **RecordingName** is set to `recording`, its **CaptureDuration** is `6 s`, and it is triggered by a Sound called *TimeLeftAndTrigger* (which we will examine more closely in a moment).

Compare that to the parameters of one of the four *Samples* called *ReadMem* that feed into the *StereoMix4* called *4 chopped samples*. This is a *Sample* that reads out of the same memory that the *MemoryWriter* writes into. Its **Sample** is set to `recording` (the same name as the **RecordingName** parameter in the *MemoryWriter*). Whenever you want to read out of the RAM that is being written into by a different Sound, you must check the **FromMemoryWriter** box in the Sound that is reading the memory; that way, Kyma knows not to bother looking for that sample or wavetable on disk but to look for it in another part of the same Sound. Notice also that the **LoopStart** of the sample comes from *Noise* and that the **LoopEnd** comes from that same *Noise* generator plus one tenth the value of `!Length`. Notice that we had to take the absolute value of the *Noise* because the output of *Noise* is in the range of $(-1, 1)$ and **LoopStart** expects values in the range of $(0, 1)$.

Now take a look at the Sound called *TimeLeft & Trigger*, the trigger for the *MemoryWriter*. This is a *SoundToEvent* module — a Sound that generates Event Values. The Event Values generated by a *SoundToEvent* look, to all the other Sounds, the same as if they had come in from the MIDI input. In this particular instance, the value to be generated is given by the expression:

```
| inputTrigger outputTriggerDuration |

"The trigger to start our trigger and how long it should be on."
inputTrigger := [env follower] L threshold: (!Threshold * 0.1).
outputTriggerDuration := 6 s.

1 - (inputTrigger * (1 - !TimeLeft asLogicValue)
    ramp: outputTriggerDuration)
```

This is a somewhat twisted way of saying that the *MemoryWriter* should be triggered when the amplitude envelope of the *GenericSource* exceeds some threshold, and that it should record for six seconds without allowing another triggering during that time.

By the way, this is an advanced example of how one can combine Event Values and expressions, so if it doesn't make total sense to you immediately, don't feel discouraged at this early stage. Just come back to look at it again later after you have had more experience with Event Values and Smalltalk programming. If you *do* understand it at this early stage then call us up and tell us so, because we are impressed!

civilian you would probably continue with $(13 \bmod 12 = 1)$ before answering.

OK, now for a verbal explanation of what this expression does. A six second ramp is triggered whenever *both* `inputTrigger` and `(1 - !TimeLeft asLogicValue)` are greater than zero. This only happens when the amplitude envelope exceeds the threshold *and* the ramp has not been triggered within the last six seconds. The actual output (and the value of `!TimeLeft` which gets fed back into this expression) is one minus the ramp function. So `!TimeLeft` goes from 1 down to 0 over the course of six seconds. And it cannot be triggered again until two conditions are met: one condition is that the ramp has had a chance to get to 0, and the other is that the amplitude of the input exceeds the threshold amplitude.

Scripts

See the tutorial called *Algorithmic Sound Construction: Your Computer is your Slave* on page 194.

Shepard's Tone

Compile, load, start (Ctrl+Space Bar) the Sound called *trick keyboard*. Play an ascending chromatic scale on the MIDI keyboard. Then play some arpeggios. Most people hear the chromatic scale as ascending in pitch, but the arpeggios are ambiguous; they seem to be going up in pitch sometimes, down in pitch at other times. Now try playing the first five notes of a C major scale: C D E F G. So far so good. Now alternate between playing C and G as an ascending perfect fifth. Despite the fact that it sounds like each step of the 5 note scale was getting higher in pitch, when you play the ascending perfect fifth, the C and the G sound as if they could be the *same* pitch. Now invert the interval, playing a descending perfect fourth from C down to G. Sounds as if the pitch is going *up* instead of down.

Named for perceptual psychologist Roger Shepard, these tones exhibit a property called “circular pitch”, that is, when used in a scale or glissando they can be perceived both as moving and remaining stationary in pitch, something like the optical illusion of the spirals on a rotating barber pole.

The effect is created by first synthesizing a complex tone consisting only of partials that are equally spaced in pitch (*i.e.* log frequency). Most commonly, the components are spaced one octave apart, but any equally spaced interval will work.

The amplitude of each component in the complex tone is determined by a fixed spectral envelope. The spectral envelope associates an amplitude with each frequency. As the components of the complex tone change in pitch, they “slide under” this fixed, unchanging spectral envelope, changing in amplitude as they change in pitch. The envelope should be a raised cosine or Hanning window shape in log frequency (pitch) space.

Continuous

Play *continous shepard's* and experiment with different intervals and glide speeds, listening to the effect. Then, double-click *continous shepard's* to see how it is constructed. Starting with the rightmost module, double click on each one in order to examine its parameters.

The rightmost Sound is a *Preset*, and it is there to preserve the last settings of the Event Values !Glide, !Int, and !Volume.

The next module, a *MIDIMapper*, is used to set the range of !Glide and !Int to more interesting ranges of values for glide speed and interval in half steps (ranging from 2 half steps as a minimum to 24 half steps or two octaves at the maximum):

```
!Glide is: (`Glide min: -0.01 max: 0.01).  
!Int is: (`Int min: 2 max: 24)
```

Feeding into the *MIDIMapper* is the source of all the sound: an *OscillatorBank*. You can see that there are 256 oscillators, and that 56 of them are scheduled on each card (the *BankSize*).

How are all these 256 oscillators controlled? The amplitude and frequency envelopes for the *OscillatorBank* come from the next module to the left, a *SyntheticSpectrumFromSounds*. A *SyntheticSpectrumFromSounds*, like all spectral source Sounds, outputs a stream of linear amplitude values on the left channel and a stream of frequency values on the right channel. In this particular case, *LogScale* in the *SyntheticSpectrumFromSounds* has been checked, meaning that the frequency values output to the right channel are in log frequency, rather than linear frequency, space. Each stream, in this case, is generated by a Sound: one Sound for the amplitudes, and one for the pitches. Let's look at the Sound that generates the pitches first.

Skip over to the leftmost Sound, the one called *freq ramp*. This is a Sample with looping that reads a ramp function over and over. Its **Frequency** is:

```
256 samp inverse + (!Glide/!Int) hz + 0.002036 hz
```

Why the 256 samp inverse? Recall that the *OscillatorBank* and the *SyntheticSpectrumFromSounds* both said there were 256 partials in the spectrum. The *OscillatorBank* expects to receive the spectrum as a stream of partials, one per sample, in order: 1, 2, ... 256, after which it expects to receive the next frame, starting over again with partial number 1. So a spectrum source has a repetition period of 256 samples;

every 256 samples, it outputs the spectrum for the next frame. We are trying to make a *synthetic* set of pitches for the spectrum. Since we know they should be evenly spaced in log frequency, we have set the *SyntheticSpectrumFromSounds* to **LogScale**, and we are using a ramp wave (*i.e.* a linear function from zero to one) that repeats every 256 samples to specify the pitches — assuming that the first partial is the lowest, then the next partial, and so on linearly until reaching the highest pitch when the ramp reaches its highest value. Why did we use a *Sample* with loop rather than an *Oscillator*? Because the *Sample*, unlike the *Oscillator*, does not interpolate between the last value before the loop and the beginning of the loop. For waveform generation, the *Oscillator* would be better, but for this specific application, we want something that drops immediately from one back to zero, without interpolation.

OK, so far, with a 256 sample period of repetition, we have a static spectrum that does not change from frame to frame and that gives us components whose frequencies are evenly spaced in pitch space.

What is the effect of increasing or decreasing the frequency of the Ramp wave?

Increasing the frequency of the ramp wave effectively increases its slope. So each corresponding point along the line is a little larger than it would have been at the original repetition rate. The effect is that the pitches of all the components go up.

Decreasing the repetition rate gives the ramp a shallower slope. So corresponding points are smaller than they were in the original ramp wave, and the pitches of all components are correspondingly lower.

So far, so good. But we don't have just a single ramp wave, we have a *repeating* ramp wave controlling the pitches of the components. A change in the repetition rate of the ramp means that it no longer lines up with the 256 sample-long frames. If we increase the repetition rate, the ramp will wrap around to zero before the end of each frame. In a sense, the repeating ramp is "rolling" with respect to the frame rate. Think of two identical tapes being played at very slightly different rates; they slowly move out of synchronization until they are so far out of synchronization that they line back up again. Or think of two sine waves that are 2 or 3 hertz apart in frequency; they gradually move out of phase with one another until they are fully 180 degrees out of phase at which point they line back up (causing a beating effect as they reinforce or cancel each other during the "roll").

It turns out that this "roll" is exactly what we want to get the circular pitch effect. Think of one point on the ramp wave. If we increase the slope of the ramp, then on each frame, this point is higher than it was on the last. This is true until it reaches the maximum value. Then on the next frame, it returns to the lowest value again. This is the result we are after: that each component increase in pitch until reaching the highest frequency and then that it wrap around and sneak back in at a subaudio frequency.

The *Gain* called **256** scales the ramp so that its range is $(0, 256)$. The *Attenuator* called *interval density* reduces that range to achieve the actual pitch interval that you specify with **!Int**. The log frequencies are set up so that one is the half sample rate and each decrease of $1/15$ is a decrease of one octave from that maximum frequency. The *Attenuator* turns your interval into a fraction of an octave and uses that to scale the range of the ramp function.

Now for the really cool part! How to specify a spectral envelope? It would be tempting to just take an *Oscillator* on the Hann wavetable and use that as the **Amplitude** input to the *SyntheticSpectrumFromSounds*. But it would be wrong! Actually for a static spectrum, it would be OK. But each amplitude is matched with its corresponding *partial* number, not necessarily with any particular pitch. And we have partials that are changing frequency on each frame. So we need some way to associate specific amplitude values with specific pitches. One way to do this would be to have a table of values that list the amplitude value for each possible frequency, and use the current frequency value to look into the table for the correct amplitude value to use.

This concept — that of a table where you can use one value as an index into the table and output the value found at that index — immediately suggests the *Waveshaper*.

The *Waveshaper* is a nonlinear transfer function that maps values between -1 and 1 to arbitrary values stored in an arbitrary-sized table. So the first task is to get our all-positive frequencies to lie between -1 and 1 . This is accomplished by the *ScaleAndOffset* called **-1 to 1**. It simply multiplies by two and subtracts one.

This scaled value is then used as an index into a Hann function (forced to zero on the first and last entries of the table) by means of the *Waveshaper* called *pch->amp*.

After passing through *overall vol* for controlling the overall amplitude level, it is finally fed in as the amplitude stream for the *SyntheticSpectrumFromSounds*.

Now that you fully understand how this Sound works, play it once more and experiment with different parameter settings.

Discrete

Now *Compile, load, start* the *trick keyboard* example. Play a chromatic scale of one octave on the MIDI keyboard. Then play the first note of the scale followed by the last note of the scale; it sounds identical. Use `!Int` to specify which interval should be the “equivalence” interval. Besides having fun turning your own mind inside out, you can also have some fun showing it to your friends.

John Platt, fellow Kyma user and professor of psychoacoustics at McMaster University, explains the effect as follows:

Each tone is ambiguous because the partials are one octave apart. So, for example, when you play a half step from 4 c to 4 c sharp, your brain has a choice of interpreting it as either a half step up, 13 half steps up, or 11 half steps down. Prior experience tells you that the smallest change is probably the correct interpretation so you hear it as one half step up. For larger intervals, though, things become more ambiguous. For example if you play a tritone (which is one half of an octave), your brain will sometimes interpret it as going up and sometimes hear it as going down. If you try playing various different tritone intervals on the trick keyboard, you can sometimes hear them as ascending and sometimes as descending; sometimes you can even hear them flip in direction in your mind in the same way that a 2 dimensional drawing of a cube can flip from one orientation to another.

Rigor

By the way, while these examples are useful for musical purposes and for demonstrating the auditory illusion of circular pitch, Professor Platt has made some refinements to give more precise control over these Sounds for psychoacoustic experimentation. If you are contemplating using these Sounds for research purposes, please contact Symbolic Sound so we can put you in touch with Professor Platt.

Stereo Placement

If a sound source is to your right or left, it is going to reach one of your ears before it reaches the other and by the time it reaches the second ear it will have lost energy — both because it has traveled further by then and because your head is probably blocking some of the energy. The bigger your head is, the more extreme the differences will be between the signal that reaches your left ear and the signal reaching your right ear.

Amplitude vs Energy

Play the Sound *Pan amplitude* and use the on-screen fader or a MIDI fader to control `!Pan`.[§] Compare that to *energy pan1* and *energy pan2*. Do you hear a difference in the loudness or position as you pan across the speakers?

That Space between your Ears

Now try *Pan phase only*. The only difference between the left and right channels is the relative delay. There is no amplitude difference between the left and right outputs. Double-click *Pan phase only* to take a look at what modules were used to construct it.

You can see that a *GenericSource* is being fed into two different delays. Then the delays are fed into a *ChannelJoiner*: one into the left input of the *ChannelJoiner*, the other into the right input. A *ChannelJoiner* creates a stereo pair out of two inputs: the left channel of the one is routed entirely to the left side of the stereo pair, the right channel of the other is routed entirely to the right side of the stereo pair. (Since the right and left channels of the delay output are the same, it is immaterial which channel is read by the *ChannelJoiner* in this particular instance).

Double-click on the Sound called *RIGHT*. Its **Delay** parameter specifies the maximum delay; **DelayScale** determines how much of that maximum delay time you want at any given moment. The strange message `normCos` stands for normalized cosine. Since so many expressions involving cosine include a factor of π , that is automatically added when you ask for the `normCos`. In other words

$$x \text{ normCos} = (\text{Float } \pi * x) \cos$$

So the expression

$$(0.5 * !\text{Pan}) \text{ normCos}$$

yields the following delays for the right and left channels relative to the value of `!Pan`:

<code>!Pan</code>	<code>(0.5 * !Pan) normCos</code>	Right Delay	<code>(0.5 * !Pan) normSin</code>	Left Delay
0	1	800 usec	0	0 usec
0.5	<code>0.5 sqrt</code>	565.6 usec	<code>0.5 sqrt</code>	565.6 usec
1	0	0 usec	1	800 usec

The only thing that you really hear is the difference between the two delays, so the effective interaural difference is 800 microseconds at the two extremes and zero when the sound source is directly in front of you (since it takes the same amount of time to get to both ears).

Do a side-by-side (*i.e.* one-after-another) comparison of *pan (phase & energy)* against *energy pan1*. What, if any, effect does adding phase delays add to the pan?

By the way, if you are interested in where the numbers come from, you can find papers on “interaural time differences” in the *Journal of the Acoustical Society of America* that explain how they come up with these time differences based on the diameter of a typical head, the speed of sound, and the assumption that the sound source is far enough away that you can consider lines drawn from the source to the two ears to be parallel.

Doppler

Imagine a sound source moving towards you. As it gets closer and closer, the delay between when the sound is produced and when the sound gets to you is getting shorter and shorter. At the moment the sound source passes directly in front of you, the delay is zero. Then as it recedes from you, the delay grows longer and longer.

You can model this phenomenon using a variable delay line with an inverted-triangle-shaped function controlling the length of the delay. In other words, the delay gets shorter and shorter, reaches zero at the apex of the triangle, and then grows longer and longer again. Try playing *moving source* to hear this effect.

Mystery Effect

Finally, try out the Sound called *phase Flipper*. You may be able to use this effect to bring one sound out from the background.

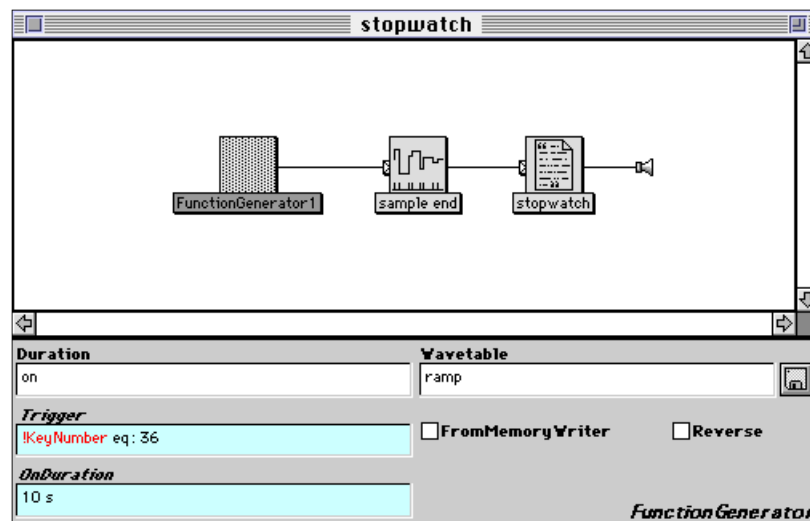
Tap Tempo

This file contains examples of Sounds that can, in various ways, match a tempo that you establish by tapping on the MIDI keyboard (or other MIDI controllers). You can use these Sounds as examples of how to set up your own Sounds for live performances with more than one performer (or for that matter, in *any* situation where you might have a need to measure the time it takes for someone to tap one key or switch followed by another).

For example, select and play the Sound called *tap tempo sample*. As per the instructions that appear on the virtual control surface, tap the 2 C on your MIDI keyboard followed by the 2 D. The sample is triggered at the same rate at which you play the two keys on the keyboard. Listen to the tempo, and try playing something twice as fast or twice as slow.

The Heart of the Clock

Double-click the Sound called *stop watch* to see how you can use Kyma Sounds to measure time.



The rightmost Sound is an *Annotation*; double-click it so you can read the explanation in the **Text** parameter:

Play 2 c to start the clock, and 2 d to stop it. Its value is one tenth of the number of seconds between start and stop. 2 c triggers a FunctionGenerator to start a ramp that takes 10 s. 2 d triggers a TriggeredSampleAndHold on that ramp.

In other words, suppose you have a friend who can run at an unerringly constant speed of 10 kilometers per hour. Imagine that you look at your watch, and at exactly 11:30 am, you yell, “Start!” Your friend starts running, and you follow alongside driving a car (not because you are lazy, just because the car has an odometer). After a while you yell “Stop,” slam on the brakes and read from the odometer that you have traveled 5 km. If your friend can really maintain a constant rate of 10 km per hour, then you know exactly how much time has passed without even looking at your watch. Since 5 km is one half of 10 km, it must have taken half an hour to run that far. So you let your friend get in the car and you go out to have lunch together.

In this Sound, the *FunctionGenerator* is the person running. Double-click on it. The **Wavetable** is Ramp (a straight line from zero up to one) and the **OnDuration** is 10 s. The way to trigger this Sound is not by yelling “Start!” but by pressing MIDI key number 36, as you can see in the **Trigger** field:

!KeyNumber eq: 36

The way to yell “Stop!” and measure how far the ramp function got is seen in the next Sound to the right, the *TriggeredSampleAndHold*. Its **Trigger**

!KeyNumber eq: 38

becomes one when you press MIDI key number 38. When the trigger becomes one, the *TriggeredSampleAndHold* measures the value of its input and holds onto it until the next time it is triggered (even if the **Input** continues to change).

Try playing *display duration of tap*. This is similar to *stopwatch* except that it converts the output of the duration measurement into an Event Value. This Event Value then shows up in the virtual control surface as a fader, and you can read its value in the numeric display at the top of the fader. Look at your watch and experiment with tapping out different durations in seconds (up to a maximum duration of 10 seconds).

Watching the Clock

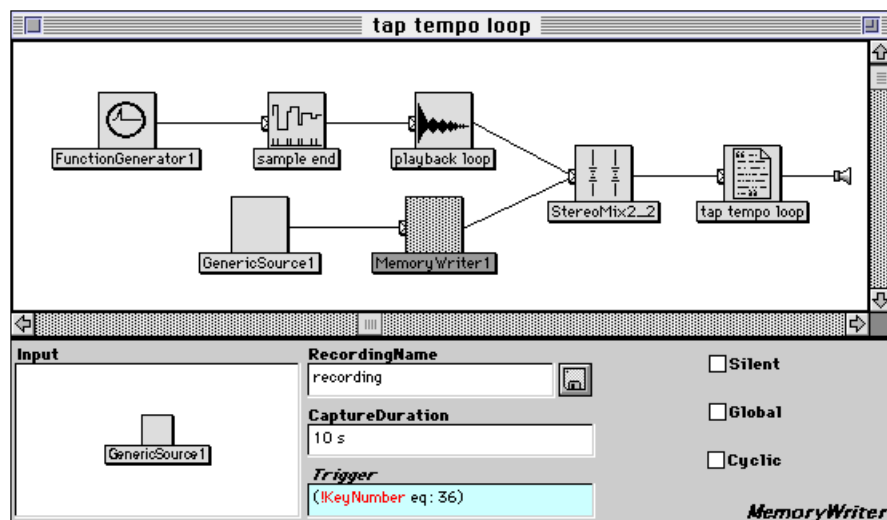
You can use this basic clock to control parameters having to do with the tempo, the rate, the duration, or other aspects of the Sound giving a sense of time or tempo.

Sequencer Rate and Delay Time

In *tap sequencer & delay* for example, the tap controls the rate of an *AnalogSequencer*. One third of the tap time is also used as the delay time of a *DelayWithFeedback*, giving a sense of triplets in the delay time.

Loop Points

Look at *tap tempo loop*. It is a mixer of a *MemoryWriter* and a memory reader (in the form of a *Sample*).



When you tap the 2 C, not only does the clock start ticking, but the *MemoryWriter* starts recording into the sample RAM. When you tap 2 D, not only is the clock stopped and measured by the *TriggeredSampleAndHold*, the 2 D also triggers the *Sample* to begin playing.

The length of the loop is the same as the duration between taps. Why? Because the **CaptureDuration** of the *MemoryWriter* is 10 seconds — exactly the same as the **OnDuration** of the ramp *FunctionGenerator*. So the proportion of the tap duration to the total duration of the recording that is being read by the *Sample*.

Time Scaling

This section is the dual of *Frequency Scaling* on page 124; you cannot change the duration of a disk track or sample without also changing its frequency and vice versa. For example, try playing *disk file rate chipmunks & monsters* and *twice as long = octave lower*. Not only does changing the rate also change the frequency, it also seems to shift the formant regions up or down, giving the impression that a larger or smaller source produced the sound (the chipmunk or monster effect).

Granular Time Stretching

Many of the same modeling techniques used in the *Frequency Scaling* section can also be used to scale the duration of a recording without affecting its frequency. For example, if we assume that the sound was produced by a train of impulses hitting a filter, we can use granular time-stretching (see for example *time-stretched virtue* and *time-stretched speech*) which slows down the rate without affecting either the fundamental frequency or the formant frequencies.

In *sample cloud stretch dur*, two *SampleClouds* are used to granulate a speech sample. Experiment with different values for !Rate, !TJitter, and !GrainDur in the virtual control surface, and then open up the Sound to see how it works.

Try playing *granular time/freq antonio*, triggering a repetition of the poem using MIDI key down. This uses several *TimeFrequencyScaler* Sounds on the same sample[‡] but once a second randomly chooses a new rate and a new frequency scale. Each time you trigger it from the keyboard it will choose different rates and different frequencies. *live random granular time/freq* is the same idea, except that you can choose the live input as the *GenericSource* and granulate the live input.

Controlling the TimeIndex of a Resynthesis

If you analyze the spectrum of a sample, you can resynthesize it using an *OscillatorBank* or a *GAOscillators*. Unlike sample playback, an oscillator gives you independent control over duration and frequency.

The **TimeIndex** of a *GAOscillators*, *SumOfSines*, or *REResonator* resynthesis is normally a linear ramp function that starts at -1 at the beginning, grows to 0 by the middle, and reaches a value of 1 by the end.

In *rate control on GA*, the **TimeIndex OnDuration** is set to:

$$2.39415 \text{ s} + ((1 - !\text{Rate}) * 10)$$

If !Rate is 1, then the time index has its normal duration and the GA reads through its envelopes at the normal rate. However, if !Rate is less than 1, the **OnDuration** is larger, and the time index takes longer to get through the amplitude envelopes, thus increasing the duration. (Try playing this Sound and adjusting !Rate in the virtual control surface). This technique is not without its artifacts; if you try *dur scaled FL GA*, you can hear features like vibrato and tremolo slow down when you slow down the rate.

The same technique can be applied to the time index that controls the amplitude and frequency envelope rate of a full-blown additive resynthesis using the *SumOfSines* or a spectral source and an *OscillatorBank*. For example, play *Kurdish dur scale*[§] and adjust the !Rate from the virtual control surface.

tantalizing celthrpGliss uses

$$\boxed{\text{exp}} \text{ L} * 100 \text{ s} + 2 \text{ s}$$

as its **OnDuration**. Since *exp* has an exponential function shape, the value of **OnDuration** gets larger as time goes on. The result is a resynthesis of a harp glissando that never seems to reach the top.

In *bass-keydown slows time*,[‡] the **OnDuration** is set to

$$5 \text{ s} + (100 * !\text{KeyDown}) \text{ s}$$

[‡] This is a fragment of a poem written and performed by composer and fellow Kyma user Antonio Barata.

[§] Analysis of sample from Thomas Tibert's *New World Order* sample collection.

[‡] Analysis of sample from Thomas Tibert's *Methods of Mayham* sample collection.

so that when a MIDI key is down, the duration jumps from 5 seconds to 105 seconds. Play this Sound and try to hit a MIDI key just at the start of one of the notes.

Play and then take a look at *warped time index*. Instead of using a *FunctionGenerator* to generate a full ramp from -1 to 1, this example uses a *WarpedTimeIndex* to control the rate of the envelopes. The *WarpedTimeIndex* takes two arrays, each containing the same number of time points: the ideal time points and the actual time points. It creates a multi-segment envelope with different slopes corresponding to different rates of time: steeper slopes go through the envelopes faster, and shallower slopes go through the envelopes more slowly. The *WarpedTimeIndex* slows down or speeds up the resynthesis such that the actual times occur at the specified ideal times.

Along these same lines, *multiseg env on time index* uses a *MultiSegmentEnvelope* generator to generate the time index. The total duration is broken down into ten segments, and the duration of each segment is hot, so you can control how long each segment should last using the faders in the virtual control surface or a MIDI fader box.

Take a look at *live freeze frame on key down(4)*. Look at the parameters of the Sound called *capture frame and loop left*. This is a *DelayWithFeedback* with the **Feedback** set to `!KeyDown` and the **Scale** set to `1 - !KeyDown`. **Scale** is an attenuator on the **Input**, so this means that when no key is being held down, the **Input** amplitude will be at the maximum of 1, and when a key is held down, the **Input** amplitude will be cut to zero. Notice, however, that whenever the **Input** amplitude is set to zero, the **Feedback** amplitude will be at its maximum (and *vice versa*); they are the opposites of one another. So when a key is held down, there is no new input, and the delay line is recirculating with no loss of amplitude on each feedback loop. In a way, the *DelayWithFeedback* is acting as an oscillator whose wavetable is whatever was captured in the delay line, and whose period is 256 samples — the size of one frame when the analysis frequency is **2 F** (see the parameter settings in *nonharmonic analysis*). The length of the delay can be any multiple of the frame length (for example 4096 samp would also work). The left and right have to be delayed separately, because *DelayWithFeedback* does a mono mix if it gets a stereo input.

Vocoder & RE Time Scaling

Try the Sound called *vocoder time-stretch*, and then double-click on it to see how it is put together. First, look at the *GenericSource*. Its **Frequency** is set to

```
60 nn hz * !Rate
```

60 nn or middle C is the default frequency given to samples that do not have a single, identifiable pitch. 60 nn (a pitch) is being converted to a frequency (by giving it units of hz). Scaling the frequency of a sample is the same as scaling its duration: a lower frequency means that you read through the sample at a slower rate, so the duration is longer.

As you know, however, slowing down a sample also lowers the frequencies of its formants. So this Sound uses a *Vocoder* to compensate for that. Double-click on the *Vocoder* to see its parameters. Notice that **SideFreq** is set to `!Rate`. So whenever `!Rate` is made smaller (making the sample frequency and formant lower in frequency), the analysis filter bank is also shifted lower in frequency. However, the resynthesis filter bank stays at its original frequency (because **InFreq** is set to 1). Thus, when the sound is “resynthesized” by passing noise through the **Input** filters, the formants are at the normal frequencies, even though the sample is being played back more slowly.

Note also, that the **TimeConstant** is also scaled by `!Rate`, because when the sample is slowed down, any changes in it happen more slowly, so the time constant controlling the amplitude follower can also be longer.

Try playing *22-vocoder rate & freq scale*, and experiment with the fader settings in the virtual control surface. This is similar to the previous example except that, instead of using *Noise* as the **Input**, this example uses an *Oscillator* on a buzz waveform with 64 equal-amplitude harmonics.

electric shaver mouth also uses a buzz-waveform *Oscillator*, but this time as the **Input** to an RE filter. In this example `!Rate` controls the **OnDuration** of the *TimeIndex* controlling the rate at which the RE filter coefficients are updated.

Timing Clock, MTC

When you use Kyma in conjunction with other audio hardware and software, you need some way to synchronize Kyma with the rest of the world.

Audio Sequencers and DAWs

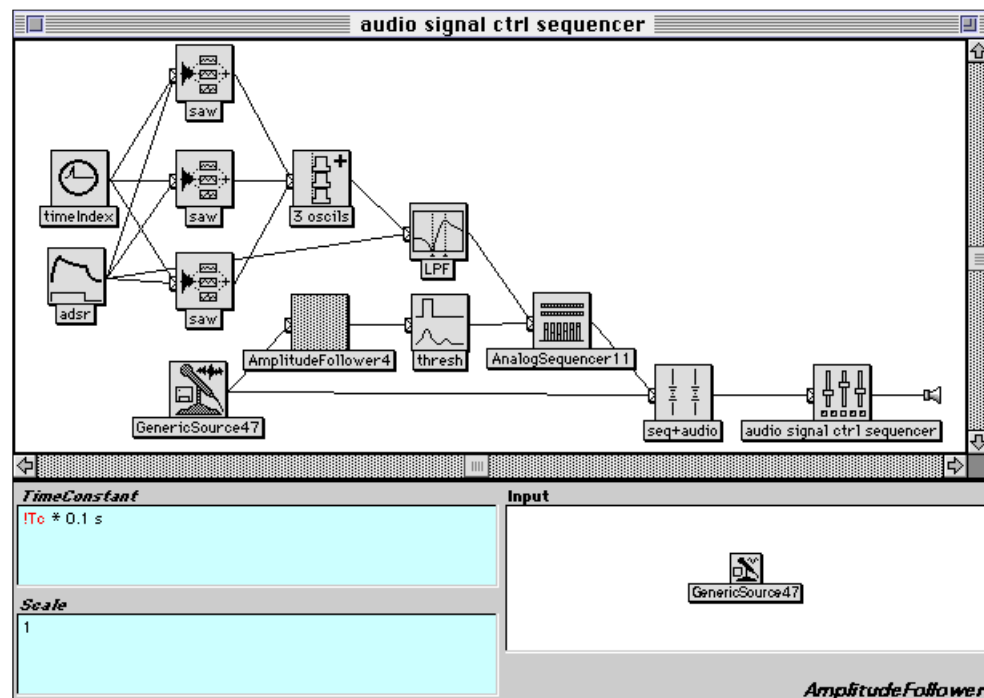
Kyma can generate audio tracks in AIFF, SD I, SD II, WAVE, and IRCAM/MTU formats, making it easy to import audio tracks into Kyma for further processing and to use the audio tracks generated in Kyma with your other software. If you import a Kyma-generated track into your audio sequencer, for example, you can adjust it graphically until it is perfectly synchronized with the other audio or MIDI tracks.

Audio Signals

Using the *ADInput* or *GenericSource*, you can bring live or recorded audio signals into Kyma, and use them to control the timing of Kyma Sounds (which would then, as a result, be synchronized to the audio signal). In general, the idea is to:

1. Get the amplitude envelope of the audio signal (by feeding it into an *AmplitudeFollower*, *PeakDetector*, or *RMSSquared*).
2. Output a trigger whenever the amplitude exceeds a certain threshold or minimum value (by feeding the amplitude envelope into a *Threshold*).
3. Use this to trigger a Kyma Sound (by pasting the *Threshold* into a **Trigger**, **Gate**, or **Step** field).

For example, try playing the Sound called *audio signal ctrl sequencer*, and then open it up to take a look at how it works. Once you have a Sound editor open on the Sound, drag the *GenericSource* further to the right to get a clearer picture of what's going on.



The *GenericSource* is being fed into an *AmplitudeFollower* which extracts an amplitude envelope from the signal. This envelope is then fed into a *Threshold* whose output is 1 when the envelope exceeds the threshold and 0 otherwise. Select *thresh*, and choose **Oscilloscope** from the **Info** menu. Adjust the time constant *!Tc* to 0.1, and *!Thresh* to 0.5. You should see and hear some clicks whenever the threshold is exceeded (actually you will hear a click when it is exceeded and another one when the amplitude drops

below the threshold and the output of the *Threshold* returns to zero). Try increasing `!Thresh` until nothing in the input exceeds it anymore.

Now look at the parameters of the *AnalogSequencer*. Notice that *thresh* has been pasted into the **Step** field. Each time the amplitude envelope on the audio signal exceeds the threshold, the sequencer will read the next set of values in its sequences. Notice that the **Durations** field is set to `0.1`. This means that all notes in the sequence are of equal duration — 0.1 seconds. It also places an upper limit on the speed of the sequence. No matter how fast the **Step** triggers come in, the sequencer will never go faster than one note every 0.1 seconds. This means that you can use the **Durations** field as a kind of mask, protecting the basic rhythm against any spurious triggers that might come from the audio signal.

Try playing *audio signal ctrl sequencer* again, this time adjusting `!Thresh` to different values, and checking/unchecking the boxes in the row along the bottom of the virtual control surface (which set the velocity values to 1 or 0 depending on whether they are checked or unchecked).

This example used the amplitude of the signal as a trigger in Kyma. You could, alternatively, use some other parameter as a trigger; for example, you could use a *FrequencyTracker* in place of the *Amplitude-Follower* and trigger whenever the *frequency* exceeds a threshold.

MIDI Note Events

If you are using a MIDI sequencer, you can synchronize Kyma with your other MIDI devices through MIDI note events. For example, take a look at *KeyDown steps sequencer*. This is similar to *audio signal ctrl sequencer* except that a `!KeyDown` event is used in place of the *Threshold* on the envelope of the audio signal. You could create a timing track on one channel of your sequencer that is never actually heard — just supplies triggers to Kyma — so that when the tempo changes in the sequencer, Kyma will adapt along everything else that is being controlled by the sequencer.

MIDI Timing Clock

If one of your MIDI devices can output a MIDI timing clock signal, you can synchronize your Kyma Sounds to changes in the rate of the MIDI clock through the judicious use of two Event Values in Kyma: `!TimingClock` and `!TimingClockDuration`. `!TimingClock` is a trigger that repeats at a rate of 24 times per beat. `!TimingClockDuration` is the duration in seconds of one `!TimingClock` trigger. In other words, if you multiply `!TimingClockDuration` by 24, you will get the duration in seconds of one beat at the current tempo.

Take a look at *MIDI Timing clock ctrl seq & del* for an example using `!TimingClock` and `!TimingClockDuration`. This Sound synchronizes both the *AnalogSequencer* and the length of a delay to an external MIDI clock.

Look first at the *DelayWithFeedback*. Its maximum **Delay** is set to `1 s` — the same as the duration of a beat in the *AnalogSequencer*. If you change the value of **Delay**, you should also change the **Durations** field of the sequencer so they are equal to each other.

The **DelayScale** is set to

```
!TimingClockDuration * 24
```

This expression is equal to the duration of one beat in terms of seconds. So, for example, if the tempo is 120, the beat is one-half second long, and this number will be `0.5`. **DelayScale** is a multiplier on the value in **Delay**, so the delay will be 0.5 seconds long. If the tempo is 60, then `!TimingClockDuration * 24` will equal 1, and the delay will be one second long.

Now look at the *AnalogSequencer*. First, notice that **Step** is set to the value

```
!TimingClock
```

This means that it is triggered 24 times per beat. But notice that **Durations** is set to

```
1
```

This acts as a mask, meaning that the sequencer cannot be triggered any faster than once per second, even if the triggers are coming in faster than that. But wait! Almost certainly you will want to trigger this at a

faster rate than once per second! Don't worry about that, because the durations are scalable — according to the **Rate**.

Notice that the **Rate** is set to

```
(!TimingClockDuration * 24) inverse
```

What does this imply? We know that rates and durations are the inverses of each other, so if `(!TimingClockDuration * 24)` is the duration of one beat in seconds, then the inverse of that value would be the *frequency* or the number of beats per second. Let's say the tempo of the MIDI clock were 120 bpm. That's two beats per second. That means the **Rate** would be equal to 2, so the duration of each beat will be scaled to one *half* second.

The Sounds in the next row of the file (*MIDI clock ctrl MIDI file rate, MIDI clock ctrl MIDI voice script, etc.*) are other examples of how you can use `!TimingClock` and `!TimingClockDuration` to synchronize various Kyma Sounds to an external MIDI clock.

MIDI Time Code (and SMPTE)

You can also use MIDI Time Code (or SMPTE if it is first translated to MTC) as a trigger in a Kyma Sound. In the trigger or gate field, enter the time when the trigger should occur, followed by the message `triggerAtTimeCode` or `gateOnAtTimeCodeForDuration:` followed by the length of time the gate should stay on.

For example, look at *trigger at time code 00:00:12.00*. This is a *DiskPlayer* whose **Trigger** field is set to

```
12 s triggerAtTimeCode
```

meaning that it will not be heard until Kyma receives the MIDI time code for 12 seconds. If you prefer, you can also specify the time in SMPTE format, for example,

```
00:00:12.00 SMPTE triggerAtTimeCode
```

In other words, 0 hours, 0 minutes, 12 seconds, and 0 frames.

To send a gate with a specific duration (rather than sending a single trigger) use `gateOnAtTimeCodeForDuration:`. For example, look at the **Gate** field of the *ADSR* in the Sound *gate on at 20 s for duration 1 s*. It is set to

```
20 s gateOnAtTimeCodeForDuration: 1 s
```

meaning that the *ADSR* will be triggered when the time 20 s is received and will stay on for 1 second before going into the release of the envelope.

Tracking Live Input

One of the most exciting aspects of real-time signal processing is that you can control synthesis and processing parameters in a very direct way, with your voice or with an instrument. This is an even more direct way of making music or of tweaking sound designs than moving MIDI faders, because you have years of experience in using your voice and/or playing your instrument, you have incredibly refined and almost subconscious control over minute aspects of its timbre and timing, and you don't have to break that concentrated and sophisticated feedback loop to push a button or move a fader (unless of course, the mixing console *is* your instrument in which case you have the same relationship to it as musicians have to their instruments of wood and skin).

This is an exciting area to explore, and it is also one of the most difficult. (Let's face it, if it were easy, it would have been done and perfected already, and you wouldn't be so interested in it anymore, because you have already implicated yourself as an inveterate innovator by having read this far into the Kyma manual in the first place).

The basic idea is that we want to use an audio signal to control a parameter (or parameters) of Kyma Sounds. What is an audio signal? Basically it is a time-varying amplitude. So let's start with a simple case — controlling the amplitude of a Kyma Sound with the amplitude of a live audio signal.

Amplitude

As you recall from having done the *Envelopes* tutorial on page 120, one way to impose an amplitude envelope onto any other Sound is to multiply the envelope and the Sound. Try this right now. Drag a **Product** into a new Sound file window (the **Product** is in the **Arithmetic** category of the prototype strip). Open it up, and replace the **FunctionGenerator** with a **GenericSource** (from the **Sources** category of the prototype strip). Change the **GenericSource** to use the **Live** source (or set it to auto-loop on the sample called **Electron**). Then play the **Product**. Sounds like ring modulation, right? That's because it *is* ring modulation. So we have uncovered a way to do ring modulation, but that is not what we had in mind; we wanted to control the amplitude of the oscillator with the amplitude of the source.

The problem is that when we think of the “amplitude” of a sound, we usually mean the amplitude *envelope*, that is a kind of averaged, slowly changing, overall amplitude, not the instantaneous amplitude that changes on every sample tick.

Look in the prototype strip under **Analysis**. Find **AmplitudeFollower** and drag it onto the line between the **GenericSource** and the **Product**. An **AmplitudeFollower** does two things: it makes all of the amplitudes positive (greater than zero) and it does a kind of averaging over several instantaneous amplitudes so as to smooth out some of the faster, smaller changes in amplitude. The **TimeConstant** parameter of the **AmplitudeFollower** gives you some control over this averaging time: the shorter the **TimeConstant**, the faster the envelope responds to changes in its input, but the less smooth it will be. Select the **AmplitudeFollower** and choose **Full waveform** from the **Info** menu, asking to plot only 3 seconds of the waveform. Then select the **Product** and listen to the amplitude envelope multiplied with the oscillator.

This immediately begs the question, what about controlling the *frequency* of the oscillator as well as its amplitude? How can you tell the frequency of an audio signal when all you have to work with is a stream of instantaneous amplitudes?

Frequency

If you look in a textbook for the definition of frequency, you will most likely be treated to a drawing of a perfect, endless-duration sine wave with vertical lines indicating the start and end of one cycle and a definition having to do with the number of cycles that transpire within one second. Fortunately for our sanity (but unfortunately for pitch tracking algorithms), real-life signals rarely have perfectly repeating waveforms at an unchanging frequency.

One module that can help identify frequency and track how it changes over time is the **FrequencyTracker** (in the **Analysis** category of the prototype strip). It takes a Sound as its input and outputs an estimate of the frequency of that Sound as a number between zero and one (which you can then scale up to the range of DC to one half of the sampling rate if you want to use it in a **Frequency** field).

Like an amplitude envelope, a pitch envelope is a feature that takes the human auditory system some amount of time to integrate and identify. The *FrequencyTracker* uses an algorithm called autocorrelation, which, as its name implies, is a measure of how correlated a signal is with itself at different delay times. Imagine that textbook sine wave again. If you were to delay that sine wave by one cycle and then compare it against itself with no delay, you would have to say that the two signals were extremely well correlated, if not identical. But if you compared them after a delay time of, say, one-tenth of one cycle, they would not be as correlated. The *FrequencyTracker* compares the signal against itself at several different delay times and picks the one delay time at which the original and delayed waveforms are the most similar. It reasons that this delay is probably the period of one cycle of the waveform, so it can then make a guess as to the frequency of the signal.

As you can tell from this description, frequency tracking is not a fool-proof or easy task, so the more clues you can give the *FrequencyTracker*, the better. If you have some idea of the frequency range of the input, you should enter that in the **MinFrequency** and **MaxFrequency** fields of the *FrequencyTracker*. The more you can narrow-in on the range, the better the *FrequencyTracker* will do.

Try playing the Sound called *oscil tracks voice*, and then open it up to see how the *AmplitudeFollower* and the *FrequencyTracker* are used to control an oscillator. Notice that there is a **Gain** on the *AmplitudeFollower* (because it tends to have low-amplitude output), and notice that the **Frequency** field of the *Oscillator* is set to:

```
FreqTrk L * SignalProcessor sampleRate * 0.5 hz
```

This expression scales the (0,1) output of the *FrequencyTracker* to the range of DC to one half of the current sample rate.

Try using your own voice as a source. If the *FrequencyTracker* has trouble tracking your voice, adjust the **MinFrequency** and **MaxFrequency** to more closely match the range of your speaking or singing voice.

In *mouth-controlled filter*, the amplitude envelope of the input controls the center frequency of a low pass filter, and the frequency of the input controls the frequency of a sawtooth oscillator that is fed into the filter. Listen to the default source, and then switch it over to **Live**, so you can try controlling it with your voice. Notice that this Sound uses a *PeakDetector* to follow the amplitude envelope of the source. Double-click the *PeakDetector* to look at its parameters. You can think of the *PeakDetector* as having two time constants: one for reacting to increases in the amplitude of the input, and the other that reacts to decreases in the amplitude. You can create, for example, an amplitude envelope that jumps immediately upward in response to attacks or onsets in the input, but that decays slowly when the input amplitude goes to zero.

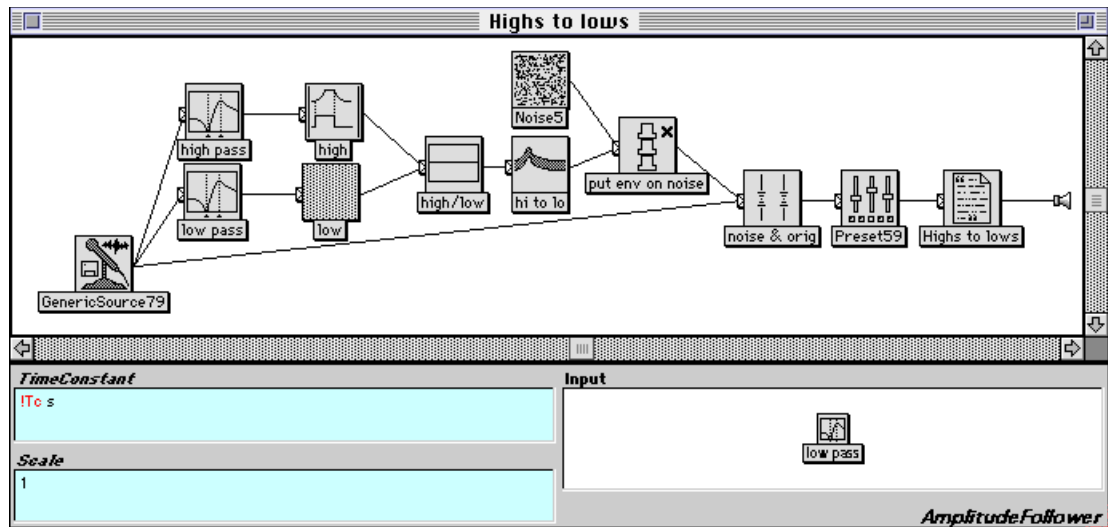
A *PeakDetector* is also used in *amplitude filter, LFO on attack*. Try playing this Sound. It is set up as a drum loop processed by a low pass filter. The amplitude envelope of the drum loop controls the cutoff frequency of the low pass filter. Notice that the attack time of the *PeakDetector* is modulated, in this case, by a low frequency oscillator. You could, alternatively, choose two different sources for this Sound: one as input to the filter and another to control the cutoff frequency of the filter. Look at the parameters of the two *GenericSources* in this Sound. One is set to read the right channel and the other is set to read the left channel of the source, so you could use the two input channels of the Cappybara for two different audio signal inputs to this Sound, listening to one through the filter and using the other signal to control the cutoff of that filter.

Timbre

So far, we have looked at modules for tracking the amplitude and the frequency of the signal. What other parameters does an audio signal have? Some books actually define timbre as those characteristics of a signal that are not pitch or loudness — sort of a “none of the above” definition. A slightly more useful (though still insufficient) definition might be: the spectrum, or the time-varying strengths of upper partials relative to the fundamental. Admittedly this is still vague, but it gives us something to work with.

We know we want to somehow differentiate between the “highs”, the “mid-range” and the “lows” or at least to monitor them independently of one another. So the first thing that springs to mind is filters. We can use filters to separate a single signal into several frequency bands.

Take a look at *Highs to lows* — an example that splits the signal into two parts: frequencies above 6000 hz and frequencies below 4000 hz, using a high-pass and a low-pass filter respectively.



The Sound takes the output of each of these filters and feeds it into an *AmplitudeFollower*. Next, it compares the energy in the high frequencies against the energy in the low frequencies by taking the ratio of the high frequencies envelope to the low frequencies in *high/low*. This ratio is smoothed by a *PeakDetector* and then used as an amplitude envelope on some pink noise. The result is that you hear a little burst of noise anytime there is a phoneme with lots of high frequency energy in the input. It is especially noticeable on the words “so”, “essence”, and “fact”.

We can take this idea of using filters to split the spectrum even further, using a bank of bandpass filters to split the spectrum into 22 to 88 bands using a *Vocoder*. This is similar to amplitude following, but on several, independent frequency bands. So the spectral characteristics of the *SideChain* signal are controlling the filters on the *Input* signal.

As an example, try *stressed out cartoon*. In this example, the spectral characteristics of the speech sample control the filtering on an oscillator input. Since the oscillator has a complex waveform, it has a relatively rich spectrum that can be shaped by the filters, resulting in intelligible speech. At the same time, the *amplitude* envelope of the speech is being used to control the *frequency* of the oscillator (nothing says you can’t cross-map amplitude to frequency or *vice versa*). Try playing this again and choosing the *Live* source. The louder you speak into the microphone, the higher and more stressed-out your voice sounds.

An Alternative Pitch Tracker

It turns out that there is another way, albeit more computationally expensive, to track the frequency besides the *FrequencyTracker* module. If you perform a *LiveSpectralAnalysis* on a signal and then use the right channel of the fundamental track, you have an estimate of the fundamental frequency of the signal. Take a look at *harm analysis ctrl oscil freq* for an example of how to do this.*

Notice that the *GenericSource* is fed into a *LiveSpectralAnalysis* called *HarmonicAnalysis-(3a - 5c)*. Double-click the *LiveSpectralAnalysis* to see its parameters. The **Harmonic** box must be checked for this to work. **LoFreq** and **HighFreq** should be set to the expected frequency range of the input. Also, take note of the value of **LowestAnalyzedFreq**, because we will use this information in the next module, which is, believe it or not a *SampleAndHold*.

Double-click *freq track* to see its parameters. This is a *SampleAndHold* whose **HoldTime** is 256 samp. Why 256 samp? Because when you select **2 F** as the **LowestAnalyzedFreq**, the *LiveSpectralAnalysis*

* Keep in mind, however, that this is a temporary and somewhat kludgy way to do this, and that it will become much easier to do in the *next* software release. So if you feel disgusted by the explanation that follows, you may just want to defer your use of this feature until the next release.

will output 256 tracks on each frame.[§] We want to sample only one of those tracks and we want to do it once per frame. So we sample one value and hold onto it for the next 255 samples until the corresponding value on the next frame. The only remaining piece of the puzzle is which of the tracks to read and how to offset the `SampleAndHold` to pick that track on each frame. The formula for this is

```
(256 - 3) samp
```

where you would substitute a different number for 256 if you were using a different **LowestAnalyzedFreq**.

Finally, double-click *track voice* to see the parameters of this *Oscillator*. Notice that the **Frequency** parameter is set to

```
freq track R * SignalProcessor sampleRate * 0.5 hz
```

This is one of the few examples that uses an `R` to get the right channel of a pasted Sound rather than the left channel. That's because the frequency envelopes are on the right output channel of the *LiveSpectralAnalysis*, and the amplitude envelopes are on the left channel.

The rest of the Sounds in this row and the next use the harmonic analysis method for tracking the frequency of the input (but could just as well have used the *FrequencyTracker* module). Try playing some (or all) of them and take a look at the ones that interest you in order to see how they were put together.

Decision Thresholds and Logical Comparisons

So far, we have been using parameters extracted from the audio signal as something like continuous controllers. It's also possible to use them in a way analogous to discrete switches or gates. The key to turning the continuous envelopes into gates or switches is to use a *Threshold* or one of the comparison operators.

For an example of using the *Threshold* on an amplitude, look at *trigger a sample on S*. This Sound is the same as the *Highs to lows* example, except that it feeds the ratio of high to low energy into a *Threshold*. When the ratio of highs to lows gets above a selected value, the *Threshold* output is one. At all other times, the output of a *Threshold* is zero. By pasting the *Threshold* into the trigger field of a *Sample*, you can trigger the sample whenever the ratio of highs to lows exceeds the selected threshold. In this case the threshold is set such that it triggers on the "S" sounds at the beginning of "so" and the ending of "essence", but does not trigger at any other time.

You can also make decisions based on the *frequency* of the input. For example, take a look at *pch selects eq*. This is a *StereoMix2* with two *GraphicEQ* inputs. The scale on the first input is

```
((22050 * freq L) gt: !Freq nn hz removeUnits) smoothed
```

and the scale on the second input is

```
((22050 * freq L) le: !Freq nn hz removeUnits) smoothed
```

First the *FrequencyTracker* is scaled to the full range of frequencies using

```
(22050 * freq L)
```

You probably noticed that it is missing the units of `hz`. Since this is not a frequency field, we don't need to include the units.

The second half of the expression

```
!Freq nn hz removeUnits
```

allows you to set a pitch in terms of note number using `!Freq`, converts the note number value to hertz, and then removes the units from the number, since we want to compare this number to another number that does not have units. (The *MIDIMapper* is scaling the range of `!Freq` to between 12 and 127 in this example).

[§] The number of tracks per frame associated with each **LowestAnalyzedFreq** value are:

1F: 512	2F: 256	3F: 128	4F: 64	5F: 32
---------	---------	---------	--------	--------

The two frequencies are compared using

le:

an operator that stands for *less than or equal to*. This means that the entire expression will be one when the input pitch is less than or equal to the pitch that you set using the `!Freq` fader in the virtual control surface. If the pitch is higher than the note number you set in the virtual control surface, then the value of the entire expression will be zero. To avoid the clicks you get from sudden changes, the message

smoothed

is added to the end of the expression. Now take a look at the expression in `Scale1` again:

```
((22050 * [freq]L) gt: !Freq nn hz removeUnits) smoothed
```

Compare it carefully against the expression in `Scale2`. What is the only difference between the two expressions? What does `gt:` stand for?

Try playing *pch selects eq* with the default settings and source. Then play it again using the **Live** source to see if you can trigger it with your voice.

For a different example of using the input frequency as a control, look at *time stops until 4 f#*.

In this example, a *TimeStopper* has its **Resume** condition set to

```
((22050 * [freq]L gt: 4 f hz removeUnits)
 * (22050 * [freq]L lt: 4 g hz removeUnits))
```

You probably recognize the basic form of the each half of this expression

```
((22050 * [freq]L gt: 4 f hz removeUnits)
```

and

```
((22050 * [freq]L lt: 4 g hz removeUnits)
```

from the previous example. But why are they being multiplied together? In a system where false is represented by zero and true is represented by one, multiplying is the same as a logical AND. In other words, both of these expressions must be true in order for the entire expression to evaluate to one (true).

This expression will only be true if the incoming pitch is *between* 4 f and 4 g. Try playing *time stops until 4 f#* with the default source. Then play it again with the **Live** source and try singing into it. The *MIDI-Voice* will play its first note and stop. It won't continue until you sing something close to a 4 f# into the microphone. So the pitch of your voice controls when the *MIDI-Voice* can finish.

They're Everywhere

There are other examples of audio signals used as continuous controllers (or in decisions based on thresholds) scattered throughout the examples. Look, for instance, in the **Output MIDI** and **Cross synthesis** files.

Tuning

In Kyma, the smallest change in frequency is about 0.0026 Hz.[§] This degree of frequency resolution means that you can use Kyma to implement microtonal tunings or other tuning systems outside the standard 12-tone equal temperament associated with most keyboard instruments.

In any **Frequency** field, you can enter a constant value directly in hertz or note number; Kyma will use that value directly to control the synthesis or processing algorithm. Additionally, you can enter an arbitrary real-time expression that maps Event Values (like `!Pitch`, `!KeyNumber`, or `!Frequency`) into a frequency value in hertz or note number.

Compute Frequency Directly from !Pitch

Try playing the Sound called *play celesta* in the **Tuning** Sound file (found in the **Examples** folder), and play a chromatic scale on your MIDI keyboard starting at middle C. Double-click on this Sound, and look at the **Frequency** parameter of the Sound called *celesta*. It is set to the value `!Pitch`, meaning that it gets its frequency from incoming MIDI note events. Thus, it is currently playing from the standard, equal-tempered scale.

Now change the value in the **Frequency** field to

```
(!Pitch * 0.5) + 30 nn
```

Compile, load, start the Sound and, once more, play a chromatic scale on the MIDI keyboard. This expression is taking the MIDI pitch and dividing it in half, so it changes half steps into quarter steps. At middle C, the MIDI note number is 60; since multiplying that by 0.5 gives us note number 30, we added 30 nn back in, so that middle C would still be middle C.

Next, try using

```
(!Pitch * 0.25) + 45 nn
```

in other words, we are going to multiply incoming MIDI note numbers by 1/4 to get eighth-step changes for every half step change on the keyboard, and we are adding 45 nn back in so that middle C is still note number 60. Now it should take 4 keyboard octaves to equal 1 octave in pitch.

The **Frequency** parameter value can also be calculated directly in hertz. Change the **Frequency** field to read

```
(!KeyNumber / 12) twoExp * 8.176 hz
```

This takes the frequency of the lowest MIDI note number and multiplies it by a ratio that is the number of half steps above note number 0, *i.e.* the definition for a the 12-tone equal tempered scale. Try the following expression to create a 10-tone scale:

```
(!KeyNumber / 10) twoExp * 8.176 hz
```

Creating Tuning Expressions using the Tuning Tool

You can use the Tuning Tool to prepare expressions to use in **Frequency** fields. Choose **Design Alternate Tunings** from the **Tools** menu to start up the tool. (See *Tools menu: Design Alternate Tunings* on page 451 for a complete description of this tool.)

[§] This number is derived from the fact that all Kyma Sounds keep at least 12 fractional bits of phase information internally (some of them, like *Sample*, keep an even larger fractional part, providing even greater accuracy, but you are always guaranteed at least that accuracy). At the pitch 4 a (440 Hz), a change of 0.0026 Hz is equal to a change in pitch of 1/100th of a cent. At 4000 Hz, it would be 1/880th of a cent, and at 30 Hz it is 1/6th of a cent.

On the main page of the tuning tool, you can choose a reference MIDI key number and the corresponding frequency, and how you want to audition the scale you are designing. After the tool has started, try playing your MIDI keyboard.

Design Alternate Tunings

Specify the MIDI note number of the tonic of the scale and the corresponding frequency (in hertz) of the tonic.

Tonic Key Number
60

Tonic Frequency
261.626

Choose how you want to audition the scale. You can either use a fixed waveform or a sample.

☒ Use Sample: celthrp4a **Choose...**

☐ Sine

☐ Saw21

Keyboard velocity is being used to alter the dynamics of the Sound being produced. However, you may wish to adjust the overall volume. You can use either this fader or Volume (MIDI continuous controller 7) on the default MIDI channel.

Volume

Scales based on 12 scale degrees per octave:

Ratio Scale **Cents Scale**

Scales based on an arbitrary number of scale degrees per octave:

Equal Tempered **Two Interval** **From Text File**

Let's use the tool to create an expression to tune the *play celesta* Sound to a *just* scale.

Click the **Ratio Scale** button. This part of the tool lets you specify each of the twelve scale degrees in terms of a ratio to the tonic. You can enter the ratios into the fields for each scale degree, and interactively try out the scale.

A just scale is built in to the tool; click the **Just** button to load it. Try playing the keyboard.

Design Alternate Tunings

Arbitrary Ratio Scales

This scale is generated from arbitrary ratios entered in the boxes below.

0	1	2	3	4	5	6	7	8	9	10	11
1	135	9	75	5	4	45	3	25	5	225	15
1	128	8	64	4	3	32	2	16	3	128	8

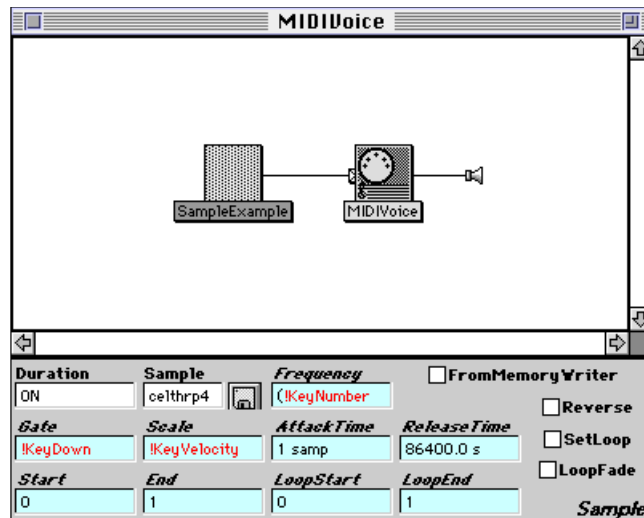
Results

Key	Freq	Ratio	Cents
60:	261.6 hz	1.0 : 1	0.0
61:	275.9 hz	1.055 : 1	92.2
62:	294.3 hz	1.125 : 1	203.9
63:	306.6 hz	1.172 : 1	274.6
64:	327.0 hz	1.25 : 1	386.3
65:	348.8 hz	1.333 : 1	498.0
66:	367.9 hz	1.406 : 1	590.2
67:	392.4 hz	1.5 : 1	702.0
68:	408.8 hz	1.562 : 1	772.6
69:	436.0 hz	1.667 : 1	884.4
70:	459.9 hz	1.758 : 1	976.5
71:	490.5 hz	1.875 : 1	1088.0

Equal **Just** **Pythagorean**

Create Example **Load** **Save** **Back**

Click the **Create Example** button; Kyma will then create an example Sound using this scale. Double click on the example Sound and edit the parameters of *SampleExample*:



Kyma has placed an expression for the just scale that we designed into the **Frequency** parameter of this Sound. To tune the *celeste* GA Sound to this just scale, copy the expression from the **Frequency** parameter and paste it into the **Frequency** parameter of the *celeste* Sound.

You can use any of the tuning methods in the tool to create an example, and either use the example directly, or copy the tuning expression from **Frequency** parameter for use in other Sounds of your own design.

A Retunable Scale

Scales based on the ratios between frequencies must be derived from a particular tonic and, thus, cannot be valid for all keys. The Sound called *tunable kbd* is an example of a just diatonic scale with a selectable tonic, so you can tune the keyboard on the fly. On MIDI channel 1, the keyboard is tuned to a just diatonic scale; switch to MIDI channel 2 and select a new tonic by playing one of the keys (it doesn't matter which octave you play; it is just looking for the pitch class), and then switch back to MIDI channel 1 to play the keyboard tuned relative to that new tonic. Alternatively, you could use two keyboards or two sequencer tracks, each on a different MIDI channel: one to set the tonic, and the other to play in the "key" of that new tonic.

Other Tuning Examples

The other tuning examples illustrate different ways to obtain the pitches of the notes in the alternate tuning system. *Half steps -> whole steps*, *just horns*, and *13-tone linear scale* all use expressions to calculate the pitch in the scale from the MIDI key number. The examples in *using a script* use the *Script* Sound to play an "instrument" in alternate tunings (see *Scripting* on page 522 for more information about scripts). The examples in *using tuning tables* use the *Waveshaper* Sound to look up the frequency for a given key number by using a wavetable that contains the scale (a "tuning table").

Wavetable Synthesis

If you look at the waveform of a sample that has an identifiable pitch and is harmonic, you immediately notice that it is (fairly) periodic — that you can identify patterns that repeat. Imagine taking one of these cycles, saving it as a wavetable, and using an oscillator to repeat it. This would take up a lot less memory than a sample and should sound the same, right? Not exactly. Musical sounds are not usually strictly periodic; they change over time in interesting and lively ways. That's why most oscillator-based synthesis techniques include some way to change the waveform over time: the examples in *Analog* used the phasing effects of detuning, the amplitude-dependent waveform changes of distortion, and variable low pass filters to create waveforms that evolve over time.

Classic Waveform Distortion Techniques

Many of the textbook “computer music” sound synthesis techniques you read about are based on the idea of wavetable synthesis with some form of controllable distortion. Distortion introduces harmonic distortion products, yielding a broader spectrum than the spectrum of the original, undistorted oscillator. Even more importantly, though, you can control how much distortion you want to add at any one time, so you can create timbres that evolve over time (all at relatively small computation costs).

Waveshaping

Imagine a sine wave shape. Then imagine what happens when you turn up the gain on it so much that it clips. It changes from a smooth sine wave to something closer to a sharp-edged square wave, and this change is reflected in the spectrum as well. Using waveshaping, you can get a similar effect, but with more control over the results. Try playing the Sound called *WS:cosine->square*. This Sound takes a cosine and uses it as the argument to a polynomial designed to distort the shape of the cosine waveform and thus add harmonics to its spectrum. Notice that the higher you put `!Volume`, the more the cosine shape begins to approach a square wave shape.

Next try the Sound called *batman head*. This Sound uses the same polynomial in the *Waveshaper*, but instead of using `!Volume` to control the amplitude of the cosine oscillator, it uses an ADSR envelope, triggered by the MIDI keyboard. When you first hit the key, the amplitude is at its maximum, the sound is a little brighter than when it dies away. This corresponds to the behavior of a lot of acoustic instruments, which have a broad, almost noisy spectrum during the attack, and which turn almost to sine waves as they decay.

Amplitude Modulation & Ring Modulation

Compile, load, start the Sound called *AM & RM*, and experiment with the fader settings for the modulator frequency and the carrier amplitude. Like the other distortion techniques, amplitude modulation can give you more frequencies out than you put in (as you can see by playing around with the amplitude of the carrier).

When you amplitude modulate a complex carrier, *each* sine wave component of the carrier generates sum and difference frequencies with the modulator. Try modulating a digital recording or your own voice coming in from the microphone using *AM&RM on a complex carrier*.

Frequency Modulation

This is probably the most familiar of all the classical synthesis techniques. It turns out that by simply modulating the phase of a sine wave oscillator, you can, in theory, generate an infinite number of additional frequencies (well, all right, most of them are at vanishingly small amplitudes, so you don't *really* get an infinite number of them, but you can easily generate more than you care to hear). Try experimenting with the fader settings in *FM lab*.

Sound designer, and fellow Kyma user Francois Blaignan contributed all of the FM Sounds across the top of this Sound file window (*i.e.* the ones that say ‘francois’ as part of their names). Try these Sounds out and then open them up to see how they are put together. Most of them use more than one frequency modulated oscillator, where each of the oscillators contributes one sub-part of the total spectrum — a kind of cross between frequency modulation and additive synthesis.

Group-additive Synthesis

In additive synthesis, you add the outputs of hundreds of sine wave oscillators together to produce a complex waveform. In GA synthesis, you add the outputs of 3 to 5 oscillators, each with a complex waveform and amplitude envelope, together to produce a complex waveform.

Imagine that you are looking at a spectrum with hundreds of sine wave components. Most of the time, you will be able to identify several components whose amplitude envelope shapes are almost identical; imagine grouping *those* sine waves together (forming a more complex waveform) and having them share a single amplitude envelope. That's where the name "group additive" synthesis comes from.

Try out *trombone GA on KBD*, *Fowels vlute*, *graphic env chopper*, and *GA vlus from MIDI file*. These are all examples of synthesis using the *GAOscillators* Sound.

Creating your own GA Wavetables

GA synthesis works best when your original sample is a single tone with an identifiable pitch and a fairly harmonic spectrum. For instance, it can work well on a single tone from a musical instrument, whereas it will probably not work well on a recording of an explosion or sample of human speech (although it can work well on sung vowels).

GA wavetables are created from spectrum files, not directly from samples. So the first step is to use the Spectral Analysis Tool to create a *quasi-harmonic spectrum file* (i.e. the ones with an "h" after the name, not the ones with an "s"), and then to use the GA Analysis Tool to generate a GA wavetable from that quasi-harmonic file.

Let's assume you already have a suitable quasi-harmonic spectral analysis file and proceed to the second step of generating GA wavetables based on that spectrum:

1. From the **Tools** menu, choose **GA Analysis From Spectrum**.
2. Click the **Browse...** button and locate the quasi-harmonic spectrum file called `aou-carla h` from the `singing` folder of the `Spectra` folder of the `Wavetables` folder. Remember, you *must* start with a quasi-harmonic spectrum (one with an "h" tacked onto the end of the name) or it will not work correctly.

Once you have selected a quasi-harmonic spectrum file, the tool will resynthesize it using sine waves so you that can verify that it is indeed the file you had intended to select.

3. By default, the number of harmonics will be set to 124 and the number of wavetables to 3. When you set these numbers, keep in mind that the analysis time grows larger when you request more harmonics or more wavetables. Three wavetables ought to be enough to represent the three vowels in this spectrum. Since the upper harmonics don't contribute as much to defining vowels as they do the consonants, it is safe to change **Number of harmonics** to 64, so you should change the 124 to a 64.
4. Click the button labeled **Create GA file**. It will ask you for a name for the GA file and ask where you want to save it. To be consistent with other GA files, tack the name of the base pitch onto the end of the name followed by "GA"; in this case it would be `aou-carla 3A GA` and save it in your own `Wavetables` folder in a subfolder named `GA`.

Now you have to wait while the Tool to performs the analysis. First it will display **Reading envelopes from the SOS analysis** and show you a progress thermometer.

Then it will display **Determining waveforms and corresponding envelopes...**

Before you know it, you will be startled by a synthetic voice singing "aou" when the analysis is finished.

5. The Tool creates a *GAOscillators* Sound for you with the newly created file already specified in the parameter fields. Open the Sound in the untitled window and set its parameters as follows:

Parameter	Value
Frequency	!Pitch
Duration	on

Then double-click *timeIndex* and change its parameters to:

Parameter	Value
Duration	on
Trigger	!KeyDown

Compile, load, start *aou-carla 3A GA* and play a 3 A on the MIDI keyboard.

To get more polyphony, drag a *MIDIVoice* between the *GAOscillators* and the output icon. Double-click the *MIDIVoice*, set **Polyphony** to 6, and change **Left** and **Right** to 0.25. **Compile, load, start** the *MIDIVoice* and try this on the keyboard in the range around middle C.

Double-click the *GAOscillators* again, and edit the parameters to:

Parameter	Value
Morph	!Morph
Analysis1	hrn 3A GA

Compile, load, start the *MIDIVoice* and try playing some quartal harmonies and using !Morph to change the timbre.

Whooshes, Hits, Bys

These are some example Sounds you may be able to use as the basis for sound effects processing. There are examples of adding subharmonic rumbling to give a sense of a large, massive object, examples of doppler and movement from left to right, and examples of flanging or whoosing sounds that can be used to give a sense of propulsion or movement.

Part II: Advanced Features

Variables: What are those Green Question Marks?

In this tutorial, we will replace the constant parameter values of a Sound with variables. Drag an *Attenuator* from the system prototypes window (**Mixers & Attenuators** category) into a Sound file window. Double-click on it in order to edit its parameters. Experiment with different values between zero and one for the left and right scales.

By adjusting the values of **Left** and **Right**, you can “place” the sound between the left and right speakers. For example, if you set **Left** to 1 and **Right** to 0, the sound seems to emanate from the left, and if you set **Left** to 0.2 and **Right** to 0.8, the sound seems to come from right of center.

One way of specifying the stereo location of a Sound is to specify how much of that Sound should come from the left speaker, and specifying that the remainder should come from the right speaker. In other words, if the portion of the Sound that comes from the left speaker is L , then the portion that comes from the right speaker is $1 - L$, since scale values can range from zero to one. This specifies a relationship between two parameters that is true no matter what the specific value of L . It requires the use of a symbolic representation for the parameters, in this case the letter L to represent some number between zero and one.

In Kyma, you can use variables to serve as symbolic representations. To specify a Kyma variable, choose a memorable name for the variable and precede it with a question mark. In the *Attenuator* Sound, enter

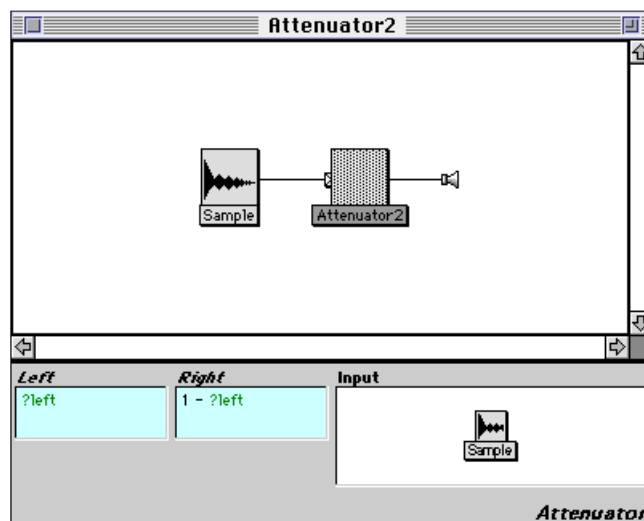
?left

for the value of **Left** and

$1 - ?left$

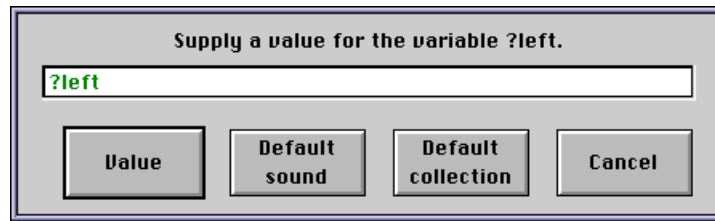
for **Right**.

The parameter fields should now look like this:



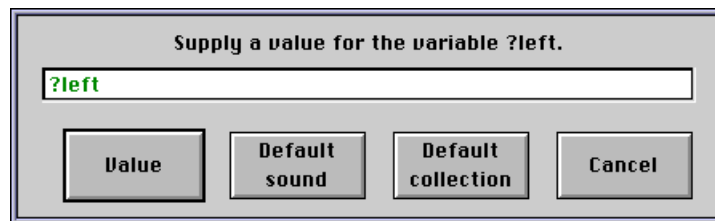
Now that you have entered a variable and an expression containing a variable as the values for **Left** and **Right**, play the *Attenuator*.

You have defined a general stereo locator Sound, but in order to play it you must supply a specific value for `?left`. A dialog box (like the one shown below) requests a value for the variable; enter a value between zero and one (be sure to use a leading zero before any decimal points).



Then click **Value** (or simply press **Enter**). Play once more. The Sound editor remembers the value that you supplied for `?left`, so you don't have to type it in each time you play.

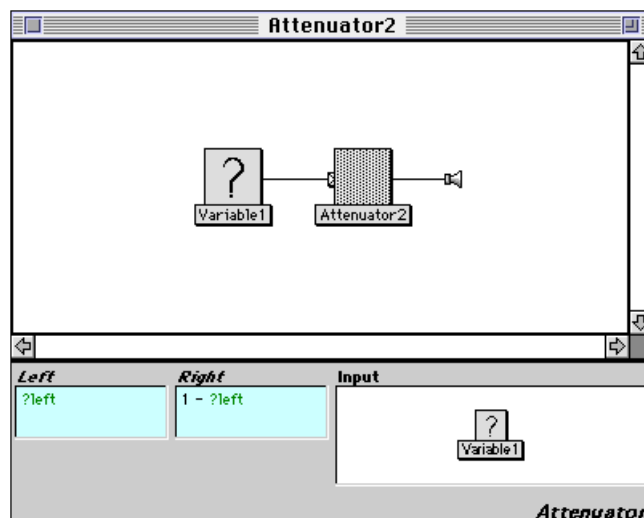
The mappings from variables to the values you have supplied in this Sound editor are called the environment of the Sound editor. Most of the **Action** and **Info** menu operations take place within this environment. To view the current environment, select **Environment** from the **Info** menu.



To clear the values in the environment and choose new values, select **Reset environment** from the **Info** menu. Play the *Attenuator* again.

Now save the *Attenuator* to the Sound file window. Play it a couple of times. Unlike the Sound editor, the Sound file window does not save the environment; you have to define an environment each time you play a Sound that has variable parameters.

Now let's make this stereo location Sound even more general by replacing the **Input** with a *Variable* Sound. Open a Sound editor on the *Attenuator*. Replace the *Attenuator's* **Input** with a *Variable* from the system prototypes window.



Now select **Compile, load, start** from the **Action** menu. Kyma will prompt you for a value for `?left` as before, and it will also prompt you for a value for `?Variable`. When Kyma asks you to supply a value for `?Variable`, you can't type in a value as you did for `?left`. Instead, select the button labeled **Default Sound**. (If this had been a *SoundCollectionVariable*, e.g. a variable representing all of the inputs of *Concatenation* or *Sum*, you should select **Default collection**.)

This associates the variable name `?Variable` with Kyma's default Sound. To change the default Sound, select one of the Sounds in the system prototypes window and select **Set default Sound** from the **Action** menu. Return to the Sound editor and reset the environment. Now try playing the *Attenuator* again.

Change the name of *Attenuator* to *StereoLocator*, and save it in your Sound file window for future reference (*i.e.* the next tutorial!)

Creating and Editing Sound Classes: Upward Mobility

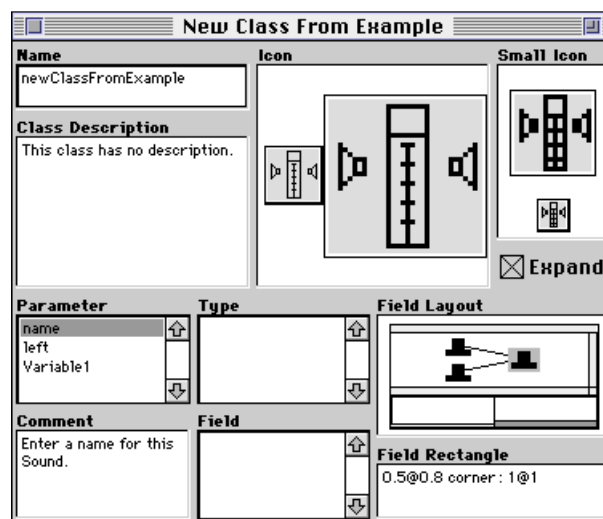
This tutorial describes how you can use the class editor to create a new Sound class with its own icon, help messages, and Sound editor.

Double-click on *StereoLocator*, the Sound that you constructed in the previous tutorial.

Recall that this Sound's *Left* is the variable `?left`, its *Right* is the expression $1 - ?left$, and its **Input** is *Variable*. Each of the variable parameters can take on an infinite number of potential values. In a sense, *StereoLocator* represents an infinite set of Sounds, all sharing a single characteristic: the portion of the Sound coming from the right speaker is one minus the portion of the Sound coming from the left speaker. *StereoLocator* actually represents a whole class of specific instances of Sounds having that characteristic.

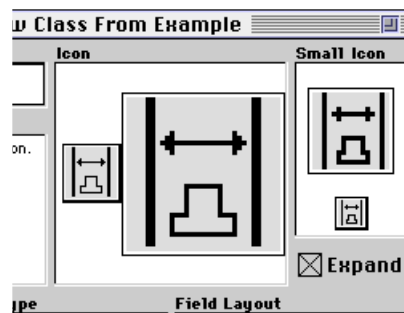
You can create a new class of Sounds based upon the example of *StereoLocator*. Close the Sound editor. Select *StereoLocator*, then, under the **Action** menu, select **New class from example**. Kyma will prompt you to provide values for `?left` and `?Variable`. Set `?left` to 0.75 and click the **Default Sound** button for `?Variable`.

Kyma then opens a class editor on this example.



Replace `newClassFromExample` with a more memorable name, like *Stereo*. Kyma's convention is to begin class names with uppercase letters.

Drag an *AbsoluteValue* from the **Arithmetic** category of the system prototypes window into the **Icon** field of the class editor. Use this starting point for designing a new icon that represents the stereo placement of a Sound between the left and right speakers.



Click in a light area to draw a dark dot, and click in a dark area to draw a light dot. If you keep holding the mouse button down while you move the mouse around, you can draw dots continuously. Sometimes it helps to click in the zoom box of the class editor window, expanding the class editor size of the screen in order to get a larger area for editing. Design both the large and the small icon.

Read the names listed in the **Parameter** field. Where have you seen these before? Apart from the `name` parameter, which is shared by all Sound classes, each parameter name corresponds to the name of a variable in the original example Sound: `?left`, and *Variable*.

Click on `Variable` in the **Parameter** field. This is the input of the new *Stereo* class. To ensure that only Sounds can be used for this parameter, select `sound` in the **Type** field. The types are listed alphabetically, and you can use the scroll bar to scroll through the list of types.

You may have noticed that after you selected sound, the list of possible parameter field types changed. Try selecting a different **Type** temporarily and watch the **Field** field. Only those fields that are legal for the selected parameter type are displayed in the **Field** field. Make sure that you have re-selected `sound` as the type for `Variable`.

Now modify the location for `Variable` in the **Field Rectangle** field:

0.5 @ 0 corner: 1 @ 0.8

and press **Enter**. This places the field in a rectangle whose upper left corner is in the middle of the editor and flush with the top. The lower right corner is flush against the right edge and with the top of the class name field.

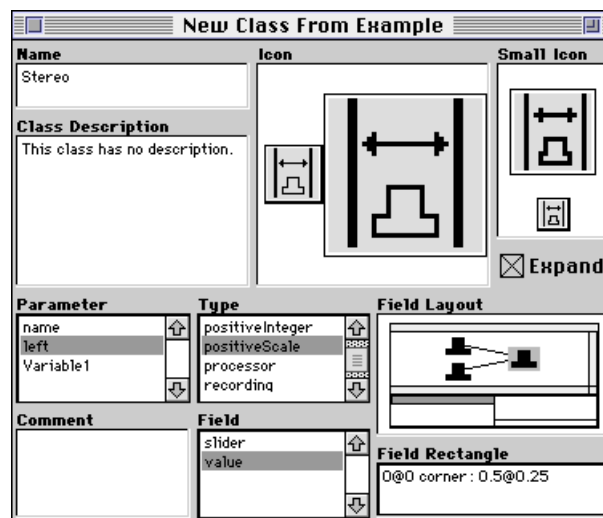
The first pair of numbers is the `x @ y` location of the upper left of the rectangle, and the second pair of numbers is the `x @ y` location of its lower right. The point `0 @ 0` is the upper left corner of the right half of the Sound editor, and its lower right is `1 @ 1`.

These numbers don't represent specific units like pixels or centimeters; they represent the relative proportions of a rectangle within a unit (1 by 1) rectangle. Thus, if you wanted a parameter field to take up the bottom half of the entire area, you would specify an upper left corner of `0 @ 0.5` and a lower right corner of `1 @ 1`.

Return to the **Parameter** field, and select `left`. The value of `left` must lie within the range of zero to one, so select `positiveScale` in the **Type** field. (The range for each variable type is shown in *Parameter Types and Ranges* on page 540). For the **Field Rectangle**, enter

0 @ 0 corner: 0.5 @ 0.25

Notice how **Field Layout** is updated when you press the **Enter** key.



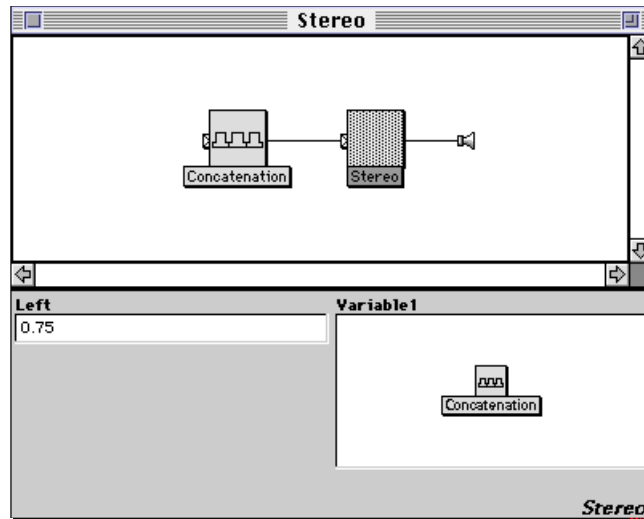
Select `Variable` in the **Parameter** field, and take note of how the **Field Layout** changes. Alternate between selecting `name`, `Variable`, and `left`.

The **Field Layout** shows a miniature version of the Sound editor for this new class. Whenever you select a parameter, its position in the parameter fields is highlighted.

Double check that **Expand** has a check mark in it. When checked, the **Expand** option will cause this new kind of Sound to expand before any transformations are applied to it. The significance of this will be explained in *Parameter Transformers: Symbolic Sounds* on page 198.

Close the class editor and save an instance of this new class. The instance of your new class should appear highlighted in the file window; its name will be *Stereo*.

Edit the new instance of class *Stereo* by double-clicking on it. Note the positions of the parameter fields on the bottom half of the Sound editor.

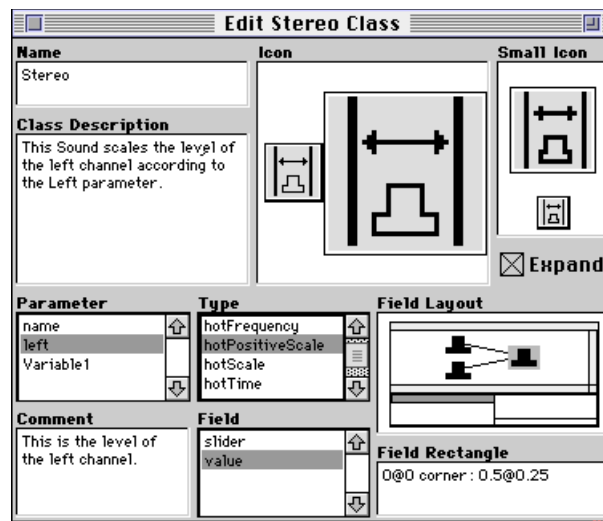


Play the Sound. From the **Modifiers** category of the system prototypes window, drag *Chopper* into the **Variable** field. Play *Stereo*. Try entering different values for the **Left** parameter and play again. Click on the parameter label of the **Left** field. It will tell you that no help is available, but we will soon fix that.

Close the Sound editor, saving the changes that you have just made. With *Stereo* still selected, choose **Expand** from the **Action** menu. Double-click on the new Sound, called *StereoLocator*, that appears in the Sound file window. Notice that this Sound is like the original *Attenuator* that we made the class out of, but this Sound has specific values assigned for the **Left**, **Right**, and **Input** parameters. When you play a Sound, it is first expanded to its most primitive components and then sent to the Capybara. You can use **Expand** from the **Action** menu to see how complex Sounds are actually constructed out of simpler component Sounds.

Close the Sound editor. Select *Stereo* again, and choose **Edit class** from the **Action** menu. **Edit class** lets you edit an already existing class. From the **Action** menu, choose **Retrieve example from class**. This saves your original example Sound in the Sound file window. Go to the Sound file window and double-click on the Sound that was just retrieved. Notice that **Retrieve example from class**, unlike **Expand**, saves the original example Sound with the variables still in it. Close the example Sound, and go back to the class editor, so we can add some on-line help messages to the new class.

First, in the **Class Description** field, enter a short description of what this class does. Then select **left** in the **Parameter** list. In the **Comment** field, remind the user (and yourself many months in the future when you try to use this Sound again but have forgotten how you set it up) that this is the level of the left channel, and that the level on the right channel is one minus this value. With **left** still selected, change the **Field** from **positiveScale** to **hotPositiveScale**.



Select **Variable1** and enter a comment for this variable as well.

Close the class editor, and save the changes to the edited class. Try out opening **Stereo** again and paste the Event Value **!Pan** into the **Left** parameter field. Try playing **Stereo** and controlling the value of **!Pan** from the virtual control surface (or MIDI controller). Once you have finished experimenting, close the Sound editor.

Now let's improve the stereo placement function. Find the example you retrieved (it should be called **StereoLocator** and should have variables in it). Edit it and replace its right parameter with

```
(1 - ?left squared) sqrt
```

You may have to choose **Large window...** (**Ctrl+L**) from the **Edit** menu to make the field large enough to see what you are doing. Close the Sound editor and keep the changes you have made.

Choose **New class from example** from the **Action** menu, supply a value for **?left** and click on **Default Sound** for **Variable**. This time we don't have to start from scratch. Drag **Stereo** from the Sound file window into the **Icon** field, and then drag it again into the **Parameter** field. This sets the icon, parameter types, and locations to those specified in class **Stereo**.

Type in **ImprovedStereo** for the name of this new class. Tab over to the **Class Description** field and enter a new description. Close the class editor and save an instance of the new class. Compare the panning of **ImprovedStereo** to that of **Stereo**.

Algorithmic Sound Construction: Your Computer is your Slave

You have by now seen multiple examples of how to construct Sounds in the graphical editor. In those cases where you can systematically describe the relationships between Sounds and parameters, you can use the **Script** Sound to construct complex Sounds for you automatically, according to an algorithm that you specify in the Smalltalk-80 programming language. Your program can be as simple as a series of events, saying when each Sound should start, or as elaborate as a new system for music composition. To specify events, all you have to learn is how to specify the start time of a Sound. For more elaborate projects, you have the power of a full programming language — Smalltalk-80 — to work with.

Let's start with simple event script and, step-by-step, develop it into a short program. Find **Script** in the **Algorithms** category of the system prototypes window, drag a copy into your Sound file window, and open a Sound editor by double-clicking on it. Replace its input **wavelet** with **Attenuator** from the system

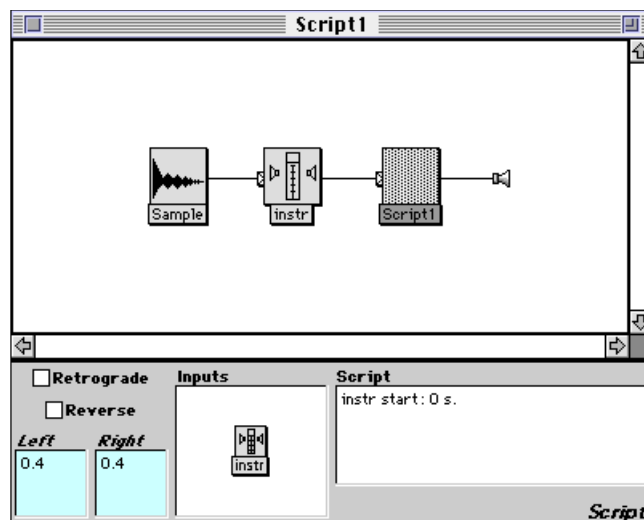
prototypes window category **Mixers & Attenuators**. Change *Attenuator's* name to *instr* (by selecting it, pressing **Enter**, and entering the new name).

In order to be able to set the parameters of *instr*, replace some of its constant values with variables. Double-click on *instr*, and replace its **Left** parameter with the variable ?left, press the **Tab** key, and replace its **Right** parameter with the expression 1 - ?left. Then edit its **Input**, replacing its **Frequency** with the variable ?freq and its **Duration** by the variable ?dur.

The **Script** parameter of a *Script* Sound consists of Smalltalk-80 expressions and events that assign start times and other parameters to its inputs. To write your own script, first double-click on *Script*. Delete the current contents of the **Script** field of the prototype. (Click once in the field to select all, and then press the **Delete** key). Now, create a simple event by setting the start time of an instance of *instr*. Type

```
instr start: 0 s.
```

into the parameter field. Play *Script*. The Sound *instr* still has variable parameters, so Kyma will ask you to provide specific values for those parameters before playing the Sound. Be sure to specify the units of the frequency and duration parameters (try 440 hz for ?freq, 1.5 s for ?dur, and 1 for ?left). If you make a mistake in supplying these values; select **Reset environment** from the **Info** menu and play *Script* again. You should hear one of the Celtic harp samples at 440 hz in the left speaker.



You can set the value of a variable parameter from the **Script** by typing the name of the variable, a colon, and then a value for that parameter. For example, to create an event that starts *instr* at 0 s with a frequency of 500 hz, a duration of 2 s, and a position centered between the two speakers, type **Ctrl+L** to enlarge the field, and then type the following:

```
instr
start: 0 s
freq: 500 hz
dur: 2 s
left: 0.5.
```

Click the **Accept** button to accept this new **Script**.

There can be any number of spaces or carriage returns separating each parameter:value pair; the set of parameter:value pairs should be terminated by a period. There should be no spaces between the parameter name and the colon; there must be a space between the colon and the parameter value.

Play the Sound as it now stands. Then try changing some of the parameter values specified in the **Script**, playing the *Script* after each change.

Add another event to the script. Copy the first event, and paste it after the original event. Start the second event at 1 s and give it the same duration and stereo position. The frequency should be one whole step higher than that of the previous event. To compute this convert 500 hz to a nn (note number) and then to add 2 half steps to it.

Now your script should look like this:

```
instr
  start: 0 s
  freq: 500 hz
  dur: 2 s
  left: 0.5.

instr
  start: 1 s
  freq: 500 hz nn + 2 nn
  dur: 2 s
  left: 0.5.
```

Try playing the Sound now.

Suppose you want to specify a whole-tone scale using the script language. We can define the pitch of any event in the whole-tone scale as being the pitch of the previous event plus 2 half steps. Let's encode this as a Smalltalk loop. Type **Ctrl+L** to enlarge the **Script** field and replace the existing script with the following:

```
| p |
p := 3 c.
1 to: 5 do: [ :i |
  instr
    start: (i - 1) s
    freq: p
    dur: 1 s
    left: i / 5.
  p := p + 2 nn].
```

Click the **Accept** button to accept this new **Script**, and play the Sound.

In this example, the Smalltalk variable *p* is declared at the beginning of the script between vertical bars. Then it is initialized to the value 3 c (once octave below middle C). Notice that there must always be spaces before and after the `:=` in an assignment statement. The code within the square brackets is executed five times; in the first iteration, *i* takes on the value 1, in the second iteration it takes on the value 2, *etc.*, until the last iteration when it takes on the value 5. An event is specified for each of the five iterations. After the event specification, *p* is incremented by two half steps, and the loop begins again.

Now try the Sound again with the following changes (and explain in words what is going on):

```
| p |
p := 3 c.
1 to: 5 do: [ :i |
  instr
    start: (i - 1 * 0.5) s
    freq: p
    dur: 1 s
    left: i / 5.
  p := p + 3 nn].
```

The starting pitch is a variable, *p*; now let's make the number of notes a variable as well. Let's call that variable *n*. Since *n* is the number of times through the event loop, we can substitute *n* for 5 in the previous program, making sure that *n* is assigned an initial value.

```
| p n |
n := 5.
p := 3 c.
1 to: n do: [ :i |
  instr
    start: (i - 1 * 0.5) s
    freq: p
    dur: 1 s
    left: (i - 1) / n.
  p := p + 3 nn].
```

Experiment with different values for *p* and *n*.

This is an adequate solution, except that we have to edit the **Script** every time we want to change the values of *p* and *n*. Let's make one final alteration to the **Script** that will enable us to control its parameters.

Replace the Smalltalk variables, which are known only within *Script*, with Kyma variables, which can be set from outside of *Script*. Now the script should look like this:

```
| p |
p := ?startPch.
1 to: ?nbrNotes do: [ :i |
    instr
        start: (i - 1 * 0.5) s
        freq: p
        dur: 1 s
        left: (i - 1) / ?nbrNotes.
    p := p + 3 nn].
```

Play the Sound. Kyma will request values for ?startPch and ?nbrNotes. Type 3c for ?startPch and 10 for ?nbrNotes. Change the name of *Script* to *pattern*, and close the Sound editor, saving the changes that you have made. Save your Sound file as well.

Duplicate *pattern*, and then edit *duplicateOfPattern* by double-clicking on it. First, change its name to *patternMaker*. Then replace *instr* with *pattern* by dragging *pattern* from the Sound file window onto *instr* in the signal flow diagram of *patternMaker*. Then modify the **Script** of *patternMaker* to read:

```
| p |
p := 4 c.
1 to: 5 do: [ :i |
    pattern
        start: (i - 1) s
        startPch: p
        nbrNotes: 5.
    p := p - 2 nn].
```

Play this Sound. What was a script in *pattern* is now being treated as an instrument in *patternMaker*. There is a blurring of the distinctions between the definitions of "instrument" and "score" in Kyma. In their place are structures built up out of other structures; how you choose to name these structures and how you choose to define "high-level" versus "low-level" is up to you.

Now change the initial pitch value to 0 nn,

```
| p |
p := 0 nn.
1 to: 20 do: [ :i |
    pattern
        start: (i - 1) s
        startPch: p + !Pitch
        nbrNotes: 5.
    p := p - 2 nn].
```

Play *patternMaker* and play trills on the MIDI keyboard. Use **Ctrl+R** to restart the Sound on the Capybara without having to load it again. At compile time, fixed values of *p* plus the Event Value !Pitch are substituted into copies of the *Sample*. Since the fixed values of *p* are so small, they are like offsets to the pitches that you are supply from the MIDI keyboard.

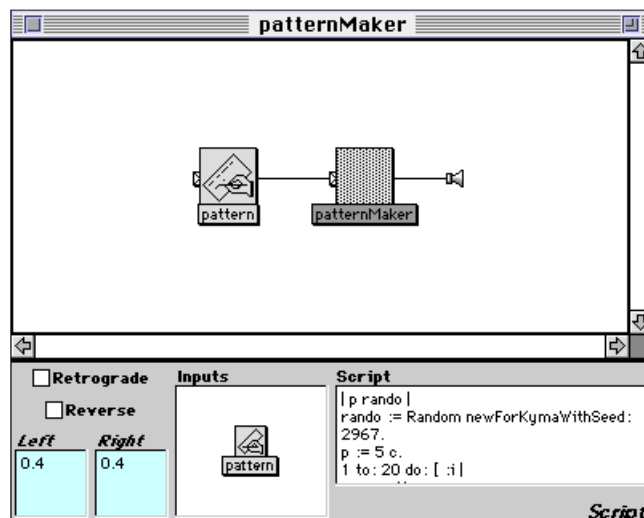
Now use a continuous control on the frequency. Since the continuous controllers are in the range from 0 to 1, multiply the value of the controller by the full range for MIDI notes: 127 nn

```
| p |
p := 0 nn.
1 to: 20 do: [ :i |
    pattern
        start: (i - 1) s
        startPch: p + (!Frequency * 127 nn)
        nbrNotes: 5.
    p := p - 2 nn].
```

As a final modification to *patternMaker*, let's add a random offset to each starting pitch. Change the initial value of p back into the audible range and add a random value between minus two half steps and plus two half steps. To do this, multiply the random number (that lies in the range of (0, 1)) by 4 (so that it lies in the range from (0, 4)). Then subtract 2 (so that it lies in the range of (-2, 2)). Make sure to include the units of nn so that the number is interpreted as the number of half steps.

```
| p rando |
rando := Random newForKymaWithSeed: 2967.
p := 5 c.
1 to: 20 do: [ :i |
    pattern
        start: (i - 1) s
        startPch: p + (rando next * 4 - 2) nn
        nbrNotes: 5.
    p := p - 2 nn].
```

Now that you have a basic loop structure, you can start experimenting with other ways to construct complex Sounds algorithmically.



Other examples of using *Scripts* to construct Sound structures can be found in the Sound files that came with Kyma (for example, the Sound called *homemade reverb* in the file called **delays, chorusing, reverb**).

Remember that a *Script* constructs a Sound first and then plays it. The Smalltalk code is not executed while the Sound is playing.

Parameter Transformers: Symbolic Sounds

In the last tutorial, we used a *Script* to generate multiple copies of a Sound that had variable parameters. This tutorial demonstrates how to use a *ParameterTransformer* to modify the parameters of a Sound that already exists.

Drag a *ParameterTransformer* from the **Algorithms** category of the system prototypes window into your Sound file window. Edit it by double-clicking on it, and replace its **Input** with a *LimeInterpreter*. A *LimeInterpreter* reads binary files generated by Lippold Haken's Lime music notation program[§] and "plays" them using Kyma Sounds. Try playing the *LimeInterpreter*. If Kyma asks you to locate a Lime binary called **waterbirds**, you can find it in a folder called Others within the Examples folder.

Now modify the **Transformation** parameter of *ParameterTransformer* to:

```
snd frequency: 100 hz.
```

Play *ParameterTransformer*. What is the effect of this **Transformation**? The *ParameterTransformer* looks through the entire Sound structure and sets any parameters named **Frequency** to 100 hz. The value of any named parameter can be changed by a *ParameterTransformer*. For example, try the following **Transformation**:

```
snd sample: 'celtHrp2a'.
```

The *ParameterTransformer* reaches into all the Sounds to the left of it in the signal flow diagram and changes any parameters named **Sample** to 'celtHrp2a'.

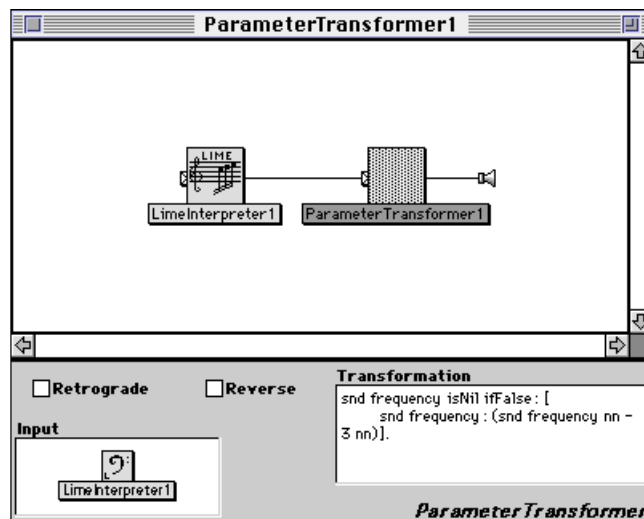
Now try changing all the durations. Change the **Transformation** to:

```
snd duration: 0.1 s.
```

Suppose you wanted to shift all the frequencies down by three half steps. The new frequency is a function of the old frequency; to access the current value of a parameter, type `snd` followed by the name of the parameter. In the following example,

```
snd frequency isNil ifFalse: [
  snd frequency: (snd frequency nn - 3 nn)].
```

the *ParameterTransformer* steps through each Sound in the structure, gets its **Frequency**, converts it to a note number, subtracts three from it, and sets the frequency parameter to the new value. If any Sound in the structure does not have **Frequency** as a parameter, it will return `nil` as its frequency. To assure that only non-nil values for frequency are used, the transformation is only executed when `snd frequency` is not nil.



All of these transformations are symbolic, based only on the name of the parameters. So, for example, this last transformation would have no effect on a Sound that did not have **Frequency** as a parameter. The transformation changes the values of parameters in the same way that you might edit the values in the parameter fields in the Sound editor before playing the Sound. A *ParameterTransformer* changes the initial values of the parameters before the Sound is played; it does not operate on the audio output of a Sound.

[§] Lime is available at <http://datura.cerl.uiuc.edu>.

Now let's make a transformation that depends on the start time of the Sound being transformed. The reserved word `time` always contains the time offset of `snd` in microseconds. The reserved word `totalDuration` contains the total duration of the *ParameterTransformer's* immediate input; it too is given in microseconds. Try the following **Transformation**:

```

snd frequency isNil ifFalse: [
    snd frequency:
        (snd frequency nn -
         (3 * (time / totalDuration)) nn)].

```

As time progresses from zero up to the number of microseconds in the total duration, the value of `(time / totalDuration)` progresses from zero up to one. The frequency of the first note has zero half steps subtracted from it: no change at all; however, as time goes on, each subsequent note has larger and larger fractions of 3 half steps subtracted from its pitch.

Not only can a new parameter value depend on time and its current value, it can also depend on the values of other parameters or combinations of parameter values. Try the following **Transformation**:

```

snd duration isNil ifFalse: [
    snd frequency: snd duration inverse * 100].

```

This transformation sets the **Frequency** to 100 times the inverse of whatever the **Duration** happens to be.

Next, open the Sound file **ParameterTransformer** from the **Advanced** folder. Try playing the Sound *independent partials*; when it asks for a frequency, enter 100 hz. Inspect *independent partials*; by double-clicking on it and looking at the **Frequency** field of each *Oscillator*. The first *Oscillator's* frequency is `?frequency`, the second one is `?frequency * 2`, the third is `?frequency * 3`, etc. When you play *independent partials*, it sounds like a single, harmonic tone.

Edit the Sound called *transformIndependentPartial*s by double-clicking on it. The **Transformation** parameter has a long comment, after which it should read:

```

snd frequency isNil ifFalse: [
    snd frequency:
        snd frequency nn + (110.0 hz * time / totalDuration)].

```

Reading this, we would expect that each repetition of *independent partials* will be at a higher frequency. When you play *transform independent partials* you hear, as expected, the frequencies getting higher with each repetition. What you may not have expected is that the sound starts out harmonic and becomes in-harmonic with subsequent repetitions. Why is that?

The sequence of events is:

1. Kyma sets the *Oscillator* frequencies to 100 hz, 200 hz, etc.
2. The *ParameterTransformer* adds a multiple of 110 hz to each repetition of *independent partials*.

Since it adds a fixed amount to each frequency, the harmonic relationship is no longer maintained. For example, if you start out with 100, 200, and 300 hz and then you add 110 to each frequency, the result is: 210, 310, and 410 hz. Since 410 hz is not a multiple of the lowest frequency, 210 hz, it is not a harmonic of 210 hz.

Is it possible to transform the frequencies in such a way as to maintain the harmonic relationship? One way would be to multiply the frequencies by a constant rather than adding a constant value to them. Another way is to create a new Sound class that maintains the harmonic relationship no matter what.

Select *independent partials* in the Sound file window and choose **New class from example** from the **Action** menu. When Kyma asks for a value for `?frequency`, enter 100 hz. Change the name of the new class from `newClassFromExample` to `HarmonicPartial`s. Then uncheck **Expand**; this tells the Sound: Apply the transformation to yourself before you expand. If you leave **Expand** checked, then the Sound will expand first and then apply the transformation to the expanded version of itself.

Close the class editor and answer that you want to save an instance of the new class; you should see a new Sound called *HarmonicPartial*s in your Sound file window. In the open Sound editor, replace *inde-*

pendent partials with *HarmonicPartial*s. Now play *transform independent partials*. The frequencies increase with each repetition, as before, but this time, the partials stay harmonic. Why does your new class behave differently from independent partials? It's because your new class has only one parameter: **Frequency**. You apply a transformation to the frequency, adding something to it. Then, your Sound computes its other frequencies based on that transformed frequency (by multiplying it by 2, 3, 4, *etc.*).

Part III: Exercises

Time/frequency Scaling using Additive Resynthesis and the Spectrum Editor

The first section is a step-by-step example of how you can use spectral analysis and resynthesis to scale the duration or frequency of your own voice.

Part I: Record your Voice

1. Use the Tape Recorder tool to record your own speaking voice. Come up with a short, meaningful or humorous phrase (or grab the nearest book or magazine and read a few lines).
2. Use the Spectral Analysis tool to create two spectral analyses (both the straight and the harmonic version) of your recorded speaking voice. The analysis tool also gives you a Sound for each analysis which you can transfer into a Sound file window. Save the Sound file window to disk before proceeding.

Part II: Time/frequency Alterations

1. Double-click on one of the *SumOfSines* created by the spectral analysis tool. Change the **Duration** to on. Set up the sound to be triggered by the MIDI keyboard. To do this, click in the **Gate** field, press **Escape**, and play two keys simultaneously on the MIDI keyboard. **Gate** should now have **!KeyDown** in it. Now play the sound (using **Ctrl+Space Bar**).

Try playing the MIDI keyboard to make sure the trigger works.

2. Time-stretch the resynthesis to ten times its original duration without changing its frequency. To do this, multiply the value in the **OnDuration** field by 10, for example, it might look something like this

5 s * 10

Load the Sound using **Ctrl+Space Bar** and trigger it using the MIDI keyboard.

3. Experiment with different time-stretching factors. Then set the **OnDuration** back to what it was originally in preparation for doing frequency scaling.
4. Control the frequency of the speech from the MIDI keyboard without changing its duration. To do this, click in the **Frequency** field, press **Escape**, and play one key on the MIDI keyboard. This should set **Frequency** to **!Pitch**. Load the Sound using **Ctrl+Space Bar**. Then play different keys on the MIDI keyboard. Which one gives you the original recording back without frequency alterations?

Part III: Spectrum Editor

By opening a spectrum editor on an analysis, you can see a graphic representation of the spectrum, listen to individual harmonics, and modify the frequency or amplitude of selected harmonics.

1. Open a spectrum editor on the harmonic analysis you did in the first part of this exercise. You can do this in several ways:

choose **Open...** from the **File** menu, set the file type to **Spectrum file**, and select the name of the file in the file dialog

double-click on the name of the spectrum file in the File Organizer

edit the *SumOfSines*, then click the disk button next to **Analysis0** while holding down the **Command** or **Control** key.



2. Once the spectrum editor opens, use the leftmost button to play a resynthesis based on this analysis. Then use the mouse to grab the yellow scrub bar and move it back and forth across the display. Next, try controlling the scrub bar by using pitch bend on the MIDI keyboard.



3. Press **1** on the computer keyboard to select and play track 1, the fundamental. Then try track 2, 3, etc. Click the selection criteria button (it is the fourth one from the right and looks like a bunch of tracks with one selected). This lets you select several tracks at once according to some criteria. Try listening to different selection criteria.



- Now, “monotonize” all of the octave harmonics. Click on the track filter button (the third one from the right). These operate on whatever harmonics are currently selected. Choose **Replace Frequency with average**. This will replace the time-varying frequency of each track with a single frequency: the average frequency the track had over the entire length of the analysis.

- Compare the sound of all harmonics against the octave harmonics alone using the leftmost play button and the one right next to it, the play-selection button.
- Now save the monotonized octave harmonics in another analysis file. The octave harmonics should still be selected. “Copy” these harmonics using **Ctrl+C**. Kyma will ask you to name the new analysis file and save it somewhere (name it something like “oct harm only”).

Click in your Sound file window to bring it to the front and use **Ctrl+V** to paste a new *SumOfSines* based on the octave harmonics file into the window. Then you can close the spectrum editor without saving the changes.

- Our next task is to try to create an endless glissando effect based on the Shepard’s tones audio illusion.

The idea is that we want to change the **Frequency** of the *SumOfSines* so that it repeatedly does a smooth glissando from a very low to a very high frequency. Double-click the *SumOfSines*. Set its **Duration** to on, and check the **Loop** box. Paste an *Oscillator* from the prototype strip into the **Frequency** field and scale and offset it as follows:

L * (300 - 30) hz + 30 hz

This will scale the *Oscillator* to the range of 30 to 300 hz.

Now we have to change the *Oscillator* so that it repeats once every 15 seconds and so that it uses Ramp rather than Sine as its wavetable. That will give us the repeating glissando function. Double-click in a white space of the signal flow diagram. Edit the *Oscillator* parameters, changing the **Wavetable** to Ramp and the **Frequency** to 15 s inverse (because the period of repetition in seconds is the inverse of frequency in hertz).

Listen to the *SumOfSines* so far. In order to avoid the discontinuity when the ramp goes back to zero every 15 seconds, let’s put a repeating amplitude envelope on the whole thing, so it fades in and fades out on each repetition.

- To make the envelope, paste another *Oscillator* from the prototypes into the **Envelope** field. Edit the *Oscillator*, changing its **Wavetable** to LinearEnvelope and its **Frequency** to 15 s inverse.

Now play the *SumOfSines* again. This time, it should fade in and fade out on each repetition.

- The next step is to add another *SumOfSines*. This one should have the same kind of glissando but it should start halfway through the first *SumOfSine’s* glissando. The idea is to gradually fade in a new glissando, just as the first glissando is reaching its top and it is fading out.

To add another Sound, we have to use a *Mixer*. Drag one from the prototypes and place it on the line between the *SumOfSines* and the speaker. Replace its default inputs with the *SumOfSines*. Then drag the *SumOfSines* from the signal flow area into the *Mixer’s* **Inputs** field. You should end up with a *Mixer* of two *SumsOfSines*. Change the *Mixer* **Left** and **Right** attenuators to 1, rather than the default 0.5.

So far we have two identical *SumOfSines*. But we need to delay both the glissando and the envelope *Oscillators* that control the frequency and amplitude of the second *SumOfSines*. Click on the tab on the left edge of the second *SumOfSines* while holding down the **Command** or **Control** key so it shows all of its inputs.

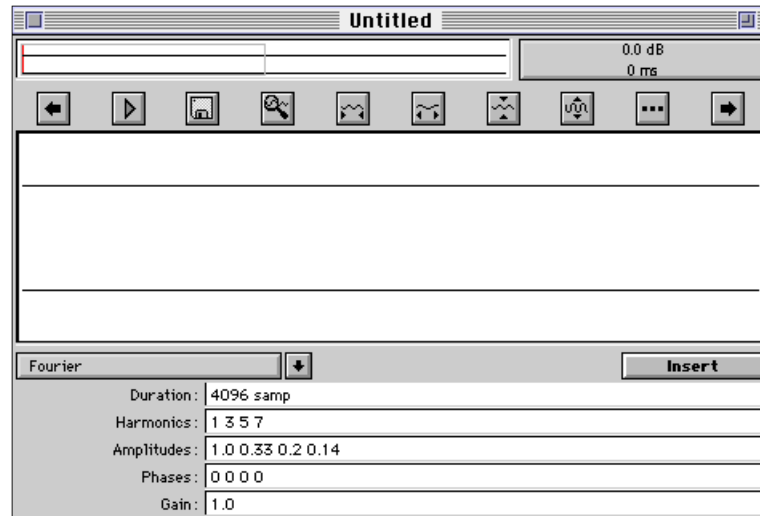
Insert a *TimeOffset* between the envelope oscillator and the *SumOfSines*. Then insert another *TimeOffset* between the glissando oscillator and the *SumOfSines*. We want to set the delay to half the duration of the glissando. So set the **SilentTime** in each of the *TimeOffsets* to 7.5 s.

- Try listening to the *Mixer*. It should give at least something of the illusion of an endlessly rising pitch, because as the higher octaves fade out, some lower octaves are fading in below them.

Creating a Wavetable using the Sample Editor

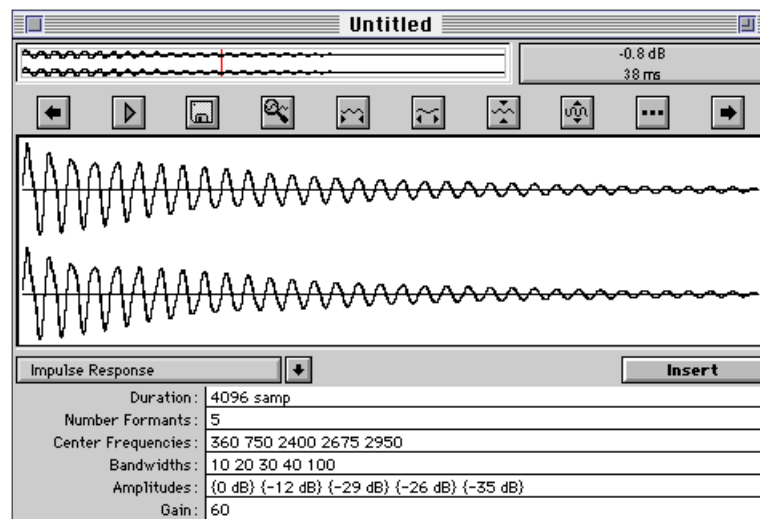
In this exercise, you will create a wavetable that contains an impulse response, useful for making percussive sounds.

First, open the sample editor. Choose **New...** from the **File** menu, set the file type to **Samples file**, and press **Enter**.



The upper half of the sample editor lets you do the usual kinds of cut, copy, and paste operations (see *Sample Editor* on page 501 for full details), and the lower half provides templates for generating new wavetables or segments of wavetables.

Scroll in the template list until you find the **Impulse Response** template. Select all of the waveform in the graphic waveform editor (the upper half) by clicking once in the waveform area and then by choosing **Select all** from the **Edit** menu. Now click the **Insert** button. **Insert** replaces the selection in the upper half with whatever you have specified in the template.



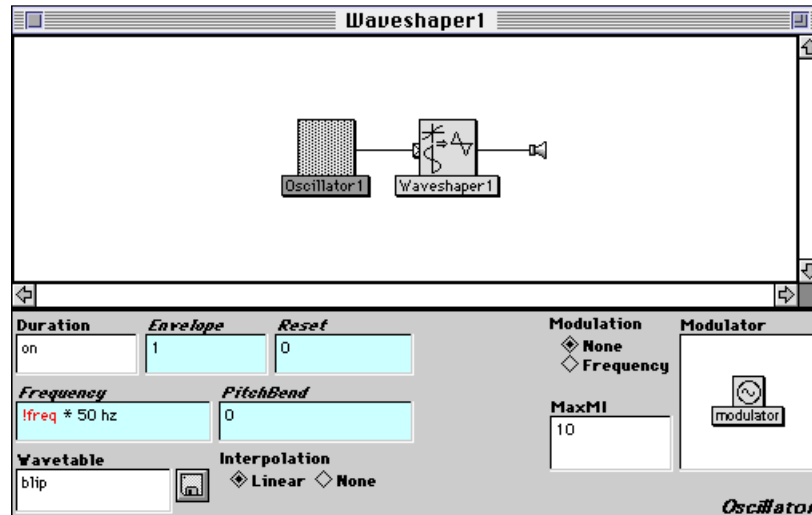
The **Impulse Response** template takes the specified center frequencies, bandwidths, and amplitudes of several formants, and it constructs a filter that matches your specification; when you click **Insert**, it inserts the response of that filter to a single impulse. This can be an interesting template to play with for creating wavetables for percussive sounds.

Try changing some of the center frequencies and then click **Insert** again. If the waveform is clipped or very small in amplitude, you may have to adjust the value of **Gain** as well. To hear the result, click on the play button above the graphic editor.

Continue experimenting with different values until you are satisfied with your percussive sound. Then close the editor, and confirm that you want to save your new wavetable. Choose the file type, the number of channels, and the number of bits per sample, press **Enter**, then enter a name for your wavetable.

From the system prototypes, drag a *Sample* into a Sound file window and double-click it to edit it. Set **Duration** to on, **Frequency** to !Pitch, **Gate** to !KeyDown, and use the disk button next to the **Sample** parameter to locate your newly created impulse response. Try playing the *Sample*. Try inserting a *Waveshaper* between the *Sample* and output speaker icon to make the audio output louder through distortion.

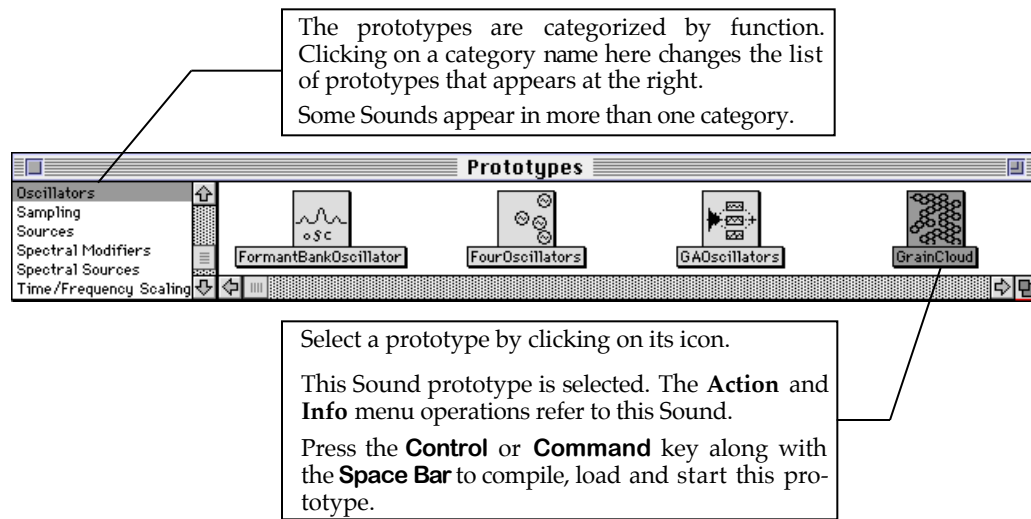
Now substitute an *Oscillator* for the *Sample*; set its **Duration** to on and its **Frequency** to !Frequency * 50 hz and its **Wavetable** to your impulse response wavetable. Play the *Oscillator* and control !Frequency using the virtual control surface.



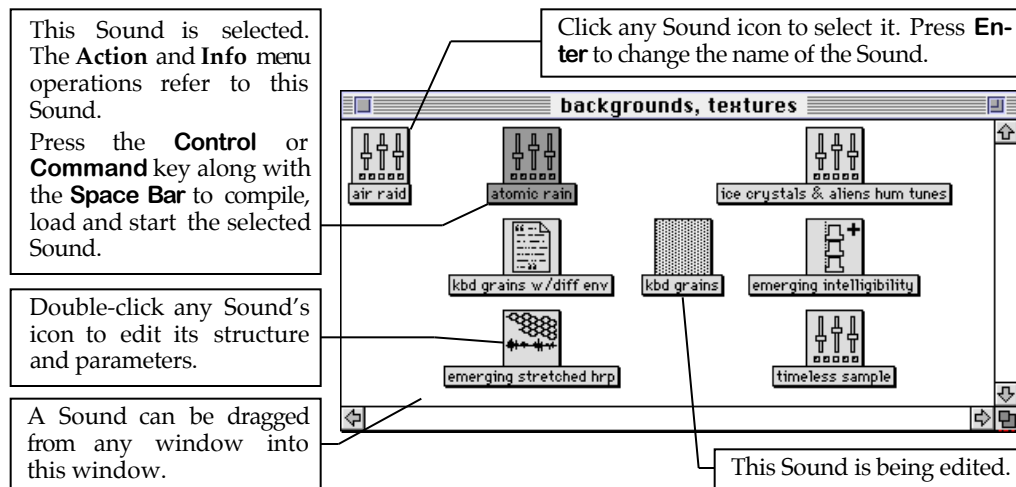
Unlike a *Sample*, which plays through its wavetable once each time you trigger it, an *Oscillator* reads through its wavetable cyclically, over and over again. Depending on the **Frequency** you specify, an *Oscillator* will step through the wavetable using larger or smaller increments; this increases or decreases the rate at which you hear the waveform repeating, thus changing the perceived pitch of the oscillator.

Kyma Quick Reference

System Prototypes Window



Sound File Window



Sound Editor Window

This is the signal flow diagram of the Sound being edited.

A stereo signal travels along each line from left to right.

A thick line with a number along it indicates that a signal is used more than once.

Click any Sound's icon to select it. Press Enter to change the name of the Sound.

Drag this line up or down to control the space devoted to the signal flow relative to the parameters.

These are the parameters of the Sound double-clicked in the signal flow diagram.

Click the close box to close the editor. Use Save from the File menu to save to disk changes made in this editor.

Double-clicking a Sound in the signal flow diagram causes the Sound's parameters to be shown below, and causes its icon to be replaced with this pattern.

The **Action** and **Info** menu operations refer to the selected Sound.

Control or **Command** with **Space Bar** compiles, loads, starts the selected Sound.

The signal flow diagram shows a sequence of processing blocks: a 'down' block, an 'osc17' block, an 'expUp' block, two 'rain cloud' blocks (rain cloud1 and rain cloud2), a 'Mixer19' block, and finally the 'atomic rain' output block. A thick line with the number '5' connects the 'expUp' block to the 'rain cloud1' block, indicating multiple signal paths.

The parameter panel for the selected sound 'atomic rain' includes the following settings:

Duration	Amplitude	Frequency	FreqJitter
ON	0.4 - ([down] L * 0.24)	[expUp] L * 533 hz	1 ramp: 60 s
Waveform	GrainEnv	GrainDur	CyclesPerGrain
sine	gaussian	[GrainDur] s	5 + [osc17] L
Seed	Pan	GrainDurJitter	PanJitter
1	0.4	0	0
MaxGrains	Density		
28	[down] L * 0.8 + 0.025		

Grain Cloud

Working with the Signal Flow Diagram

Click on the icon's tab to show hidden inputs. Clicking again will hide the inputs.

Click and drag an icon to reposition it. Hold down the **Shift** key to move the Sound's inputs at the same time.

To insert a Sound into the signal flow, drag the Sound onto a line connecting two Sounds.

The black border added to this field indicates that it is the *active field*. The **Edit** menu operations always refer to the active field. Click in a field to make it active.

To replace a Sound with a *copy* of another Sound, drag the other Sound on top of an icon. Hold down the **Control** or **Option** key to replace a Sound with the other Sound without making a copy. Alternatively, use **Paste** or **Paste special...** from the **Edit** menu.

To delete a Sound, click on its icon to select it, then press **Delete** or choose **Clear** from the **Edit** menu. **Undo** from the **Edit** menu will undo the deletion.

The signal flow diagram is identical to the one in the previous section, showing the sequence of processing blocks and the thick line with the number '5'.

The parameter panel for the selected sound 'atomic rain' is also identical to the one in the previous section, showing the same settings for Duration, Amplitude, Frequency, FreqJitter, Waveform, GrainEnv, GrainDur, CyclesPerGrain, Seed, Pan, GrainDurJitter, PanJitter, MaxGrains, and Density.

Grain Cloud

Working with the Parameter Fields of a Sound

This Sound is being edited. Double-click a different Sound to edit its parameters.

A parameter field with a white background and non-italic name is a *constant parameter*. Constant parameters have a fixed value for the duration of the Sound.

The duration of a Sound is how long it runs on the Capybara, and is found by taking the longest duration of any input or parameter of the Sound.

Many Sounds have a **Duration** parameter. Use ON (or on) to keep a Sound turned on indefinitely. Or, use a value with units, for example, 10 s or 1.5 h.

A parameter field with a cyan background and italic name is a *hot parameter*. Hot parameters can be altered while the Sound is playing.

This is the name of one of the Sound's parameters. Click on the name for help on how to set the parameter.

This is the kind of Sound being edited. Click here for a description of the Sound type and its parameters.

Click this button to choose a file from a standard file dialog. Hold down the **Control** or **Command** key while clicking the button to open an editor on the file.

Hot parameter fields can contain Event Values controlled by MIDI or the Virtual Control Surface. Event Values can be inserted by hitting the **Escape** key and moving a MIDI controller or playing the MIDI keyboard, choosing **Paste hot...** from the **Edit** menu, or by typing an exclamation point (!) followed by the name of the Event Value.

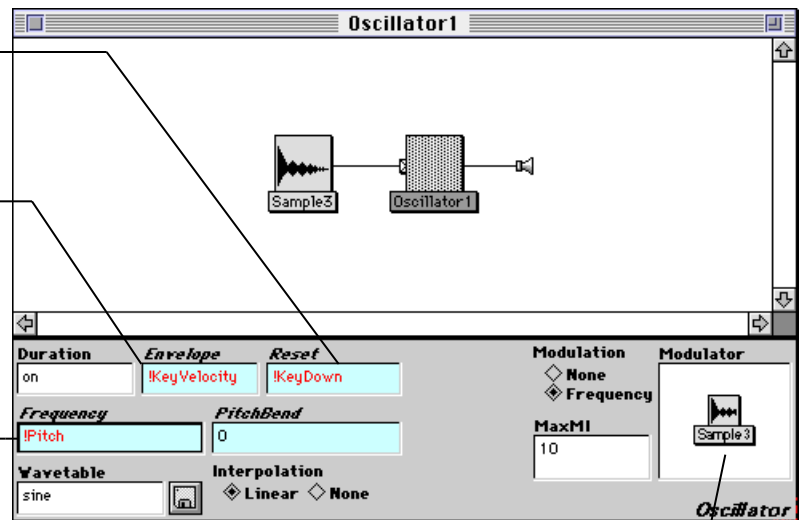
Hot parameter fields can contain Sounds. To insert a Sound into a hot parameter field, **Copy** the Sound, and then **Paste** it into the field. Use L for left, R for right, or M for mono mix of the channels.

If a parameter field flashes when you try to play the Sound, it indicates an error in the value or expression in the field.

Can't see everything in a parameter field? Choose **Large window...** from the **Edit** menu to expand the field to full screen size.

A parameter field can be set to !Pitch, !KeyDown, or !KeyVelocity by hitting **Escape** and playing 1, 2, or 3 keys simultaneously on the MIDI keyboard.

The black border added to this field indicates that it is the *active field*. The **Edit** menu operations always refer to the active field. Click in a field to make it active. Alternatively, press **Tab** to advance the active field through each of the fields.



Signal inputs to a Sound appear in both the signal flow diagram and in the Sound's parameters. To change the input, **Cut**, **Copy**, **Paste**, or drag Sounds into the parameter field, or onto the icon of the old input in the signal flow diagram. The signal flow diagram will not be updated until you double-click a different Sound or double-click in the background of the signal flow diagram.

Specifying Units in Parameter Fields

Example	Meaning
on	about 2 years
3 days	3 days (4320 minutes)
2.1 h	2.1 hours (126 minutes)
3 m	3 minutes
4.7 s	4.7 seconds
100 ms	100 milliseconds (0.1 seconds)
100 usec	100 microseconds (0.0001 seconds)
5 samp	5 samples
4 beats	4 beats at current value of MM
01:37:42.7 SMPTE	1 hour, 37 minutes, 42 seconds, and 7 frames
440 hz	440 hertz
60 nn	MIDI note number 60 (middle C)
4 c	note number 60
4 c sharp	note number 61
4 c flat	note number 59
4 do	note 60 (from <i>solfege</i> scale: do, re, mi, fa, so, la, ti)
default	“natural” duration or frequency of sample file
4 c removeUnits	60, the value with units removed

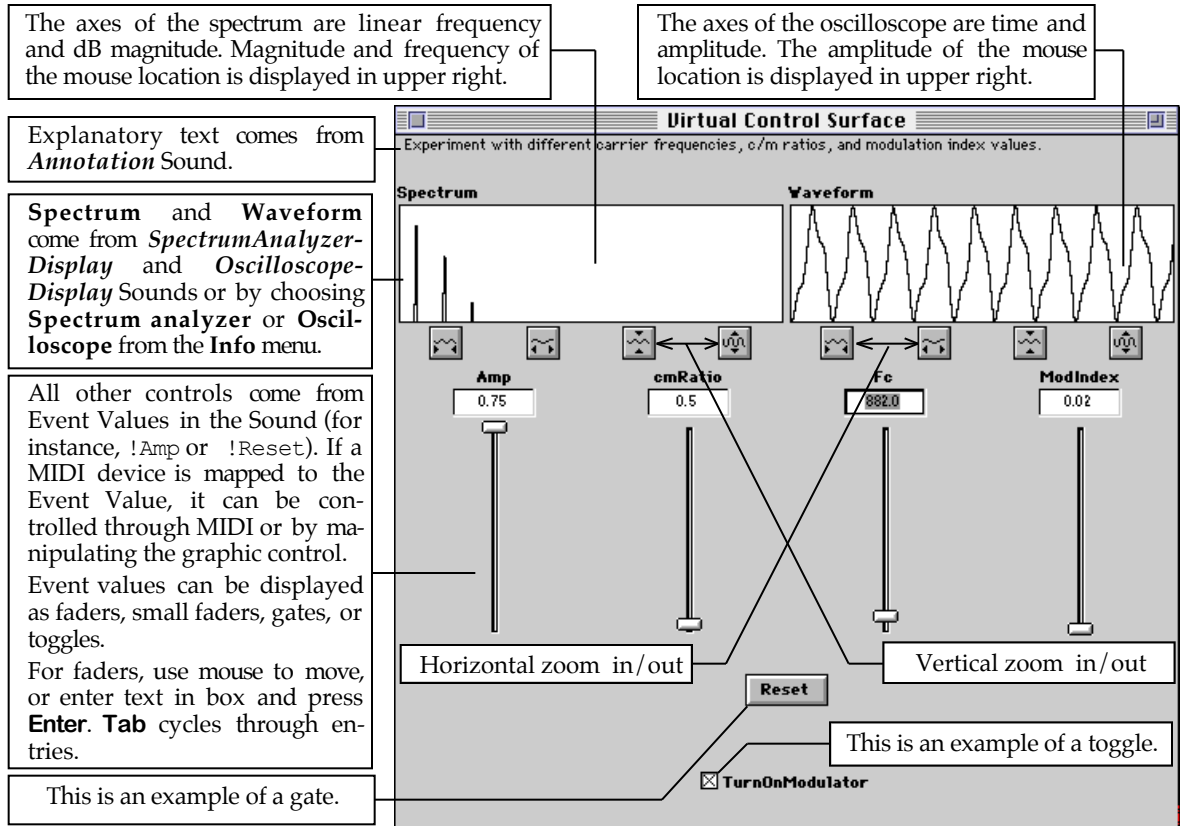
Time Functions in Parameter Fields

Message	Explanation	Example
ramp	ramp from 0 to 1 over 1 second when triggered	!KeyDown ramp
ramp:	same as ramp:, except use given duration	!KeyDown ramp: 10 s
repeatingRamp	same as ramp, except repeats until trigger turns off	!KeyDown repeatingRamp
repeatingRamp:	same as ramp:, except repeats until trigger off	!KeyDown repeatingRamp: 2.3 s
fullRamp	ramp from -1 to 1 over 1 second when triggered	!KeyDown fullRamp
fullRamp:	same as fullRamp:, except use given duration	!KeyDown fullRamp: 10 s
repeatingFullRamp	same as fullRamp, except repeats until trigger off	!KeyDown repeatingFullRamp
repeatingFullRamp:	same as fullRamp:, except repeats until trigger off	!KeyDown repeatingFullRamp: 2.3 s
bpm:	periodic trigger output at given rate when trigger on	!KeyDown bpm: 60
bpm:dutyCycle:	same as bpm:, except output duty cycle is given	!KeyDown bpm: 30 dutyCycle: 0.60
random	random numbers generated at given time intervals	1 s random
nextRandom	random number generated on each trigger	!KeyDown nextRandom
smooth:	linearly ramp to new value over the given duration	!cc01 smooth: 0.5 s
smoothed	same as smooth: with an argument of 100 ms	!cc01 smoothed

Real-Time Expressions in Parameter Fields

Message	Explanation	Example	Value
+	addition	3.0 + 2.0	5.0
-	subtraction	3.0 - 2.0	1.0
*	multiplication	3.0 * 2.0	6.0
/	division	3.0 / 2.0	1.5
//	truncating division	3.0 // 2.0	1
mod:	modulo	3.0 mod: 2.0	1
negated	additive inverse	3.0 negated	-3.0
inverse	multiplicative inverse	3.0 inverse	0.3333333
**	exponentiation	3.0 ** 2.0	9.0
sqrt	square root	2.0 sqrt	1.4142
exp	powers of e	1 exp	2.71828
twoExp	powers of 2	3 twoExp	8
log	logarithm base 10	100 log	2.0
twoLog	logarithm base 2	8 twoLog	3.0
db	decibel conversion	-20 db, -6 db	0.1, 0.5
inSOSPitch	SOS pitch conversion	11025.0 inSOSPitch	0.5
truncated	truncation	3.2 truncated, -3.6 truncated	3, 4
rounded	round to nearest integer	3.2 rounded, 3.6 rounded	3, 4
abs	absolute value	-3.0 abs, 3 abs	3, 3
sign	sign	-10 sign, 0 sign, 10 sign	-1, 0, 1
clipTo01	clip to lie in (0, 1) interval	-5 clipTo01, 0.5 clipTo01, 1.2 clipTo01	0, 0.5, 1
vmin:	select minimum value	3 vmin: 2	2
vmax:	select maximum value	3 vmax: 2	3
cos	cosine in radians	3.14159 cos	-1
normCos	$\cos(x)$	1 normCos	-1
sin	sine in radians	3.14159 sin	0
normSin	$\sin(x)$	1 normSin	0
asLogicValue	0 is false, 1 is true	-1 asLogicValue, 2 asLogicValue	0, 1
eq:	equal	2 eq: 3	0
ne:	not equal	2 ne: 3	1
gt:	greater than	2 gt: 3	0
lt:	less than	2 lt: 3	1
ge:	greater or equal	2 ge: 3	0
le:	less or equal	2 le: 3	1
neg:zero:pos:	choose value	-3 neg: 0.1 zero: 0.2 pos: 0.3	0.1
true:false:	conditional evaluation	0 true: 2 twoExp false: 3 twoExp	8
of:	array indexing	3 of: #(0.1 0.2 0.3 0.4 0.5 0.6)	0.4

Virtual Control Surface



Global and Local Maps

```
!EventValue is: `EventSource.

!EventValue is: (`EventSource channel: 2).

!EventValue is: (`EventSource min: 100 max: 500).

!EventValue is: (`EventSource min: 0 max: 10 grid: 2).

!EventValue is: (`EventSource taper: #log).

!EventValue is: (`EventSource displayAs: #smallFader).

!EventValue is:
  (`EventSource
    channel: 2;
    min: 100 max: 500 grid: 0.1;
    taper: #log;
    displayAs: #smallFader).
```

Use EventValue as the name of EventSource.

Specify that EventSource comes from MIDI channel 2. The channel must be between 1 and 16. If this option is omitted, the default MIDI channel will be used.

Scale the (0,1) range of EventSource to (100,500).

Constrain the allowable values of EventSource to multiples of 2 between 0 and 10.

Map the (0,1) range of EventSource with a log taper. Either #log or #linear can be used. If this option is omitted, #linear will be used.

Show EventValue as a small fader in the virtual control surface. The possible display types are #fader, #smallFader, #gate, #toggle, or #nothing. If this option is omitted, #fader will be used.

Combine any of the above by separating them with a semicolon.

See *Sources of Event Values* on page 473 for a list of allowable event sources.

MIDI Script Messages

Example	Remarks
<code>self keyDownAt: 1 s duration: 0.25 s frequency: 100 hz velocity: 0.8.</code>	Fully specified note event: !Key-Down is 1 at 1 s and 0 at 1.25 s, !KeyNumber is set to 100 hz nn removeUnits, !KeyVelocity is set to 0.8
<code>self keyDownAt: 1 s duration: 0.25 s frequency: 4 c sharp.</code>	!KeyVelocity defaults to 1
<code>self keyDownAt: 1 s frequency: 60.2 nn duration: 0.25 s.</code>	Alternate order of line above
<code>self keyDownAt: 1 s duration: 0.25 s velocity: 0.8.</code>	!KeyNumber defaults to mapper's LowPitch parameter
<code>self keyDownAt: 1 s duration: 0.25 s.</code>	!KeyNumber defaults to mapper's LowPitch parameter, !Key-Velocity defaults to 1
<code>self keyDownAt: 1 s.</code>	!KeyNumber defaults to mapper's LowPitch parameter, !Key-Velocity defaults to 1, duration defaults to 10 ms
<code>self controller: !Volume setTo: 0.5 atTime: 5 s.</code>	!Volume jumps to 0.5 at 5 s
<code>self controller: !Volume slideTo: 1 byTime: 6 s.</code>	!Volume slides to 1 by 6 s in 10 steps
<code>self controller: !Volume slideTo: 0.75 steps: 100 byTime: 10 s.</code>	!Volume slides to 0.75 by 10 s in 100 steps
<code>self controller: !Volume slideTo: 0 stepSize: 0.01 byTime: 20 s.</code>	!Volume slides to 0 by 20 s in steps of size 0.01

See *MIDI Scripts* on page 522 for information about **EventCollections**.

File Organizer

Files and folders are listed here with color coding.

Underline: open folder

Gray: unopened folder

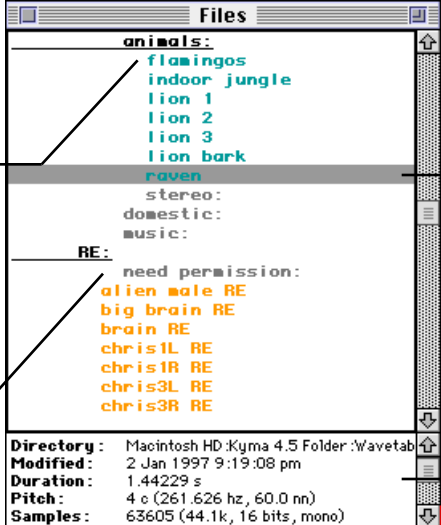
Turquoise: sample file

Purple: spectrum file

Red-brown: GA file

Yellow-orange: RE file

Green: MIDI file



Click file to select it. Use arrow keys to move selection up and down within the window. Double-click file to edit it. Press **Control** or **Command** along with **Space Bar** to hear the file. Drag the file into a Sound file window to create a Sound that uses this file.

Double-click a folder to show or hide the contents of the folder.

Information on the selected file or folder.

Spectrum Editor

Overview

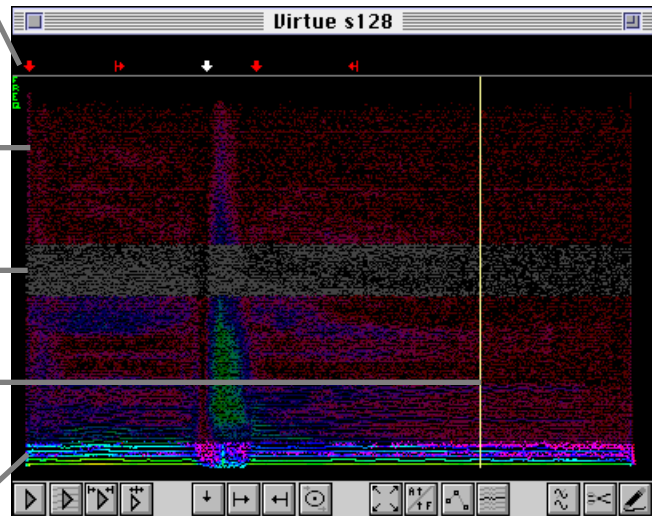
Markers indicate labeled time points, and time segments for cutting or auditioning.

Each line represents a sine wave *track*. Color indicates the amplitude, vertical position indicates the frequency.

Gray tracks have zeroed (cleared) amplitudes.

Scrub bar can be moved with the mouse, with pitch-bend from MIDI, or left or right arrow keys.

Brightly colored tracks are selected.



Navigating and Selecting

Select by clicking on individual tracks, or by drawing a box around a region to select. Hold the shift key to add to or remove from the current selection.

Hold down the **Control** or **Command** key while drawing a box to zoom in on a specific region.

Plays selection outside of start and end markers.

Plays selection between start and end markers.

Plays selection.

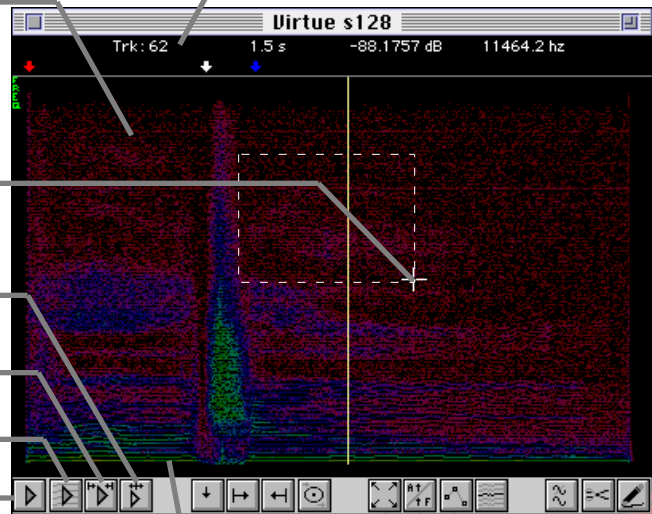
Plays entire spectrum.

Zoom all the way out.

Switches between time-frequency and time-amplitude display modes.

Switches between isolated-dot and connected-dot between spectral frames.

Information is displayed here about the track under the crosshair cursor.



Select track between 1 and 10 by pressing a number key (1-9, 0). Up and down arrow keys select next larger or smaller track.

Markers

Selected marker's time and name are displayed here.

Click marker to select it. Scrub bar moves to marker when selected. Press **Enter** to rename marker; use **Delete** or **Cut** from **Edit** menu to remove marker.

Sets pre- and post-roll play times.

Creates end marker at scrub bar location.

Creates start marker at scrub bar location.

Creates marker at scrub bar location.

Start and end markers.

The screenshot shows the Virtue s128 software interface. At the top, a title bar reads 'Virtue s128'. Below it is a time display showing '0.7 s'. The main area is a spectrogram with a color-coded selection. A vertical line (scrub bar) is positioned at 0.7 seconds. Red arrows on the top edge indicate start and end markers. A toolbar at the bottom contains various icons for editing and playback.

Editing

Zero the amplitude of the selection by choosing **Clear** from **Edit** menu.

Create a Sound to play the selection by choosing **Copy** from **Edit** menu, then **Paste** into Sound file window or Sound editor.

Switch in and out of drawing mode.

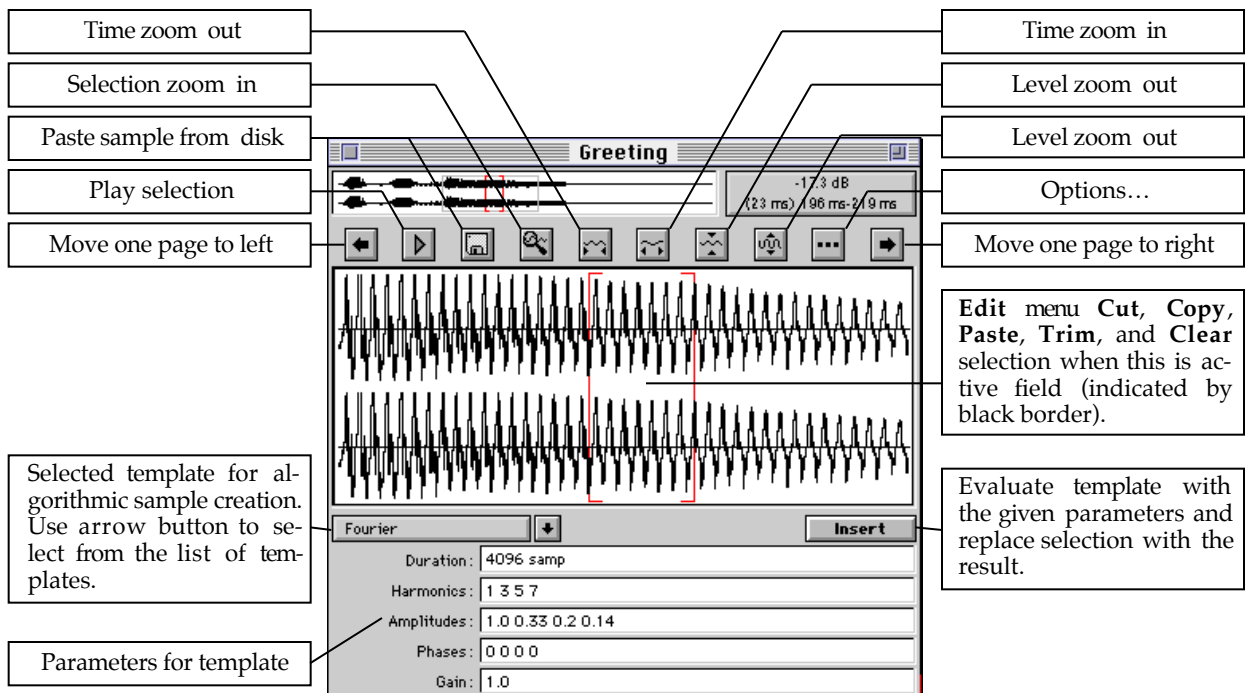
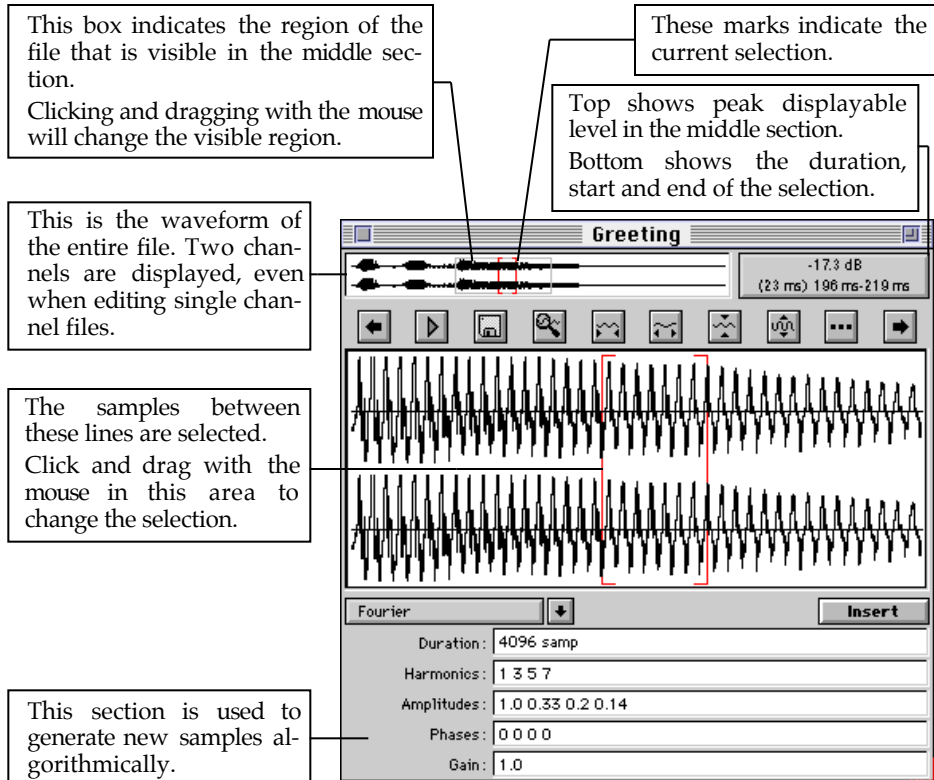
Cut the time between the start and end markers.

Modify the selection in various ways.

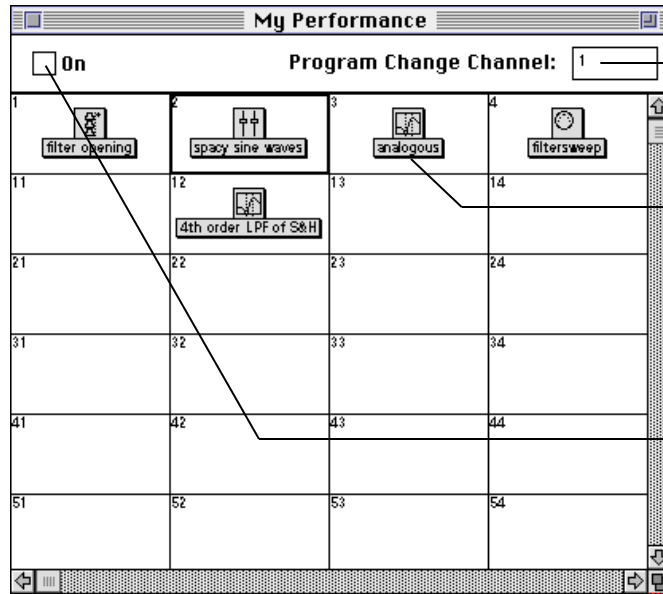
Select tracks and frames based on various criteria.

This screenshot is identical to the one in the 'Markers' section, showing the Virtue s128 software interface with a spectrogram and various editing callouts. The title bar, time display, spectrogram, and toolbar are all present and labeled with the same callouts as in the previous image.

Sample File Editor



Compiled Sound Grid



Enter MIDI channel on which to receive program change messages and press **Enter**.

Drag Sound into square to associate it with the MIDI program number in upper left. Double-click to edit the Sound. Press **Enter** to rename the Sound.

Choose **Compile to disk...** from the **Action** menu to precompile the Sounds in the grid. Use **On** to start and stop the grid.

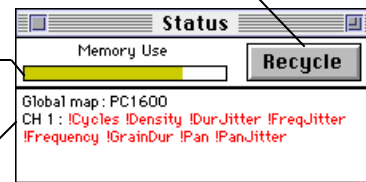
When **On** is checked, click in a square or send a MIDI program change message to load and start the Sound.

Status from the File Menu

Click **Recycle** if Memory Use gets too high (bar is colored red).

Indicates amount of memory in use.

Indicates global map, A/D usage, the MIDI channels and Event Values in use by last Sound loaded.



Status from the DSP Menu

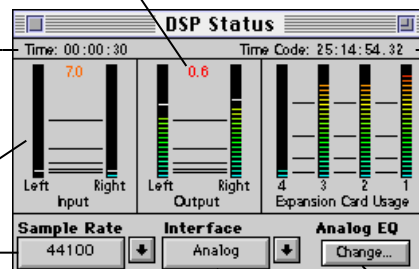
Output peak level indicator. Click mouse in window to clear. Possible clipping if it shows >0<. Input has a peak level indicator as well.

Time since last Sound was started. Does not update after Sound ends or when DSP is stopped.

Input and output level indicators. Each line is 10 dB.

Choose a sample rate from the pull down list.

Choose analog or digital audio interface from the pull down list.

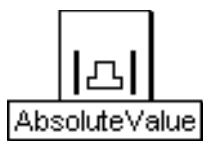


Last time received in MIDI time code messages received by DSP MIDI input.

Approximate measure of current computational load on each expansion card. Each line indicates approximately 20% usage.

Click this button to configure the equalizer on the analog output.

Prototypes Reference



AbsoluteValue

Math Category

The output of an AbsoluteValue is the absolute value of its Input. (Choose Full waveform from the Info menu to see that the output waveform contains only positive values, rather than values above and below zero).

Input

This is the Sound whose absolute value is taken.



ADSR

Envelopes & Control Signals Category

Generates a traditional four-segment envelope with attack, initial decay, sustain, and release segments. As long as Gate is 0, no envelope is generated. When Gate becomes positive, the attack and decay are generated. The envelope continues to decay towards the SustainLevel until the Gate value returns to zero at which time the envelope enters its release portion.

In a prototype with an Envelope parameter field (Oscillator, for example) you can use the ADSR as the Envelope parameter. Envelope generators can also be used to control other parameters (such as Frequency or OnDuration). To apply an envelope to any Sound, use the Sound and an envelope generator as Inputs to the VCA prototype. (The VCA simply multiplies its two inputs by each other).

AttackTime

Time for the envelope to reach the maximum amplitude (attenuated by Scale) once it has been triggered.

DecayTime

Time for the envelope to decay from the maximum level down to the SustainLevel.

SustainLevel

Level that will be sustained for as long as Gate remains nonzero.

ReleaseTime

Time for envelope to decay from SustainLevel down to 0.

Type

Choose between linear envelope segments or exponential segments.

Scale

Overall level for the envelope. IF you have the Linear box checked, you can set Scale to a negative value to generate envelopes that go from 0 down to a negative value. (Note that this makes most sense when using the ADSR to control parameters other than amplitude, e.g. when using this as a pitch envelope).

Gate

Enter a 1 in this field to make the envelope last exactly as long as the Sound is on.

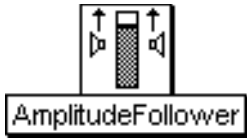
If you use an EventValue (for example, !KeyDown) in this field, the envelope can be retriggered as often as you like for as long as this Sound is on.

When Gate becomes positive, the envelope is started; when Gate becomes zero, the envelope is released.

Legato

Legato affects the behavior of the envelope when triggered by the Gate field.

When Legato has a zero or negative value, the envelope will rapidly reset to zero before beginning the re-attack. When Legato has a positive value, the attack will begin from the current envelope value (without first resetting to zero).



AmplitudeFollower

Tracking Live Input Category

Follows the amplitude of the Input by taking an average of the absolute values of individual input samples. This is similar to the RMS but requires less computation. TimeConstant controls how long the average runs; thus, the longer the TimeConstant, the smoother the output (but the less quickly it can respond to transients in the Input).

Input

This is the Sound whose amplitude is tracked.

TimeConstant

This controls the response time. Longer timeConstants result in smoother outputs at a cost of losing some of the detail in the attacks. Short timeConstants result in outputs that respond more immediately to attack transients but that may not be as smooth for the steady state portions. For a constant input at maximum amplitude, this is the time required for the output to reach 60% of the full output amplitude. (Note that the output may never reach the maximum possible amplitude since it is the average of the squares of the amplitudes).

Scale

Attenuates the input amplitude.



AnalogSequencer

Sequencers Category

Generates sequences of note events and continuous controller values for EventValues in parameters of the Input. There is a sequence of MIDI notenumbers, durations, duty cycles and velocities to supply MIDI note events to the Input, and ExtraValues lets you supply a sequence of values for any continuous controller EventValues in the Input's parameters.

The length of all sequences is the length of the longest sequence; any shorter sequences will repeat their final values in order to be as long as the longest sequence. For example, if you want all values in a sequence to be 0.5, you need only enter the number a single time, because it will be repeated for as many times as necessary to make it as long as the other sequences.

If Step is a constant 1, then the Durations and the DutyCycles are used to determine when !KeyDown events should be generated and how long each key should remain down. If no other units of time are used and if the value of Rate is 1, the numbers in the Durations sequence are interpreted in seconds. Rate is a divisor on the length of each duration, so if Rate is greater than 1, the durations will be shorter, and if Rate is less than 1, they will be longer.

Use Step to step through the sequences one by one, according to a trigger. For example, you could use !KeyDown to control the step rate from a MIDI sequencer, or you could use 1 bpm: (!Speed * 1024) to control it with an internal metronome, or you could paste in an audio signal that has been passed through an amplitude follower and a threshold to trigger each element of the sequences in synch with an audio signal, or you could use !TimingClock to step using the MIDI clock from a software sequencer or external synthesizer.

To trigger using the Step field, you should set all Durations to the same value; this value (taken along with the value of DutyCycle and Rate) will determine the duration of each note, and the Step trigger will specify the onset time of the note. The value you pick for the Durations will act as a kind of "mask" on the speed of the step triggers. Notes cannot be triggered any more often than the minimum values specified in the Durations field. This can be helpful if you are triggering from the !TimingClock (24 triggers per beat) or an audio signal with lots of peaks in it, because it will force the sequencer to ignore triggers that occur faster than at the desired rate.

Input

EventValues anywhere in this Sound can be controlled by the arrays of key events or continuous controller values sequenced by this AnalogSequencer.

Left

Attenuator on the left channel amplitude.

Right

Attenuator on the right channel amplitude.

Gate

When this value changes from 0 to 1, it restarts the sequences. If Loop is checked, the sequences will repeat for as long as Trigger is nonzero.

Use a constant value of 1 to get an infinitely repeating sequence. Use an EventValue such as !KeyDown to restart and stop the sequence interactively.

Step

Step to the next set of values in the sequence when this value changes from a zero to a number greater than zero. If Step is changing faster than the specified durations, some of the steps will be ignored. In other words, you can use the sequence of durations as a kind of mask, constraining the minimum durations to be at least those specified in the duration sequence.

Rate

This is the rate at which the sequences are traversed. When Rate is 1, all Durations are interpreted as seconds, when Rate is 2, the Durations are half as long, and when Rate is 0.5 the Durations are twice as long.

Loop

Check this box to loop back to the StartIndex once the EndIndex has been reached.

StartIndex

This is the starting position of the sequence. Each position in the sequence is numbered from 0 up to the length of the sequence minus 1.

EndIndex

This is the ending position of the sequence. Each position in the sequence is numbered from 0 up to the length of the sequence minus 1.

Polyphony

The sequencer is monophonic, but this allows some overlap between the release of one event and the attack of the next. Polyphony specifies how many events can be heard overlapping each other at any one time.

Durations

A sequence of durations. If no units are used, the numbers are assumed to be in seconds. Each Duration, in conjunction with the corresponding DutyCycle, is used to determine when to send each !KeyDown and how long to keep it down.

DutyCycles

A sequence of duty cycles where the duty cycle is the fraction of the beat during which the note is on. For example, a duty cycle of 0.5 means that the note is on for half the beat and off for the second half of the beat. A duty cycle of 0 would mean that the note is never on, and a duty cycle of 1 would mean that the note is continuously on and never turns off.

Pitches

A sequence of MIDI notenumbers that will supply the pitches to any !Pitch or !KeyNumber EventValues in the parameters of the Input.

Velocities

A sequence of values between 0 and 1 that will supply the values to any !KeyVelocity EventValues in the parameters of the Input.

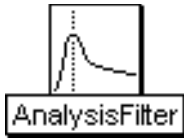
ExtraValues

Use this field to specify a sequence of changes for any non-keyboard EventValues in the Input. The syntax is:

#(<!EventValue> <val1> <val2>, ..., <valn>)

For example, to send a sequence of 3 values for !Morph and for !Pan, you would use something like the following:

#!Morph 0 0.25 1)
#!Pan 0 0.5 1)



AnalysisFilter

Tracking Live Input Category

Bandpass filter designed to isolate the individual harmonics of its Input. It has quadrature output (i.e. the left channel output is the cosine part of the Input signal at that frequency; the right channel is the sine part of the Input at that frequency); thus you can use a QuadratureOscillator to frequency shift this harmonic using single side band ring modulation.

The filter is designed such that, if you were to add together the output of filters set at each harmonic from 0 to the harmonic closest to half the sampling rate, the amplitude of that sum would be 1.

Input

This is the Sound that gets filtered.

Fundamental

This is the assumed fundamental frequency. In most cases, you should set it to be the same as the Input frequency.

Harmonic

This is the number of the harmonic that the filter will try to isolate. In other words, the center frequency of the filter will be the Fundamental * this Harmonic.



Annotation

Variables & Annotation Category

An Annotation contains a Text commenting on its Input. It does not affect the sound of the Input in any way; it is just a comment. The Text of an Annotation shows up in the Virtual control surface whenever the Sound is loaded. (The text of any Annotations that occur in Sounds to the left of this one will appear below this one's Text in the Virtual control surface).

Input

The Text refers to this Sound.

Text

This is a textual description of the Input.



AR

Envelopes & Control Signals Category

Generates an envelope with the specified attack and release times with either linear or exponential segments.

In a prototype with an Envelope parameter field (Oscillator, for example) you can use an envelope generator directly as the Envelope parameter. Envelope generators can also be used to control other parameters (such as Frequency or OnDuration). To apply an envelope to any Sound, use the Sound and an envelope generator as Inputs to the VCA prototype. (The VCA simply multiplies its two inputs by each other).

AttackTime

Time required for the envelope to reach its maximum value (as attenuated by Scale) whenever its Gate value becomes positive.

ReleaseTime

Time it takes for the envelope to return to 0 once Gate changes from a positive number back to zero.

Type

Choose between linear and exponential segment shapes.

Scale

Overall attenuation on the envelope generator values.

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released.



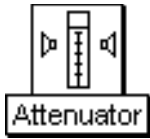
ArcTan

Math Category

The output of ArcTan is the four-quadrant arctangent of the ratio of the right channel input to the left channel input.

Input

This is the Sound whose arctangent is taken.



Attenuator

Level, Compression, Expansion Category

An Attenuator multiplies the left and right channels of its Input by the values in Left and Right. It can be used to "place" the Input between the speakers; for example, if the Input is multiplied by 1.0 in the right channel and by 0 in the left channel, it will seem as if the source of the Input is located to the right of the listener.

Input

This is the Sound whose left and right channels will be attenuated.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



AudioInput

Sampling Category

An AudioInput represents the analog or digital inputs on the back of the Capybara. If Digital Input is selected in the DSP Status window, then this represents the digital input.

If there are (preamplified) microphones connected to the inputs then this Sound represents the input from those microphones. You can also connect the audio outputs of some other line-level sound generator (like a CD or DAT) to the input.

The individual channel check boxes control which audio input channel(s) will be used. If only one channel is checked, it will be output on both channels of the AudioInput.

Channel11

Check this box to use the channel 1 audio input of the signal processor.

Channel12

Check this box to use the channel 2 audio input of the signal processor.

Channel13

Check this box to use the channel 3 audio input of the signal processor.

Channel14

Check this box to use the channel 4 audio input of the signal processor.

Channel15

Check this box to use the channel 5 audio input of the signal processor.

Channel16

Check this box to use the channel 6 audio input of the signal processor.

Channel17

Check this box to use the channel 7 audio input of the signal processor.

Channel18

Check this box to use the channel 8 audio input of the signal processor.



AveragingLowPassFilter

Filters Category

Low-pass filter that operates by taking a running average of the stream of Input values. The length of the running average is the period of the cutoff frequency. The Cutoff frequency and its harmonics are cancelled out by the filter (and frequencies close to the cancelled frequencies are attenuated).

Input

This is the Sound that is to be low-pass filtered.

Cutoff

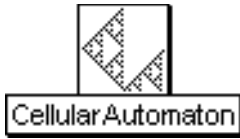
Frequencies above the cutoff will be attenuated by the filter.

Wavetable

This is the wavetable that is used to keep the running average of the last few samples. Select Private if you want the next free wavetable and do not need to reference this same segment of memory again. (If you want to reference this same memory segment from another Sound, type in a unique name for the wavetable and use that same name when accessing the memory from the other Sound.)

Scale

This is the attenuation on the input. For the full amplitude use +1.0 or -1.0 (or 0 dB); any factor whose absolute value is less than 1 will attenuate the output.



CellularAutomaton

Scripts Category

This Sound is based on the one-dimensional cellular automata described by Stephen Wolfram in his book, *Theory and Applications of Cellular Automata*. The state, an n-place binary number, where n is the number of Inputs, determines which of the Inputs is turned on and which is turned off in a given generation. An integer, rule, is used to determine the next state.

Inputs

Each Sound is associated with a position in the state array, starting with position 1 at the upper left and moving from top to bottom and left to right. These Sounds form a sort of "chord" in which individual Sounds are turned on or turned off according to whether they correspond to a position in the state array that contains a 1 or a position that contains a 0.

Rule

If you look at the rule and the state as 8-bit binary numbers, you can use the rule to compute the next generation of the state. Number the positions of the digits 0 to 7 counting from right to left. To compute the next generation of position P of the state: Take the number at position P, P+1 and P-1 (i.e. the number at position P and its neighbors to the right and to the left). Think of this as a three digit binary number N that you are going to use as an index into the binary number representing the rule. Look at position N of the rule. The value at position N of the rule is the value of position P in the next generation of the state.

For example, if the rule is 2r10101110 and the state is 2r10010101, let's compute the next generation state of position 3. The current value at position 3 of the state is '0'. Looking at position 3 and its two neighbors as a binary number, we get '101' or the number 5. Using this number as an index, look at position 5 of the rule; it is '1'. So the value of position three in the next generation is '1'. When you reach the rightmost or leftmost digit in the state, make the assumption that there is an infinite number of zeroes extending both leftwards and rightwards.

Iterations

The value of iterations is the number of generations.

InitialState

Imagine the initial state as an n-bit binary number where n is the size of the collection of inputs. Each digit in the binary number corresponds to an input; use a 1 to indicate that a Sound is initially on, a 0 to indicate that it is initially off.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

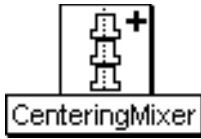
You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



CenteringMixer

Mixing & Panning Category

Lines up all of the Inputs' duration midpoints with the longest Input's midpoint and outputs the mix of all the Inputs. The "midpoint" is defined as half the duration of the longest Input.

Inputs

The duration midpoint of each of these Sounds occurs at the same time.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

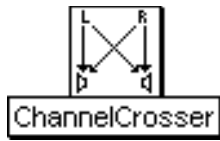
You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



ChannelCrossover

Mixing & Panning Category

A Crossover lets you switch any portion of the left channel signal into the right channel and vice versa.

Input

The left and right channels of this Sound can be attenuated and mixed using the sliders.

LeftInLeft

This is the portion of the left input that appears in the left output.

RightInLeft

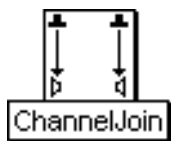
This is the portion of the right input that appears in the left output.

LeftInRight

This is the portion of the left input that appears in the right output.

RightInRight

This is the portion of the right input that appears in the right output.



ChannelJoin

Mixing & Panning Category

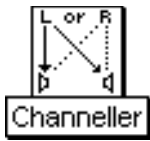
This Sound places the left channel of Left into the left output channel and the right channel of Right into the right output channel.

Left

The left channel of this Sound is output to the left channel.

Right

The right channel of this Sound is output to the right channel.



Channeller

Mixing & Panning Category

If LeftChannel is checked, the left channel of Input is output on both channels.

If RightChannel is checked, the right channel of Input is output on both channels.

If both are checked, the Input is passed through unchanged.

If neither is checked, there will be no output.

Input

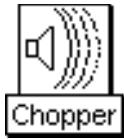
Either the left or right channel of this Sound will be output.

LeftChannel

If only this control is checked, the left channel of Input will be output on both channels.

RightChannel

If only this control is checked, the right channel of Input will be output on both channels.



Chopper

Envelopes & Control Signals Category

Multiplies the Input by a stream of "grains" or envelopes (one cycle of the selected wavetable), each lasting for GrainDuration with InterGrainDelay silence in between. During the InterGrainDelay the Input is multiplied by zero.

Input

This Sound is multiplied by an alternating stream of grains and inter-grain silences.

GrainDuration

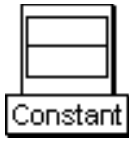
This is the duration of each grain. (Duration should always be greater than 0.)

InterGrainDelay

This is the delay time between grains. (Duration should always be greater than 0.)

Wavetable

One cycle of the selected wavetable will be used as the envelope of each grain.



Constant

Math Category

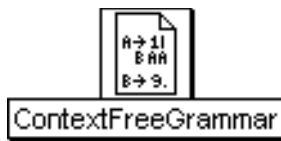
The output of a Constant is its Value. If Value is set to a number, the output of the Constant is the same number over its entire duration.

If you paste an Event Value into the Value field of a Constant, the Constant's output is equal to the Event Value. This is useful for processing Event Values as if they were Sounds. For example, if you were to paste !cc07 into the Value field of this Constant, you could then feed the Constant into a delay, put it into a waveshaper, multiply it by a sine wave oscillator, or perform any number of other signal processing operations on it.

Value

Enter a value from -1.0 to 1.0. You can also paste Event Values or Sounds into this field (since their values fall within the range of -1 to 1).

If you paste !Pitch or !KeyNumber into this field, you must divide it by 127 in order to scale it down to the range of 0 to 1; if you then use this Constant in the Frequency field of another Sound, remember to multiply the Constant by 127 nn in order to scale it back into the range of 0 nn to 127 nn.



ContextFreeGrammar

Scripts Category

A Sound consisting of Concatenations and CenteringMixers of the Inputs is generated from the startExpression by rewriting according to the production rules of a context-free grammar. A seed is used for repeatable results.

Inputs

These Sounds act as the terminals of the production rules; in order from top to bottom, left to right they are referred to in the productions as s1, s2, ..., sn.

Drag a folder or any number of individual Sounds into this field.

RewriteRules

Production rules should be of the following form,

Variable -> option { | option }*.

A single uppercase letter is followed by -> and then one or more options separated by the delimiter, |. Each option consists of a fully parenthesized expression followed by a number in brackets indicating the relative weighting of that choice. Each expression should consist of binary combinations of Sounds, s1 - sn where n is the number of Inputs, and Variables (single uppercase letters), separated by a comma for Concatenation or plus for a CenteringMixer. For example, the following production rule will always generate palindromes:

```
A -> ((s1 , A) , (s1)) [2] |
      ((s2 , A) , (s2)) [2] |
      ((s3 , A) , (s3)) [2] |
      (s1) [1] | (s2) [1] | (s3) [1].
```

This assumes that there are at least 3 Inputs.

StartExpression

This expression is rewritten using the production rules. It should consist of some combination of variables, A - Z, and terminals, s1 - sn where n is the number of subSounds, separated by one of the operators, comma or plus (where a comma represents a Concatenation and a plus represents a CenteringMixer). The starting expression represents a Sound; it should be fully parenthesized. Some legitimate examples of startExpressions are:

```
(A)
(A + (s1 , B))
((E , Z) + ((T , U) , (s3 , (X , s5)))),
```

where A, B, E, Z, T, U, and X all appear on the left hand sides of production rules and there are at least 5 Inputs.

Seed

Type in an integer less than 65535, for example, 34897.

MaxRewrites

This is an upper bound on the number of times the startExpression will be rewritten.

MaxSize

This puts an upper bound on the number of Sounds that will be generated by the grammar.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

1 (no attenuation)
0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

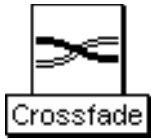
You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1 (no attenuation)
0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



Crossfade

Level, Compression, Expansion Category

Crossfades between its two Sound inputs while optionally also panning and attenuating the result.

Pan

A Pan value of 0 places the sound entirely in the left speaker, and a Pan value of 1 places it entirely in the right. Values inbetween those extremes make the Input source appear as if it were placed somewhere inbetween the two speakers.

Scale

Attenuates the crossfaded signal.

Snd1

This Sound will be crossfaded with Snd2.

Snd2

This Sound is crossfaded with Snd1.

Fade

0 corresponds to entirely Snd1, and 1 corresponds to entirely Snd2. Values in between correspond to mixtures of Snd1 and Snd2.

Type

Choose between a straight, linear crossfading function (which, psychoacoustically, "jumps" in the middle) and a power function that will sound, psychoacoustically, as if it were a linear change from one Sound to the other. The linear fade function is sometimes the more desirable one when crossfading between control functions (See also the Interpolation prototype).



DelayWithFeedback

Reverb, Delay, Feedback Category

Delays the Input signal, optionally feeding some of that delayed signal back and adding it to the current Input.

Type

Choose between Comb and Allpass filters. Both Comb and Allpass are delays with feedback. Allpass also adds some of the direct signal to the output in order to make the long term frequency response flat. With Comb selected, you will not hear the Input until after the first delay. With Allpass, you will hear the direct Input immediately.

Input

This is the signal to be delayed.

Scale

An attenuation factor on the Input (where 1 is full amplitude and 0 is completely attenuated so there is no more Input).

Feedback

Controls the amount of the delayed signal that is fed back and added to the Input. It is the attenuation on the feedback signal (where 1 or 0 dB feeds back the full amplitude signal and adds that to the current Input signal).

Delay

The maximum delay time. The proportion of this time that is actually used is determined by multiplying this value by DelayScale. Kyma needs to know the maximum possible delay in order to allocate some memory for this Sound to use as a delay line, but the actual delay can vary over the course of the Sound from 0 s up to the value of DelayTime.

DelayScale

The proportion of DelayTime that is actually used as the delay, where 0 is no delay, and 1 is equal to the value in the DelayTime field.

Wavetable

In almost all situations, this should be set to Private, so Kyma can allocate some unused wavetable memory to be used as a delay time for this program. (The only time you would want to name this wavetable is if you would like multiple delays or resonators to share a single delay line. In that case, you would type a name for the wavetable and make sure that the other delays use the same name for their wavetables.)

Prezero

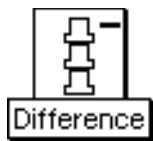
Check this box to start with an empty delay line when this program starts. If Prezero is not checked, the delay line may have garbage in it from previous programs. This can have interesting, if unpredictable, effects and, in some sense, models a physical object or resonator which would maintain its "state" between excitations.

Interpolation

When Linear is selected, changes to DelayScale are linearly interpolated, causing smoother changes to the delay.

When None is selected, changes to DelayScale are not interpolated, resulting in zipper noise.

For fixed delays, it is better to select None, since that uses fewer DSP resources.



Difference

Math Category

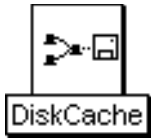
Outputs the difference of Input and minusInput.

Input

The Sound in the MinusInput field will be subtracted from this Sound.

MinusInput

MinusSound is subtracted from Sound.



DiskCache

Sampling Category

Stores the Input as a disk recording when Record is clicked. When Record is unclicked, the input is played off the disk rather than computed in real time.

This Sound can be useful when you are trying to reduce the amount computation required by one branch of a large Sound structure. Whenever you make a change to the Input, remember to click Record and recapture the new version of Input on the disk.

Input

When Record is checked, the Input is recorded into a disk file. When Record is not checked, the recording of the Input is played, not the Input itself.

FileName

Enter a memorable name for the samples file that will be used to "cache" the Input Sound.

Record

Check this box when you want to record the Input. Uncheck it when you want to play back the previously recorded input.



DiskPlayer

Sampling Category

This Sound plays back recordings from the disk file specified in `FileName`, starting at the time specified in `FilePosition` and continuing for the amount of time specified in `Duration` and at the rate specified in `RateScale`. Whenever `Trigger` changes to a positive number, the playback restarts from `FilePosition` and plays again.

To treat this as a sample controlled from the MIDI keyboard, set `FileName` to the name of the sample, set `Duration` to the total time during which you would like to be able to trigger the sample, set `Trigger` to `!KeyDown` (or a MIDI switch), and set `RateScale` to the ratio of the desired frequency to the original recorded frequency, for example

`!Pitch hz / 2 a hz`

to use the MIDI keyboard to control a sample whose recorded pitch was 2 a.

FileName

Enter the name of a Samples file or use the Browse button to select a file from the standard file list dialog. The file can be a recording made in Kyma, a recording imported from another program, or a sample from a CD-ROM (as long as the file is in one of the formats listed in the Kyma manual).

FilePosition

This is the start time within the recording. In other words, you don't have to start playback at the beginning of the file; instead, you can start some amount of time into the file.

RateScale

This is the rate of playback. For example, use 1 to play back at the original rate, 0.5 for half speed, 2 for twice as fast, etc.

Trigger

When the `Trigger` becomes nonzero, one event is triggered. You can trigger several events over the course of the total `Duration` of this program as long as the value of `Trigger` returns to zero before the next trigger. Some example values for `Trigger` are:

<code>1</code>	(plays once with no retriggering)
<code>0</code>	(the sound is silent, never triggered)
<code>!KeyDown</code>	(trigger on MIDI key down)
<code>!cc64</code>	(trigger when controller 64 > 0)

You can also paste another signal into this field, and events will be triggered every time that signal changes from zero to a positive value. (See the manual for a complete description of hot parameters, Event Values, and the Global map files).



DiskRecorder

Sampling Category

Records its Input to disk for CaptureDuration when its Trigger becomes positive.

Input

This is the Sound to be recorded onto disk.

FileName

The name of the samples file where Input will be recorded.

Format

Choose the samples file format. Kyma can record or playback any of these, but if you are going to export this sample to be used in another application, choose a format that the other application can read.

WordSize

This is the number of bits used for each sample point. 24-bit words take up the most memory but will provide the best dynamic range and signal to noise ratio. 8-bit words provide the least dynamic range but also take up the least amount of space on disk. If you are going to export this samples file for use in another application, choose a word size appropriate for that application. For example, if you are creating an alert sound for a Windows application, you would choose 8-bit words and the WAV format. But if you have 18-bit converters and digital I/O, and this is an audio track that you want to use in ProTools, you would probably want to choose 24-bit words and the SD-II format.

Channels

Choose between monaural and stereo recording. If both channels of the Input are identical, choose Mono since it will take half the disk space of a stereo file.

Trigger

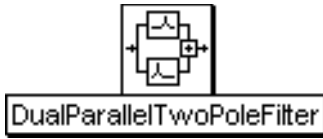
When Trigger becomes positive, the Input will be recorded into the specified file for the specified CaptureDuration. You cannot retrigger without replaying the DiskRecorder (so you will not accidentally write over what you have just captured on disk).

CaptureDuration

Amount of time to record the Input to disk when triggered. This duration can be shorter than the duration of the Input, so, for example, you could have a 1 day long ADInput, and just capture 3 s of it when you trigger the recording. To automatically set the CaptureDuration to the full duration of the Input, enter 0 s here.

Gated

When checked, Gated makes the trigger act as a gate. This means that its input is only recorded when Trigger is a positive value (for example, if Trigger is !KeyDown, the input is only recorded while the key is held down).



DualParallelTwoPoleFilter

Filters Category

Two parallel second-order filter sections having fixed zeroes (at the complex location 1%0). The output of this Sound is the sum of the outputs from the two filter sections.

This Sound is useful if you already know the complex pole locations of the filters that you want.

Otherwise, use the TwoFormantElement prototype; it will give you the same results but with more intuitive parameters (like Frequency and Bandwidth).

Input

This is the Sound that is filtered.

Pole1

This is the pole of the first filter. Type a complex number of the form, $r \% i$, where r is the x coordinate in the z-plane and i is the y coordinate in the z-plane. Try 0.0%0.9 for a high-pass filter, or 0.9%0.0 for a low-pass. If $(r^2 + i^2)$ is greater than 1.0, the filter output will overflow. The second pole of this filter is automatically the complex conjugate of this one, so you don't have to specify it.

Scale1

This is the scale factor on the first filter section. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.

Pole2

This is the pole of the second filter. Type a complex number of the form, $r \% i$, where r is the x coordinate in the z-plane and i is the y coordinate in the z-plane. Try 0.0%0.9 for a high-pass filter, or 0.9%0.0 for a low-pass. If $(r^2 + i^2)$ is greater than 1.0, the filter output will overflow. The second pole of this filter is automatically the complex conjugate of this one, so you don't have to specify it.

Scale2

This is the scale factor on the second filter section. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.



DynamicRangeController

Level, Compression, Expansion Category

Compresses or expands the Input's dynamic range as a function of the SideChain's amplitude envelope.

The Input and output levels will be the same except when the SideChain amplitude envelope crosses the specified Threshold.

If Compressor is selected, the Input will be attenuated whenever the SideChain amplitude exceeds the Threshold.

If Expander is selected, the Input amplitude will be boosted whenever the SideChain amplitude falls below the Threshold.

You specify the Ratio of the input to the output amplitude.

Typical uses include: Limiting (compression with a very large ratio and high threshold), Gating (expansion with extremely small ratio and low threshold), Ducking (set attack and decay to 1 or 2 seconds, put the signal you want to duck in an out at the Input and put the controlling signal at the SideChain, and also mix the SideChain with the output of the DynamicRangeController), making short percussive sounds seem louder (compress, then increase the overall gain), and smoothing out extreme changes in amplitude (particularly useful when recording to media with limited dynamic range, such as cassette tape or videotape).

SideChain

The amplitude envelope of this signal affects the amplitude envelope on the Input. Whenever the SideChain's envelope crosses the Threshold, the Input's dynamic range will be either compressed or expanded.

Input

This is the signal whose dynamic range will be compressed or expanded.

Type

Choose Compressor to compress all amplitudes above the threshold to a narrower dynamic range.

Choose Expander to expand the dynamic range of all amplitudes below the threshold to a wider dynamic range.

Ratio

This is the ratio of the Input amplitude to the output amplitude. For compression, it should be greater than 1. For expansion, it should be less than 1. (This only affects the Input when the SideChain crosses the Threshold).

Threshold

This is the point at which a graph of Input to output level changes from a straight line to a curved line.

In compression, the Input is unchanged when the SideChain amplitude is below this threshold. When the SideChain amplitude exceeds this threshold, the Input is attenuated.

In expansion, the Input is unchanged when the SideChain amplitude is above this threshold. When the SideChain amplitude falls below this threshold, the Input is boosted.

AttackTime

Relates to how quickly or slowly the output amplitude will be modified whenever the Threshold is crossed. This controls how quickly the envelope follower on the SideChain reacts to increases in the SideChain's amplitude.

ReleaseTime

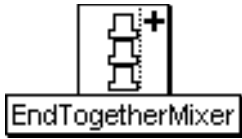
Relates to how quickly or slowly the output amplitude returns to normal whenever the Threshold is crossed. This controls how quickly the envelope follower on the SideChain reacts to decreases in the SideChain's amplitude.

Delay

Typically set to equal to the attack plus the decay time. This is to compensate for the delay introduced by the envelope follower on the SideChain.

Gain

Besides changing the relative levels within the signal, compression and expansion usually change the overall level of the output signal as well. Use Gain to adjust the overall level up or down.



EndTogetherMixer

Mixing & Panning Category

Forces all of its Inputs to end at the same time (if necessary, by delaying the start time of the shorter Inputs).

Inputs

All of these Inputs end at the same time.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

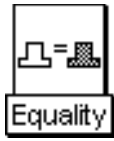
You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



Equality

Math Category

Whenever InputA equals InputB (plus or minus the Tolerance), the output of this Sound is one; at all other times, the output is zero.

Tolerance

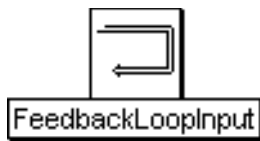
This is the amount of deviation from equality that is still to be considered equality.

InputA

InputA is compared, sample point by sample point, against the Sound in InputB.

InputB

InputB is compared, sample point by sample point, against the Sound in InputA.



FeedbackLoopInput

Xtra Category

A FeedbackLoopInput and FeedbackLoopOutput must always be used as a pair sharing the same Connection name, start time, and duration. The FeedbackLoopInput writes into the delay line specified in Connection, and the FeedbackLoopOutput reads out of that same delay line.

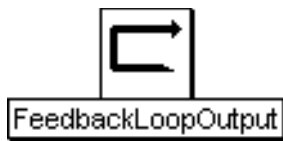
(This differs from other ways of doing feedback in that it allows Kyma's scheduler to put the input to the delay line and the output from the delay line on different expansion cards--freeing up more computation time for processing modules that are in the loop. For simpler cases of feedback, use the DelayWithFeedback or simply write into memory with a MemoryWriter and read out of it with a TimeOffset Sample, Oscillator, or TriggeredTableRead.)

Input

This is the signal that is to be delayed.

Connection

This is the name of the delay line (It must be the same as that specified in the corresponding FeedbackLoopOutput).



FeedbackLoopOutput

Xtra Category

A FeedbackLoopInput and FeedbackLoopOutput must always be used as a pair sharing the same Connection name, start time, and duration. The FeedbackLoopInput writes into the delay line specified in Connection, and the FeedbackLoopOutput reads out of that same delay line.

(This differs from other ways of doing feedback in that it allows Kyma's scheduler to put the input to the delay line and the output from the delay line on different expansion cards--freeing up more computation time for processing modules that are in the loop. For simpler cases of feedback, use the DelayWithFeedback or simply write into memory with a MemoryWriter and read out of it with a TimeOffset Sample, Oscillator, or TriggeredTableRead.)

Connection

This is the name of the delay line and must be the same as that specified in the corresponding FeedbackLoopInput.

Delay

Specify a delay time between 12 samp and 2048 samp. For longer delays, feed the output of this Sound into a DelayWithFeedback. Shorter delays may cause clicking in the output.



FFT

Spectral Analysis FFT Category

The FFT takes an input from the time domain and produces an output signal in the frequency domain or vice versa. Length is the length of the FFT.

Two independent time domain signals are present: one in the left channel and one in the right.

The frequency domain signal repeats every $\text{Length}/2$ samples, alternating between the spectrum of the left channel time domain signal and the spectrum of the right channel time domain signal. The frequency signal is output in frequency order: 0 hz, F_s / Length , $2 * F_s / \text{Length}$, etc. Each frequency domain sample has the real part in the left channel and the imaginary part in the right channel.

When Inverse is not checked, the Input is a time domain signal and the output is a frequency domain signal. When Inverse is checked, the Input is a frequency domain signal and the output is a time domain signal.

Input

This is the signal that will be transformed from the time domain to the frequency domain or vice versa.

Length

This is the window length of the FFT.

Inverse

[Click here](#) to perform an inverse FFT (to convert from a spectrum back to a time-domain waveform).



Filter

Filters Category

An IIR filter of the specified type, cutoff frequency, and order with gain or attenuation on the input and an attenuator on the amount of feedback.

Input

This is the Sound to be filtered.

Type

Choose:

LowPass to attenuate all frequencies above the cutoff Frequency.

HighPass to attenuate all frequencies below the cutoff Frequency.

AllPass to allow all frequencies to pass through unattenuated (but phase shifted by $(-90 * \text{Order})$ degrees at the specified Frequency, with smaller phase shifts at frequencies below that and larger ones for frequencies above).

Frequency

The cutoff frequency for the filter can be specified in units of pitch or frequency. When Feedback is close to 1, the filter will tend "ring" at this frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Q

Q is related to Bandwidth.

For AllPass filters, this affects the size of the phase shift on frequencies around the center Frequency (higher Q corresponds to a narrower band of frequencies that will be phase-shifted).

For LowPass and HighPass filters, this control has no effect.

Scale

This is the attenuation or gain on the Input. The typical maximum for scale is 1, but it can be set as high as 2 if necessary.

HighPass filters generally require lower Scale values than LowPass filters.

Feedback

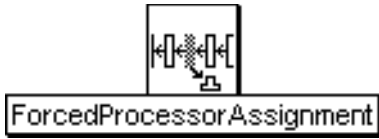
The higher the Feedback, the longer the filter will ring in response to an input.

This is the amount of filtered signal that is fed back in and added to the Input (when low pass is selected, the feedback is negative). Negative feedback values are the same as the positive ones but 180 degrees out of phase.

To simulate a traditional analog sound, use a 4th order low pass or high pass filter, and use feedback to increase the "resonance".

Order

The order of the filter corresponds to the number of poles. In general, the higher the order of the filter, the sharper the cutoff, and the more real-time computation required.



ForcedProcessorAssignment

Sampling Category

Forces the Input to be scheduled on the specified expansion card's processor.

In almost all cases, it is better to let Kyma automatically handle the scheduling of Sounds on different processors. However, this Sound lets you override the default scheduling and force a particular Sound to be computed on a particular processor. Reasons for doing this might include: wanting to record a sample into the wavetable memory of a specific expansion card (or range of cards) and being able to read out of that same memory later; or trying to reschedule a Sound by hand if Kyma's scheduling of it doesn't keep up with real time (doing your own processor allocation by hand is tricky and time-consuming and it is not necessarily recommended! See the Tutorial entitled "What is Real Time Really" for other ways to reduce the computational complexity of a Sound that can't keep up with real time).

Processor

Number of the expansion card that Input will be scheduled on.

Input

Sound that will be forced onto the specified expansion card.



FormantBankOscillator

Xtra Sources Category

Synthesizes a filtered pulse train where the filter is based on the shape of the FormantImpulse and on the formant frequencies, amplitudes, and bandwidths that you supply in the Spectrum parameter (which is usually a SyntheticSpectrumFromArray Sound).

Frequency

This is the fundamental frequency of the pulse train input.

Spectrum

This should be a spectral source (typically a SyntheticSpectrumFromArray) and should specify center frequencies, bandwidths, and amplitudes for the desired formants.

CascadeInput

Whatever Sound you place at this input will be added to the output of the current Sound. You can use this to cascade several FormantBankOscillators or to mix the output of a different kind of Sound with the output of this Sound.

FirstFormant

The lowest numbered formant that you would like to read from the spectrum specification. This is almost always going to be 1 (but can be set to a different number if you would like to skip over some formants or if this is one in a chain of cascaded FormantBankOscillators).

NbrFormants

The number of formants that you would like to synthesize (which can be less than the number of formants specified in the Spectrum input).

FormantImpulse

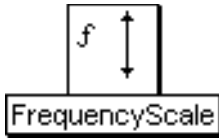
This is a wavetable containing the impulse response for each of the formant filters. In other words, if you were to hit one of the formant filters with a single 1 followed by an infinite string of zeroes, this would be the output of the filter.

NbrImpulses

This is the maximum number of simultaneous impulse responses that can be generated. Leave it at its default value unless you hear the sound breaking up (in which case you can try a smaller number).

AllowFormantAtDC

In nearly all situations, you should leave this box unchecked. You can use this option to synthesize a pulse train whose shape is stored in the FormantImpulse wavetable. (To do this, you also have to specify a spectrum that is a single formant centered at 0 hz or DC).



FrequencyScale

Frequency & Time Scaling Category

Scales the frequency of the Input by the value specified in FrequencyScale. It does this by "granulating" the input and then either overlapping the grains to scale up in frequency, or leaving time between the grains to scale down in frequency.

Input

This is the Sound whose frequency is to be scaled.

FreqTracker

In order to scale the frequency, we have to know the frequency.

The normal setup is to have a FrequencyTracker as the Input in this field. It could also be a Constant or a FunctionGenerator if you already have a good frequency estimate (or intentionally want to supply a different frequency estimate).

FrequencyScale

The frequency of the input will be multiplied by this value.

For example, to shift up by an octave, the FrequencyScale should be 2, and to shift down an octave, the scale should be 0.5. To shift up by 3 half steps, you would use:

$$2^{**} (3/12)$$

To shift down by 7 half steps, you would use:

$$2^{**} (-7/12)$$

To continuously shift between 0 and 4 half steps under control of !Frequency, you could use:

$$2^{**} (!\text{Frequency} * 4 / 12)$$

MaxScale

This should be equal to the FrequencyScale, or, if FrequencyScale is an Event Value, this should be the maximum value of the Event Value.

The larger MaxScale is, the more computation time is required by the FrequencyScale.

Window

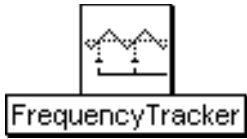
This function is used as a window or envelope on each grain.

Delay

Use this to delay the Input so that it lines up with the FrequencyTracker's estimate of its frequency (since the FrequencyTracker has to have at least two cycles of the Input before it can make its frequency estimate).

It should be a power of two number of samples. Use the following to calculate the delay based on the value of MinFrequency that you have set in the FreqTracker that feeds into this module:

(2 raisedTo: ((1.0 / 120 "!-- this should be replaced by the minFrequency in hertz!") s samp removeUnits log: 2) ceiling) * 2 samp



FrequencyTracker

Tracking Live Input Category

Outputs a continuously updated estimate of the frequency of the Input.

Set the range of frequencies (MinFrequency to MaxFrequency) to be as narrow as possible given what you know about the range of the instrument or voice you are tracking. It is recommended that you leave Confidence, Scale, Emphasis, and Detectors at their default values until you have gotten a reasonably good frequency tracker for a given Input. Then, if you want to fine-tune the tracker, experiment with making small changes to these values, one at a time (starting by increasing the number of Detectors), so you can be sure of what effect each one will have on the tracking. If the tracking gets worse instead of better, revert back to the default values.

The output of the FrequencyTracker falls within the range of 0 to 1; to use this value in a frequency field, multiply it by the maximum possible frequency:

`SignalProcessor sampleRate hz * 0.5`

Input

Estimates the frequency of this Sound.

MinFrequency

This is the lowest expected input frequency. In general, try to set this as high as possible given what you know about the input. For example, if you are frequency tracking a recording or sample and know the lowest frequency, enter it here. Or, for example, if you are tracking the frequency of a live violin, you know that there won't be any frequencies lower than that of the lowest open string 3 g, so if you were to enter 3 e here, you know you would be safe.

MaxFrequency

This is the highest expected input frequency. In general, set this as low as you can given what you know about the Input. But don't underestimate the value, because higher frequencies will then be misidentified as being an octave lower than they really are.

Confidence

This is a measure of how confident the FrequencyTracker must be of a new estimate before it lets go of the previous estimate. In other words, this is a control on how easily the FrequencyTracker will change to a new estimate when the Input frequency is changing over time. A Confidence value of 1 means that the tracker must be 100% sure of its new estimate before giving up the previous estimate; since the tracker is not omniscient it never feels *that* sure, so the result is that it sticks with its very first guess throughout the entire Input, no matter how much the Input's frequency changes. Setting the Confidence to 0 means that the tracker will output every guess even if it is not confident at all, resulting in a lot of spurious frequency estimates. Carefully adjust the Confidence to some value between these two extremes, fine-tuning this setting depending on the Input.

Scale

This is an attenuator on the Input to the FrequencyTracker. In general, the input must be attenuated, since the FrequencyTracker uses an autocorrelation which requires summing the contributions of at least 1000 sample points at a time.

Detectors

This determines the sensitivity of the frequency tracking. Try starting with a value of 10, and then experiment with more or fewer if you want to try fine tuning the frequency tracking. (More is not necessarily better; there is some optimal number of detectors for each circumstance.)

Emphasis

This is a frequency-dependent weighting giving preference to higher frequency estimates. The range of this value is 0 to 1, where 1 is the highest weighting and 0 means to do no weighting. The recommended value is 1.



FunctionGenerator

Envelopes & Control Signals Category

Reads the specified Wavetable for the specified OnDuration whenever it receives a Trigger.

Useful for envelope generation or for reading recordings stored in the wavetable memory (see also the Sample prototype).

In a prototype with an Envelope parameter field (Oscillator, for example) you can use the FunctionGenerator directly as the Envelope parameter. A FunctionGenerator can also be used to control other parameters (such as Frequency or OnDuration). To apply an envelope to any Sound, use the Sound and an envelope generator as Inputs to the VCA prototype. (The VCA simply multiplies its two inputs by each other).

Trigger

When the Trigger becomes nonzero, one event is triggered. You can trigger several events over the course of the total Duration of this program as long as the value of Trigger returns to zero before the next trigger. Some example values for Trigger are:

1	(plays once with no retriggering)
0	(the sound is silent, never triggered)
!KeyDown	(trigger on MIDI key down)
!F1	(trigger when MIDI switch > 0)

You can also paste another signal into this field, and events will be triggered every time that signal changes from zero to a nonzero value. (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

OnDuration

This is the duration of each triggered event. It should be the same length or shorter than the value in Duration (which is the total length of time that this program is available to be triggered). Think of Duration as analogous to the total lifetime of a piano string, and OnDuration as the duration of each individual note that you play on that piano string. The OnDuration must be greater than zero, and you must specify the units of time, for example:

2 s	(for 2 seconds)
2 ms	(for 2 milliseconds)
200 usec	(for 200 microseconds)
2 m	(for 2 minutes)
2 h	(for 2 hours)
2 days	
2 samp	(for 2 samples)
1 / 2 hz	(for the duration of one period of a 2 hz signal)

Wavetable

This is the name of the function that will be generated (or the sample that will be played) each time the FunctionGenerator is triggered.

FromMemoryWriter

Check FromMemoryWriter when the wavetable does not come from a disk file but is recorded by a MemoryWriter in real time.



Gain

Level, Compression, Expansion Category

Gain can be used to boost the amplitude above the maximum of 1.0. This can be useful when you want to multiply by numbers greater than 1 or when you have a low amplitude Sound.

If you need variable gain, you can use an Attenuator with variable loss (using Left and Right scales) on the Sound before feeding it into the Gain.

Input

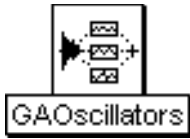
This is the Sound whose amplitude will be boosted.

Left

This is the scale on the left channel. It can be any value.

Right

This is the scale on the left channel. It can be any value.



GAOscillators

Xtra Sources Category

Additive synthesis using oscillators with complex waveforms (rather than sine waves). Each oscillator has its own amplitude envelope and all oscillators share the same frequency deviation envelope.

TimeIndex

This controls the position within the frequency and amplitude envelopes, where -1 points to the beginning of the envelopes, 0 points to the middle, and 1 points to the end of the envelopes. To move linearly from the beginning to the end of the envelopes, use a FunctionGenerator whose wavetable is a FullRamp, or use the EventValue fullRamp generator. For example,

!KeyDown fullRamp: 3 s

would go from -1 up to 1 whenever a MIDI key goes down.

Analysis0

This is the GA analysis file used when Morph is set to 0. A GA analysis file is an AIFF file containing each of the wavetables, followed by each of the amplitude envelopes, followed by the frequency deviation envelope. Each of the waveforms is 4096 samples long, and each of the amplitude envelopes and the frequency envelope is the same length as each other (but this length varies from analysis to analysis).

To create your own analysis file from a sample, first do a spectral analysis of the sample, and then generate a GA analysis file from that. Both of these operations can be performed using tools found in the Tools menu. (See the tutorial on GA analysis/synthesis for full details).

Analysis1

This is the GA analysis file used when Morph is set to 1. A GA analysis file is an AIFF file containing each of the wavetables, followed by each of the amplitude envelopes, followed by the frequency deviation envelope. Each of the waveforms is 4096 samples long, and each of the amplitude envelopes and the frequency envelope is the same length as each other (but this length varies from analysis to analysis).

To create your own analysis file from a sample, first do a spectral analysis of the sample, and then generate a GA analysis file from that. Both of these operations can be performed using tools found in the Tools menu. (See the tutorial on GA analysis/synthesis for full details).

Envelope

This is an overall envelope on all of the enveloped oscillators.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch (key number plus pitch bend)
!KeyNumber nn (MIDI notenumber)
4 c + (!Frequency * 9 nn) (continuous controller from 4 c to 4 a)

Morph

This controls a crossfade between each of the waveforms and envelopes in Analysis0 with each of the waveforms and envelopes in Analysis1.



GenericSource

Xtra Sources Category

This Sound can represent the live input, a sample read from the disk, or a sample read from RAM. If Ask is checked, you can choose between these three kinds of sources each time you recompile the Sound.

Source

Select the live input, an audio track on disk, or a sample in RAM.

LeftChannel

Check this box to monitor the left channel of the source signal.

RightChannel

Check this box to monitor the right channel of the source signal.

Sample

Enter the name of a sample file or click the disk icon to choose it from the file dialog.

Autoloop

Check here to loop the sample or disk.

Trigger

Whenever Trigger changes from 0 to a nonzero value, it will replay the disk or sample in RAM from the beginning.

AttackTime

Attack time for a linear envelope applied to the sample source.

ReleaseTime

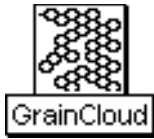
Release time for a linear envelope applied to the sample source.

Scale

Scales the amplitude of the source.

Frequency

For samples in RAM or on disk, this controls the playback frequency. For samples that are unpitched, the original frequency is assumed to be 4 c (60 nn).



GrainCloud

Xtra Sources Category

Generates a cloud of short-duration grains of sound, each with the specified Waveform and each with an amplitude envelope whose shape is given by GrainEnv. The density of simultaneous grains within the cloud is controlled by Density, with the maximum number of simultaneous grains given by MaxGrains.

Amplitude controls an amplitude envelope over the *entire* cloud (each individual grain amplitude is controlled by GrainEnv). Similarly, Duration is the duration of the entire cloud, not of each individual grain.

You can control the Frequency, stereo positioning, and duration of each grain as well as specifying how much (if any) random jitter should be added to each of these parameters (giving the cloud a more focused or a more dispersed sound, depending on how much randomness is added to each of the parameters).

Waveform

The waveform of the oscillator inside each grain.

GrainEnv

Defines the shape of each grain's amplitude envelope. To minimize clicks, choose a wavetable that starts and ends on zero.

MaxGrains

Maximum number of grains that can be playing at any one time. The smaller this number, the less computational power the GrainCloud requires (but the less dense the texture you can generate). For even denser textures, put more than one GrainCloud into a Mixer, and give each GrainCloud a different Seed value.

Amplitude

An overall level or amplitude envelope applied to the entire cloud. Note that this is independent of the amplitude envelope on each individual grain.

Density

Controls the number of new grains that can start up at any one time. Small Density values result in a sparse texture; large Density values generate a dense texture.

GrainDur

The duration of an individual grain.

The duration of each grain is a function of three parameters:

$$\text{GrainDur} + \text{CyclesPerGrain} + (\text{GrainDurJitter} * 2 * (\text{GrainDur} + \text{CyclesPerGrain}))$$

To specify the number of waveform cycles within each grain (implies that higher frequency grains will have shorter duration than lower frequency grains and assures that every grain will contain an integer number of full cycles of the waveform):

$$\text{GrainDur} = 0 \text{ s}$$

$$\text{CyclesPerGrain} = \text{<number of cycles in each grain>}$$

To specify a constant duration, no matter what the frequency of the waveform within each grain (implying that high frequency grains will have more cycles in them than low frequency grains):

GrainDur = <desired grain duration>
CyclesPerGrain = 0

GrainDurJitter

The amount of random jitter added to the grain duration value when a new grain starts up. When GrainDurJitter is 0, every new grain will have the same duration. At the other extreme, when it is set to 1, the durations vary randomly from 0 to twice the specified duration.

CyclesPerGrain

The integer number of full cycles of the waveform that should occur inside each grain. Use this parameter to specify grains that are shorter for high frequencies than they are for low frequencies. If you prefer uniform grain durations over all frequencies, set this parameter to zero and use GrainDur to set the grain duration.

Frequency

Frequency of the oscillator within each grain.

FreqJitter

The amount of random jitter added to the Frequency value when a new grain starts up. When FreqJitter is 0, the frequency inside each grain will be equal to the value in the Frequency parameter. When it is 1, each grain will have a different, randomly selected frequency.

It is defined as:

$$(1 + (<\text{randomNumber}> * \text{FreqJitter})) * \text{Frequency}$$

so when FreqJitter is 1, the frequency can range from 0 hz up to twice the specified Frequency (100% up is an octave up, while 100% down is DC). When FreqJitter is zero, no random deviations are added to the Frequency.

Pan

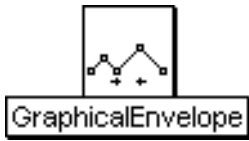
Stereo position of each grain (where 0 is hard left, 0.5 is in the middle, and 1 is hard right).

PanJitter

The amount of random jitter added to the Pan value when a new grain starts up. The larger this number, the more diffuse the apparent location, and the smaller the number, the more localized the sound.

Seed

A starting point for the random number generator. It should be a number between 0 and 1. Each different seed number results in a different (but repeatable) sequence of random numbers. When adding several GrainClouds with the same control parameters together in a Mixer, give each of them a different seed in order to ensure that each of them has *different* random jitter added to its parameters (otherwise, they will just double each other).



GraphicalEnvelope

Envelopes & Control Signals Category

Similar to the ADSR envelope, except that you can graphically specify an arbitrary number of segments and can specify loop points.

Typical uses include amplitude envelopes, pitch envelopes, and time index functions.

Envelope

Use this field to edit the envelope.

To add an envelope breakpoint, click the mouse while holding down the Shift key. Click and drag a breakpoint to move it. Click a breakpoint and press the Delete key to delete it. The button at the lower right of this field is used to control the looping behavior of the selected breakpoint.

See the section on Parameter Settings in the manual for more information.

Level

This is a scale factor (from 0 to 1) for attenuating the overall output level.

Gate

When Gate changes from zero to a nonzero, the envelope will be triggered. Gate must return to zero again before the envelope can be retriggered. If you have specified beginning and ending segments for a loop, the envelope will repeat the loop segments for as long as the Gate is nonzero. If the ending loop segment ends on a higher or lower value than the start of the beginning segment, the entire looped portion will get larger or smaller each time it is repeated (because each segment has a *slope* associated with it, not the absolute values at each point).

Rate

This is the rate at which the envelope is played.

Use 1 to play it back as shown in the Envelope parameter field (where each heavy vertical line represents one second), 0.5 to make the envelope last twice as long, 2 to make it play twice as fast, etc.



GraphicEQ

Filters Category

This gives you independent control over the levels of seven, octave-wide bands ranging in center frequency from 250 hz up to 16,000 hz. You can use it for attenuating or accentuating subparts of the input spectrum. When all levels are set to 1, you should hear the original input signal with no change.

Input

This is the Sound to be filtered or equalized.

CF250Hz

Controls the level of a band from DC up to about 4 f.

CF500Hz

Controls the level of an octave band from about 4 f to 5 f (350-670 hz).

CF1000Hz

Controls the level of an octave band from about 5 f to 6 f (670-1340 hz).

CF2000Hz

Controls the level of an octave band from about 6 f to 7 f (1340-2794 hz).

CF4000Hz

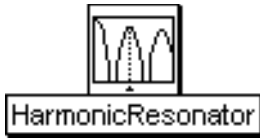
Controls the level of an octave band from about 7 f to 8 f (2794-5588 hz).

CF8000Hz

Controls the level of an octave band from about 8 f to 9 f (5588-11,175 hz).

CF16000Hz

Controls the level of an octave band from about 9 f to 10 f (11,175 up to half the sampling rate).



HarmonicResonator

Filters Category

A filter that has resonances at the specified Frequency and all of its harmonics.

Input

This is the Sound that will be filtered.

DecayTime

This is the time it will take for the input amplitude to decay to -60 dB below its initial amplitude.

Brightness

The higher this value, the longer it will take for the high frequency partials to die away, resulting in a brighter timbre.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Wavetable

Type in the name of a wavetable to use as a delay line, or select Private to let Kyma choose some free wavetable memory for you.

Prezero

Check this box if you want to assure that the delay line is clear before it is used in this Sound. Leaving it unchecked simulates a physical resonator by allowing the filter to remember its state between excitations.

Scale

This is an attenuator on the input Sound. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.



HighShelvingFilter

Filters Category

Boosts or cuts the frequencies above the specified cutoff frequency.

Input

This is the Sound to be filtered.

CutoffFreq

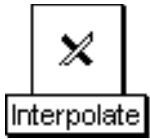
Frequencies above this will be boosted or cut by the specified amount.

BoostOrCut

Indicate the amount of boost or cut in units of dB. Negative values indicate a cut, positive values a boost.

Scale

Attenuator on the input.



Interpolate

Math Category

A linear combination of two inputs. The left channel of Input0 is multiplied by 1-leftInterp, the left channel of Input1 is multiplied by leftInterp, and the two are added together (and the same for the right channels and rightInterp).

This Sound is useful for interpolating between control functions and envelopes (for example, interpolating between two sets of analysis envelopes and using the result as an input to an OscillatorBank results in a spectral "morph" from one spectrum to another before it is fed into the OscillatorBank).

Input1

When the Interpolation value is 0, this Sound is at full amplitude and Input2 is at zero amplitude.

Input2

When the Interpolation value is 1, this Sound is at full amplitude and Input1 is at zero amplitude.

LeftInterp

This parameter controls the left channel of the output. 0 results in an output of Input1 alone, and 1 results in an output of Input2 alone. Any values between 0 and 1 result in a mix of Input1 and Input2 to be output. (If Inputs are spectral sources, this channel is the amplitude envelopes.)

RightInterp

This parameter controls the right channel of the output. 0 results in an output of Input1 alone, and 1 results in an output of Input2 alone. Any values between 0 and 1 result in a mix of Input1 and Input2 to be output. (If Inputs are spectral sources, this channel is the frequency envelopes.)



IteratedWaveshaper

Distortion & Waveshaping Category

This is like waveshaping, except that the output of the waveshaper is fed back into the waveshaper for the specified number of Iterations before the result is finally output.

This algorithm was submitted by Agostino Di Scipio.

Iterations

Specify the number of times the output of the waveshaper should be fed back through the shaping function before the output.

Input

Each sample of this Sound is used as an index into the shaping function stored in the Wavetable.

Scale

This attenuation is applied prior to feeding the output back into the waveshaper for each iteration.

Wavetable

This is the shaping function. An input value of zero indexes into the middle of this table, minus one indexes into the beginning, and plus one indexes into the end of the table.



KeyMappedMultisample

Sampling Category

This provides a quick way to map a large bank of samples to specific ranges of a MIDI keyboard. It can be used for mapping large numbers of samples taken from musical instruments to narrow ranges on the keyboard (much as is done on a standard sampler) or for being able to select and trigger large banks of sound effects in real time (from a sequencer or from the MIDI keyboard).

To specify the samples that belong in the same bank, place all of them in the same folder or directory. Kyma will only look at the top level of your directory, so any folders within that folder will be ignored. The ordering of the samples within that file will be interpreted to be alphabetical by name (when ordering is important).

The files within the directory must all be mono or all stereo; mixtures of mono and stereo files are not guaranteed to be interpreted correctly.

Frequency

Use Default here if you want the Frequency to equal the pitch of the recorded sample. The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released and will finish playing through the current sample and then stop.

If the sample file has loop points stored in its header, Kyma will loop the sample for as long as Gate remains positive (so, for example, as long as the MIDI key is held down).

Velocity

If By Base Pitch is checked, this value will pick between several samples of the same base pitch but different velocity ranges as long as you have set those ranges in the header of the samples file.

FirstSample

Use the disk icon to browse, and then select one sample file within the folder or directory containing all

the samples to be mapped.

LoFreq

Lowest frequency on the keyboard that will trigger one of the samples.

HiFreq

Highest frequency on the keyboard that will trigger one of the samples.

Mapping

The policy for mapping note number to samples file:

OnePerHalfStep: assign each samples file in order to the next half step on the keyboard. If you run out of samples files start over again from the first file. This is a good mode for triggering sound effects from the keyboard since you know that each half step will trigger a different sample.

EquallySpaced: Give each samples file an equal-sized range of the keyboard. The first samples file in the list gets the lowest range of keys, the next file gets the next block of keys, etc. This is especially useful when you have arranged the files alphabetically in your directory from the lowest to the highest originally recorded pitches.

ByBasePitch: Use the base pitch (as specified in the header of AIFF files) to assign samples to the frequencies closest to their originally recorded pitch. This is the best policy to use when you have a set of samples that covers the range of a musical instrument, since it will result in the least distortion of the samples if you can play them as close as possible to their original pitches.

ByPitchRange: Assigns each sample to the pitch range specified in the sample file header. When the ranges overlap, sample files whose names sort later in the alphabet take precedence.

AttackTime

Duration of the attack of an envelope applied to the sample.

ReleaseTime

Duration of the release of an envelope applied to the sample.

Scale

Overall level of the sample.

Loop

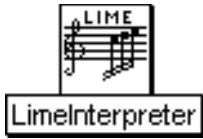
Click here if you want to loop the sample using the loop points specified in the header of the samples file. (Applies only to samples read from RAM, not those read directly from disk).

FromDisk

Click here to indicate that the sample should be read directly from disk rather than from sample RAM on the Capybara. You can use this option when Kyma tells you that you have run out of sample RAM (because your sample is too long or you have requested too many samples).

NoTransposition

Click here to indicate that the samples should not be transposed from their originally recorded pitches. This is so you can use the keyboard to trigger sound effects or long disk files without changing the duration or frequency of the recordings.



LimeInterpreter

Scripts Category

Reads binary files produced by the Lime music notation program and maps values to parameters of Kyma Sounds. This allows you to "play" scores produced in Lime using Kyma Sounds as the instruments.

FileName

This is the name of a binary file created and saved in Lime.

Inputs

These Sounds are treated as templates. Each name should begin with a letter and contain only alpha-numeric characters; this field will reject any Sounds with "illegal" names. You can reference these Sounds by name in the Script field.

Script

The script contains Smalltalk code that reads and interprets data from the specified Lime binary file. See the manual for a more details about Smalltalk.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



LiveSpectralAnalysis

Tracking Live Input Category

This Sound should be used as the Spectrum parameter of an OscillatorBank. It analyzes the Input and produces amplitude and frequency envelopes for controlling a bank of oscillators.

Input

The output of the LiveSpectralAnalysis is the spectrum of this Input.

LowestAnalyzedFreq

Check the highest frequency that will still lie below the lowest fundamental frequency of the Input. The lower this frequency, the more time-smearing and delay, so pick the highest one that will still encompass the fundamental.

The frequency value you select will also determine how many bandpass filters are used in the analysis and, therefore, the number of tracks or partials generated by the analysis. The lower the frequency, the more partials that are generated:

- 1 F: 512
- 2 F: 256
- 3 F: 128
- 4 F: 64
- 5 F: 32

If you use this LiveSpectralAnalysis to control an OscillatorBank, this is the maximum number of oscillators that you should specify in the OscillatorBank (you can specify fewer of them, but specifying more of them will not result in any additional partials).

AmpScale

This is the overall amplitude level for all the partials.

FreqScale

This scales the frequency of all the oscillators without affecting the timing or duration of the amplitude envelopes. There is no limit on the range, so to control it continuously use:

!Freq * 10

Or to control it from a MIDI keyboard use:

!KeyNumber nn hz / 60 nn hz

Response

This is the time response of the filters. Experiment to find the best time response that does not add distortion to the sound. This specifies the bandwidth of the bandpass filters used in the analysis: "BestFreq" is the narrowest bandwidth, "BestTime" is the widest bandwidth, and the others are intermediate bandwidths.

Harmonic

Some kinds of live morphs work better with Harmonic checked, but you should experiment with it both checked and unchecked. It is a little trickier to do the harmonic analysis, so you should avoid checking this box except in situations where it is required.

Once you check this box, you must also set several other parameters: LowFreq, HighFreq, InitFreq, TrackedHarmonic, UnpitchedThreshold.

UnpitchedOnly

Set this value to 1, and adjust UnpitchedThreshold if you would like to hear the unpitched parts of the sound only (transients, clicks, consonants, noise, etc). It requires that you have Harmonic checked.

UnpitchedThreshold

Set UnpitchedOnly to 1, and adjust this value if you would like to hear only the unpitched parts of the Input (clicks, consonants, transients, noise). This requires that you have Harmonic checked.

TrackedHarmonic

Only required if you have Harmonic set. A pitch-follower attempts to track whichever harmonic you indicate in this field. Usually it is 1 for the fundamental, but if a higher harmonic is stronger, it may be easier to track harmonic 2, 3 or higher. The harmonic's frequency must lie between LowFreq and HighFreq.

LowFreq

Only required if you have Harmonic checked. This is the lowest frequency you expect to see in the tracked harmonic.

HighFreq

Required only when you have Harmonic checked. This is the highest frequency you expect to encounter in the tracked harmonic.

InitFreq

Required only if Harmonic is checked. This is an estimate of the initial frequency of the tracked harmonic.

FundamentalOnly

This only applies when Harmonic is checked. Check this box if you would like to listen to the estimated fundamental frequency. It can help you judge whether adjusting LowFreq or HighFreq might result in a better estimate.



LowShelvingFilter

Filters Category

Boost or cut the spectrum below the specified cutoff frequency.

Input

This is the Sound to be filtered.

CutoffFreq

Frequencies below this will be boosted or cut by the specified amount.

BoostOrCut

Indicate the amount of boost or cut in units of dB. Negative values indicate a cut, positive values a boost.

Scale

Attenuator on the input.



Matrix4

Spatializing Category

This Sound is a four-input four-output matrix mixer. The four input Sounds are routed and mixed to the four output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

In1

One of the four input Sounds.

In2

One of the four input Sounds.

In3

One of the four input Sounds.

In4

One of the four input Sounds.

InsToOut1

This parameter is a list of four mixing levels. These levels are used to mix the four inputs into an output for channel 1.

InsToOut2

This parameter is a list of four mixing levels. These levels are used to mix the four inputs into an output for channel 2.

InsToOut3

This parameter is a list of four mixing levels. These levels are used to mix the four inputs into an output for channel 3.

InsToOut4

This parameter is a list of four mixing levels. These levels are used to mix the four inputs into an output for channel 4.



Matrix8

Spatializing Category

This Sound is a eight-input eight-output matrix mixer. The eight input Sounds are routed and mixed to the eight output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

In1

One of the eight input Sounds.

In2

One of the eight input Sounds.

In3

One of the eight input Sounds.

In4

One of the eight input Sounds.

In5

One of the eight input Sounds.

In6

One of the eight input Sounds.

In7

One of the eight input Sounds.

In8

One of the eight input Sounds.

InsToOut1

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 1.

InsToOut2

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 2.

InsToOut3

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 3.

InsToOut4

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 4.

InsToOut5

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 5.

InsToOut6

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 6.

InsToOut7

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 7.

InsToOut8

This parameter is a list of eight mixing levels. These levels are used to mix the eight inputs into an output for channel 8.



MemoryWriter

Sampling Category

When Trigger becomes positive, records the Input into the wavetable memory of the signal processor for the length of time specified in CaptureDuration.

Any Sounds that read wavetables can be used to play back this recording (for example, FunctionGenerator, Sample, and others).

Input

The output of this Sound is recorded into the wavetable memory of the signal processor.

CaptureDuration

The length of time to record the Input. Enter 0 s if you want to record Input for its full duration.

Global

Click here to record the Input into the wavetable memory on all expansion cards (otherwise, it will be recorded only into the memory of the expansion card on which the MemoryWriter happens to get scheduled, and Kyma will be forced to schedule the playback Sound on that same card. If you make the recording global, it is much easier for Kyma to schedule the playback Sounds, because it can schedule them on any cards, knowing that the recording is available in the memory of all the cards.)

Cyclic

When Cyclic is selected, the MemoryWriter does a "looping" recording. In other words, it records for the specified CaptureDuration; then, if Trigger is still positive, it wraps around to the beginning of the recording and continues recording the Input, overwriting what it had previously recorded there.

RecordingName

Enter a name for the sample that you are recording into the wavetable memory. Use this same name in the playback Sounds, so they can find the sample in the wavetable memory. Any Sound that reads from the wavetable memory can also read the sample that you are writing into the memory with MemoryWriter. Sounds like Sample and FunctionGenerator read arbitrarily long tables, whereas Sounds like Oscillator will use only the first 4096 entries of the named wavetable (only the first 4096 sample points).

Silent

Click here if you would like to record the Input silently, without also monitoring it at the same time.

Trigger

When the Trigger becomes nonzero, the recording is triggered. You can trigger several events over the course of the total Duration of this program as long as the value of Trigger returns to zero before the next trigger. Some example values for Trigger are:

- | | |
|----------|--|
| 1 | (plays once with no retriggering) |
| 0 | (the sound is silent, never triggered) |
| !KeyDown | (trigger on MIDI key down) |
| !F1 | (trigger when MIDI switch > 0) |

You can also paste another signal into this field, and events will be triggered every time that signal changes from zero to a nonzero value. (See the manual for a complete description of hot parameters,

EventValues, EventSources, and Map files).



MIDIFileEcho

MIDI Out Category

This Sounds reads up all MIDI events on the specified range of channels from the designated file and then echoes them to the MIDI output.

It does not output MIDI within Kyma but copies the MIDI file directly to the DSP MIDI output.

LowChannel

The lowest MIDI channel to echo.

HighChannel

Highest MIDI channel to echo.

FileName

A MIDI file



MIDI Mapper

MIDI In Category

Defines its Input as a MIDI voice of the specified polyphony that takes its input from the specified MIDI input channel within the given range of pitches either in real time or from a MIDI file. Left and Right are attenuators on the left and right channels of the audio output of this Sound.

A local map supplied in the Map field overrides the global MIDI map for any Event Values in its Input. If you don't need to override the global map, use MIDI Voice instead.

Input

Input (including all of its inputs) is the Sound associated with this MIDI voice. If any of Input's parameters are Event Values, they will be mapped to Event Sources by the Map parameter (which overrides the currently select global map but only for Input)

Map

Enter any mappings from Event Values to Event Sources that should be *different* in Input (and its inputs) than they are in the currently selected global map. If an Event Value is not defined here in the local map, Kyma will use the global map to determine its Event Source.

The syntax for a mapping is:

!EventValueName is: 'EventSourceName

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

1 (no attenuation)
0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1 (no attenuation)
0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Channel

The MIDI Mapper only pays attention to this incoming MIDI channel (or MIDI events on this channel of the

MIDI file).

Set Channel to 0 to use whatever channel is specified in the global map.

Source

Choose between live MIDI input, reading from a MIDI file, or receiving MIDI events specified in the Script field.

MidiFile

Read the MIDI event stream from this file if MIDI File is selected as the Source. Use the Browse button to bring up a standard file list and select the filename from the list.

Polyphony

Number of simultaneous MIDI note events possible on this voice. For example, if you specify a Polyphony value of 4, Kyma makes 4 copies of the Input Sound, so any one of them can be triggered at any time and all four can be sounding at the same time. The higher the value of Polyphony, the more computation time is required per sample tick.

LowPitch

The lowest MIDI pitch that this voice responds to. Be sure to include units of pitch or frequency with the value. (For this particular Sound, if you specify this value as a frequency, Kyma will round to the nearest equal-tempered MIDI notenummer).

This allows you to map different regions of the MIDI note range to different voices and to define keyboard splits.

HighPitch

The highest MIDI pitch that this voice responds to. Be sure to include units of pitch or frequency with the value. (For this particular Sound, if you specify this value as a frequency, Kyma will round to the nearest equal-tempered MIDI notenummer).

This allows you to map different regions of the MIDI note range to different voices and to define keyboard splits.

Script

When Source is set to Script, this program sends MIDI events to the Input (just as if these events were being read from a MIDI file). See the manual for more information on algorithmically generating and manipulating MIDI events.

To specify a MIDI event, use:

```
self keyDownAt: <aTime> duration: <aDur> frequency: <aFreq> velocity: <aVel>.
```

Be sure to include units on the start time, duration, and frequency values, and specify velocity within a range of 0 to 1. Frequency can be any value specified in hz or nn; you are not limited to the pitches from the 12-tone equal tempered scale. All arguments must be real values (as opposed to EventValues).

As a shortcut, you can drop any of the tags, for example, the following are all valid:

```
self keyDownAt: 0 s.
```

```
self keyDownAt: 3 s duration: 10 beats.
```

```
self keyDownAt: 0 beats duration: 10 beats frequency: 4 c.
```

```
self keyDownAt: 5 beats duration: 0.25 beats frequency: 4 c + 0.5 nn velocity: 0.75.
```

This field is actually a Smalltalk program, so you can use Smalltalk expressions or control structures to generate these events algorithmically, for example:

```
1 to: 12 do: [ :i |
```

```
self keyDownAt: (i - 1) beats duration: 0.25 beats frequency: 4 c + i nn velocity: (i / 12.0)].
```

or:

```
| r t |  
r := Random newForKymaWithSeed: 66508.  
t := 0.  
100 timesRepeat: [  
    t := t + r next.  
    self keyDownAt: t s duration: 0.25 beats frequency: (r next * 1000 + 60) hz velocity: r next].
```

You can also create sequences and mixes of "notes" and "rests" or collections of MIDI events, each associated with its own time tag.

To create a rest object, use:

```
Rest durationInBeats:
```

To create a note, use any of the following creation messages:

```
Note durationInBeats:  
Note durationInBeats:frequency:  
Note durationInBeats:frequency:velocity:  
Note durationInBeats:velocity: frequency:  
Note durationInBeats: frequency:durationInBeats:velocity:
```

To create a sequence of events (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection) use:

```
EventSequence events: <anArrayOfEvents>.
```

To create a mix of events which all start at the same time (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection) use:

```
EventMix events: <anArrayOfEvents>.
```

To create a collection of events, each of which has a starting time associated with it (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection, and the starting time is specified in beats) use:

```
TimedEventCollection startingBeats: <anArrayOfBeatsWithNoUnits> events: <anArrayOfEvents>.
```

To play a Note, Rest, EventSequence or EventMix, use:

```
<anEvent> playOnVoice:onBeat:bpm:  
<anEvent> playOnVoice:  
<anEvent> playOnVoice:bpm:  
<anEvent> playOnVoice:onBeat:
```

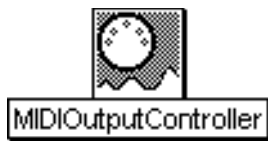
Transformations that can be applied to Notes, Rests, EventSequences, EventMixes or TimedEventCollections include:

```
dim: <aDurationScaleFactor>  
trsp: <anIntervalOfTranspositionInHalfSteps>  
dbl: <anIntervalOfDoublingInHalfSteps>  
retrograde
```

Transformations that can be applied to EventSequences, EventMixes or TimedEventCollections include:

```
randomOrder  
randomizeTimesUsing: <aRandomStream>  
pickingEventsUsing: <aRandomStream>  
totalBeats: <durInBeats>  
quantizeTo: <shortestDur>  
maxSpacing: <longestDur>
```

For examples using these creation and manipulation methods, see MIDI scripts in the manual.



MIDIOutputController

MIDI Out Category

This Sound outputs its Value parameter to the MIDI output on the specified channel as the specified continuous controller.

Channel

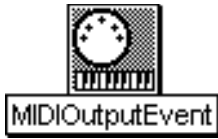
MIDI output channel.

ControllerNumber

This is the MIDI continuous controller number.

Value

This is the value that will be output as the controller data. Paste a Sound in here to turn a Sound into a MIDI controller output.



MIDIOutputEvent

MIDI Out Category

When Gate becomes positive, a note-on message with the current values of Frequency and Amplitude is sent as the note number and velocity on the given MIDI channel. When Gate returns to zero, a note-off message will be sent.

Frequency

There is no help available for this parameter.

Amplitude

There is no help available for this parameter.

Channel

There is no help available for this parameter.

Gate

There is no help available for this parameter.



MIDIOutputEventInBytes

MIDI Out Category

This Sounds sends an uninterpreted sequence of bytes to the MIDI output. You can use it to send arbitrary MIDI events.

Bytes

Enter the MIDI message as a sequence of numbers separated by spaces. If you want to specify them in hex, precede the number with 16r, for example:

16rFF



MIDI Voice

MIDI In Category

Defines its Input as a MIDI voice of the specified polyphony that takes its input from the specified MIDI input channel within the given range of pitches either in real time or from a MIDI file. Left and Right are attenuators on the left and right channels of the audio output of this Sound.

Input

Input (including all of its inputs) is the Sound associated with this MIDI voice. If any of Input's parameters are Event Values, they will be mapped to Event Sources by the Map parameter (which overrides the currently select global map but only for Input)

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Channel

The MIDI Voice only pays attention to this incoming MIDI channel (or MIDI events on this channel of the MIDI file).

Set Channel to 0 to use whatever channel is specified in the global map.

Source

Choose between live MIDI input, reading from a MIDI file, or receiving events specified in the Script field.

MidiFile

Read the MIDI event stream from this file if MIDI File is selected as the Source. Use the Browse button to bring up a standard file list and select the filename from the list.

Polyphony

Number of simultaneous MIDI note events possible on this voice. For example, if you specify a Polyphony value of 4, Kyma makes 4 copies of the Input Sound, so any one of them can be triggered at any time and all four can be sounding at the same time. The higher the value of Polyphony, the more computation time is required per sample tick.

LowPitch

The lowest MIDI pitch that this voice responds to. Be sure to include units of pitch or frequency with the value. (For this particular Sound, if you specify this value as a frequency, Kyma will round to the nearest equal-tempered MIDI notenummer).

This allows you to map different regions of the MIDI note range to different voices and to define keyboard splits.

HighPitch

The highest MIDI pitch that this voice responds to. Be sure to include units of pitch or frequency with the value. (For this particular Sound, if you specify this value as a frequency, Kyma will round to the nearest equal-tempered MIDI notenummer).

This allows you to map different regions of the MIDI note range to different voices and to define keyboard splits.

Script

When Source is set to Script, this program sends MIDI events to the Input (just as if these events were being read from a MIDI file). See the manual for more information on algorithmically generating and manipulating MIDI events.

To specify a MIDI event, use:

```
self keyDownAt: <aTime> duration: <aDur> frequency: <aFreq> velocity: <aVel>.
```

Be sure to include units on the start time, duration, and frequency values, and specify velocity within a range of 0 to 1. Frequency can be any value specified in hz or nn; you are not limited to the pitches from the 12-tone equal tempered scale. All arguments must be real values (as opposed to EventValues).

As a shortcut, you can drop any of the tags, for example, the following are all valid:

```
self keyDownAt: 0 s.
self keyDownAt: 3 s duration: 10 beats.
self keyDownAt: 0 beats duration: 10 beats frequency: 4 c.
self keyDownAt: 5 beats duration: 0.25 beats frequency: 4 c + 0.5 nn velocity: 0.75.
```

This field is actually a Smalltalk program, so you can use Smalltalk expressions or control structures to generate these events algorithmically, for example:

```
1 to: 12 do: [:i |
  self keyDownAt: (i - 1) beats duration: 0.25 beats frequency: 4 c + i nn velocity: (i / 12.0)].
```

or:

```
| r t |
r := Random newForKymaWithSeed: 66508.
t := 0.
100 timesRepeat: [
  t := t + r next.
  self keyDownAt: t s duration: 0.25 beats frequency: (r next * 1000 + 60) hz velocity: r next].
```

You can also create sequences and mixes of "notes" and "rests" or collections of MIDI events, each associated with its own time tag.

To create a rest object, use:

Rest durationInBeats:

To create a note, use any of the following creation messages:

Note durationInBeats:

Note durationInBeats:frequency:

Note durationInBeats:frequency:velocity:

Note durationInBeats:velocity: frequency:

Note durationInBeats: frequency:durationInBeats:velocity:

To create a sequence of events (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection) use:

EventSequence events: <anArrayOfEvents>.

To create a mix of events which all start at the same time (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection) use:

EventMix events: <anArrayOfEvents>.

To create a collection of events, each of which has a starting time associated with it (where an event is a Note, a Rest, an EventSequence, an EventMix, or a TimedEventCollection, and the starting time is specified in beats) use:

TimedEventCollection startingBeats: <anArrayOfBeatsWithNoUnits> events: <anArrayOfEvents>.

To play a Note, Rest, EventSequence or EventMix, use:

<anEvent> playOnVoice:onBeat:bpm:

<anEvent> playOnVoice:

<anEvent> playOnVoice:bpm:

<anEvent> playOnVoice:onBeat:

Transformations that can be applied to Notes, Rests, EventSequences, EventMixes or TimedEventCollections include:

dim: <aDurationScaleFactor>

trsp: <anIntervalOfTranspositionInHalfSteps>

dbl: <anIntervalOfDoublingInHalfSteps>

retrograde

Transformations that can be applied to EventSequences, EventMixes or TimedEventCollections include:

randomOrder

randomizeTimesUsing: <aRandomStream>

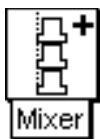
pickingEventsUsing: <aRandomStream>

totalBeats: <durInBeats>

quantizeTo: <shortestDur>

maxSpacing: <longestDur>

For examples using these creation and manipulation methods, see MIDI scripts in the manual.



Mixer

Mixing & Panning Category

Adds all of its Inputs together. Mixes the outputs of all the Sounds in the Inputs field.

Inputs

Inputs are all added together (mixed) so they will be heard simultaneously.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



Monotonizer

Frequency & Time Scaling Category

Removes pitch changes from the input and uses the specified Frequency instead.

Input

Any frequency changes in the Input will be flattened out or removed by this module.

Frequency

This is the new frequency of the monotonized input.

MinInputPitch

This is the lowest frequency you expect in the input. It must include units: hz for a frequency or nn for a notenumber.

MaxInputPitch

This is the highest frequency you expect in the input. It must include units: hz for a frequency or nn for a notenumber.



MultifileDiskPlayer

Sampling Category

This is similar to DiskPlayer except that you specify an array of disk file names rather than a single disk file name.

The value of Index determines which file will play on the next retrigger. (An index of 0 chooses the first file in the array, an index of 1 chooses the second, etc.) Only one disk file will play at any one time, but the choice of file can be made in real time. (To get more than one disk file to play simultaneously, feed this Sound into a MIDIVoice and set the desired polyphony).

You can use a single Rate for all disk files, or make Rate a function of the Index if you want different files to play at different rates.

The MultiFileDiskPlayer can be used whenever you need real-time random access to several different disk recordings through a keyboard or MIDI controller. For example, it can be used to choose from a set of live sound effects and synchronize them by hand to a film, to create a disk-based sampler with a different sample for every key on the keyboard, to perform a composition made up of several, long disk recordings, or (if the trigger is linked to the FrequencyTracker or EnvelopeFollower) as a synchronizable "tape part" that responds to a live performer.

FileNames

List each of the file names that could be triggered, enclosing each of them within single quotes. The Index corresponds to the placement of the filename in this field. In other words, an index of 0 selects the first filename in the field, and index of 1 selects the second filename, etc.

RateScale

This is the rate of playback. For example, use 1 to play back at the original rate, 0.5 for half speed, 2 for twice as fast, etc.

Trigger

When the Trigger becomes nonzero, one event is triggered. You can trigger several events over the course of the total Duration of this program as long as the value of Trigger returns to zero before the next trigger. Some example values for Trigger are:

1	(plays once with no retriggering)
0	(the sound is silent, never triggered)
!KeyDown	(trigger on MIDI key down)
!cc64	(trigger when controller 64 > 0)

You can also paste another signal into this field, and events will be triggered every time that signal changes from zero to a positive value. (See the manual for a complete description of hot parameters, Event Values, and the Global map files).

Gated

NOT IMPLEMENTED YET.

Index

This is an integer that selects which of the disk files should be played when the next trigger is received. An index of 0 selects the first file. If the index is less than 0, it selects the 0th file (the first file in the list). If the index is larger than the length of the file list, it selects the last file in the list.



MultiplyingWaveshaper

Level, Compression, Expansion Category

Multiplies Input by a value read from the Wavetable at an index supplied by the NonlinearInput and attenuates or amplifies the result by multiplying it by Scale.

Can be used as a computationally inexpensive dynamic range controller if the NonlinearInput is a signal fed through a peak detector or RMS detector and the Input is that same signal delayed by some amount. In this situation, the Wavetable describes the attenuation of the output amplitude as a function of input amplitude.

To design a new input-output characteristic function, open the Sample/Wavetable editor and use the InputOutputCharacteristic template to generate a new transfer function with the desired compression/expansion parameters.

NonlinearInput

The output of this Sound is used as an index into the Wavetable.

Input

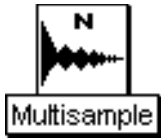
This Sound is multiplied by the value from the Wavetable that is indexed by the NonlinearInput.

Scale

This is a gain control for the output. It can be any positive number.

Wavetable

This is the transfer function that the NonlinearInput indexes into. When used as a dynamic range control, this function describes a multiplier on the output amplitude as a function of the input amplitude.



Multisample

Sampling Category

This provides a quick way to select from a number of samples. The sample files are listed in the Samples field, and the Index field is used to determine which sample file to play whenever the Gate changes to a positive value.

Frequency

Use 0 hz here if you want the Frequency to equal the pitch of the recorded sample. The frequency can be specified in units of pitch or frequency. Different frequencies are obtained by changing the size of the increment through the recorded sample. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released. If the sample file has loop points stored in its header, Kyma will loop the sample for as long as Gate remains positive (so, for example, as long as the MIDI key is held down).

Samples

Takes a list of samples file names, each within single quotes.

Index

An expression whose value is the index into the list of filenames: 0 selects the first file in the list, 1 the second, and so on.

AttackTime

Duration of the attack of an envelope applied to the sample.

ReleaseTime

Duration of the release of an envelope applied to the sample.

Scale

Overall level of the sample.

Loop

Click [here](#) if you want to loop the sample using the loop points specified in the header of the samples file.



MultisegmentEnvelope

Envelopes & Control Signals Category

Similar to the ADSR envelope, except that you can specify an arbitrary number of segments and can specify loop points. See also MultiSlopeFunctionGenerator and GraphicalEnvelope. Use the GraphicalEnvelope except in those cases where you need hot BreakPoints or Levels.

Typical uses include amplitude envelopes, pitch envelopes, and time index functions.

Durations

Enter the durations of each segment of the envelope. You must include the units of time and enclose the duration and its units within curly braces, for example

{!Length s} or {2 s}

The number of Durations must be one less than the number of BreakPoints.

BreakPoints

These are the amplitude values at the endpoints of each segment. There should always be one more breakpoint than there are segment durations. Every time there is a change in slope or a "break" in the line corresponding to the envelope, you have to specify the amplitude at that point (including the very last point in the envelope, since it does not necessarily have to end on a zero). These numbers can be any value from 0 to 1. If you enter a larger value, the amplitude of the envelope will approach that number at the rate required to reach that number in the given duration, but it will stick at the value of 1 once that has been reached.

StartLoop

The number of the first segment included in the loop (where the segments are numbered from 1 to the number of segments).

EndLoop

The number of the last segment included in the loop (where the segments are numbered from 1 to the number of segments).

Level

This is a scale factor (from 0 to 1) for attenuating the overall output level.

Gate

When Gate changes from zero to a nonzero, the envelope will be triggered. Gate must return to zero again before the envelope can be retriggered. If you have specified beginning and ending segments for a loop, the envelope will repeat the loop segments for as long as the Gate is nonzero. If the ending loop segment ends on a higher or lower value than the start of the beginning segment, the entire looped portion will get larger or smaller each time it is repeated (because each segment has a *slope* associated with it, not the absolute values at each point).



MultislopeFunctionGenerator

Envelopes & Control Signals Category

This is similar to the MultiSegmentEnvelope, except that you specify time points and *slopes* between the time points (rather than time points and the levels at those time points), and you cannot loop the envelope.

GraphicalEnvelope is easier to use than the MultiSlopeFunctionGenerator except in those situations where you need *hot* TimePoints and/or Slopes.

The resting value of this envelope is 1. Each time it is triggered, it generates the envelope exactly once.

TimePoints

These are the time points at which the slope of the envelope should change. There should be one more TimePoint than there are Slopes because the slopes specify the slope of a line *between* adjacent pairs of TimePoints.

You must include the units of time and enclose the time point and its units within curly braces, for example

`{!TimePoint1 s}` or `{2 s}`

Slopes

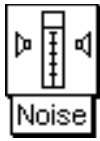
Specify a slope for each pair of adjacent TimePoints (you should end up with one more TimePoint than you have slopes). A slope of 1 is a 45 degree angle, slopes of less than 1 are shallower, and slopes of greater than 1 are steeper. Negative slopes go downward at the same angle as the positive slopes go upward.

Level

This is an overall amplitude scale on the entire envelope.

Gate

When this changes from a zero to a number larger than zero, the envelope is generated exactly once. The resting value of the envelope is the maximum amplitude (1).



Noise

Xtra Sources Category

This generates white or pink noise, which is a stream of pseudo-random numbers. Choose different initial states to generate different streams of random numbers.

To get random numbers at a lower rate, use something like:

1 ms random

in a hot parameter field.

InitialState

Type in a seed value within the range of -1 to 1, excluding zero. (If you type a zero it will be changed to a 1 by the program).

Pink

Check this box to generate pink rather than white noise.



Oscillator

Xtra Sources Category

The Wavetable is treated as a single cycle of a periodic function. There are options for interpolation and modulation. In general, the more options that are selected and more parameters that are time-varying, the more complicated the computation of the Oscillator and the fewer of them you can compute in real time.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Wavetable

Select a wavetable for the oscillator. The Oscillator expects wavetables with 4096 entries.

Modulation

Select whether or not there should be frequency modulation.

Modulator

If Modulation has been set to frequency, then this Sound is the Modulator (otherwise it is ignored). Usually the Modulator is another Oscillator, but it can be any Sound.

MaxMI

This is the value of the modulation index when the Modulator is at its full amplitude.

Interpolation

Choose linear if you would like to interpolate between the values read from the wavetable.

Envelope

This is an attenuator on the output of the Oscillator. Enter 1 (or 0 dB) for the full amplitude. For a time-varying amplitude, paste in a Sound (such as AR, ADSR, or FunctionGenerator) or an Event Value (such as !Volume) in this field.

PitchBend

This is a deviation from the specified Frequency computed as:

$\text{actualFreq} := \text{Frequency} + (\text{Frequency} * \text{pitchBend}).$

The maximum pitchBend value is 2 and the minimum value is 0.

Reset

When reset is nonzero, it resets the phase to zero. In other words, it sets the wavetable index to its initial position.



OscillatorBank

Xtra Sources Category

Generates the sum of several oscillators on the specified waveform, each with its own frequency and amplitude envelope.

NbrOscillators

This is the number of oscillators that will be added together. Each oscillator is associated with a partial from the time-varying spectrum given in the Spectrum field.

BankSize

This is the number of oscillators that will be synthesized at a time. This is important since the signal processor has a maximum number of oscillators it can add at a single time (typically 50-56).

For instance, if NbrOscillators is 100 and BankSize is 50, this Sound will add up two groups of 50 oscillators.

Wavetable

This is the waveform used by all the oscillators.

Spectrum

The Spectrum controls the amplitude and frequency envelopes for each oscillator. This should come from one of the Sounds in the Spectral Sources or Spectral Modifiers categories of the System Prototypes.



OscilloscopeDisplay

Tracking Live Input Category

Displays the Input as an oscilloscope trace on the Virtual control surface. Use the buttons along the bottom of the display to zoom in or out in the time or amplitude dimensions. The value at the cursor point (where the red cross hairs meet) is displayed in the upper left. Clicking on the display freezes it so you can hold down the mouse over specific points to read their exact values.

An Oscilloscope can be placed anywhere along the signal flow path; it does not necessarily have to be the final Sound in a signal flow path (it could, for example, be displaying the Input to the Sound that is actually being heard). If a Sound has more than one Oscilloscope within it, all the traces will be displayed side by side in the Virtual control surface.

You can also view the oscilloscope trace of any Sound by selecting the Sound and then choosing Oscilloscope from the Info menu. (But the menu method only allows you to view one Sound at a time on the Oscilloscope and does not allow you to adjust the trigger frequency for a stable display).

Input

The amplitude of this Sound is continuously displayed on the Virtual control surface, as if by an oscilloscope.

Trigger

In order to see a picture of the waveform that does not drift across the screen, use a PulseTrain here, and set the repetition period of the pulses to equal the inverse of the Input's frequency. That way, the Oscilloscope is triggered once every Input period, and you will see a single period of the Input in the display window.



Output4

Spatializing Category

This Sound routes the four input Sounds to the four output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

Out1

This Sound is routed to output channel 1 of the signal processor.

Out2

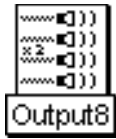
This Sound is routed to output channel 2 of the signal processor.

Out3

This Sound is routed to output channel 3 of the signal processor.

Out4

This Sound is routed to output channel 4 of the signal processor.



Output8

Spatializing Category

This Sound routes the eight input Sounds to the eight output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

Out1

This Sound is routed to output channel 1 of the signal processor.

Out2

This Sound is routed to output channel 2 of the signal processor.

Out3

This Sound is routed to output channel 3 of the signal processor.

Out4

This Sound is routed to output channel 4 of the signal processor.

Out5

This Sound is routed to output channel 5 of the signal processor.

Out6

This Sound is routed to output channel 6 of the signal processor.

Out7

This Sound is routed to output channel 7 of the signal processor.

Out8

This Sound is routed to output channel 8 of the signal processor.



OverlappingMixer

Mixing & Panning Category

Overlaps the start times of its Inputs by the specified OverlapTime.

Inputs

These Sounds will be played one after another, overlapping with each other by the amount of time specified in OverlapTime. The ordering is determined by their position in the Inputs field: left to right and top to bottom.

OverlapTime

This is the amount of time that each Input overlaps with the previous Input. Be sure to include the units of time.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

- 1 (no attenuation)
- 0 (maximum attenuation)
- !Fader1 (continuous controller sets level)
- !KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



Pan

Mixing & Panning Category

Places the Input between the left and right speakers and optionally attenuates the overall output.

Input

This is the signal being attenuated and positioned between the speakers.

Pan

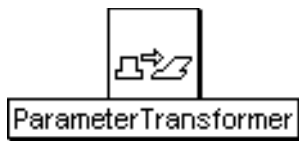
A Pan value of 0 places the sound entirely in the left speaker, and a Pan value of 1 places it entirely in the right. Values inbetween those extremes make the Input source appear as if it were placed somewhere inbetween the two speakers.

Scale

Attenuates the Input.

Type

When Power is selected, the Input is about as loud for Pan = 0.5 as it is for Pan = 0 and Pan = 1. When Linear is selected, the Input is softer at the midpoint than it is at the two extremes.



ParameterTransformer

Scripts Category

Parameters of the Input can be altered or set by statements made in the Transformation field (for full details, see the corresponding tutorial and chapter in the manual). All transformations take place symbolically (in other words, these are not signal processing transformations but transformations to the parameters fields **before** the Input is compiled and loaded into the signal processor--before it has started generating sound).

Input

The parameters of this Sound (and, in turn, any inputs **it** might have) can be set or modified by statements in the Transformation field.

Transformation

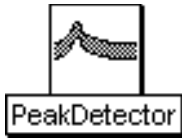
Here are two examples of simple modifications. For examples of more elaborate transformations, see the manual.

To set the all parameters named "frequency" to 400 hz, type:

`snd frequency: 400 hz.`

To double all frequencies, type:

`snd frequency isNil ifFalse: [snd frequency: snd frequency * 2].`



PeakDetector

Tracking Live Input Category

Outputs an amplitude envelope for its Input by tracking increases in the Input's absolute value.

Responds to increases in the Input amplitude within the specified AttackTime and responds to decreases in Input amplitude within the specified ReleaseTime. Scale is an attenuator on the Input amplitude.

Input

This is the Sound whose amplitude is being tracked.

AttackTime

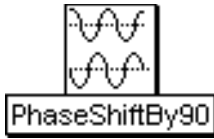
This is the shortest attack time that will be tracked by the PeakDetector. You are specifying that any faster increases in amplitude should be smoothed over.

ReleaseTime

This is the shortest decay time that will be tracked by the PeakDetector. You are specifying that any faster decreases in amplitude should be smoothed over.

Scale

This is an attenuator on the input.



PhaseShiftBy90

Math Category

This is a combination of two filters tuned to do a 90 degree phase shift between the left and right channels at the specified Frequency (by shifting one channel backwards 45 degrees and the other channel forward 45 degrees).

Expand the prototype SingleSideBandRM for an example of how to use this Sound as the Envelope of a QuadratureOscillator to do single side band ring modulation on Input.

NOTE: These filters are very sensitive to Input amplitude. Try attenuating the Input amplitude by 0.05 and gradually adjusting it upwards until you hear distortion and then backing it off a little. (It helps to also look at the output of the PhaseShiftBy90 on the Info Oscilloscope as you adjust the Input amplitude).

Frequency

This is the only frequency at which the 90 degree phase shift occurs. Frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Input

This is the Sound whose left and right channels will shifted 90 degrees out of phase from each other (but only at the specified Frequency).



PresenceFilter

Filters Category

Acts as a band pass or band reject (notch) filter. Specify a center frequency, a bandwidth, and indicate the boost or cut amount in units of dB (negative values are cuts, positive values boosts).

Input

This is the Sound to be filtered.

CenterFreq

The center of the boost or cut region of the spectrum.

Bandwidth

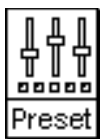
The width of the boost or cut region of the spectrum.

BoostOrCut

Indicate the amount of boost or cut in units of dB. Negative values indicate a cut, positive values a boost.

Scale

Attenuator on the input.



Preset

MIDI In Category

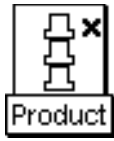
Click the Set to Current Event Values button to save the current settings of faders and buttons in the Virtual Control Surface (and/or the current settings of all MIDI controllers). The current values are written into the EventValues field as a reminder (but they are not editable). Each time you play this Sound, any EventValues in its Input will be initially set to the saved values; you can then change them using the Virtual Control Surface or MIDI.

Input

Any EventValues in this Sound will be initially set to the values shown in the EventValues field. If that field is empty, set the EventValues to reasonable initial values, and then click the Set to Current Event Values button.

EventValues

These are initial values for the EventValues in the Input. If this field is blank, click on the Set to Current Event Values button.



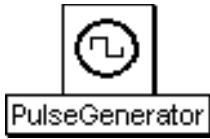
Product

Math Category

Outputs the product of its Inputs. If there are two, audio frequency inputs, this does ring modulation. If one of the Inputs changes at sub-audio frequencies and the other is at audio frequencies, the effect will be like applying an amplitude envelope to the Input that is at audio frequencies.

Inputs

The output of the Product is the product of all the Sounds in this field.



PulseGenerator

Xtra Sources Category

Generates a bandlimited square wave of the specified DutyCycle. The square wave always has a zero DC offset regardless of the pulse width setting; this means that the minimum and maximum of the waveform will change as the pulse width is changed.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Modulation

Select whether or not there should be frequency modulation.

Modulator

If Modulation has been set to frequency, then this Sound is the Modulator (otherwise it is ignored).

MaxMI

When Modulation is set to frequency, this is the value of the modulation index when the Modulator is at its full amplitude.

Interpolation

Choose linear if you would like to interpolate between the values read from the wavetable.

Envelope

This is an attenuator on the output. Enter 1 (or 0 dB) for the full amplitude. For a time-varying amplitude, paste in a Sound (such as AR, ADSR, or FunctionGenerator) or an Event Value (such as !Volume) in this field.

DutyCycle

The proportion of each period that the waveform is in the "up" portion of its cycle. (If you add up all the sample points in a cycle, the sum is zero, no matter what the duty cycle; when the duty cycle is 0.5 the waveform is above zero half the time and below zero for the other half cycle.).

Reset

When reset is nonzero, it resets the phase to zero. In other words, it sets the wavetable index to its initial position.



PulseTrain

Xtra Sources Category

If VariableDutyCycle is unchecked, then PulseTrain's value is 1 for the first sample of each period and zero for the remainder of each period. If VariableDutyCycle is checked, then DutyCycle controls how much of each period's is spent outputting 1 and how much is spent outputting 0.

Period

The amount of time for each period.

If you want a period corresponding to a certain frequency, for example 440 hz, use:

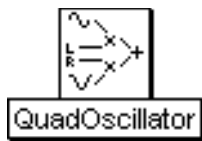
440 hz inverse

VariableDutyCycle

Check here to control the percentage of the period that the output should be one. If unchecked, the output will be one for exactly one sample per period.

DutyCycle

Enter a value between 0 and 1, where 0 means that the output is never 1, 0.5 means that it is 1 for half of each period, and 1 means that it is 1 for all of each period.



QuadOscillator

Xtra Sources Category

Multiplies the left channel of Envelope by a sine wave oscillator and the right channel of Envelope by a cosine oscillator. The output is the sum of the ring-modulated left and right channels. If the Envelope has the same signal but 90 degrees out of phase in the left and right channels, the lower sideband will be cancelled out, leaving only the upper sideband (the sum of the frequencies of the Envelope and the QuadratureOscillator).

Expand the SingleSideBandRM prototype for an example of how to use this as a nonharmonic frequency shifter.

Frequency

This is the frequency of the sine and cosine oscillators. The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Envelope

The left channel of Envelope will be multiplied by a sine and the right channel by a cosine. If this Sound has gone through a PhaseShiftBy90 (forcing its left and right channels to be 90 degrees out of phase with each other at a specified frequency), then putting it through the QuadratureOscillator will perform single side band ring modulation. In this configuration, only the upper sideband is heard. To get the lower sideband alone, use a negative frequency for the QuadratureOscillator (or else swap the left and right channels of Envelope using a ChannelCrossover).



RandomSelection

Scripts Category

Chooses Sounds from the sample space of all Inputs and assigns them start times that are randomly generated according to an exponential distribution of delay times having the specified averageDelay time. The value in Iterations is the total number of Sounds. Seeds are supplied so that the results are repeatable.

Inputs

This serves as the sample space. Sounds in the result are randomly chosen from among these Sounds.

Weights

Supply the relative likelihoods of each Input (where Inputs are ordered according to their positions in the field from top to bottom and left to right). You must specify the same number of likelihoods as there are Inputs. The likelihoods are normalized, so you can use numbers in any range.

AverageDelay

Supply an average delay time for the Input Sounds. (Durations should always be greater than 0.)

Iterations

This is the total number of randomly selected Sounds.

DelaySeed

This is a seed value for the exponential distribution of delay times. Type in an integer less than 65535, for example, 35409.

SampleSeed

This is a seed value for the sample space of subSounds. Type in an integer less than 65535, for example, 35425.

Left

This controls the level of the left input channel. The maximum value is 1 and the minimum is -1. The left channel of the input is multiplied by the value of this parameter. Some example values for Left are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
---	------------------

0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



REResonator

Filters Category

This is a time-varying filter whose coefficients have been derived by analyzing a digital recording (a "sample") using the RE Analysis Tool. RE (resonator/exciter) analysis assumes that the sound was produced by an excitation signal feeding into a resonator. This Sound is the resonator and its input is the excitation.

The most striking results occur when the analyzed signal is from a source whose resonances change dramatically over time (e.g. human speech, singing, instruments like the didgeridoo, mouth harp, or tabla). For analyses of instruments or other sound sources that do not change shape very much over time, the REResonator will sound like a fixed, unchanging filter.

Input

This is the new excitation that you are substituting for the original excitation. Be sure to use extreme attenuation of your input.

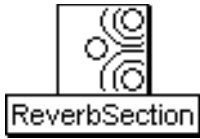
Broadband signals such as noise or pulse trains work best as inputs, because they cover more of the spectrum and will be able to excite all the resonances of the filter.

Wavetable

This is a table of time-varying filter coefficients produced by the RE analysis. Use the RE Analysis Tool to create your own sets of filter coefficients.

TimeIndex

This determines where to read from the sequence of filter coefficients. A value of -1 reads the beginning set of coefficients, and a value of 1 reads the last set of coefficients. A FunctionGenerator whose wavetable is a FullRamp will go through the coefficients in time order. To go through the coefficients at the original rate, set the duration of the FunctionGenerator to be the same as the duration of the original, analyzed sample. Use longer or shorter durations to stretch or compress time.



ReverbSection

Reverb, Delay, Feedback Category

Same as DelayWithFeedback except that the characteristics are specified in terms of DecayTime, the time it takes for the delayed and fed-back input to die away 60 dB below its initial level.

You can use combinations of these Sounds and others to build your own reverberation algorithms.

Type

Choose between comb and allpass filters. Both comb and allpass are delays with feedback. Allpass also adds some of the direct signal to the output in order to make the long term frequency response flat.

Input

This signal is delayed and added to itself.

Scale

An attenuation factor on the Input (where 1 is full amplitude and 0 is completely attenuated so there is no more Input).

DecayTime

This is the time it takes for the signal to die away to 60 dB below its original level.

Delay

The maximum delay time. The proportion of this time that is actually used is determined by multiplying this value by DelayScale. Kyma needs to know the maximum possible delay in order to allocate some memory for this Sound to use as a delay line, but the actual delay can vary over the course of the Sound from 0 s up to the value of DelayTime.

DelayScale

The proportion of DelayTime that is actually used as the delay, where 0 is no delay, and 1 is equal to the value in the DelayTime field.

Wavetable

In almost all situations, this should be set to Private, so Kyma can allocate some unused wavetable memory to be used as a delay time for this program. (The only time you would want to name this wavetable is if you would like multiple delays or resonators to share a single delay line. In that case, you would type a name for the wavetable and make sure that the other delays use the same name for their wavetables.)

Prezero

Check this box to start with an empty delay line when this program starts. If Prezero is not checked, the delay line may have garbage in it from previous programs. This can have interesting, if unpredictable, effects.

Interpolation

When Linear is selected, changes to DelayScale will be interpolated, causing smoother changes to the delay.

When None is selected, changes to DelayScale are not interpolated, resulting in zipper noise.
For fixed delays, it is better to select None, since that uses less DSP resources.



RhythmicCellularAutomaton

Scripts Category

This Sound is based on the one-dimensional cellular automata described by Stephen Wolfram in his book "Theory and Applications of Cellular Automata". The state, an n-place binary number, where n is the number of Inputs, determines when the Input is turned on and when it is turned off in a given generation. An integer, rule, is used to determine the next state.

This differs from the CellularAutomaton in that the state is interpreted "horizontally" (rhythmically) rather than "vertically" (harmonically). Each generation of the automaton is interpreted as a rhythmic pattern, where a 1 means the Input is turned on, and a 0 means the Input is turned off (for the duration of the Input).

Input

This Sound is repeated in a pattern determined by the state of the automaton in each generation. When a value in the state array is 1, the Sound is played, and when a value in the state array is 0, there is a silence of the same duration as the Sound. This field accepts only one Sound at a time.

Rule

If you look at the rule and the state as 8-bit binary numbers, you can use the rule to compute the next generation of the state. Number the positions of the digits 0 to 7 counting from right to left. To compute the next generation of position P of the state: Take the number at position P, P+1 and P-1 (i.e. the number at position P and its neighbors to the right and to the left). Think of this as a three digit binary number N that you are going to use as an index into the binary number representing the rule. Look at position N of the rule. The value at position N of the rule is the value of position P in the next generation of the state.

For example, if the rule is 2r10101110 and the state is 2r10010101, let's compute the next generation state of position 3. The current value at position 3 of the state is '0'. Looking at position 3 and its two neighbors as a binary number, we get '101' or the number 5. Using this number as an index, look at position 5 of the rule; it is '1'. So the value of position three in the next generation is '1'. When you reach the rightmost or leftmost digit in the state, make the assumption that there is an infinite number of zeroes extending both leftwards and rightwards.

Generations

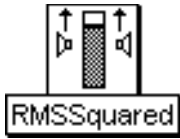
This is the number of generations.

InitialState

Imagine the initial state as an n-bit binary number where n is the size of the collection of Inputs. Each digit in the binary number corresponds to a Sound in the Input collection; use a 1 to indicate that a Sound is initially on, a 0 to indicate that it is initially off.

GenerationLength

This is the number of bits in each generation.



RMSSquared

Tracking Live Input Category

This can be used to get an estimate of the amplitude envelope of the Input. The output is

$$\text{input}^2 * \text{timeConst} + \text{prev} * (1 - \text{timeConst})$$

This is the root mean square of the input without the final square root at the end.

Input

This is the Sound whose amplitude is tracked.

TimeConstant

This controls the response time. Longer timeConstants result in smoother outputs at a cost of losing some of the detail in the attacks. Short timeConstants result in outputs that respond more immediately to attack transients but that may not be as smooth for the steady state portions. For a constant input at maximum amplitude, this is the time required for the output to reach 60% of the full output amplitude. (Note that the output may never reach the maximum possible amplitude since it is the average of the squares of the amplitudes).

Scale

Attenuates the input amplitude.



RunningMax

Math Category

Output is the maximum of all Input amplitudes seen so far, from the start of the Sound until the current time. To reset the maximum to zero and restart on calculating the running max, set Reset to a nonzero value. By the end of the Sound, if there have been no resets, the value is the maximum of all the Input's sample points.

Input

This is the Sound whose maximum amplitude over its entire duration is being computed.

Reset

When this Sound becomes nonzero, it resets the running maximum.



RunningMin

Math Category

The output of this Sound is the minimum amplitude of its Input as seen so far. Whenever Reset is nonzero, the current minimum is thrown away, and the process starts over again. If Reset is always zero, the final value of this Sound is the minimum of all the output values of its Input.

Input

This is the Sound whose minimum amplitude is being computed.

Reset

Whenever this Sound is nonzero, the min is reset to the maximumAmplitude and the process of keeping track of the minimum seen so far begins again.



Sample

Sampling Category

Plays the specified Sample from the wavetable memory of the signal processor. If there is a loop stored in the header of the sample file or if you have SetLoop checked, the sample will play once up through the LoopEnd; then it will loop back to LoopStart and continue looping for as long as Gate has a positive value; when Gate returns to zero, the sample will play on through LoopEnd to the end of the sample file.

Frequency

Use 0 hz here if you want the Frequency to equal the pitch of the recorded sample. The frequency can be specified in units of pitch or frequency. Different frequencies are obtained by changing the size of the increment through the recorded sample. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released and will finish playing through the sample and then stop.

If the sample file has loop points stored in its header, Kyma will loop the sample for as long as Gate remains positive (so, for example, as long as the MIDI key is held down).

Sample

Choose a sample from among those stored in the Wavetables folder or directory. When you compile/load/start, Kyma will read the sample from the hard disk of the host computer and load it into the wavetable memory (the sample RAM) of the signal processor. This Sound then reads the sample from the memory of the signal processor, not directly off the disk.

SetLoop

Check this box if you would like to set the loop points using the LoopStart and LoopEnd parameter fields.

LoopStart

When SetLoop is checked, this is the start point of the loop (otherwise it is ignored). Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the start point should occur. (To compute the

exact time within the sample where the start point occurs, multiply LoopStart's value by the total duration of the sample. For example, if your sample is 5 seconds long and LoopStart is set to 0.2, then the beginning of the loop is 1 second into the sample.)

LoopEnd

When SetLoop is checked, this is the end point of the loop (otherwise it is ignored). Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the end point should occur. (To compute the exact time within the sample where the end point of the loop occurs, multiply LoopEnd's value by the total duration of the sample. For example, if your sample is 5 seconds long and LoopEnd is set to 0.4, then the end of the loop occurs at 2 seconds into the sample.)

LoopFade

When checked, this puts a quick fade in at the beginning of a loop and a quick fade out at the end to help minimize any clicks due to discontinuities in the waveform between the beginning and end of the looped section.

Start

This is the start point of playback within the sample. Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the start point should occur. (To compute the exact time within the sample where the start point occurs, multiply Start's value by the total duration of the sample. For example, if your sample is 5 seconds long and Start is set to 0.2, then the beginning of the playback is 1 second into the sample.)

End

This is the end point of the sample playback. Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the end should occur. (To compute the exact time within the sample where the end occurs, multiply End's value by the total duration of the sample. For example, if your sample is 5 seconds long and End is set to 0.4, then the end of the playback occurs at 2 seconds into the sample.)

FromMemoryWriter

Check FromMemoryWriter when the wavetable does not come from a disk file but is recorded by a MemoryWriter in real time.

AttackTime

Duration of the attack of an envelope applied to the sample.

ReleaseTime

Duration of the release of an envelope applied to the sample.

Scale

Overall level of the sample.



SampleAndHold

Sampling Category

A SampleAndHold holds onto the current value of its Input for the duration specified in HoldTime. While it is holding onto this value, it ignores any changes in its Input's value. When HoldTime has expired, a SampleAndHold looks at its Input's value again, and holds onto THAT value for HoldTime and so on.

This effectively lowers the sample rate on the Input.

Try pasting this Sound into another Sound's Frequency field and multiplying it by the desired range of frequencies and adding an offset frequency to it, for example:

$$4 \text{ c} + ([\text{SampleAndHold}] * 12 \text{ nn})$$

where [SampleAndHold] is a this Sound copied and pasted into another Sound's Frequency field.

Input

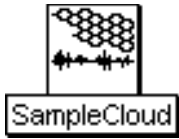
This is the Sound whose output is periodically sampled by the SampleAndHold.

HoldTime

The amount of time that each sampled value is held before the Input is sampled again. If you think of the SampleAndHold as downsampling its input, then this is the period of the new, lower sample rate.

OffsetTime

This is the amount of time to initially wait before starting the process of sampling and holding.



SampleCloud

Sampling Category

Generates a cloud of short-duration grains, each using GrainEnv as an amplitude envelope on a short segment of sound taken from the specified Sample at a point in the sample given by the TimeIndex. The density of simultaneous grains within the cloud is controlled by Density, with the maximum number of simultaneous grains given by MaxGrains. Amplitude controls an amplitude envelope over the *entire* cloud (each individual grain amplitude is controlled by GrainEnv). Similarly, Duration is the duration of the entire cloud, not of each individual grain. You can control the Frequency, stereo positioning, time point within the sample, and the duration of each grain as well as specifying how much (if any) random jitter should be added to each of these parameters (giving the cloud a more focused or a more dispersed sound, depending on how much randomness is added to each of the parameters).

Sample

Enter the name of a mono sample file or click the disk icon to choose a file from the file dialog. This is the source material for each of the short duration grains.

GrainEnv

This is the shape of the amplitude envelope on each grain. The wavetables in the Windows category make the classic, smooth grain envelopes, and some of the shapes in Impulse Responses also give interesting results.

MaxGrains

This is the maximum number of simultaneous grains. The smaller this number, the less computational power the SampleCloud requires (but the less dense the texture you can generate). On a Cappybara-66 you should be able to get around 28 simultaneous grains per cloud. For even denser textures, put more than one SampleCloud into a Mixer, and give each cloud a different Seed value.

Amplitude

This is an overall level applied to the entire cloud. Paste an envelope generator into this field to give an overall envelope to the cloud.

Density

Small Density values result in a sparse texture; large Density values generate a dense texture. This controls the average number of new grains starting up at any given point in time.

GrainDur

This is the duration of each individual grain.

GrainDurJitter

Adds some amount of random jitter to the grain durations. When set to 1, the durations vary randomly from 0 to twice the specified duration. When this is set to 0, all grains will have a duration of GrainDur. In other words, the actual grain duration for each grain is:

$$\text{GrainDur} + (<\text{rand}> * \text{GrainDurJitter} * \text{GrainDur})$$

where <rand> is a random number between -1 and 1.

Pan

This is the stereo position of each new grain where 0 is hard left, 0.5 is in the middle, and 1 is hard right.

PanJitter

This is the amount of random deviation added to the pan position. The larger this number, the more diffuse the apparent location, and the smaller the number, the more localized the sound.

Seed

This should be a number between 0 and 1. The seed provides a starting point for the random number generator, so each different seed results in a different (but repeatable) sequence of random numbers. When adding several SampleGrains with the same control parameters together in a Mixer, give each of them a different seed in order to ensure that each of them has *different* random jitter added to its parameters (otherwise, they will just double each other).

FromMemoryWriter

Check this box to granulate the live input or to granulate a sample that is being changed by a MemoryWriter as the granulation is going on. This SampleCloud should be fed into a Mixer along with a MemoryWriter that is recording something into the sample that you are granulating with the SampleCloud. The SampleCloud should be fed through a TimeOffset of at least 1 sample, so it is reading *after* the sample is written.

TimeIndex

This is a pointer into the Sample memory. -1 points to the beginning of the sample, 0 points to the middle, and 1 points to the end of the sample. Grains are selected from this point and from random positions in the neighborhood (whose size is determined TimeIndexJitter) around this point.

To read through the sample in linear, forward time, you can use something like:

!KeyDown fullRamp: 10 s

which will scan the sample from beginning to end over the course of 10 seconds each time it receives a MIDI note event.

To remove the element of time from the sample, set TimeIndex to a fixed position like 0 (the middle of the sample), and increase TimeIndexJitter to its maximum value. Then grains will be chosen at random from all different time points within the sample.

TimeIndexJitter

TimeIndex is a time point in the Sample, and TimeIndexJitter is an amount of random deviation forward or backward in time from the one point specified TimeIndex. A TimeIndexJitter of zero means that all grains will be chosen from the single point specified in TimeIndex, whereas a TimeIndexJitter of 1 means that grains may be chosen at random from any time point in the entire sample.



SamplesFromDiskSingleStep

Sampling Category

As long as the Trigger is greater than zero, the SamplesFromDiskSingleStep will read samples from the disk file; if the Trigger is less than or equal to zero, the last sample read will be output. Gate resets the pointer to the beginning of the file.

FileName

This is the name of a sample file that you have previously created either in Kyma or in another application.

FilePosition

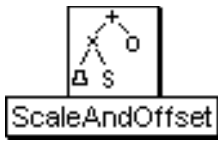
This is the first sample point to play back.

Trigger

As long as the Trigger is greater than zero, the SamplesFromDiskSingleStep will read samples from the disk file; if the Trigger is less than or equal to zero, the last sample read will be output. PulseTrain is a good Sound to use as a source of periodic triggers, and by putting an Event Value in the PulseTrain's Period field, you can control the rate at which the triggers occur.

Gate

Each time this value becomes positive, the Sound will start over again from the beginning of the sample. Enter a 1 in this field to play the Sound exactly once. If you use an EventValue (for example, !KeyDown) in this field, you can restart the sound multiple times.



ScaleAndOffset

Math Category

The output of this Sound is:

$$(\text{Input} * \text{Scale}) + \text{Offset}$$

This can be useful for changing the minimum value and range of a control signal before using it to control another Sound, as for example, in scaling or offsetting the left and right channel outputs of a SpectrumFromRAM before they are fed into an OscillatorBank. (However, for those cases when the control signal is pasted directly into a hot parameter field, it may be more straightforward to just use regular arithmetic to scale or offset the value in the parameter field itself).

Input

The output of this Sound is multiplied by Scale and then the added to Offset.

LeftScale

Multiplier on the left channel. The range of allowable values is -2 to +2.

LeftOffset

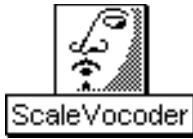
Offset on the left channel. The range of allowable values is -1 to +1.

RightScale

Multiplier on the right channel. The range of allowable values is -2 to +2.

RightOffset

Offset on the right channel. Offset on the left channel. The range of allowable values is -1 to +1.



ScaleVocoder

Filters Category

Vocoder whose center frequencies are tuned to a base pitch and a scale.

Input

This is the source material to be filtered by the SideChain-controlled filters. This Sound is heard directly, through the filters (whereas the SideChain is never heard directly). For example, if you want to make an animal talk, put a sample of the animal sound here and put a sample of speech (or use a microphone) as the SideChain.

The best Inputs tend to be fairly broad band signals that have energy in each of the frequency bands covered by the resynthesis filter bank. For example, Noise or an Oscillator on a waveform with lots of harmonics (such as Buzz128) will work well because they generate energy over the full frequency range.

SideChain

Sometimes referred to as the "modulation", this Sound is never heard directly; it controls the amplitudes of the filters in the bank.

TimeConstant

This determines how quickly the amplitude envelopes on the filters will respond to changes in the SideChain. For precise, intelligible results, use values less than 0.1 s. For a more diffuse, reverberated result, use a longer TimeConstant.

NbrBands

This is the number of band pass filters in the filter bank.

BankSize

This is the number of filters per processor. Type

default

to get the standard number of filters per processor. If you are running out of time, try reducing the default size, for example

default * 0.75

Tonic

This is the tonic or first pitch in the scale.

Intervals

This is the interval pattern of the scale in half steps. For example, a major scale would be

0 2 4 5 7 9 11

Arithmetic expressions should be enclosed in curly braces, for example

{!SmallInterval1 rounded nn}

The scale can have any number of steps, and the steps are repeated in each octave for as many bands as you have specified.

SideLevel

Controls the level of the SideChain Sound before it is fed into the analysis filters.

InputLevel

Controls the level on the input Sound before it goes through the filters.

InBandwidth

Control on the bandwidth of the filters on the Input Sound.

SideBandwidth

Control on the bandwidth of the filters on the Sidechain Sound.

Tone

A tone control where higher values emphasize higher frequencies, and lower values emphasize lower frequencies. Rolloff determines the width of the tone control filter.

Rolloff

This controls the steepness of the edges of a weak tone control filter on the Input. Use 1 if the edges should rolloff precipitously at LoCutoff and HiCutoff. Use smaller numbers if you would like the attenuation to start sooner and take longer.

Gain

You can boost or cut the overall output level here.



Script

Scripts Category

A Script is a handy way to construct Sounds algorithmically (rather than piecing them together graphically in the Sound editor). The constructed Sound will be a Mixer of several Inputs, each with its own start time (or TimeOffset).

A Script is like any other Sound in that it can be used as an Input to a more complex Sound; for example, a Script can contain variables and can even be used as an Input to another Script.

Inputs

Use the script to schedule each of these Sounds at a specific time and to supply values for any variables in the Sound's parameters. (Script actually uses each of these Input Sounds as a template or model for creating new instances of the Sounds with specific values at specific times. Multiple instances of a Sound can be scheduled from the script by specifying simultaneous start times or overlapping durations.)

Script

The script supplies start times for the Sounds in the Inputs field and, optionally, sets the values of any variable parameters. To specify an event in the script, type:

`<name of an Input> start: <aTime in s or samp> {<variableParameterName>: <aValue>}`.

In other words, type the name of an Input Sound, a space, the word "start" followed by a colon and then a space, a start time followed by units of samp or s or beats, and then any number of `<parameter: value>` pairs followed by a period. The `<parameter: value>` pairs consist of the name of a variable in the Input, a colon, a space, and then a value for that variable. To specify the length of a beat, assign the desired metronome setting to the variable MM. If an Input takes another Sound as an argument, you can supply it from the script as a parenthesized event with no start time.

Any Smalltalk expression can appear in the script, including temporary variable declarations and control structures like loops.

See the manual for more details and examples.

Left

This controls the level of the left output channel. The maximum value is 1 and the minimum is -1. The left channel of the output is multiplied by the value of this parameter. Some example values for Left are:

1 (no attenuation)
0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

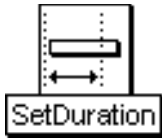
Right

This controls the level of the right output channel. The maximum value is 1 and the minimum is -1. The right channel of the output is multiplied by the value of Right. Some example values for Right are:

1 (no attenuation)

0 (maximum attenuation)
!Fader1 (continuous controller sets level)
!KeyVelocity (MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



SetDuration

Time & Duration Category

Sets the duration and start time of its input. (It is the equivalent of dragging the input Sound into a timeline and changing its duration and start time graphically). Without the SetDuration the Input Sound's program would continue running indefinitely; with the SetDuration you can specify that it should stop after a given amount of time.

Input

Duration sets the duration of this Sound and StartTime sets its start time relative to the SetDuration.

StartTime

Start time of the Sound in Input relative to the start time of the SetDuration. Must be a value greater than zero. Examples of startTimes:

0
1 samp
440 hz

Duration

Duration of the Input. Can be specified in seconds, samples, or in terms of frequency (where the duration will be the duration of one period at that frequency). Must be a value greater than zero. Examples of durations:

1 s
44100 samp
440 hz



SetRange

Math Category

This maps the output range of the Input to the specified range of newMin to newMax.

Input

The output of this Sound is scaled to a range of newMin to newMax. Set oldMin and oldMax to the current output range of this Sound (typically -1.0 to 1.0 or 0 to 1.0). For example, a FunctionGenerator that steps through the wavetable #ramp has a range of 0 to 1.0, but if the wavetable is #sine the range is -1.0 to 1.0.

OldMin

The current minimum value of the Input. This is typically -1.0 (for full range wavetables) or 0 (for wavetables like #ramp that never go negative).

OldMax

The current maximum output of the Input. Typically, this is the full amplitude: 1.0.

NewMin

This is the new minimum output.

NewMax

This is the new maximum output.



SimplePitchShifter

Frequency & Time Scaling Category

Shift the pitch of the input up or down by an interval (given in half steps).

Input

The frequency of this input will be shifted up or down by the given interval. Works best on monophonic inputs with a strong formant structure.

Interval

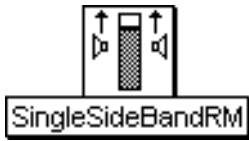
A positive or negative number of halfsteps by which to shift the input's pitch up or down. This does not have to be an integer but can include fractions of halfsteps.

MinInputPitch

This is the lowest frequency you expect in the input. It must include units: hz for a frequency or nn for a notenumber.

MaxInputPitch

This is the highest frequency you expect in the input. It must include units: hz for a frequency or nn for a notenumber.



SingleSideBandRM

Frequency & Time Scaling Category

Does nonharmonic frequency scaling of the input. Takes the input and does a 90 degree phase shift between the left and right channels at the frequency specified in the Frequency parameter field. Multiplies this by a QuadratureOscillator with sine in the left and cosine in the right. The resulting ring modulation gives you sum and difference frequencies but, because they are 90 degrees out of phase, the difference frequency is mostly cancelled out, leaving you with single side band modulation. Expand to see how this is put together.

Input

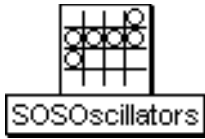
This signal will be ring modulated to scale its frequency by the specified FreqScale.

Frequency

This is the frequency at which there will be perfect cancellation of the difference frequency side-band. The further the input is from this frequency, the less cancellation there will be and the more the result will be like regular ring modulation.

FreqScale

Any part of the input that was at Frequency will be scaled by this ratio.



SOS Oscillators

Xtra Sources Category

Generates the sum of several oscillators on the specified waveform, each with its own frequency and amplitude envelope.

Spectrum

This should be either a SpectralShape or an SOSAnalysis. The Spectrum controls the amplitude and frequency envelopes for each oscillator.

CascadeInput

The left channel of this input is mixed with the outputs of the oscillators.

NbrOscillators

This is the number of oscillators that will be added together. Each oscillator is associated with a partial from the Input analysis, starting from the partial number associated with the firstOscillator.

FirstOscillator

This is the partial number to be resynthesized by the first oscillator in the bank. For example, set this to 1 if you want the lowest frequency oscillator to correspond to the fundamental. If you want to skip the fundamental, set this to 2.

If you are mixing two or more OscillatorBanks, they can cover different portions of the spectrum. For instance, one OscillatorBank might have 1 as its FirstPartial and 10 as the number of partials; the next OscillatorBank might have 11 as its FirstPartial and 10 as its number of partials; and a third might have 21 as its FirstPartial and 10 as its number of partials.

Wavetable

This is the waveform used by all the oscillators.



SoundCollectionVariable

Variables Category

This represents a collection of Sounds. It can appear in any parameter field that takes more than one Sound. It is typically used when creating new Sound classes that have an arbitrary number of inputs.



SoundToGlobalController

Tracking Live Input Category

Takes a number, a pasted Sound, or an Event expression as its input and generates a corresponding EventValue (either a single event or a continuous controller stream) which, to all other Kyma Sounds, looks the same as EventValues coming from the Virtual control surface or from an external MIDI source.

GeneratedEvent

Enter an EventValue name (including the exclamation point prefix) for the generated EventValue.

Value

Paste a Sound or enter a number or EventValue here. The constant or time-varying value here will be translated into an EventValue named in GeneratedEvent.



SpectralShape

Spectral Sources Category

A SpectralShape sets the frequencies and amplitudes of oscillators in an OscillatorBank according to the Spacing and SpectralEnvelope parameters. This kind of Sound makes sense only when used as an Input to an OscillatorBank. Frequencies are output on the right channel and their corresponding amplitudes are output on the left channel.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Spacing

This is the spacing between the partials and should be specified in units of frequency. To specify harmonic partials, set the Spacing to be the same as the Frequency. For example, if you have set Frequency to !KeyNumber nn, then setting Spacing to !KeyNumber nn will tell the OscillatorBank to generate harmonics of !KeyNumber nn.

NbrPartial

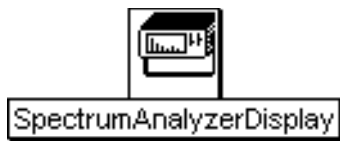
This is the number of (amplitude,frequency) pairs that the SpectralShape will supply to an OscillatorBank. For example, if there are 20 partials, this Sound will output the amp1 and freq1 on the first sample, amp2 and freq2 on the second sample, and on through amp20 and freq20 on the 20th sample. Then it will start over again with amp1 and freq1.

Wavetable

The shape stored in this wavetable is interpreted as the shape of the spectrum, from 0 hz up to half the sampling rate. An OscillatorBank can use this table to set the amplitude of each of its oscillators according to that oscillator's frequency. For example, if the frequency falls in a region with a low amplitude in this table, it will be attenuated in the OscillatorBank. To see the spectral envelope, open this file using File open with the file type set to Samples file. If the OscillatorBank waveform is Sine, and you have chosen harmonic spacing, then this shape will be something like a filter acting on a bandlimited pulse train (equal amplitude, harmonically spaced sine waves).

Scale

Used as an overall amplitude scale applied equally to all of the oscillators.



SpectrumAnalyzerDisplay

Tracking Live Input Category

A real-time spectrum analyzer. Displays the spectrum of the Input on the Virtual control surface. Use the buttons below the display to zoom in or out in the frequency or magnitude dimensions. The value at the cursor point (where the red cross hairs meet) is displayed in the upper left. Clicking on the display freezes it so you can hold down the mouse over specific points to read their exact values.

A SpectrumAnalyzer can be placed anywhere along the signal flow path; it does not necessarily have to be the final Sound in a signal flow path (it could, for example, be displaying the spectrum of the Input to the Sound that is actually being heard). If a Sound has more than one SpectrumAnalyzer within it, all the spectra will be displayed side by side in the Virtual control surface.

You can also view the real-time spectrum of any Sound by selecting the Sound and then choosing Spectrum analyzer from the Info menu. (But the menu method only allows you to view one Sound at a time on the SpectrumAnalyzer and does not allow you to adjust the windowing function or the length of the FFT, except by changing the Preferences).

Input

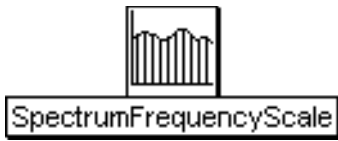
The spectrum of this Sound is continuously displayed on the Virtual control surface, as if by a real-time spectrum analyzer.

Window

Window weighting function applied to the analysis window of the FFT used to compute the spectra.

Length

Length of the FFT. Ideally it should be the same as the number of samples in the period of the lowest frequency or fundamental frequency.



SpectrumFrequencyScale

Spectral Modifiers Category

Takes a spectral source (which must be harmonic) as its input and scales the frequency envelopes without changing the amplitude envelopes. This allows you to shift the pitch of the resynthesis, while leaving the formants at their original frequencies. The SpectrumFrequencyScale should be fed to an OscillatorBank in order to resynthesize the newly scaled spectrum.

Spectrum

Should be a Sound from the spectral sources category of the Prototypes (based on an harmonic analysis).

Scale

All frequencies in the spectrum will be multiplied by this scale factor. For example, use 2 to scale up by one octave, 1 for no change, 0.5 for down by one octave. You can get other intervals by using a ratio of two pitches that have been converted to hertz. For example, to get a half step up, you could use

$4\text{ c sharp hz} / 4\text{ c hz}$
or to shift down by a perfect fifth, you could use
 $4\text{ c hz} / 4\text{ g hz}$

To control the pitch from the MIDI keyboard, use the ratio of !Pitch to the original pitch of the recording. For example, if the original recording is a 3rd octave b, you could use

$!Pitch\text{ hz} / 3\text{ b}$



SpectrumInRAM

Spectral Sources Category

This Sound is used only as the Spectrum input to an OscillatorBank.

It reads an analysis file that contains a series of spectra indexed by TimeIndex. It outputs a spectrum as a sequence of (amplitude,frequency) pairs on every sample tick for nbrPartials samples. After nbrPartials samples, it starts over again from the fundamental and outputs the entire spectrum again.

Frequency

Use Default to leave the frequency unchanged from the original analysis. Otherwise, the frequency envelopes will be altered to scale the base pitch of the analysis to the value listed in this parameter field.

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Level

This is a control on the overall amplitude of all the partials. Enter 1 to leave all amplitudes as they are; numbers larger than one result in a gain, and numbers less than one result in attenuation.

TimeIndex

This selects where we are in the series of spectral snapshots. The first snapshot is at -1, the middle snapshot is at 0, and the last snapshot is at 1. To go through the series in linear time, use a FunctionGenerator whose Duration equals the duration of the original recording and whose Wavetable is FullRamp (which goes from -1 to 1). Change the Duration of the FunctionGenerator to go through the spectra at different rates. Change the wavetable to go through the spectra in a different order.

Analysis

Use a spectrum file from the Wavetables folder or directory. These files came from spectral analyses performed on digital recordings by the Spectral Analysis Tool or by Lemur.

NbrPartials

This is the number of partials you want to output for the resynthesis. Use Default to output all of the partials in the file.

FirstPartial

This is the first analyzed partial that you want to output--usually it is partial number 1. If you want to skip over some of the lower partials, enter a higher number here.



SpectrumLogToLinear

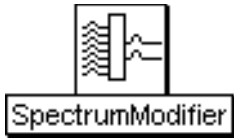
Spectral Modifiers Category

A spectrum can be in one of two forms: linear frequency or logarithmic frequency. This Sound converts a logarithmic frequency spectrum input into a linear frequency spectrum output.

Generally, a spectrum that comes from a spectrum file has logarithmic frequencies, and a spectrum generated in real time has linear frequencies.

Spectrum

This logarithmic frequency spectrum input is converted to linear frequency and then output.



SpectrumModifier

Spectral Modifiers Category

A SpectrumModifier takes one of the Sounds from the Spectral Sources category of the Prototype strip as its input and modifies the spectrum. To resynthesize the modified spectrum, feed the SpectrumModifier into the spectrum input of an OscillatorBank.

In order to modify the output of a spectral source, the SpectrumModifier selects or rejects tracks of the spectrum according to some criteria, and then it optionally scales and offsets each frequency and/or amplitude value of the selected tracks.

Decide whether to select or reject the tracks that meet the criteria.

Then decide whether the rejected tracks should have their amplitudes set to zero or whether they should simply pass through unaffected by the scale and offset modifications.

Then set the selection (or rejection) criteria, including frequency range, track number range, or amplitude range. The frequency and amplitude hysteresis values can prevent tracks that are close to the selected range from popping in and out as they cross the threshold. Probability is the likelihood (ranging from 0 up to 1) that a track will be selected (or rejected) on each frame.

Finally, you can choose to scale and/or offset either the frequency or amplitude (or both) on each frame of each selected track.

Spectrum

This is the spectrum that will be modified; it should be one of the classes of Sound found in the Spectral Sources category (e.g. LiveSpectralAnalysis, SpectrumInRAM). The SpectrumModifier assumes linear (rather than log) frequencies, so you may see a dialog asking you to insert a SpectrumLogToLinear module inbetween the spectral source and the SpectrumModifier.

Select

Check this to specify the criteria for *selection*. Otherwise, the tracks that meet the criteria will be *rejected*. Unchecking this box is like placing a logical NOT after all of the selection criteria.

LoTrack

Enter an integer track number. Only this track and higher-numbered tracks will be selected.

HiTrack

Enter an integer track number. Only this track and lower-numbered tracks will be selected. To be certain of selecting all tracks, enter a number much larger than the highest possible track number (e.g. 10000).

LoFreq

Enter a pitch or frequency with units. On each frame, a track will be selected if the value of the frequency envelope on that frame is at this frequency or a higher frequency. Use FreqHysteresis to prevent tracks from popping in and out on each frame if they are wavering around this frequency.

HiFreq

Enter a pitch or frequency with units. On each frame, a track will be selected if the value of the frequency envelope on that frame is at this frequency or a lower frequency. Use FreqHysteresis to prevent tracks from popping in and out on each frame if they are wavering around this frequency.

FreqHysteresis

Enter a frequency or pitch with units that is smaller than the value of LoFreq. If a track is currently selected, its frequency will have to drop this much *lower* than LoFreq in order to be rejected. If a track is currently unselected, it will have to be this much *higher* than LoFreq in order to become selected. (And similarly, the frequency of a selected track must be this much *higher* than HiFreq in order to be deselected, and the frequency of a rejected track would have to be this much lower than HiFreq in order to switch from being rejected to selected).

Adjust this value to keep tracks that are close to LoFreq or HiFreq from switching between on and off on every frame.

FreqScale

Multiply the frequency of each selected track by this number between 0 and 1.

FreqOffset

Add this number (between 0 and 1) to the frequency value of each selected track.

LoAmp

Enter an amplitude value between 0 and 1 (or from -1000 to 0 dB). On each frame, a track will be selected if the value of the amplitude envelope on that frame is at this amplitude or a higher amplitude. Use AmpHysteresis to prevent tracks from popping in and out on each frame if they are wavering around this amplitude.

HiAmp

Enter an amplitude value between 0 and 1 (or from -1000 to 0 dB). On each frame, a track will be selected if the value of the amplitude envelope on that frame is at this amplitude or a lower amplitude. Use AmpHysteresis to prevent tracks from popping in and out on each frame if they are wavering around this amplitude.

AmpHysteresis

Enter a number between 0 and 1 but smaller than the value of LoAmp. If a track is currently selected, its amplitude will have to drop this much *lower* than LoAmp in order to be rejected. If a track is currently unselected, it will have to be this much *higher* than LoAmp in order to become selected. (And similarly, the amplitude of a selected track must be this much *higher* than HiAmp in order to be deselected, and the amplitude of a rejected track would have to be this much lower than HiAmp in order to switch from being rejected to selected).

Adjust this value to keep tracks that are close to LoAmp or HiAmp from switching between on and off on every frame.

AmpScale

Multiply the amplitude of each selected track by this number between 0 and 1.

AmpOffset

Add this number (between 0 and 1) to the amplitude value of each selected track.

Probability

Enter a likelihood from 0 to 1. Numbers larger than 1 will be clipped to 1 (the maximum likelihood). On each frame and for each track, this is the likelihood that the track will be selected on this frame. Use 1 to say that the track will be selected 100% of the time, use 0.5 to give it a 50-50 chance of being selected, and use 0 to indicate that it will never be selected. You can make the likelihood a function of the track number. For example,

TrackNumber / 128

would make the higher tracks more likely to be selected on each frame than the lower tracks, and:

$(\text{TrackNumber} - 1) \bmod 2$

would make the even-numbered tracks 100% likely, and the odd-numbered tracks 0% likely (because an odd number minus 1 is an even number, and an even number modulo 2 is zero, while an odd number modulo 2 is 1).

Seed

Enter a number from -1 to 1. This is the seed for the random number generator used in conjunction with the value of Probability to determine whether a track should be selected on a given frame.

HearAll

Check this box to hear all the tracks, both the selected and the rejected. Uncheck it to set the rejected tracks' amplitudes to zero.

(Only the selected tracks are affected by the FreqScale, FreqOffset, AmpScale, and AmpOffset, so check the HearAll box to hear all tracks but modify only the selected tracks).



SpectrumOnDisk

Spectral Sources Category

This can be used in place of a SpectrumInRAM as the input to an OscillatorBank. The difference is that this reads the analysis file directly off the disk, rather than first loading the analysis file into RAM. This is helpful for SOS analyses that are too long to fit into sample RAM.

Unlike the SpectrumInRAM, this Sound can only go through the analysis envelopes in forward-time order. (The SpectrumInRAM TimeIndex parameter lets you read the analysis at any point in time and in any time order with any function). In this Sound, you can, however, control the Rate at which you go forward through the analysis file.

Frequency

Use Default to leave the frequency unchanged from the original analysis. Otherwise, the frequency envelopes will be altered to scale the base pitch of the analysis to the value listed in this parameter field.

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenummer)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenummer)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Level

This is a control on the overall amplitude of all the partials.

Enter 1 to leave all amplitudes as they are; numbers larger than one result in a gain, and numbers less than one result in attenuation.

RateScale

This controls the rate at which the analysis is read: use 1 to read it at the original rate, numbers greater than 1 to read through it faster, and numbers less than 1 to read through it more slowly.

FileName

Click the Browse button to be able to select the file name from a list of names in the standard file dialog.

NbrPartial

This is the number of partials you want to output for the resynthesis. Use Default to output all of the partials in the file.

FirstPartial

This is the first analyzed partial that you want to output--usually it is partial number 1. If you want to skip

over some of the lower partials, enter a higher number here.

Trigger

When this number becomes positive, the Sound will start over again at the beginning of the analysis file.



SqrtMagnitude

Math Category

This is the square root of the sum of the left and right channels squared. If the square root of the sum of the squares is greater than 1.0, this Sound saturates at 1.0. It can be useful in doing spectral analysis where the left channel is defined to be the real part and the right channel as the imaginary part of a complex number. You could also use this as a strange kind of measure of the instantaneous "distance" between two signals, one in the left and one in the right.

Input

The output is the square root of the sum of the squares of the left and right channels of this Sound.



StereoInOutput4

Spatializing Category

This Sound routes the two stereo input Sounds to the four output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

Out12

This Sound will be routed to channels 1 and 2.

Out34

This Sound will be routed to channels 3 and 4.



StereoInOutput8

Spatializing Category

This Sound routes the four stereo input Sounds to the eight output channels of the signal processor.

This Sound only works properly as the rightmost Sound in the signal flow diagram.

Out12

This Sound will be routed to channels 1 and 2.

Out34

This Sound will be routed to channels 3 and 4.

Out56

This Sound will be routed to channels 5 and 6.

Out78

This Sound will be routed to channels 7 and 8.



StereoMix2

Mixing & Panning Category

Adds the outputs of the Sounds in the In1 and In2 fields, each with the specified Pan and Scale (attenuation) value. The overall output can also be panned and attenuated.

Left

This controls the level of the left output channel. The maximum value is 1 and the minimum is -1. The left channel of the mix is multiplied by the value of this parameter. Some example values for Left are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right output channel. The maximum value is 1 and the minimum is -1. The right channel of the mix is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

In1

The output of this Sound will be added to the output of the Sound in In2.

Pan1

The stereo position of In1. (0 is hard left and 1 is hard right).

Scale1

Attenuation on In1. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.

In2

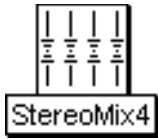
This Sound is added to the Sound in the In1 field.

Pan2

The stereo position of In2. (0 is hard left and 1 is hard right).

Scale2

Attenuation on In2. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.



StereoMix4

Mixing & Panning Category

Adds the outputs of In1, In2, In3, and In4, each with its own Pan position and Scale (attenuation). Scale and Pan control the attenuation and stereo position of the overall mix.

Left

This controls the level of the left output channel. The maximum value is 1 and the minimum is -1. The left channel of the mix is multiplied by the value of this parameter. Some example values for Left are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right output channel. The maximum value is 1 and the minimum is -1. The right channel of the mix is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

In1

The output of this Sound will be added to the output of the Sounds in In2, In3, and In4.

Pan1

The stereo position of In1. (0 is hard left and 1 is hard right).

Scale1

Attenuation on In1. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.

In2

The output of this Sound will be added to the output of the Sounds in In1, In3, and In4.

Pan2

The stereo position of In2. (0 is hard left and 1 is hard right).

Scale2

Attenuation on In2. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.

In3

The output of this Sound will be added to the output of the Sounds in In1, In2, and In4.

Pan3

The stereo position of In3. (0 is hard left and 1 is hard right).

Scale3

Attenuation on In3. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.

In4

The output of this Sound will be added to the output of the Sounds in In1, In2, and In3.

Pan4

The stereo position of In4. (0 is hard left and 1 is hard right).

Scale4

Attenuation on In4. 1 (or 0 dB) is no attenuation, and 0 is fully attenuated.



SumOfSines

Xtra Sources Category

Resynthesizes sounds from the spectral analyses stored in Analysis0 and Analysis1. The dbMorph parameter interpolates between the amplitudes of Analysis0 and Analysis1, and the pchMorph parameter interpolates between the pitches in Analysis0 and Analysis1.

OnDuration

This is the duration of each triggered event. It should be the same length or shorter than the Duration which is the total length of time that this program is available to be triggered. Think of Duration as analogous to the total lifetime of a piano string, and OnDuration as the duration of each individual note that you play on that piano string. The OnDuration must be greater than zero, and you must specify the units of time, for example:

2 s (for 2 seconds)
2 ms (for 2 milliseconds)
200 usec (for 200 microseconds)
2 m (for 2 minutes)
2 h (for 2 hours)
2 days
2 samp (for 2 samples)
1 / 2 hz (for the duration of one period of a 2 hz signal)

Frequency0

Frequency of the resynthesis based on Analysis0. Use 0 hz to default to the base frequency as stored in the samples file.

Frequency1

Frequency of resynthesis based on Analysis1. Use 0 hz to default to the base frequency as stored in the samples file.

Analysis0

Select a spectrum file from the dialog that you get when you click on the disk button next to this field.

The spectrum file contains frequency and amplitude information for resynthesizing an analyzed sound using banks of sine wave oscillators.

Analysis1

Select a spectrum file from the dialog that you get when you click on the disk button next to this field.

The spectrum file contains frequency and amplitude information for resynthesizing an analyzed sound using banks of sine wave oscillators.

DBMorph

Specifies how much of the amplitude envelopes of each of the envelopes is present in the resynthesized sound. A value of zero specifies that the amplitude envelopes come from Analysis0 only, a value of one specifies Analysis1 only, and values between specify mixtures of the two analyses.

Use a continuous controller or a control signal here to morph continuously between the two sets of amplitude envelopes.

PchMorph

Specifies how much of the frequency envelopes of each of the envelopes is present in the resynthesized sound. A value of zero specifies that the frequency envelopes come from Analysis0 only, a value of one specifies Analysis1 only, and values between specify mixtures of the two analyses.

Use a continuous controller or a control signal here to morph continuously between the two sets of frequency envelopes.

NbrPartials

This is the total number of sine wave oscillators used to resynthesize the analyzed sound. Try increasing the number of partials to hear the effect on the sound. There will be some maximum number above which there is no longer any improvement in the perceived quality of the sound. The more partials you request, the more computation this algorithm requires, so choose the minimum number of partials that still gives you acceptable sound quality.

BankSize

This specifies the number of oscillators per bank. If you get a message that you are running out of real time, try larger or smaller bank sizes.

TimeIndex

The analyzed sounds are like sequences of spectral snapshots. This value describes which snapshot to resynthesize. A FunctionGenerator with Fullramp as its Wavetable is a straight line from - 1 to 1, and this moves forward through the spectra in linear time. Try different functions (or use a continuous controller) to go backwards through the sequence of spectra or to vary the rate at which you are stepping through the spectra.

This parameter is only active if CtrlTime is checked.

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released.

This parameter is ignored if CtrlTime is checked.

Loop

Check this box if you would like to set the loop points using the LoopStart and LoopEnd parameter fields.

This parameter is ignored if CtrlTime is checked.

LoopStart

When Loop is checked, this is the start point of the loop (otherwise it is ignored). Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the start point should occur. (To compute the exact time within the sample where the start point occurs, multiply LoopStart's value by the total duration of the sample. For example, if your sample is 5 seconds long and LoopStart is set to 0.2, then the beginning of the loop is 1 second into the sample.)

LoopEnd

When Loop is checked, this is the end point of the loop (otherwise it is ignored). Enter a value in the range from 0 to 1, where 0 is the beginning of the sample and 1 is the end of the sample. In other words, this is the proportion of the total sample duration when the end point should occur. (To compute the

exact time within the sample where the end point of the loop occurs, multiply LoopEnd's value by the total duration of the sample. For example, if your sample is 5 seconds long and LoopEnd is set to 0.4, then the end of the loop occurs at 2 seconds into the sample.)

CtrlTime

The analyzed sounds are like sequences of spectral snapshots. This sound provides two ways to move through these spectral snapshots.

If CtrlTime is not checked, then the snapshots will be played back in forward order over the duration given in the OnDuration field. The playback will start whenever Gate becomes positive. If Loop is checked, the playback will loop between the StartLoop and EndLoop points within the analysis for as long as Gate is positive.

If CtrlTime is checked, then the snapshot played back is controlled directly by the value in the TimeIndex field.

Envelope

This is an attenuator on the output of the Oscillator. Enter 1 (or 0 dB) for the full amplitude. For a time-varying amplitude, paste in a Sound (such as AR, ADSR, or FunctionGenerator) or an Event Value (such as !Volume) in this field.



SyntheticSpectrumFromArray

Spectral Sources Category

Creates a synthetic spectrum from two arrays: an array of amplitude values for each track in the frame, and an array of frequency values for each track in the frame (and, if SendBandwidths is checked, a corresponding array of bandwidths for each of the tracks as well). A SyntheticSpectrumFromArray should be fed to an OscillatorBank, FormantBankOscillator, or VocoderChannelBank in order to synthesize the partials, formants, or bank of vocoder filters. The SyntheticSpectrumFromArray produces a set of envelopes for controlling the parameters of an OscillatorBank, FormantBankOscillator, or VocoderChannelBank.

NbrPartials

This is the number of partials (or filters) to synthesize. In most cases, it should be the same as the size of the Frequencies array; however, you can specify a slower update rate for the envelopes by using a larger number here. The time between updates of the control envelopes is equal to the number you specify here but in units of samples. If you enter 128 here, for example, the envelopes will be updated every 128 samples (that is about every 3 milliseconds if your sampling rate is 44.1 kHz).

LogScale

Check this box to output the Frequencies (and, optional Bandwidths) in log rather than linear frequency. In most cases, this box should be unchecked; the only time it should be checked is if you want to manipulate the frequency envelopes in pitch space rather than in hertz.

SendBandwidths

Check this box to send bandwidth information. Bandwidths are required for controlling the filters of a FormantBankOscillator or a VocoderChannelBank, but they are not required for controlling the oscillators in an OscillatorBank.

Envelope

This is an overall amplitude envelope.

Amplitudes

Enter an array of amplitude values separated by spaces. Enclose any arithmetic expressions or units within curly braces, for example:

```
!Amp1 {!Amp2 * 0.5} {-6 db} !KeyVelocity {!KeyDown ramp: 5 s} {0.1 s random}
```

The number of amplitude values should be the same as the number of frequency values (and optional bandwidth values). If the frequency, amplitude (and bandwidth if used) arrays are different sizes, the smallest array will be used, and any extra values in the other two arrays are thrown away.

Frequencies

Enter an array of frequency values separated by spaces. If you leave off the units, the values will be interpreted as frequencies in hertz. Enclose any arithmetic expressions or frequencies with units within curly braces, for example:

```
609 {!Freq1 * 1000} {2048 hz} {60 nn} {5 c}
```

The number of frequency values should be the same as the number of amplitude values (and optional bandwidth values). If the frequency, amplitude (and bandwidth if used) arrays are different sizes, the

smallest array will be used, and any extra values in the other two arrays are thrown away.

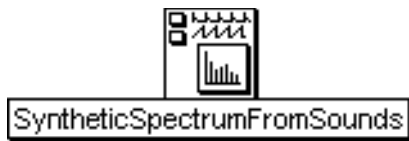
Bandwidths

This array is optional and need only be set if the SendBandwidths box is checked. Bandwidths are required by the FormantBankOscillator and VocoderChannelBank, but they are not required by the OscillatorBank.

Enter an array of bandwidth values separated by spaces. If you leave off the units, the values will be interpreted as frequencies in hertz. Enclose any arithmetic expressions or frequencies with units within curly braces, for example:

609 {!Freq1 * 1000} {2048 hz} {60 nn} {5 c}

The number of bandwidth values should be the same as the number of amplitude and frequency values. If the frequency, amplitude and bandwidth arrays are different sizes, the smallest array will be used and any extra values in the other two arrays are thrown away.



SyntheticSpectrumFromSounds

Spectral Sources Category

Generates a synthetic spectrum whose amplitudes, frequencies (and optionally, bandwidths) are controlled by two input Sounds. One input supplies the amplitudes and the other supplies the frequencies (optionally alternating with bandwidths). You can think of each cycle of the input Sounds as defining one frame of the spectrum. If the input Sounds change from cycle to cycle, then the spectrum will also change from frame to frame.

A SyntheticSpectrumFromSounds (like other Sounds in the Spectral Sources category) outputs spectral envelopes in the following format:

Left Channel: Amp1 Amp2 ... AmpN

Right Channel: Freq1 Freq2 ... FreqN

For each frame, Amp1 is the amplitude of the first partial (and Freq1 is the frequency or pitch of the first partial), Amp2 is the amplitude of the second partial (corresponding with Freq2), and AmpN is the amplitude of the highest numbered partial (specified in NbrPartials). Then the whole sequence repeats for the next frame of the spectrum. Because of this repetition rate, the output of the SyntheticSpectrumFromSound has a kind of periodicity to it, where the period is the equal to the same number of samples as there are partials in each frame.

Amplitudes

If the period of this Sound in samples is equal to NbrPartials, then one cycle of this Sound defines one frame's worth of amplitudes for the synthesized spectrum. (For example, to synthesize 80 partials, set the Frequency of this Sound to 80 samp inverse if you want the cycles to line up with frames). If the repetition rate of this Sound is lined up with the number of partials in each frame of the spectrum, then the waveform of each cycle of this Sound will correspond to a kind of spectral envelope for each frame of the spectrum. For example, if you select ExponRev as the waveform of an oscillator whose period is 80 samples and set NbrPartials to 80, then each frame of the spectrum will have high amplitudes on its lower-numbered partials and lower amplitudes on the upper partials. Even more interesting is to make this Sound's frequency adjustable within a narrow range so you can create spectral envelopes that "drift" because their repetition rates are slightly out of phase with the number of partials being generated on each frame.

FrequenciesAndBandwidths

If the period of this Sound in samples is equal to NbrPartials, then one cycle of this Sound defines one frame's worth of frequencies (or pitches if you have LogScale checked) for the synthesized spectrum. (For example, to synthesize 80 partials, set the Frequency of this Sound to 80 samp inverse if you want the cycles to line up with frames). If the repetition rate of this Sound is lined up with the number of partials in each frame of the spectrum, then the waveform of each cycle of this Sound will provide the frequencies for each partial in one frame of the spectrum. For example, if you select Ramp as the waveform of an oscillator whose period is 80 samples and set NbrPartials to 80, then, in each frame of the spectrum, the lower-numbered partials will have low frequencies, and the higher-numbered partials will have high frequencies. Even more interesting is to make this Sound's frequency adjustable within a narrow range so you can create spectra that "drift" because their repetition rates are slightly out of phase with the number of partials being generated on each frame.

If the SyntheticSpectrumFromSounds is controlling something that requires bandwidth (like FormantBankOscillator or VocoderChannelBank), and you have checked the SendBandwidths box, then

every other value of this Sound will be interpreted as a bandwidth, rather than a frequency.

NbrPartials

This is the number of partials in the synthetic spectrum. It should be greater than or equal to the number of oscillators or filters in the Sound being controlled by the SyntheticSpectrumFromSounds.

LogScale

Check this box to output log-frequency (pitch) envelopes rather than frequency envelopes.

IncludesBandwidths

Check here if the synthetic spectrum is feeding into a Sound that can use bandwidth information (e.g. FormantBankOscillators and VocoderChannelBanks).



TextFileInterpreter

Scripts Category

Reads and interprets a line at a time from a text file. This Sound can be used to interpret scores prepared for Music N languages (such as csound) and map the parameters to the variable parameters of Kyma Sounds, essentially treating Kyma Sounds as "instruments". It can also be used more generally to read data from text files and map them to the parameters of Kyma Sounds.

This is like the Script in that it constructs a new Sound algorithmically and then plays it; it does NOT set up fixed "patches" and then update the parameters by reading them out of the text file as the Sound is playing.

FileName

This is the name of a text file created in Kyma or another program. In other programs, save a file as text-only (ASCII) if you want to use it as a Kyma text file.

Inputs

These Sounds are treated as templates. Each name should begin with a letter and contain only alpha-numeric characters; this field will reject any Sounds with "illegal" names. You can reference these Sounds by name in the Script field.

Script

Enter the Smalltalk code to read and interpret data from the specified text file. See the manual for a detailed description and example scripts.

Left

This controls the level of the left output channel. The maximum value is 1 and the minimum is -1. The left channel of the output is multiplied by the value of this parameter. Some example values for Left are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).

Right

This controls the level of the right input channel. The maximum value is 1 and the minimum is -1. The right channel of the input is multiplied by the value of Right. Some example values for Right are:

1	(no attenuation)
0	(maximum attenuation)
!Fader1	(continuous controller sets level)
!KeyVelocity	(MIDI key velocity controls the amplitude)

You can also paste another signal into this field, and the amplitude will vary with the output amplitude of the pasted signal (something like an LFO controlling the attenuation). (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



Threshold

Tracking Live Input Category

The output of a Threshold is 1 when its Input amplitude exceeds the specified threshold; otherwise it is 0. The smaller the value of hysteresis, the more sensitive the Threshold is to momentary changes in the Input amplitude.

When trying to detect when an amplitude is exceeded, it is usually a good idea to put your input through an AmplitudeFollower or PeakDetector first so you are detecting when the amplitude *envelope* exceeds the threshold rather than when individual sample points might cross the threshold.

Input

When this Sound's amplitude exceeds the threshold, the output of the Threshold will be a 1 (i.e. the maximum deviation).

Threshold

When the amplitude of the Input is less than the threshold (plus or minus half the hysteresis), the output of this sound is zero. Otherwise the output is 1.

Hysteresis

The larger the hysteresis, the less sensitive the Sound will be to small changes in the Input amplitude. Hysteresis comes from the Greek husteros, come later or behind. This is the tendency of this Sound to stay in its previous state (either 1 or 0).



TimeControl

Time & Duration Category

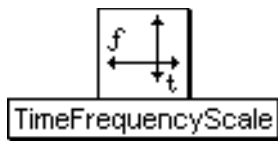
Slows down or speeds up the rate that time is progressing in its Input by controlling how the time counter is incremented on the signal processor. This affects only the start time of events within the Input (e.g. if Input is a Script or Concatenation or contains TimeOffsets), not the sample rate.

Input

The duration of this Sound can be shortened or lengthened depending on the value of Rate.

Rate

This is the rate that time progresses. For example, use 1 to increment time at the normal rate, 0.5 for half speed, 2 for twice as fast, etc.



TimeFrequencyScale

Frequency & Time Scaling Category

Simultaneously time stretches and/or frequency scales a disk recording or a sample stored in wavetable memory.

FrequencyScale

The frequency of the input will be multiplied by this value.

For example, to shift up by an octave, the FrequencyScale should be 2, and to shift down an octave, the scale should be 0.5. To shift up by 3 halfsteps, you would use:

2 raisedTo: (3/12)

To shift down by 7 half steps, you would use:

2 raisedTo: (-7/12)

Rate

This is the rate of playback. The value should be less than or equal to 1, because the Sound can only do time stretching, not time compression. For example, use 1 to play back at the original rate, 0.5 for half speed, 0.25 for one quarter of the speed, etc.

Gate

Enter a 1 in this field to play the Sound exactly once for the duration you have specified in the Duration field.

If you use an EventValue (for example, !KeyDown) in this field, the Sound can be retriggered as often as you like within the duration specified in the Duration field.

When Gate becomes positive, the Sound is heard; when Gate becomes zero, the Sound is released.

MinInputFreq

This is the minimum frequency you expect to hear at the input. Follow the usual conventions for specifying frequencies.

MaxInputFreq

This is the maximum frequency you expect to hear at the input. Follow the usual conventions for specifying frequencies.

MaxFreqScale

This is the largest scale that will be applied to the frequency (the maximum allowable is 4).

Detectors

This determines the sensitivity of the frequency tracking. Try starting with a value of 10, and then experiment with more or fewer if you want to try fine tuning the frequency tracking. (More is not necessarily better; there is some optimal number of detectors for each circumstance.)

FromDisk

When the box is checked, read the recording directly from the disk. Otherwise, look for it in wavetable

memory.

Sample

If FromDisk is checked, this is the name of the disk file. Otherwise, this should be the name of a sample in the Wavetables list or the name of a segment of wavetable memory being recorded into by a MemoryWriter prior to or in parallel with this Sound.

FromMemoryWriter

Check FromMemoryWriter when the wavetable does not come from a disk file but is recorded by a MemoryWriter in real time.



TimeOffset

Time & Duration Category

Offsets the start time of its Input by the specified SilentTime. If Retrograde or Reverse is set, a Constant zero is concatenated to the end of Input; this can be useful for adding some silence to the end of an input to a reverberator or echo in order to give the reverberation time to die away.

Input

This Sound's start time is delayed by the amount of time specified in SilentTime. If retrograde or reverse (but not both) is true, the silence will follow this Sound.

SilentTime

Amount of time to delay the start time of the Input. It can be any amount of time from 0 to the maximum possible duration. If retrograde or reverse (but not both) is true, this silence follows the Input.



TimeStopper

Time & Duration Category

Allows Input to be loaded into the signal processor and start playing but then stops any further progress of time on the signal processor. Time resumes only when the value in the Resume field becomes nonzero. For example, even if Input had a duration of 1 samp, it would last until Resume became nonzero. If the input has multiple events in it that occur sequentially, only the first one will take place immediately; the others will occur only after Resume becomes nonzero.

Input

This Sound is loaded and started but it will not terminate unless Resume becomes nonzero.

Resume

Time is stopped until this value becomes something other than 0. You could use an EventValue (such as !KeyDown) in this field to control when time should progress. By putting a Threshold Sound here, you can make the progress of time depend on the amplitude of another Sound (such as the ADInput). Use an Equality prototype to make time depend on an Event Value or Sound reaching an exact value.

ResumeOnZero

Click here if you would like time to resume when Resume equals 0 (rather than resuming whenever the value in Resume becomes nonzero).



TriggeredSampleAndHold

Sampling Category

When triggered, reads a value from Input and holds onto it until triggered again. This is like SampleAndHold except that the sampling only occurs on triggers, not periodically.

Input

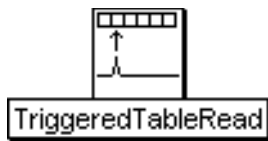
A sample of this Sound is read each time the Trigger becomes positive.

Trigger

When the Trigger becomes positive, one event is triggered. You can trigger several events over the course of the total Duration of this program as long as the value of Trigger returns to zero before the next trigger. Some example values for Trigger are:

- 1 (plays once with no retriggering)
- 0 (the sound is silent, never triggered)
- !KeyDown (trigger on MIDI key down)
- !F1 (trigger when MIDI switch > 0)

You can also paste another signal into this field, and events will be triggered every time that signal changes from zero to a nonzero value. (See the manual for a complete description of hot parameters, EventValues, EventSources, and Map files).



TriggeredTableRead

Sampling Category

As long as the Trigger is greater than zero, the TriggeredTableRead will read samples from the Wavetable; if the Trigger is less than or equal to zero, the last sample read will be output. Gate resets the pointer to the beginning of the Wavetable.

Trigger

As long as the Trigger is greater than zero, the TriggeredTableRead will read samples from the Wavetable; if the Trigger is less than or equal to zero, the last sample read will be output. PulseTrain is a good Sound to use as a source of periodic triggers, and by putting an Event Value in the PulseTrain's Period field, you can control the rate at which the triggers occur.

Wavetable

Select a wavetable or a sample. A single sample point is read from this table each time Trigger becomes positive.

Gate

Each time this value becomes positive, the Sound will start over again from the beginning of the Wavetable. Enter a 1 in this field to play the Sound exactly once. If you use an EventValue (for example, !KeyDown) in this field, you can restart the sound multiple times.

IgnoreLoops

Click here if the Wavetable (or sample) has loop points specified in the header and you want to ignore the loop points.

FromMemoryWriter

Check FromMemoryWriter when the wavetable does not come from a disk file but is recorded by a MemoryWriter in real time.



TunableVocoder

Filters Category

Tune the base frequency of the vocoder and control the spacing of the center frequencies of the filters.

Input

The sound source that you hear going through the filter bank.

SideChain

The spectrum of this Sound controls the amplitudes of each filter in the filter bank.

TimeConstant

The smaller the time constant, the more quickly the amplitude envelopes respond to changes in the side chain spectrum. Longer time constants result in a less precise sound (and give a reverberated effect).

NbrBands

This is the number of filters desired.

BankSize

This is the number of filters that should be scheduled on each processor. Ordinarily you should leave this set to default. If you run out of realtime processing, you can try reducing the bankSize, as for example

0.75 * default

LogSpacing

Check this box to specify the spacing between center frequencies as an interval in pitch space. Uncheck the box if you prefer to specify the spacing as a frequency in hertz.

AnalysisFreq

This is the lowest center frequency in the filter bank operating on the SideChain.

SynthesisFreq

This is the lowest center frequency in the filter bank operating on the Input.

AnalysisLevel

This is an attenuator on the amplitude of the SideChain before it goes through the filters.

SynthesisLevel

This is an attenuator on the amplitude of the Input before it goes through the filters.

AnalysisSpacing

This is the spacing between the center frequencies of the filters on the SideChain. Use nn as the units if LogFrequency is checked. Use hz as the units if LogFrequency is unchecked.

SynthesisSpacing

This is the spacing between the center frequencies of the filters on the Input. Use nn as the units if LogFrequency is checked. Use hz as the units if LogFrequency is unchecked.

AnalysisBW

This is a control on the bandwidth of the filters on the SideChain.

SynthesisBW

This is a control on the bandwidth of the filters on the Input.

Tone

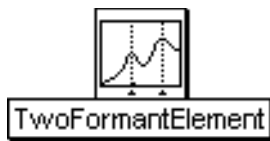
This is a tone control. Higher values emphasize the filters with higher center frequencies. Lower values emphasize the filters with lower center frequencies. (Rolloff determines the narrowness of this filter).

Rolloff

This is a control on the bandwidth of the Tone filter. Set this to 0 if all filters should have equal weight. The bigger the value of Rolloff, the sharper the cutoff on the effect of the Tone filter.

Gain

You can boost or cut the final output amplitude here.



TwoFormantElement

Filters Category

A TwoFormantElement is realized as a DualParallelTwoPoleFilter; however, rather than specifying the filter in terms of pole locations, you specify the desired center frequency and bandwidth of the two formants.

Input

This is the Sound to be filtered.

Formant1

This is the center frequency of the first formant.

Bandwidth1

This is the bandwidth of the lower formant region. The narrower the bandwidth, the more "pitched" the formant will sound--also the more likely the the filter is to overflow.

Scale1

This controls the amplitude of the first formant. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.

Formant2

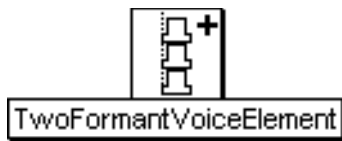
This is the center frequency of the second formant.

Bandwidth2

This is the bandwidth of the upper formant region. The narrower the bandwidth, the more "pitched" the formant will sound--also the more likely the the filter is to overflow.

Scale2

This controls the amplitude of the second formant. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.



TwoFormantVoiceElement

Xtra Sources Category

An excitation signal similar to a glottal pulse (with a randomly chosen rate of vibrato) is used as the input to a pair of parallel second-order filter sections that simulate two of the formants of the vocal cavity.

Frequency

The frequency can be specified in units of pitch or frequency. The following are all ways to specify the A above middle C:

440 hz	(in hertz or cycles per second)
4 a	(as the 4th octave A)
69 nn	(as a MIDI notenumber)
4 c + 9 nn	(as 9 half steps above middle C)
1.0 / 0.00227273 s	(inverse of a period at 44.1 kHz sample rate)

The following are examples of how to control the frequency using MIDI, the virtual control surface, or a third-party program:

!Pitch	(key number plus pitch bend)
!KeyNumber nn	(MIDI notenumber)
4 c + (!Frequency * 9 nn)	(continuous controller from 4 c to 4 a)

Formant1

For an [IY] sound, try a center frequency of 238 hz. This will be the center frequency of the first formant. This is not the fundamental frequency of the TwoFormantVoiceElement but the center of an emphasized region of the spectrum.

Bandwidth1

For an [IY] sound, try a a bandwidth of 70 hz. This is the bandwidth of the lower formant region. The narrower the bandwidth, the more "pitched" the formant will sound--also the more likely the the filter is to overflow.

Scale1

For an [IY] sound, scale this formant to 0.3. This controls the amplitude of the first formant. For the full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.

Formant2

For an [IY] sound, try a center frequency of 1741 hz. This will be the center frequency of the second formant. This is not the fundamental frequency of the TwoFormantVoiceElement but the center of an emphasized region of the spectrum.

Bandwidth2

For an [IY] sound, try a a bandwidth of 100 hz. This is the bandwidth of the upper formant region. The narrower the bandwidth, the more "pitched" the formant will sound--also the more likely the the filter is to overflow.

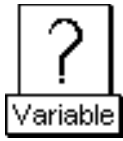
Scale2

For an [IY] sound, scale this formant to 1.0. This controls the amplitude of the second formant. For the

full amplitude use +1.0 or -1.0; any factor whose absolute value is less than 1 will attenuate the output.

Seed

Supply an integer less than 2^{30} as a seed for the random number generator that controls the vibrato rate.



Variable

Variables & Annotation Category

A Variable is a placeholder that represents a single Sound. You can assign a value to the Variable in a Script or related Sound by typing the name of the Variable followed by a colon, a space, and then the name of the Sound that you want to assign to the variable.



VCA

Envelopes & Control Signals Category

Multiplies its Inputs together. To apply an amplitude envelope to a Sound, use the Sound and the envelope generator as inputs to this Sound.

Inputs

These Sounds are multiplied together. Typically the inputs are a Sound and the amplitude envelope that you want to apply to the Sound.



Vocoder

Filters Category

The Vocoder applies the spectral character of the SideChain Sound onto the Input Sound. What you hear is the Input Sound filtered by the SideChain Sound. For example, you can use this module to apply the spectral characteristics of human speech (the SideChain) onto any other sample or synthetic sound (the Input). On some analog vocoders, the SideChain input is called the "modulation" input.

The Vocoder is implemented as two filter banks--an analysis bank and a resynthesis bank. The analysis bank is used to measure the amount of energy in each frequency band of the SideChain Sound. The resynthesis bank is used to filter the Input Sound. There is an amplitude follower on the output of each of the filters in the analysis bank. The resulting amplitude envelopes are then applied to the corresponding filters in the resynthesis bank. In this way, the SideChain controls the amplitude envelopes on the resynthesis filters.

Input

This is the source material to be filtered by the SideChain-controlled filters. This Sound is heard directly, through the filters (whereas the SideChain is never heard directly). For example, if you want to make an animal talk, put a sample of the animal sound here and put a sample of speech (or use a microphone) as the SideChain.

The best Inputs tend to be fairly broad band signals that have energy in each of the frequency bands covered by the resynthesis filter bank. For example, Noise or an Oscillator on a waveform with lots of harmonics (such as Buzz128) will work well because they generate energy over the full frequency range.

SideChain

Sometimes referred to as the "modulation", this Sound is never heard directly; it controls the amplitudes of the filters in the bank.

TimeConstant

This determines how quickly the amplitude envelopes on the filters will respond to changes in the SideChain. For precise, intelligible results, use values less than 0.1 s. For a more diffuse, reverberated result, use a longer TimeConstant.

NbrBands

This is the number of band pass filters in the filter bank. In other words, this is the number of equally-spaced frequency bands between LowCF and HighCF, inclusive.

BankSize

This is the number of filters per expansion card. In general, you should be able to get about 10 to 11 filters per card. Experiment with fewer or more filters per card to optimize the efficiency of your particular Sound.

InputLevel

Controls the level on the input Sound before it goes through the filters.

SideLevel

Controls the level of the SideChain Sound before it is fed into the analysis filters.

LowCF

This is the center frequency of the lowest bandpass filter in the bank.

HighCF

This is the center frequency of the highest bandpass filter in the bank.

InFreq

This is a scale factor on all of the center frequencies in the "resynthesis" bank.

SideFreq

A scale factor on all of the center frequencies in the "analysis" filter bank.

Bw

A control on the bandwidth of all the bandpass filters in both the analysis and the resynthesis filter banks.

Pitch

Check here if you would like the bandpass filters to be spaced equally in pitch space from the LowCF to the HighCF. If you uncheck this box, the filters will be spaced equally in frequency space.

LoCutoff

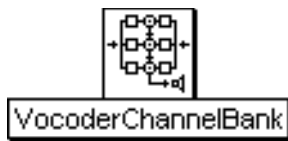
Everything below this frequency should drop off in amplitude according to the slope specified in Rolloff. This is a weak tone-control-style filter applied to the Input Sound before it is fed to the resynthesis filter bank. Use it to attenuate the low end if the output is too boomy (or set it to 0 hz if you want to give your subwoofers something to do).

HiCutoff

Everything above this frequency should drop off in amplitude according to the slope specified in Rolloff. This is a weak tone-control-style filter applied to the Input Sound before it is fed to the resynthesis filter bank. Use it to attenuate the high end if the output is too piercing or trebly. Set it to an even higher frequency if the output sounds muffled or low pass and you would like to boost the high end.

Rolloff

This controls the steepness of the edges of a weak tone control filter on the Input. Use 1 if the edges should rolloff precipitously at LoCutoff and HiCutoff. Use smaller numbers if you would like the attenuation to start sooner and take longer.



VocoderChannelBank

Filters Category

For most situations, you should use a Vocoder rather than the VocoderChannelBank, because the Vocoder is a higher-level Sound with higher-level parameters and controls. Use a VocoderChannelBank only in those situations requiring independent control over the center frequency, amplitude, and bandwidth of each filter in both the analysis and the resynthesis filter banks.

The VocoderChannelBank works by feeding the sidechain input through a bank of bandpass filters (the analysis filters), extracting an amplitude envelope from the output of each of those filters, applying the extracted amplitude envelopes to a second bank of filters (the synthesis filters), and feeding the input through that bank of filters.

To see an example of how the VocoderChannelBank can be used, drag a Vocoder into a Sound file window and expand it; it expands into cascaded VocoderChannelBanks.

CascadeInput

The cascaded input is added to whatever output is produced by this VocoderChannelBank. Use it to cascade several VocoderChannelBanks when you need more filter banks than can fit on a single expansion card (usually around 11).

Input

This is the Sound that is actually heard through the filter bank. The output of the VocoderChannelBank is the sound of the Input but filtered through formants of the SideChain.

The kinds of Inputs that work best are those that are broadband and continuous enough to excite all of the filters in the bank at all times.

SideChain

This is the Sound that controls the amplitude envelopes on each of the filters in the filter bank. The formants of the SideChain will be imposed on the basic sound characteristics of the Input.

The kinds of SideChains that work best are those with strong formants that change noticeably over time (e.g. human speech, tablas, mouth harps).

InputParameters

This should be a Sound from the spectral sources category (most typically the SyntheticSpectrumFromArray). Use the spectral source Sound to specify the center frequencies, amplitudes, and bandwidths for all the filters in the synthesis bank.

To use the same settings on both the analysis filters and the synthesis filters (as it is in the classic channel vocoder), hold down the option key and drag this Sound into the SideChainParameters field.

SideChainParameters

This should be one of the Sounds from the spectral sources category (most typically the SyntheticSpectrumFromArray). Use the spectral source Sound to specify the center frequencies, amplitudes, and bandwidths for all the filters in the analysis bank.

To use the same settings on both the analysis filters and the synthesis filters (as it is in the classic channel vocoder), hold down the option key and drag this Sound into the InputParameters field.

TimeConstant

Controls the reaction time of the envelope follower on each of filters in the side chain bank. Smaller numbers should be used for better intelligibility, larger numbers for a more diffuse, reverberated result.

First

This is the number of the first filter in this bank. If this is the first VocoderChannelBank in a cascade, this number will be 1, but the next VocoderChannelBank in the cascade should start at (1 + Count). The total number of filters in the entire cascade should be equal to the NbrPartials specified in the InputParameters and the SideChainParameters.

Count

This is the total number of filters in this bank. You can get about 11 filters per card on a Capybara-66. To get more filters, feed this Sound into the cascade input of another VocoderChannelBank.



WaitUntil

Time & Duration Category

Don't start the Input Sound until the Resume condition is true. This is like TimeStopper except that a WaitUntil won't even let its Input start until the Resume value becomes nonzero (and a TimeStopper lets its Input get started but won't let it end until Resume becomes nonzero).

Input

Input will not start until Resume becomes nonzero.

Resume

Input will not start playing until this value becomes positive.

If this field contains a Sound, the Input will not resume until the Sound in this field ends.

ResumeOnZero

Click here if you want the Input to start whenever Resume becomes zero (rather than whenever it is no longer zero).



WarpedTimeIndex

Time & Duration Category

Can be used as the input to any Sound that requires a time index (e.g. GAOscillators, REResonator, SampleCloud, SpectrumInRAM). Generates a time index with a variable slope so that it can move more quickly through some parts of sound and more slowly through others. You provide a set of current time points that should be adjusted forward or backward in time to match a set of ideal time points, and it generates a time index function that moves from -1 up to 1 but with a varying slope.

IdealTimePoints

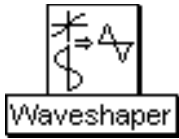
Enter a chronological sequence of time points with units, enclosed within curly braces and separated by spaces. These are the new time points. Time will be sped up or slowed down to make the CurrentTimePoints line up with these ideal time points.

CurrentTimePoints

Enter a chronological sequence of time points with units, enclosed within curly braces and separated by spaces. These are the time points that should be moved forward or backward in time in order until they line up with the IdealTimePoints.

Trigger

Each time this value changes from a zero to a number greater than zero, the time function starts over again from the beginning.



Waveshaper

Distortion & Waveshaping Category

The Input is used as an index into the table specified in ShapingFunction (if ShapeFrom is set to Wavetable) or as the input to a polynomial whose coefficients are those listed in the Coefficients parameter field (if ShapeFrom is set to Polynomial).

Unless the ShapingFunction or polynomial is a straight line, the Input will be nonlinearly distorted. The distortion adds harmonics to the synthesized or sampled Input. Since polynomials tend to be close to linear around zero and less linear the further they are from zero, low amplitude Inputs will be less distorted than high amplitude Inputs. This tends to match the behavior of physical instruments (which sound "brighter" when played louder) and also of electronic components like amplifiers which produce harmonic distortions of their inputs at high amplitudes.

A Waveshaper can also be used to map non-signal Inputs to new values according to the ShapingFunction or polynomial. For example, if the Input were a Constant whose Value were !Pitch, the full range of MIDI notenumbers could be remapped by a Waveshaper to frequencies of an alternate tuning system as stored in a table (the ShapingFunction).

Input

This Sound is used as an index into the ShapingFunction (or as the input into the polynomial described the list of Coefficients).

Interpolation

Choose whether to use only integer values to index into the ShapingFunction or whether to use the fractional part of the Input value to interpolate between the values actually stored in the table to values that would fall "inbetween" the table entries if the actual values were connected by a straight line.

ShapeFrom

Choose whether to use a function stored in a table (Wavetable) or a polynomial computed on the fly using the Coefficients (Polynomial).

ShapingFunction

Select the wavetable that will be used to map the Input to the output.

Coefficients

Enter a list of coefficients A0 A1 A2 ... An (separated by spaces) for a polynomial of the form:

$$A0 + A1x + A2x^2 + A3x^3 + \dots + Anx^n$$

where Input is x.

FromMemoryWriter

Check FromMemoryWriter when the shaping function does not come from a disk file but is recorded by a MemoryWriter in real time.

Sound Classes by Category

Sources & Generators	FormantBankOscillator	260
Sources & Generators	DiskPlayer	247
Sources & Generators	DynamicRangeController	250
Sources & Generators	TwoFormantElement	386
Sources & Generators	Oscillator	307
Sources & Generators	GenericSource	268
Sources & Generators	GrainCloud	269
Sources & Generators	AudioInput	230
Sources & Generators	SumOfSines	368
Sources & Generators	Attenuator	229
Sources & Generators	OscillatorBank	309
Sources & Generators	PulseGenerator	321
Sources & Generators	Sample	334
Sources & Generators	Mixer	298
Sources & Generators	SampleCloud	337
Sources & Generators	TimeFrequencyScale	378
Sources & Generators	MultiplyingWaveshaper	301
Sources & Generators KBD Ctrl	OscillatorBank	309
Sources & Generators KBD Ctrl	ScaleVocoder	341
Sources & Generators KBD Ctrl	Mixer	298
Sources & Generators KBD Ctrl	FormantBankOscillator	260
Sources & Generators KBD Ctrl	GAOscillators	266
Sources & Generators KBD Ctrl	GrainCloud	269
Sources & Generators KBD Ctrl	HarmonicResonator	273
Sources & Generators KBD Ctrl	KeyMappedMultisample	277
Sources & Generators KBD Ctrl	SumOfSines	368
Sources & Generators KBD Ctrl	Filter	257
Sources & Generators KBD Ctrl	Product	320
Sources & Generators KBD Ctrl	IteratedWaveshaper	276
Additive synthesis	OscillatorBank	309
Additive synthesis	DynamicRangeController	250
Additive synthesis	SumOfSines	368
Additive synthesis	SOSOscillators	349
Additive synthesis	Product	320
Compression/Expansion	DynamicRangeController	250
Compression/Expansion	Mixer	298

Compression/Expansion	Gain	265
Cross synthesis	DynamicRangeController	250
Cross synthesis	REResonator	327
Cross synthesis	Vocoder	391
Delays-Mono	DelayWithFeedback	243
Delays-Mono	Mixer	298
Disk	DiskCache	246
Disk	DiskPlayer	247
Disk	DiskRecorder	248
Disk	GenericSource	268
Disk	SamplesFromDiskSingleStep	339
Distortion & Waveshaping	Mixer	298
Distortion & Waveshaping	MultiplyingWaveshaper	301
Drum machines	StereoMix4	366
Envelopes & Control Signals	ADSR	220
Envelopes & Control Signals	Product	320
Envelopes & Control Signals	Gain	265
Envelopes & Control Signals	AR	227
Envelopes & Control Signals	Constant	239
Envelopes & Control Signals	FunctionGenerator	264
Envelopes & Control Signals	GraphicalEnvelope	271
Envelopes & Control Signals	Oscillator	307
Envelopes & Control Signals	MultisegmentEnvelope	304
Envelopes & Control Signals	MultislopeFunctionGenerator	305
Envelopes & Control Signals	PeakDetector	316
Envelopes & Control Signals	PulseTrain	323
Envelopes & Control Signals	SampleAndHold	336
Envelopes & Control Signals	Difference	245
Envelopes & Control Signals	TriggeredSampleAndHold	382
Envelopes & Control Signals	TriggeredTableRead	383
EQ	StereoMix2	365
EQ	Vocoder	391
EQ	Gain	265
EQ	GraphicEQ	272
EQ	PresenceFilter	318
EQ	HighShelvingFilter	274
EQ	LowShelvingFilter	282
Filters	AnalysisFilter	225
Filters	DualParallelTwoPoleFilter	249

Filters	GraphicEQ	272
Filters	HighShelvingFilter	274
Filters	LowShelvingFilter	282
Filters	PresenceFilter	318
Filters	TunableVocoder	384
Filters	TwoFormantElement	386
Filters	Vocoder	391
Filters	VocoderChannelBank	393
Filters-Mono	Filter	257
Filters-Mono	AnalysisFilter	225
Filters-Mono	AveragingLowPassFilter	231
Filters-Mono	ScaleVocoder	341
Filters-Mono	HarmonicResonator	273
Filters-Mono	Mixer	298
Filters-Stereo	ChannelJoin	236
Flanging & Chorus-Mono	Mixer	298
Frequency & Time Scaling	OscillatorBank	309
Frequency & Time Scaling	Annotation	226
Frequency & Time Scaling	FrequencyScale	261
Frequency & Time Scaling	Monotonizer	299
Frequency & Time Scaling	QuadOscillator	324
Frequency & Time Scaling	SimplePitchShifter	347
Frequency & Time Scaling	Gain	265
Frequency & Time Scaling	SpectrumFrequencyScale	354
Frequency & Time Scaling	Mixer	298
Gain & Level	Attenuator	229
Gain & Level	Gain	265
Global controllers	Attenuator	229
Global controllers	SoundToGlobalController	351
Granulating & Chopping-Mono	Chopper	238
Granulating & Chopping-Mono	Mixer	298
Granulating & Chopping-Mono	Product	320
Inputs	AudioInput	230
Inputs	GenericSource	268
Looping	Sample	334
Looping	AnalogSequencer	222
Math	AbsoluteValue	219
Math	ArcTan	228

Math	Difference	245
Math	Equality	253
Math	Interpolate	275
Math	Constant	239
Math	PhaseShiftBy90	317
Math	Product	320
Math	DelayWithFeedback	243
Math	RunningMax	332
Math	RunningMin	333
Math	SampleAndHold	336
Math	ScaleAndOffset	340
Math	SetRange	346
Math	SqrtMagnitude	362
Math	TriggeredSampleAndHold	382
Math	VCA	390
MIDI In	MIDIMapper	289
MIDI In	MIDIVoice	295
MIDI In	Preset	319
MIDI Out	MIDIFileEcho	288
MIDI Out	MIDIOutputController	292
MIDI Out	MIDIOutputEvent	293
MIDI Out	MIDIOutputEventInBytes	294
Mixing & Panning	CenteringMixer	234
Mixing & Panning	ChannelJoin	236
Mixing & Panning	Channeller	237
Mixing & Panning	Crossfade	242
Mixing & Panning	EndTogetherMixer	252
Mixing & Panning	Difference	245
Mixing & Panning	Mixer	298
Mixing & Panning	Output8	312
Mixing & Panning	Output4	311
Mixing & Panning	OverlappingMixer	313
Mixing & Panning	Pan	314
Mixing & Panning	StereoInOutput4	363
Mixing & Panning	StereoInOutput8	364
Mixing & Panning	StereoMix2	365
Mixing & Panning	StereoMix4	366
Modulation	Oscillator	307
Modulation	Product	320

Modulation	Gain	265
Outputs	Matrix4	283
Outputs	Matrix8	284
Processing analyzed spectra	OscillatorBank	309
Processing analyzed spectra	SumOfSines	368
Reverb-Mono	ReverbSection	328
Reverb-Spatializing	Mixer	298
Reverb-Spatializing	StereoInOutput4	363
Reverb-Stereo	Mixer	298
Sampling	Sample	334
Sampling	DiskCache	246
Sampling	DiskPlayer	247
Sampling	DiskRecorder	248
Sampling	ForcedProcessorAssignment	259
Sampling	GenericSource	268
Sampling	MemoryWriter	286
Sampling	Filter	257
Sampling	SampleAndHold	336
Sampling	Mixer	298
Sampling	TriggeredSampleAndHold	382
Sampling	TriggeredTableRead	383
Scripts	CellularAutomaton	232
Scripts	ContextFreeGrammar	240
Scripts	StereoMix4	366
Scripts	LimeInterpreter	279
Scripts	ParameterTransformer	315
Scripts	RandomSelection	325
Scripts	RhythmicCellularAutomaton	330
Scripts	Script	343
Scripts	MIDIVoice	295
Scripts	TextFileInterpreter	375
Sequencers	AnalogSequencer	222
Spatializing	ChannelCrosser	235
Spatializing	Channeller	237
Spatializing	StereoMix4	366
Spatializing	Difference	245
Spatializing	Matrix4	283
Spatializing	Matrix8	284

Spatializing	Output8	312
Spatializing	Output4	311
Spatializing	StereoInOutput4	363
Spatializing	StereoInOutput8	364
Spatializing	StereoMix2	365
Spatializing	Mixer	298
Spectral Analysis-FFT	ChannelJoin	236
Spectral Analysis-FFT	Mixer	298
Spectral Modifiers	SpectrumFrequencyScale	354
Spectral Modifiers	SpectrumLogToLinear	356
Spectral Modifiers	SpectrumModifier	357
Spectral Processing-Live	Gain	265
Spectral Processing-Live	OscillatorBank	309
Spectral Sources	LiveSpectralAnalysis	280
Spectral Sources	SyntheticSpectrumFromSounds	373
Spectral Sources	SpectralShape	352
Spectral Sources	SpectrumInRAM	355
Spectral Sources	SpectrumOnDisk	360
Spectral Sources	SyntheticSpectrumFromArray	371
Time & Duration	SetDuration	345
Time & Duration	TimeControl	377
Time & Duration	TimeOffset	380
Time & Duration	TimeStopper	381
Time & Duration	WaitUntil	395
Time & Duration	WarpedTimeIndex	396
Tracking Live Input	Gain	265
Tracking Live Input	Threshold	376
Tracking Live Input	FrequencyTracker	262
Tracking Live Input	OscilloscopeDisplay	310
Tracking Live Input	PeakDetector	316
Tracking Live Input	SpectrumAnalyzerDisplay	353
Variables & Annotation	Annotation	226
Variables & Annotation	SoundCollectionVariable	350
Variables & Annotation	Variable	389
Visual Displays	SoundToGlobalController	351
Visual Displays	OscilloscopeDisplay	310
Visual Displays	SpectrumAnalyzerDisplay	353
Vocoders	ScaleVocoder	341
Vocoders	TunableVocoder	384

Xtra	FeedbackLoopInput	254
Xtra	FeedbackLoopOutput	255
Xtra Sources	GAOscillators	266
Xtra Sources	MultifileDiskPlayer	300
Xtra Sources	Multisample	302
Xtra Sources	OscillatorBank	309
Xtra Sources	TimeFrequencyScale	378
Xtra Sources	Mixer	298

Sound Classes by Name

AbsoluteValue	Math	219
ADSR	Envelopes & Control Signals	220
AnalogSequencer	Sequencers	222
AnalogSequencer	Looping	222
AnalysisFilter	Filters-Mono	225
AnalysisFilter	Filters	225
Annotation	Variables & Annotation	226
Annotation	Frequency & Time Scaling	226
AR	Envelopes & Control Signals	227
ArcTan	Math	228
Attenuator	Global controllers	229
Attenuator	Gain & Level	229
Attenuator	Sources & Generators	229
AudioInput	Inputs	230
AudioInput	Sources & Generators	230
AveragingLowPassFilter	Filters-Mono	231
CellularAutomaton	Scripts	232
CenteringMixer	Mixing & Panning	234
ChannelCrossover	Spatializing	235
ChannelJoin	Spectral Analysis-FFT	236
ChannelJoin	Mixing & Panning	236
ChannelJoin	Filters-Stereo	236
Channeller	Spatializing	237
Channeller	Mixing & Panning	237
Chopper	Granulating & Chopping-Mono	238
Constant	Math	239
Constant	Envelopes & Control Signals	239
ContextFreeGrammar	Scripts	240
Crossfade	Mixing & Panning	242
DelayWithFeedback	Math	243
DelayWithFeedback	Delays-Mono	243
Difference	Spatializing	245
Difference	Mixing & Panning	245
Difference	Math	245
Difference	Envelopes & Control Signals	245
DiskCache	Sampling	246

DiskCache	Disk	246
DiskPlayer	Sampling	247
DiskPlayer	Disk	247
DiskPlayer	Sources & Generators	247
DiskRecorder	Sampling	248
DiskRecorder	Disk	248
DualParallelTwoPoleFilter	Filters	249
DynamicRangeController	Cross synthesis	250
DynamicRangeController	Compression/Expansion	250
DynamicRangeController	Additive synthesis	250
DynamicRangeController	Sources & Generators	250
EndTogetherMixer	Mixing & Panning	252
Equality	Math	253
FeedbackLoopInput	Xtra	254
FeedbackLoopOutput	Xtra	255
Filter	Sampling	257
Filter	Filters-Mono	257
Filter	Sources & Generators KBD Ctrl	257
ForcedProcessorAssignment	Sampling	259
FormantBankOscillator	Sources & Generators KBD Ctrl	260
FormantBankOscillator	Sources & Generators	260
FrequencyScale	Frequency & Time Scaling	261
FrequencyTracker	Tracking Live Input	262
FunctionGenerator	Envelopes & Control Signals	264
Gain	Tracking Live Input	265
Gain	Spectral Processing-Live	265
Gain	Modulation	265
Gain	Gain & Level	265
Gain	Frequency & Time Scaling	265
Gain	EQ	265
Gain	Envelopes & Control Signals	265
Gain	Compression/Expansion	265
GAOscillators	Xtra Sources	266
GAOscillators	Sources & Generators KBD Ctrl	266
GenericSource	Sampling	268
GenericSource	Inputs	268
GenericSource	Disk	268
GenericSource	Sources & Generators	268
GrainCloud	Sources & Generators KBD Ctrl	269

GrainCloud	Sources & Generators	269
GraphicalEnvelope	Envelopes & Control Signals	271
GraphicEQ	Filters	272
GraphicEQ	EQ	272
HarmonicResonator	Filters-Mono	273
HarmonicResonator	Sources & Generators KBD Ctrl	273
HighShelvingFilter	Filters	274
HighShelvingFilter	EQ	274
Interpolate	Math	275
IteratedWaveshaper	Sources & Generators KBD Ctrl	276
KeyMappedMultisample	Sources & Generators KBD Ctrl	277
LimeInterpreter	Scripts	279
LiveSpectralAnalysis	Spectral Sources	280
LowShelvingFilter	Filters	282
LowShelvingFilter	EQ	282
Matrix4	Spatializing	283
Matrix4	Outputs	283
Matrix8	Spatializing	284
Matrix8	Outputs	284
MemoryWriter	Sampling	286
MIDIFileEcho	MIDI Out	288
MIDIMapper	MIDI In	289
MIDIOutputController	MIDI Out	292
MIDIOutputEvent	MIDI Out	293
MIDIOutputEventInBytes	MIDI Out	294
MIDIVoice	Scripts	295
MIDIVoice	MIDI In	295
Mixer	Xtra Sources	298
Mixer	Spectral Analysis-FFT	298
Mixer	Spatializing	298
Mixer	Sampling	298
Mixer	Reverb-Stereo	298
Mixer	Reverb-Spatializing	298
Mixer	Mixing & Panning	298
Mixer	Granulating & Chopping-Mono	298
Mixer	Frequency & Time Scaling	298
Mixer	Flanging & Chorus-Mono	298
Mixer	Filters-Mono	298

Mixer	Distortion & Waveshaping	298
Mixer	Delays-Mono	298
Mixer	Compression/Expansion	298
Mixer	Sources & Generators KBD Ctrl	298
Mixer	Sources & Generators	298
Monotonizer	Frequency & Time Scaling	299
MultifileDiskPlayer	Xtra Sources	300
MultiplyingWaveshaper	Distortion & Waveshaping	301
MultiplyingWaveshaper	Sources & Generators	301
Multisample	Xtra Sources	302
MultisegmentEnvelope	Envelopes & Control Signals	304
MultislopeFunctionGenerator	Envelopes & Control Signals	305
Oscillator	Modulation	307
Oscillator	Envelopes & Control Signals	307
Oscillator	Sources & Generators	307
OscillatorBank	Xtra Sources	309
OscillatorBank	Spectral Processing-Live	309
OscillatorBank	Processing analyzed spectra	309
OscillatorBank	Frequency & Time Scaling	309
OscillatorBank	Additive synthesis	309
OscillatorBank	Sources & Generators KBD Ctrl	309
OscillatorBank	Sources & Generators	309
OscilloscopeDisplay	Visual Displays	310
OscilloscopeDisplay	Tracking Live Input	310
Output4	Spatializing	311
Output4	Mixing & Panning	311
Output8	Spatializing	312
Output8	Mixing & Panning	312
OverlappingMixer	Mixing & Panning	313
Pan	Mixing & Panning	314
ParameterTransformer	Scripts	315
PeakDetector	Tracking Live Input	316
PeakDetector	Envelopes & Control Signals	316
PhaseShiftBy90	Math	317
PresenceFilter	Filters	318
PresenceFilter	EQ	318
Preset	MIDI In	319
Product	Modulation	320
Product	Math	320

Product	Granulating & Chopping-Mono	320
Product	Envelopes & Control Signals	320
Product	Additive synthesis	320
Product	Sources & Generators KBD Ctrl	320
PulseGenerator	Sources & Generators	321
PulseTrain	Envelopes & Control Signals	323
QuadOscillator	Frequency & Time Scaling	324
RandomSelection	Scripts	325
REResonator	Cross synthesis	327
ReverbSection	Reverb-Mono	328
RhythmicCellularAutomaton	Scripts	330
RunningMax	Math	332
RunningMin	Math	333
Sample	Sampling	334
Sample	Looping	334
Sample	Sources & Generators	334
SampleAndHold	Sampling	336
SampleAndHold	Math	336
SampleAndHold	Envelopes & Control Signals	336
SampleCloud	Sources & Generators	337
SamplesFromDiskSingleStep	Disk	339
ScaleAndOffset	Math	340
ScaleVocoder	Vocoders	341
ScaleVocoder	Filters-Mono	341
ScaleVocoder	Sources & Generators KBD Ctrl	341
Script	Scripts	343
SetDuration	Time & Duration	345
SetRange	Math	346
SimplePitchShifter	Frequency & Time Scaling	347
SOS_Oscillators	Additive synthesis	349
SoundCollectionVariable	Variables & Annotation	350
SoundToGlobalController	Visual Displays	351
SoundToGlobalController	Global controllers	351
SpectralShape	Spectral Sources	352
SpectrumAnalyzerDisplay	Visual Displays	353
SpectrumAnalyzerDisplay	Tracking Live Input	353
SpectrumFrequencyScale	Spectral Modifiers	354
SpectrumFrequencyScale	Frequency & Time Scaling	354

SpectrumInRAM	Spectral Sources	355
SpectrumLogToLinear	Spectral Modifiers	356
SpectrumModifier	Spectral Modifiers	357
SpectrumOnDisk	Spectral Sources	360
SqrtMagnitude	Math	362
StereoInOutput4	Spatializing	363
StereoInOutput4	Reverb-Spatializing	363
StereoInOutput4	Mixing & Panning	363
StereoInOutput8	Spatializing	364
StereoInOutput8	Mixing & Panning	364
StereoMix2	Spatializing	365
StereoMix2	Mixing & Panning	365
StereoMix2	EQ	365
StereoMix4	Spatializing	366
StereoMix4	Scripts	366
StereoMix4	Mixing & Panning	366
StereoMix4	Drum machines	366
SumOfSines	Processing analyzed spectra	368
SumOfSines	Additive synthesis	368
SumOfSines	Sources & Generators KBD Ctrl	368
SumOfSines	Sources & Generators	368
SyntheticSpectrumFromArray	Spectral Sources	371
SyntheticSpectrumFromSounds	Spectral Sources	373
TextFileInterpreter	Scripts	375
Threshold	Tracking Live Input	376
TimeControl	Time & Duration	377
TimeFrequencyScale	Xtra Sources	378
TimeFrequencyScale	Sources & Generators	378
TimeOffset	Time & Duration	380
TimeStopper	Time & Duration	381
TriggeredSampleAndHold	Sampling	382
TriggeredSampleAndHold	Math	382
TriggeredSampleAndHold	Envelopes & Control Signals	382
TriggeredTableRead	Sampling	383
TriggeredTableRead	Envelopes & Control Signals	383
TunableVocoder	Vocoders	384
TunableVocoder	Filters	384
TwoFormantElement	Filters	386
TwoFormantElement	Sources & Generators	386

Variable	Variables & Annotation	389
VCA	Math	390
Vocoder	Filters	391
Vocoder	EQ	391
Vocoder	Cross synthesis	391
VocoderChannelBank	Filters	393
WaitUntil	Time & Duration	395
WarpedTimeIndex	Time & Duration	396

Reference

Menu Operations



The menus available in Kyma are **File**, **Edit**, **DSP**, **Action**, **Info**, and **Tools**. Only those menu operations that can be applied in the current context will be highlighted; all other menu operations will be dimmed and unselectable. Each menu and menu item is explained in the following sections.

File	
New...	⌘N
Open...	⌘O
Open any...	
Play...	
Import...	
Close	⌘W
Save	⌘S
Save as...	
Choose window...	
System prototypes	
Choose global map...	
Virtual control surface	
File organizer	
Status	
Quit	⌘Q

The **File** menu (see page 419) is used to create or modify files, file editors, and other Kyma windows.

Edit	
Undo	⌘Z
Cut	⌘H
Copy	⌘C
Paste	⌘V
Paste special...	
Paste hot...	⌘H
Clear	
Trim	
Evaluate	⌘Y
Select all	⌘A
Find...	⌘F
Find again	⌘G
Replace...	⌘J
Replace again	⌘T
Large window...	
Zoom in	⌘]
Zoom out	⌘[
Clean up	
View...	
Preferences...	

The **Edit** menu (see page 425) is used to edit Sounds, samples, or text within the active window.

DSP	
Stop	⌘K
Restart	⌘R
Status	
Configure MIDI...	
MIDI notes off	⌘M
Initialize DSP	

The **DSP** menu (see page 433) is used to configure and monitor the Capybara.

Action	
Compile, load, start	⌘P
Compile & load	
Record to disk...	
Compile to disk...	
Collect	
Duplicate	⌘D
Expand	⌘E
Revert	
Set default sound	
Set default collection	
Find prototype...	⌘B
Edit class	
New class from example	
Retrieve example from class	

The **Action** menu (see page 436) is used to manipulate the selected Sound(s) within the active window.

Info	
Get info	⌘I
Describe sound	
Structure as text	
Environment	
Reset environment	
Full waveform	
Oscilloscope	
Spectrum analyzer	

The **Info** menu (see page 439) is used to obtain information on the selected Sound(s) within the active window.

Tools	
Tape Recorder	⌘0
Spectral Analysis	⌘1
Synchronizing Spectra	⌘2
RE Analysis	⌘3
GA Analysis from Spectrum	⌘4
Design Alternate Tunings	⌘5

The **Tools** menu (see page 442) is used to start up high level tools for designing alternate tunings and for recording or analyzing sample files.



Many of the menu items have a keyboard short-cut listed to the right. All of these shortcuts are two-key combinations: the **Control** or the **Command** (marked as **⌘** or **⌘**) key and a letter key. Usually the letter is the first letter of the operation name (e.g., **Ctrl+S** or **⌘S** for **Save...** from the **File** menu).

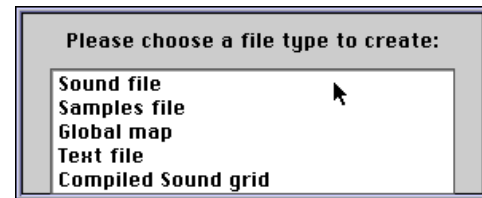
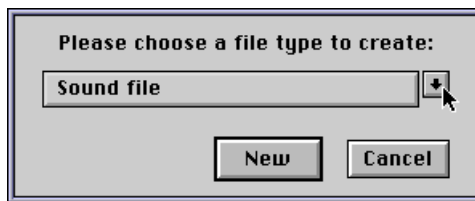
Kyma operating on a Macintosh OS computer uses a single menu bar in the standard position. Kyma operating on a Windows 95 computer uses a menu bar on each window; the menu bar for each window manipulates the contents of only that window. Windows-based versions of Kyma have an additional **Launcher** window not present on the Macintosh; the **Launcher** window contains most of the menu bar operations in pop-up menu form.[§]

File Menu

Operations under the **File** menu are used to create or modify files, file editors, and other Kyma windows.

File menu: New...

To create a new file choose **New...** from the **File** menu. **New...** opens a dialog box requesting the type of the new file.



To select a type, push down on the arrow to see a list of file types. Drag the mouse to the desired file type and release the mouse button. Click the **New** button or press **Enter** to create a new file.

Types of Files Usable in Kyma

The following table lists the kinds of files that Kyma can use.

File Description	File Type	Open	Create	Play	Import
Sound file	KYM0 / KYM	yes	yes	no	yes
AIFF sample file	AIFF / AIF	yes	yes	yes	yes
IRCAM/MTU samples	sf / SF	yes	yes	yes	yes
SD-I sample file	SFIL / SD1	yes	yes	yes	yes
SD-II sample file	Sd2f / ---	yes*	yes*	yes*	no
WAV sample file	.WAV / WAV	yes	yes	yes	yes
Spectrum file	KYM8 / SPC	yes	yes	yes	yes
Compiled Sound file	KYM9 / KYC	no	yes	yes	yes
Compiled Sound grid	PMAP / PRG	yes	yes	no	yes
Global map	KYM8 / MAP	yes	yes	no	yes
LIME file	Lime / LIM	no	no	no	yes
MIDI file	Midi / MID	no	no	yes	yes
Preferences file	PREF / PRE	no	no	no	no
Text file	TEXT / TXT	yes	yes	no	yes
Tool file	KYM7 / PCI	yes	no*	no	yes

[§] On Windows computers, if you close all of the windows in Kyma without exiting the Kyma application, you would no longer have access to a menu bar, since the menu bar appears in each window. The Launcher window contains most of the items found in the menu bar, so you can still open files, exit Kyma, *etc.* even in this case.

About the table:

File Description describes the kind of file.

File Type lists the Macintosh file type (4 letters) and the Windows file extension (3 letters).

Open indicates whether you can open an editor on files of this type via **Open...** in the **File** menu.

Create indicates whether you can use Kyma to create files of this type.

Play indicates whether Kyma can play files of this type via **Play...** in the **File** menu or the File Organizer.

Import indicates whether Kyma can import files of this type via **Import...** in the **File** menu.

About the files:

A **Sound file** contains a collection of Sound objects.

The five flavors of **sample files** contain digital recordings in various formats.

A **spectrum file** contains amplitude and frequency envelopes for additive resynthesis.

A **compiled Sound file** contains the Cappybara program for a specific Sound.

A **compiled Sound grid** is a collection of Sounds and their associated compiled Sound files.

A **global map** file contains text that describes a relationship between MIDI devices and Kyma Event Values.

A **LIME file** contains music notation created by the Lime application.[‡]

A **MIDI file** contains a MIDI sequence created by a MIDI sequencer application.

A **preferences file** contains the settings of Kyma's user preference items.

A **text file** contains text.

A **tool file** contains a high level tool, made up of a state machine with a graphic user interface, created in the developer's version of Kyma.

Where the File Type Information is Stored

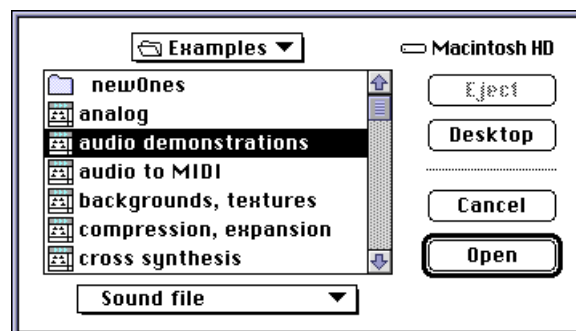
The Macintosh and Windows operating systems store the file type in different ways.

On Macintosh computers, there is a four character *file type* value stored in the resource fork of each file. Applications running on the Macintosh use this file type value to determine whether a specific file is a text file, MIDI file, AIFF sample file, *etc.*

On Windows computers, the file type is stored in the three character *extension* that follows the period in each file's name. Applications running on Windows use this extension to determine whether a specific file is a text file, MIDI file, AIFF sample file, *etc.*

File menu: Open...

To open a file that already exists, choose **Open...** from the **File** menu.



[‡] Lime is available at: <http://datura.cerl.uiuc.edu>.

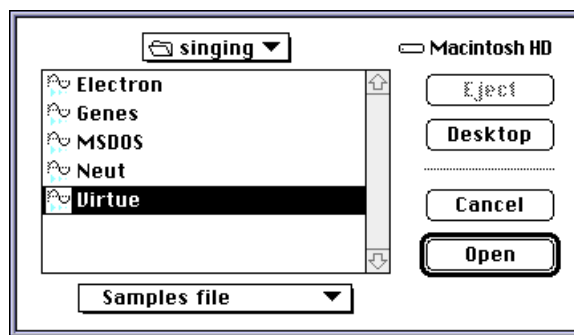
In the file list, select the file type and the name of the file to be opened. To select a type, push down on the arrow to see a list of file types. Drag the mouse to the desired file type and release the mouse button. The list of files will show files of the selected type only. Either double-click on the name of the desired file in the list, or click on the name of the desired file and click **Open** to open. A list of files that can be opened is given in the section called *Types of Files Usable in Kyma* on page 419.

File menu: Open any...

To open a file regardless of its file type, choose **Open any...** from the **File** menu. In the file list, select the name of the file to be opened and click **Open**, or double-click on the file name. Kyma will open the file with the appropriate file editor. A list of files that can be opened is given in the section called *Types of Files Usable in Kyma* on page 419.

File menu: Play...

To play a file without opening an editor on the file, choose **Play...** from the **File** menu. (To play a Sound, see *Action menu: Compile, load, start* on page 436.)



In the file list, select the file type and the name of the file to be played. To select a type, push down on the arrow to see a list of file types. Drag the mouse to the desired file type and release the mouse button. The list of files will show files of the selected type only. Either double-click on the name of the desired file in the list, or click on the name of the desired file and click **Open** to play. A list of files that can be played is given in the section called *Types of Files Usable in Kyma* on page 419.

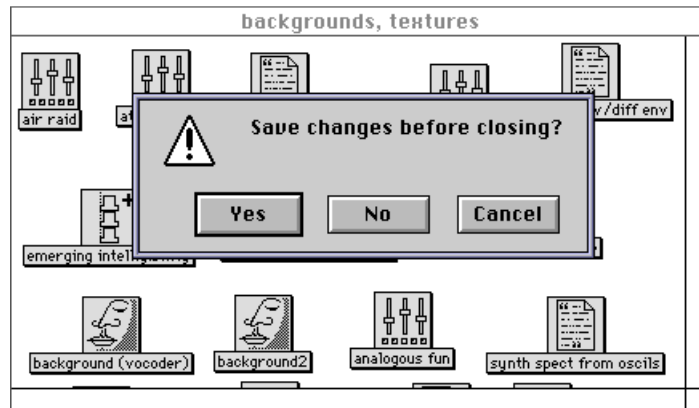
File menu: Import...

The Macintosh and Windows operating systems use different methods of storing the type of a file (see *Where the File Type Information is Stored* on page 420 for more information). When moving Kyma files from Windows to Macintosh computers (also, when moving files from the Internet to Macintosh computers), the file type is not available in the place normally used by the Macintosh for storing this information. Without this file type information, these files will not be usable by Kyma.

On Macintosh computers, choosing **Import...** from the **File** menu opens a file list. In the file list, select each file that has been moved to the Macintosh and click **Open**. Kyma will read the file type (encoded as the three letter extension at the end of each file name) and store the proper file type information in the selected file. Click **Cancel** when all files have been imported.

File menu: Close

Close closes the active window. Additionally, any window can be closed by clicking the window's close box (upper left corner of windows on the Macintosh; upper right corner of windows on Windows 95).



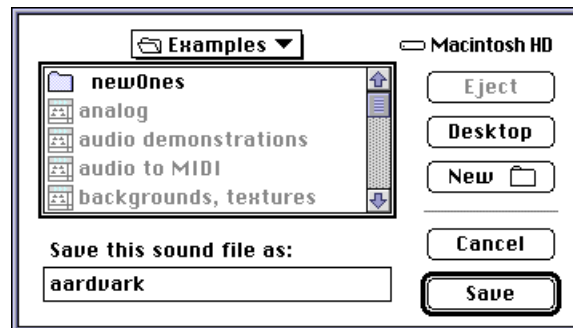
If there are changes in the window that have not been saved, Kyma will ask whether the changes in the window should be saved.

File menu: Save...

Choose **Save...** from the **File** menu to save the contents of the active window into its current file.

File menu: Save as...

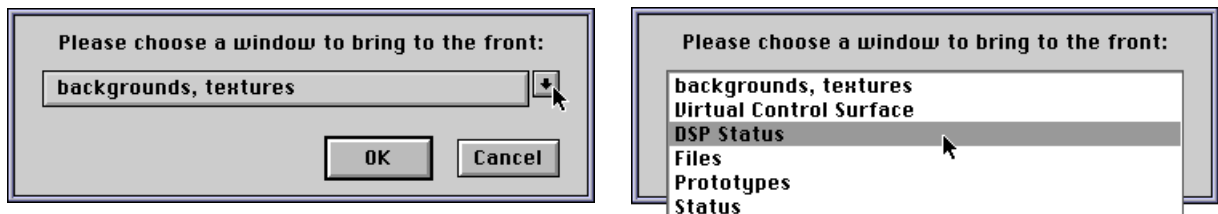
Choose **Save as...** from the **File** menu to save the contents of the active window to a different file.



In the file list, select the folder and enter the name of the file to store the contents of the active window. Click **Save** to save the contents of the window into the file.

File menu: Choose window...

It can be hard to locate a specific window when many overlapping windows are opened simultaneously. To cycle through the open windows, bringing each window in turn to the front, use either **Ctrl+** or **Ctrl+** . To bring a window to the front directly, choose **Choose window...** from the **File** menu.

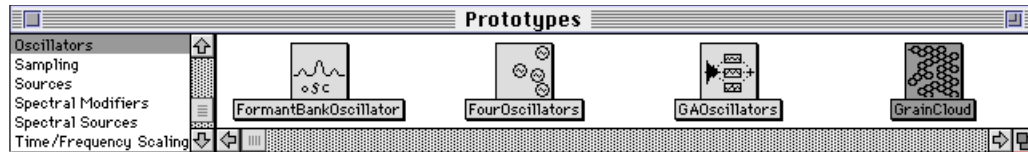


To select a window, push down on the arrow to see a list of the windows. Drag the mouse to the desired window and release the mouse button. Click **OK** to bring the selected window to the front.

File menu: System prototypes

The system prototypes window is a palette containing an example of each Sound built in to Kyma. These prototypical Sounds are modified and combined with other Sounds when doing sound design and composition in Kyma. (See *System Prototypes* and the Sound File Window on page 458 for more information.)

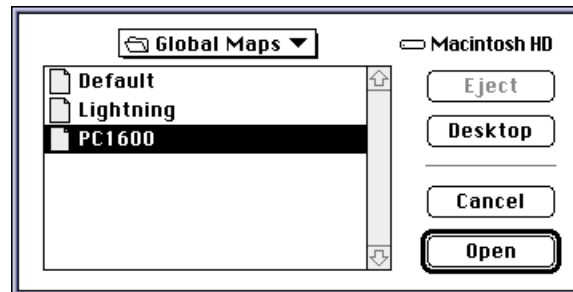
Choose **System prototypes** from the **File** menu to open the system prototypes window.



File menu: Choose global map...

The global map specifies a mapping between MIDI values received by the Cappybara (for example, values from continuous controller #18 on MIDI channel 1) and memorable names (for example, !Frequency). Sounds designed using these descriptive names can be used on any Kyma System, regardless of the MIDI devices connected to the Cappybara, by simply modifying the global map to reflect the available MIDI devices. (See *Event Values, Virtual Control Surface, and Global Map* on page 472 for more information.)

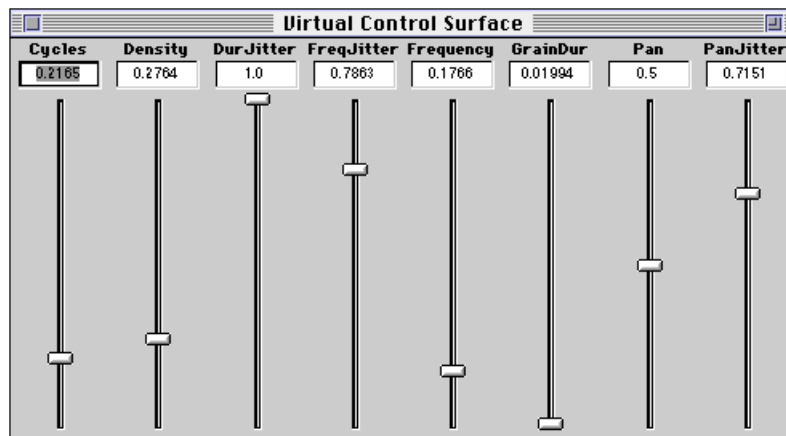
Choose **Choose global map...** from the **File** menu to open a file list form which to select the global map.



File menu: Virtual control surface

The virtual control surface displays the adjustable parameters of the Sound that was last compiled and loaded. The appearance of the controls within the window is set by the global map and any maps contained within the loaded Sound (see *Virtual Control Surface and Mappings* on page 483). As a preferences option, the virtual control surface can be opened whenever a Sound is compiled and loaded, see *Performance...* on page 430.

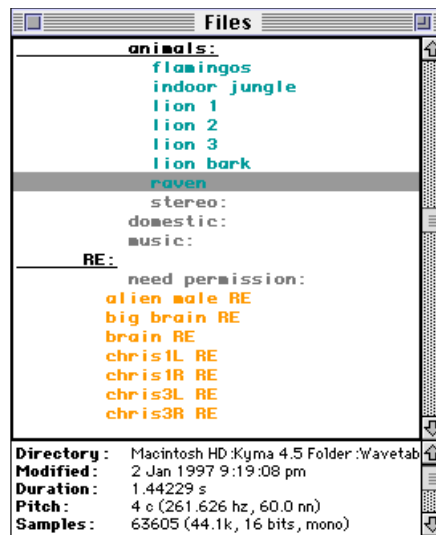
To open the virtual control surface, choose **Virtual control surface** from the **File** menu.



For more information, see *Virtual Control Surface* on page 481.

File menu: File organizer

Open the file organizer by choosing **File organizer** from the **File** menu. The file organizer provides a way to quickly access, edit, and audition sound-producing files on the hard disk of the host computer.



The file organizer lists the files on the host computer's hard disks, color-coded according to the type of each file, and indented to indicate the folder hierarchy. The color conventions are:

Description	Color Code
Unopened disk or folder	Gray
Open disk or folder	Underlined Black
Sample, EX	Turquoise
Spectrum	Purple
GA	Red-brown
RE	Yellow-orange
MIDI	Green

Double-click on a disk or folder to hide or show the folders and files contained in it. Select a file or folder by clicking on it once with the mouse. Use the arrow keys and the page up and down keys to move around in the list.

The information stored in the header of the selected file is displayed in the lower part of the file organizer window. The items displayed include the full path name for the file, the date and time on which it was last modified, and other information specific to the file type.

To hear the selected file, use **Ctrl+Space Bar**. Kyma plays samples files directly. For analysis files (spectrum, GA, or RE), Kyma uses the analysis to synthesize the sound. For MIDI files, Kyma constructs a minimal "orchestra" and plays the notes of the MIDI file on those simple "instruments".

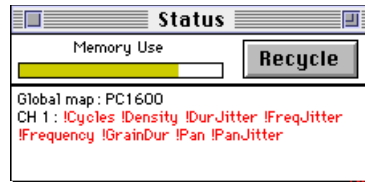
To edit the selected file, press **Enter** or double-click on the file. This will open the appropriate Kyma editor or, if there is an external editor specified for this file type (see *File Editors...* on page 430), it will open the preferred program for editing files of this type.

To create a Sound based on a file, drag the file from the file organizer into a Sound file window. To set the value of any parameter field in the Sound editor, drag a file name from the file organizer into the pa-

parameter field. If it is a parameter that makes sense for the file, the file will paste *its* value for that parameter into the field.

File menu: Status

To monitor the status of memory on the host computer, choose **Status** from the **File** menu. The following window will appear:



Memory Use

The horizontal bar indicates the percentage of available memory being used by the host computer. On color monitors, the display is color coded as follows: cyan for light usage, green and yellow for moderate usage, and red for heavy usage.

The **Recycle** button in the upper right is for recycling memory. Kyma tries to keep memory recycled automatically whenever this window is opened; however, if the bar turns red and Kyma starts to get sluggish, click the **Recycle** button to cause Kyma to recycle memory.

You should leave this window opened all of the time so that Kyma can automatically recycle memory for you.



When the **Recycle** button is clicked, in addition to scanning the host computer's memory for objects that are no longer in use, Kyma removes any information about recently accessed files and folders. This means that it is a good idea to click **Recycle** whenever a file or folder has been changed or moved to another location using another application (for example, using the Finder on the Macintosh, File Manager on Windows 3.1, or Windows Explorer on Windows 95).

MIDI Input

The bottom section of the status window displays information related to Event Values, MIDI input, and the A/D used in the last Sound that was loaded. The status window indicates the following:

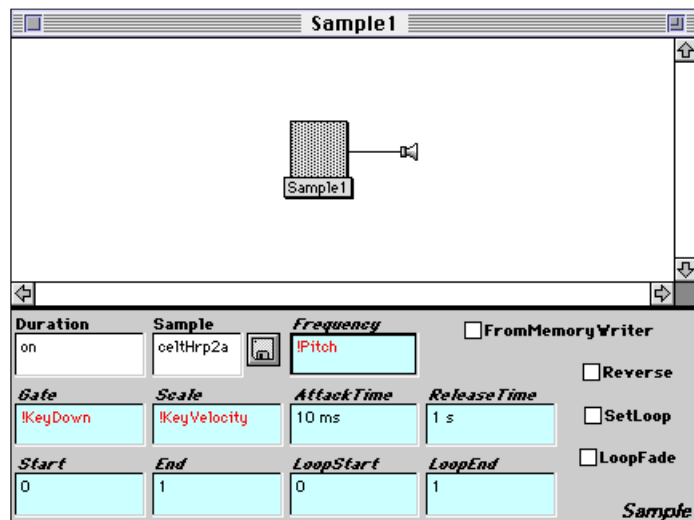
- the name of the global map being used to define Event Values
- whether the Sound expects input from the A/D
- the MIDI channels (if any) on which the Sound expects keyboard input
- the Event Value names (if any) that control the Sound's parameters

File menu: Quit

Choosing **Quit** from the **File** menu causes Kyma to close all open windows (asking to save changes for any window whose contents have been changed), save the preferences, and exit the Kyma application. Any Sound playing on the Cappybara will be stopped.

Edit Menu

Operations under the **Edit** menu are used to edit Sounds, samples, and text in the *active window* or *active field* within the active window. In windows that include several editable fields (for example, the Sound editor or the sample file editor), the active field will have a dark border drawn around it.



In this window, the **Frequency** field is active.

A field can be activated by clicking the mouse within its borders, or by using the **Tab** key to cycle through all editable fields.

Edit menu: Undo

Undo cancels the effect of the last editing operation.

Edit menu: Cut

Cut removes the selection (whether it is a Sound, samples, or text) from the active field and places it into the clipboard.

Edit menu: Copy

Copy places a copy of the selection (whether it is a Sound, samples, or text) into the clipboard.

Edit menu: Paste

Paste replaces the selection in the active field with the contents of the clipboard.

Edit menu: Paste special...

Paste special... applies to both text fields and Sound fields.

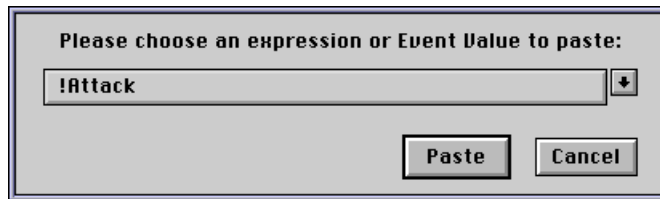
In text fields, **Paste special...** presents a choice of the last several text selections cut or copied. Use the pull down list to select the text, and click the **Paste** button to paste that text into the active field.



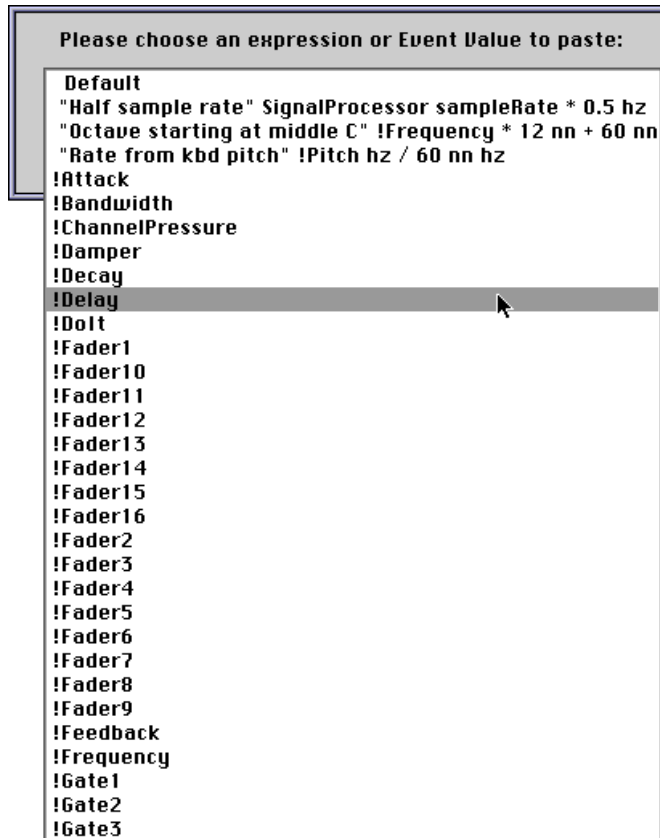
In Sound fields, **Paste special...** pastes exactly the Sounds that are in the clipboard, not a copy of the Sounds that are in the clipboard. This is one way to cause different Sounds to share the same input.

Edit menu: Paste hot...

Paste hot... presents a list of commonly used expressions together with all of the Event Values in the current global map.



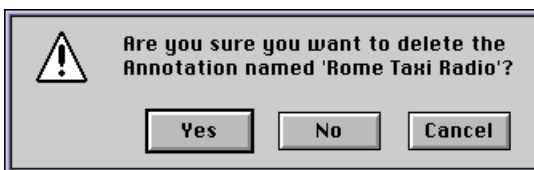
Use the pull down list to select an Event Value or expression, and click **Paste** to paste it into the hot parameter field. Alternatively, type all or part of the name of the expression and hit **Enter**.



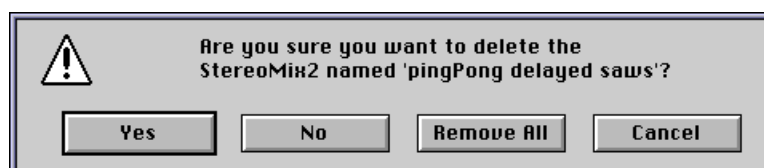
Edit menu: Clear

Clear from the **Edit** menu and the **Delete** key accomplish the same result: the selection is removed without being saved in the clipboard.

Before Kyma removes any Sound, it will verify that you really want to delete it.



If you have several Sounds selected and would like to remove all of them, click **Remove All**.



Edit menu: Trim

The **Trim** operation is the complement of the **Clear** operation. Whereas **Clear** removes the selection, leaving the unselected items untouched, **Trim** removes the unselected items, leaving the selection untouched.

Edit menu: Evaluate

Evaluate tries to evaluate the selection as a Smalltalk-80 expression, and, if successful, inserts the result immediately following the selection. (See *The Smalltalk-80 Language* on page 513 for an introduction to Smalltalk-80.) If there is an error, the field will flash and Kyma will either insert a message into the text indicating the error or it will present dialog explaining the error.

Edit menu: Select all

Choosing **Select all** from the **Edit** menu causes the entire contents of the active field to be selected.

Edit menu: Find...

Find... is used to search the active field for the text entered into the dialog. Kyma begins the search after the current selection, and selects the next occurrence of the text that it finds. Kyma will cause the host computer to beep if the text cannot be found.

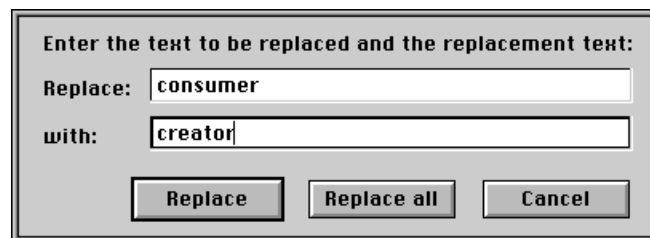


Edit menu: Find again

Find again searches the active field for the text that was last entered in **Find...** dialog. Kyma begins the search after the current selection, and selects the next occurrence of the text that it finds. Kyma will cause the host computer to beep if the text cannot be found.

Edit menu: Replace...

Replace... is used to replace an occurrence of one text with another text. Enter the text to be replaced and the replacement text, and then click **Replace** to replace the first occurrence or click **Replace all** to replace all occurrences. Kyma will cause the computer to beep if the text cannot be found.



Edit menu: Replace again

Replace again replaces the next occurrence of the text to be replaced with the replacement text. Kyma will cause the computer to beep if the text cannot be found.

Edit menu: Large window...

Large window... expands the active field to fill the entire screen. Click **Accept** to save the contents of the large window back into the active field, or click **Cancel** to discard the changes.

Edit menu: Zoom in

Certain parameter fields in the Sound editor (for example, the **Envelope** field of *GraphicalEnvelope*) and certain fields of other editors (the spectrum area in the spectrum editor) can be viewed with different magnifications. **Zoom in** increases the magnification of the field, showing more detail.

Edit menu: Zoom out

Zoom out performs the opposite of **Zoom in**: it decreases the magnification of the field, showing less detail.

Edit menu: Clean up

In fields or windows that contain Sounds, **Clean up** positions the Sounds into straight rows while attempting to keep them from overlapping. When applied to the signal flow diagram in the Sound editor, **Clean up** restores the original layout of the signal flow diagram.

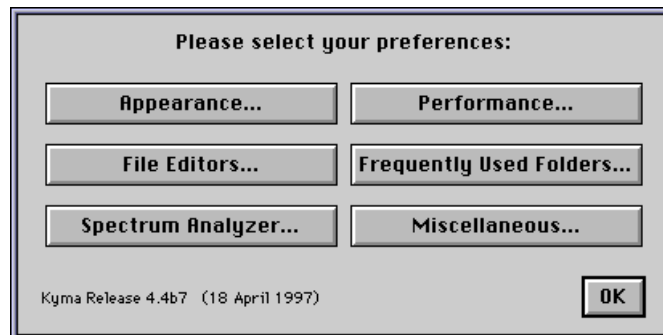
Edit menu: View...

View... controls the appearance of a Sound file window, a Sound editor, or the file organizer. For Sound file windows, **View...** controls the size of the icons in the window. For Sound editors, **View...** controls icon size and icon connection style. For file organizers, **View...** controls how folders are displayed.

Edit menu: Preferences...

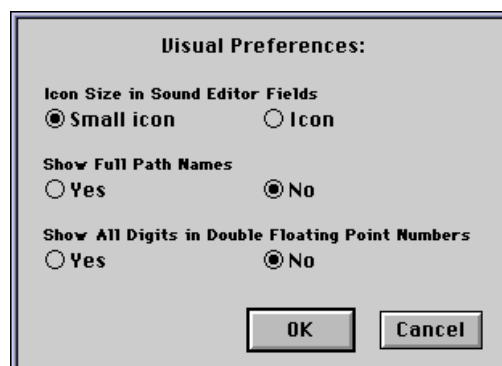
The global preference items can be set by choosing **Preferences...** from the **Edit** menu. These preferences are automatically saved in a file (Kyma Preferences on the Macintosh or `kyma.pre` on Windows) when exiting Kyma.

The six categories of preferences are explained in detail below.



The Kyma release number and release date is shown in the lower left of the dialog.

Appearance...



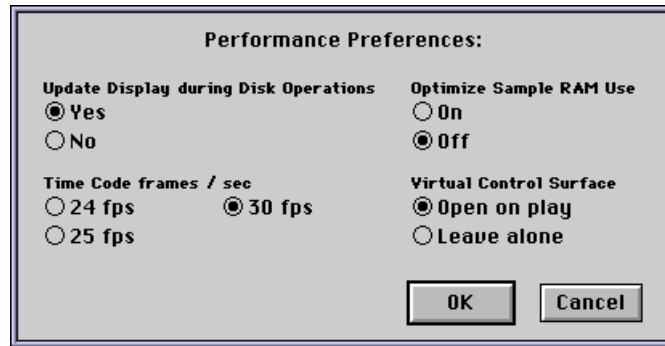
The appearance preferences control the appearance of certain items in the parameter fields of Sounds:

Icon Size in Sound Editor Fields controls the size of Sound icons in **Input** or other Sound fields.

Show Full Path Names controls whether fields that display file names (for example, **Sample**) show the entire file path or only the name of the file.

Show All Digits in Double Floating Point Numbers controls whether double precision floating point numbers are displayed with all twelve digits or with only six digits.

Performance...



The performance preferences control how Kyma should behave when compiling, loading and starting Sounds.

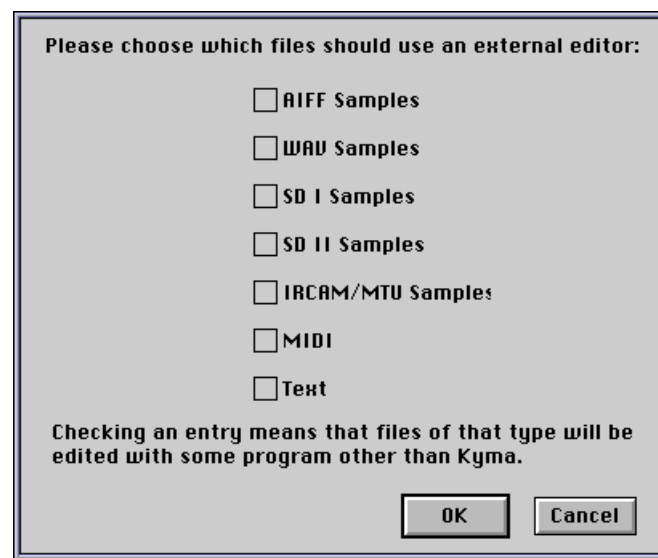
Update Display during Disk Operations controls whether Kyma will allow editing and other forms of user interaction when playing a Sound that uses the host computer's hard disk (for example, *DiskPlayer* or *DiskRecorder*).

Optimize Sample RAM Use controls how efficiently Kyma will use the Cappybara sample RAM. If turned off, the Kyma compiler will operate more quickly, but each sample file referenced will be loaded into all Cappybara expansion cards, possibly wasting sample RAM. If this option is turned on, the Kyma compiler will load each sample file referenced into the Cappybara expansion cards that use the sample.

Time Code frames / sec controls how Kyma interprets SMPTE times and durations (for example, 1:23:45.12 SMPTE) in the parameter fields of Sounds. (Use **30 fps** for either 29.97 or 30 fps).

Virtual Control Surface controls whether the virtual control surface should be opened and brought into view or left in its current state whenever a Sound is played.

File Editors...



The file editors preferences control the selection of editors for different kinds of files. If the box is checked in front of a file type, then Kyma will use a different application to edit files of that type. On computers using Windows, the application chosen will be the same as the one that the Windows File Manager or Windows Explorer uses when a file is opened. On computers using the Macintosh operating system, Kyma will prompt for the application to use.

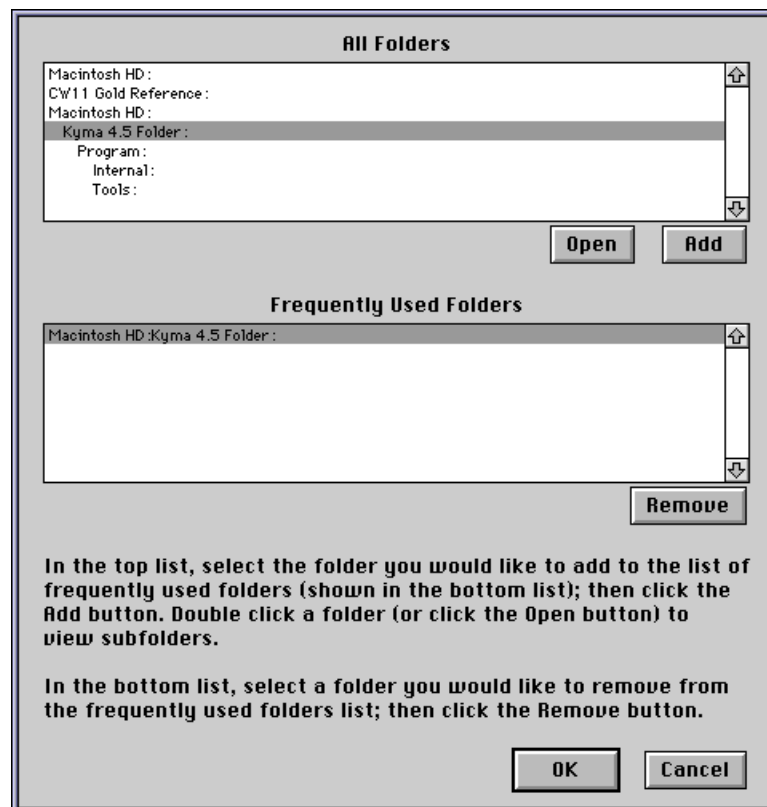


Choosing **Open...** from the **File** menu (see page 420) and selecting a file will always use Kyma's built-in editor for files of that type. The editors described here are used when editing a file from the File Organizer (see page 424) or when editing a file from the parameter field of a Sound (by clicking on the field's disk button while holding down either the **Option** or **Control** keys).

Frequently Used Folders...

The frequently used folders (or directories) preferences controls where Kyma looks to locate files. Select folders in the upper list and click **Open** to display any folders inside of the selected folder. Click **Add** to add the selected folder to the list of frequently used folders. To remove a folder from the list of frequently used folders, select the folder in the bottom list and click **Remove**.

Kyma will use these folders *and the folders that they contain* to locate files it cannot find immediately. Therefore, it is unnecessary to add folders contained in any folder already in the frequently used folders list.



Because Kyma removes any path information from files that it cannot locate immediately, it is important that there is *only one* file with a given name among all of the folders and subfolders of the frequently used folders. Keep in mind that Kyma may select *any* of the files with the same name among the frequently used folders.

How Kyma locates files



While Kyma is locating a file, it changes the cursor to some eyeglasses; the eyeballs within the lens move as Kyma searches folders for the desired file.

Kyma uses the following procedure to locate files:

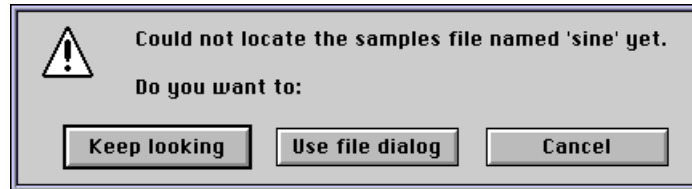
If the file name has a complete path and the file exists, Kyma uses the file name directly.

Kyma removes any path information from the name.

If that file name has been found recently, Kyma uses the file that was found at that time.

Kyma checks the most recently used folders for a file with the same name.

Kyma checks each folder and subfolder in the list of frequently used folders for a file with the same name. During this process, Kyma will periodically put up this dialog:



Click **Keep looking** to have Kyma continue checking the frequently used folders list, click **Use file dialog** to locate the file using a file list, or click **Cancel** to stop Kyma.

At this point, Kyma has exhausted all automatic ways of finding the file and opens a file list.



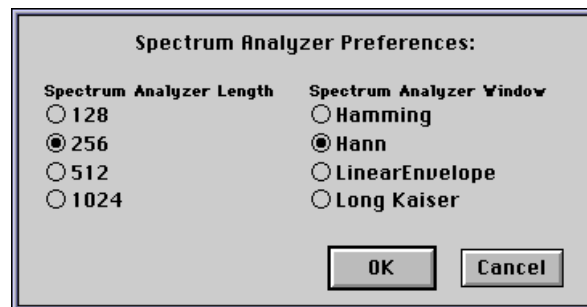
Because Kyma keeps track of recently accessed files and folders, it is important to inform Kyma whenever a file or folder has been changed using another application (for example, renaming or deleting files using the Finder on the Macintosh, File Manager on Windows 3.1, or Windows Explorer on Windows 95). To do this, click on the **Recycle** button in the Status window (see **File menu: Status** on page 425).

Substituting for missing files

If you use Sounds developed by someone else on another computer, that person might have used a different set of files or might have created some files you do not have on your computer. When you try to load a Sound that uses a file you do not have, a dialog box will prompt you to locate the file. The same thing could happen if you rename a file on the host computer's hard disk but have not changed the reference to that file name in the Sounds' parameter fields.

To have Kyma substitute a different file in place of the missing file, choose the replacement file from the file list. This substitution will remain in effect until either the **Recycle** button in the Status window is clicked or the Capybara is initialized via **Initialize DSP** under the **DSP** menu.

Spectrum Analyzer...

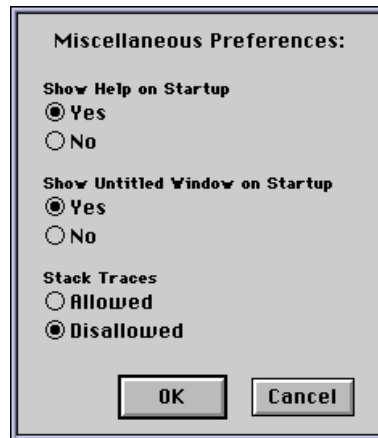


The spectrum analyzer preferences control the settings for the spectrum analyzer that Kyma creates when **Spectrum analyzer** from the **Info** menu is chosen. (Kyma also includes a separate *SpectrumAnalyzer* Sound that displays the spectrum of its input, but the Sound is unaffected by these preference settings.)

Spectrum Analyzer Length controls the length of the FFT used by the spectrum analyzer. The larger the number the better the frequency resolution.

Spectrum Analyzer Window controls the window function applied to the signal before using the FFT. Each of the window functions has different properties; in general, resolution increases in the following order: **LinearEnvelope**, **Hamming**, **Hann**, and **Long Kaiser**.

Miscellaneous



The miscellaneous preferences control a few random items.

Show Help on Startup controls whether a help window will be opened when Kyma is started.

Show Untitled Window on Startup controls whether an untitled Sound file window will be opened when Kyma is started.

Stack Traces controls whether Kyma allows low-level debugging stack traces when errors occur. Generally, this should be left set to **Disallowed** unless instructed to do otherwise by Symbolic Sound.

DSP Menu

Operations under the **DSP** menu are used to configure and monitor the Capybara.

DSP menu: Stop

To stop a Sound while it is playing, choose **Stop** from the **DSP** menu.



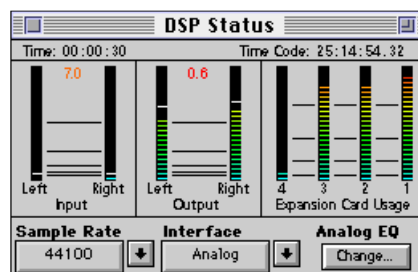
If **Update Display during Disk Operations** in the **Performance...** preferences (see page 430) is turned off and the Sound uses the computer's hard disk, click the mouse while holding down the **Shift** key to stop the Sound.

DSP menu: Restart

Once a Sound has been loaded into the Capybara, it can be restarted immediately by choosing **Restart** from the **DSP** menu. This is often used in conjunction with **Compile & load** from the **Action** menu (see page 436).

DSP menu: Status

The DSP status window configures the audio inputs and outputs, and, while a Sound is running, it displays the expansion card usage and the input and output levels. To open the DSP status window, select **Status** from the **DSP** menu.



Time and Time Code Displays

The time display at the upper left of the status window is reset to zero each time you start a Sound. The display format is *hours:minutes:seconds*. When the Sound finishes (or when the Sound is stopped by choosing **Stop** from the **DSP** menu), the time display stops.

The time code display in the upper right of the status panel shows incoming MIDI time code. The display format is *hours:minutes:seconds.frames*.

Input and Output Levels

The two bars at the left of the window show the left and right input levels. The left and right output levels are shown by the pair of bars to the right of the input level bars. The marks alongside the bars represent steps of 10 dB.

Between each pair of bars is a peak level indicator. The peak level indicator will show >0< if the signal *may* have clipped. The peak level indicators can be reset by clicking the mouse.

Expansion Card Usage

The number of bars displayed in the Expansion Card Usage section of the status window will depend upon the number of expansion cards installed in your Capybara. Shown above is an example with four expansion cards. Each bar indicates the current computational load on each expansion card relative to the card's ability to compute Sounds in real time. When the expansion cards are in normal operation (processing in real time), each mark indicates approximately 20% expansion card usage.



If your Capybara is unable to process a complex Sound in real time, you can use these bars to determine when that happens. The bar representing the most heavily-taxed expansion card will “stick” against the top of the scale until the card's processing time catches up with real time. By noting when the bars run out of headroom, you can pinpoint which part of the Sound needs to be simplified or recorded into a disk cache to allow the expansion cards to return to real time computation. Alternatively, you can lower the sample rate and play the Sound again.



When the Capybara is unable to process a complex Sound in real time, the usage bar display does not accurately display the usage of the expansion cards.

Sample Rate

Use the pull down menu in the **Sample Rate** field to change the sampling rate of the Capybara. There are seven sampling rates available via the menu; they are divisions of the compact disk sampling rate: 44100, 22050 and 11025 samples per second; divisions of the digital audio tape sampling rate: 48000, 24000 and 12000 samples per second; and the extended-play DAT sampling rate of 32000 samples per second. The higher you set the sampling rate, the higher the frequencies that you can synthesize or reproduce; you can produce frequencies up to half of the sampling rate. However, the higher the sampling rate, the less time the Capybara has to compute each sample.



In general, use the highest sampling rate that still allows the Capybara enough time to compute each sample. If an algorithm is particularly complex, it may be necessary to lower the sampling rate (allowing more computation time per sample) or to store the sound on the host computer's hard disk for later real time playback.

Interface

Use the **Interface** pull down menu to choose between the analog and digital audio interfaces. If a digital device is plugged into the Capybara's digital input, the source will determine the sample rate, overriding Kyma's setting (the sample rate setting display will not automatically update, however). For example, if the source is operating at a 32 khz sample rate, Kyma will operate at that sample rate.

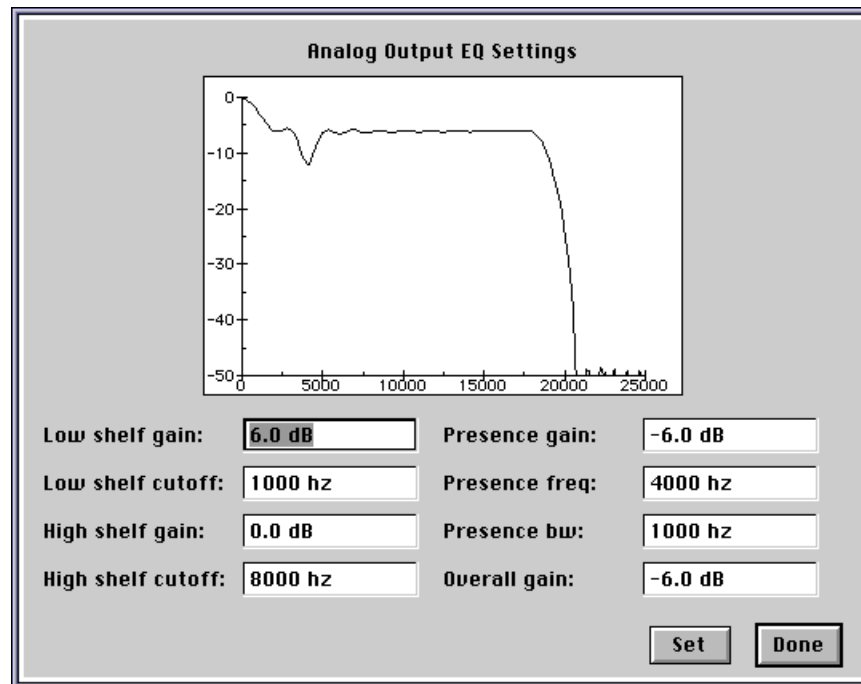


Even though the device controls the sample rate of the signal processor, Kyma has no way of finding out that sample rate value. So, whenever using the digital audio input, remember to make sure that the sample rate shown in the Sample Rate field is the same as the sample rate of the digital device.

Analog EQ

The analog audio output of the Capybara has an equalization section that can be used to adjust the frequency content of the output signal. Click on the **Analog EQ** button to configure this equalization section.

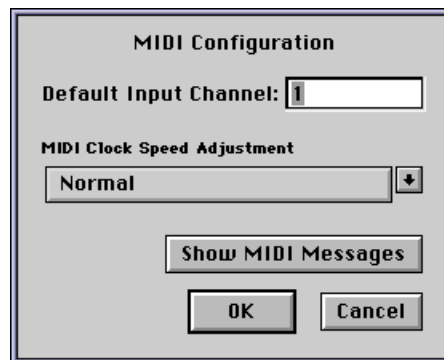
The upper part of the configuration dialog shows the frequency response of the analog equalizer. The fields below are used to set the parameters for the overall gain, a low shelving filter, a high shelving filter, and a presence filter. Click the **Set** button to update the equalizer and the frequency response graph. Click **Done** when finished.



To disable the action of any of the filters specify 0.0 dB for the filter's gain parameter. The default equalizer setting has all filters disabled, giving a flat frequency response between DC and 40% of the sampling rate.

DSP menu: Configure MIDI...

The Capybara's MIDI interface is configured by choosing **Configure MIDI...** from the **DSP** menu.



The **Default Input Channel** is the MIDI channel used by Sounds that are controlled by MIDI but have no *MIDI Voice*, *MIDI Mapper*, or other Sound to specify the MIDI channel to use. *MIDI Voices* and *MIDI Mappers* with **Channel 1** set to 0 use the **Default Input Channel** as well.

The **MIDI Clock Speed Adjustment** pull down menu is used to speed up or slow down the Capybara's MIDI clock. Some MIDI devices (including some personal computer MIDI interfaces) use a MIDI clock that is incompatible with the normal clock setting. Use this menu to try slower and faster MIDI clock

speeds that may be compatible with the MIDI device. This option has no effect on the responsiveness or speed of processing of MIDI events on the Capybara.

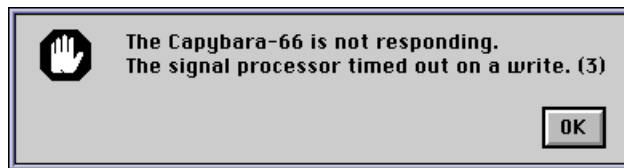
Clicking the **Show MIDI Messages** button opens a window that displays information about the continuous controller and keyboard MIDI messages as they are received by the Capybara. This can be useful when troubleshooting MIDI configurations.

DSP menu: MIDI notes off

MIDI notes off sends a note-off event (key up) for every note on every MIDI channel. The note-off events are sent to the Capybara MIDI out connector, silencing any MIDI synthesizers attached to the Capybara.

DSP menu: Initialize DSP

Use **Initialize DSP** from the **DSP** menu to initialize the Capybara. There are only a few circumstances where this will be needed. Generally, in any of these circumstances, Kyma will display a dialog (such as the one below) to indicate a problem.



If you see this warning, click **OK**, and then choose **Initialize DSP** from the **DSP** menu.

Action Menu

Operations under the **Action** menu are used to manipulate the selected Sound(s) within the active window.

Action menu: Compile, load, start

Selecting **Compile, load, start** from the **Action** menu causes the selected Sound to be compiled and then loaded into the digital signal processor. **-Space Bar** or **Ctrl+Space Bar** are the key-equivalents. Often the compiler will be able to start the Capybara playing while it loads the rest of the Sound.

If more than one Sound and/or Sound collection is selected, the Sounds will be loaded one after another in order from top to bottom and left to right.



While the Sound is being compiled, the cursor changes to a watch. If the Sound uses any sample files, the cursor will change to an animated picture of a waveform being downloaded to the Capybara while the sample files are being transferred from the hard disk into the RAM. Once Kyma begins sending information to the Capybara, the cursor changes to an animation of a Sound transferring data to the Capybara.

While the Sound is playing, the cursor changes:



If the Sound does not use the computer's hard disk, the cursor will be a hollow arrow.

If the Sound uses the computer's hard disk, and **Update Display during Disk Operations** in the **Performance...** preferences (see page 430) is turned on, the cursor will be a partially-filled arrow.

If the Sound uses the computer's hard disk, and **Update Display during Disk Operations** in the **Performance...** preferences is turned off, the cursor will be a floppy disk.

To stop a Sound while it is playing, choose **Stop** from the **DSP** menu. If the cursor looks like a floppy disk, *click the mouse while holding the **Shift** key down*. When the Sound has finished or has been stopped, the cursor will change back to the normal fully-filled arrow.

Action menu: Compile & load

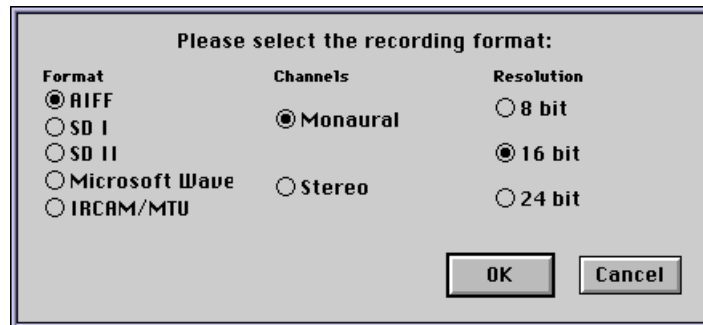
Choose **Compile & load** to compile and load the selected Sound into the Capybara without immediately starting the Sound. Once the Sound has been compiled and loaded, start it by selecting **Restart** from the

DSP menu. This is useful for loading a Sound before a performance or for *very* dense Sounds with many structure changes that must be completely downloaded before being started.

Action menu: Record to disk...

Select **Record to disk...** to make a digital recording of the selected Sound into a sample file on the hard disk. (See *Samples* on page 493 for additional information about digital recordings.)

A dialog box asks you to specify how the Sound should be recorded: pick a file format; decide whether to record in stereo or monaural; and choose a bit resolution of 8, 16, or 24-bits.

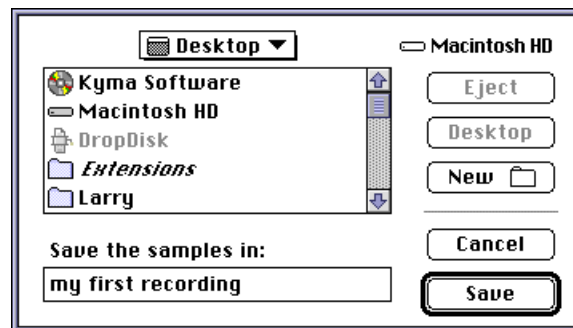


If your Sound is the same in the left and right channels, recording in monaural will save disk space. Similarly, if you know that the recorded samples will only be played back over 16-bit converters, you can save disk space by choosing the 16-bit recording format.



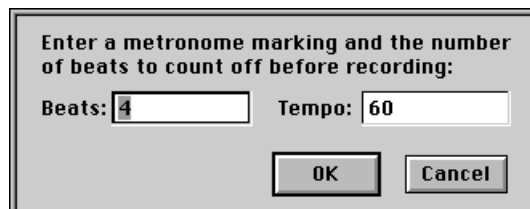
In monaural mode, only the left channel of the selected Sound is recorded to disk.

Choose the file format, number of channels, and resolution, and click **OK** or press **Enter**. Then enter the name of the sample file in which to place the digital recording:



and click **Save** or press **Enter**.

If the selected Sound uses live input (either MIDI or audio input), Kyma will prompt for the length and tempo of a metronome count-off. When you click **OK** the Capybara will beep for the specified number of beats at the specified tempo. Recording begins on the beat after the count-off.



If you don't want a count-off, enter zero for the number of beats.

Finally, Kyma will compile, load, and start the selected Sound, recording the audio into the sample file. To stop the recording prematurely, click the mouse button while holding down the **Shift** key.



Even if the selected Sound cannot play in real time (and is heard with clicks in the audio output), the digital recording made will be correct and will be free of clicks.

Action menu: Compile to disk...

Compile to disk... stores a compiled version of the selected Sound onto the disk. Compiled Sound files can be played by choosing **Play...** from the **File** menu (see page 421). Compiled Sound files are also used by the compiled Sound grid.

Compiled Sound files and the compiled Sound grid are explained in detail in *Compiled Sound Grid* starting on page 485.

Action menu: Collect



Collect... applies only to groups of selected Sounds in Sound file windows. It prompts for the name of the collection and places the selected Sounds into a new Sound collection (icon shown on the left) with the given name.

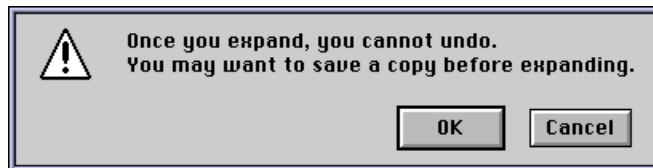
Action menu: Duplicate

Duplicate applies only to the selected Sound in a Sound file window. It makes a copy of the selected Sound(s) and/or Sound collections(s), and puts them into the Sound file window. A duplicate of a Sound collection contains a duplicate of each Sound that was in the original collection.

Action menu: Expand

More than half of the prototypes supplied with Kyma are constructed out of other Sounds. Selecting **Expand** from the **Action** menu makes this lower level construction available for editing and inspection. **Expand** often is used to debug complex Sounds and Sound classes.

Expand can be applied in the Sound file window and the Sound editor. In the Sound file window, a copy of the expanded Sound is saved in the window and no changes are made to the selected Sound. However, in a Sound editor, **Expand** replaces the selected Sound with its expansion; this operation is not reversible.



Action menu: Revert

Revert applies only to the Sound editor. It discards all changes made to the Sound whose parameters are displayed in the lower half of the Sound editor.

Action menu: Set default Sound

Kyma keeps a default Sound that it can use as a value for variables that are Sounds. To change the value of this default Sound, select a Sound and choose **Set default sound**. See *Variables* on page 510 for more information.

Action menu: Set default Sound collection

Kyma also keeps a default Sound collection that it can use as a value for *SoundCollectionVariables*. To change the value of this default Sound collection, select a group of Sounds and choose **Set default collection**. See *Variables* on page 510 for more information.

Action menu: Find prototype

Find prototype... applies only to prototype windows. Type all or part of the name of a Sound into the dialog, and click **OK** or press **Enter**. Kyma will select the named Sound. If more than one Sound matches the name entered, choose one of the Sounds from the list.

Action menu: Edit class

Edit class can be applied in Sound file windows and Sound editors. Choose **Edit class** to modify aspects of the selected Sound's class. Class creation and modification are described in *The Class Editor* on page 536.

Action menu: New class from example

New class from example can be applied only in Sound file windows. It defines a new Sound classes based upon the selected Sound. Class creation and modification are described in *The Class Editor* on page 536.

Action menu: Retrieve example from class

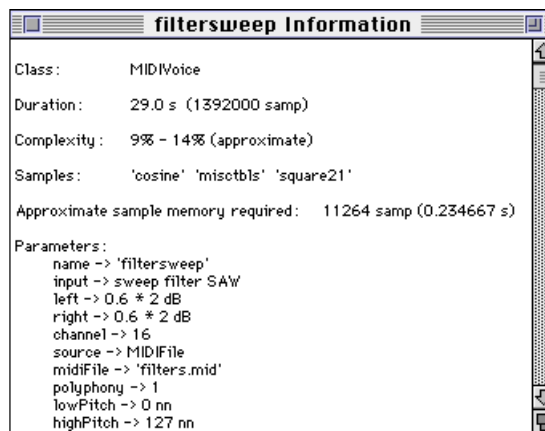
Retrieve example from class, active only in the Sound class editor, saves the original Sound on which this class was based, preserving any variables that were in the original Sound. Class creation and modification are described in *The Class Editor* on page 536.

Info Menu

Operations under the **Info** menu are used to obtain information on the selected Sound(s) within the active window.

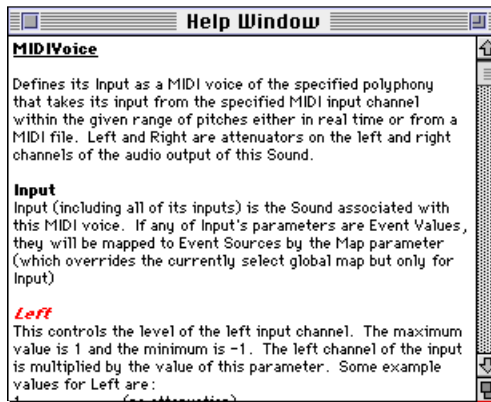
Info menu: Get info

Get info opens a window containing information about the selected Sound, including its Sound class, duration (in samples and seconds), complexity (useful as a rough estimate of the relative efficiencies of different Sounds), variable names (if any), Event Value names (if any), sample files, and parameter names and values.



Info menu: Describe sound

Describe sound opens a help window with a description of the selected Sound's class and its parameters.



Info menu: Structure as text

Structure as text opens a window containing a textual representation of the Sound structure. Each Sound is represented by the name of the Sound followed by a list of its inputs' names enclosed in parentheses. Below each Sound, each of its inputs is listed in the same manner but indented. The level of indenting indicates the number of levels between the Sound and the final output Sound.



Info menu: Environment

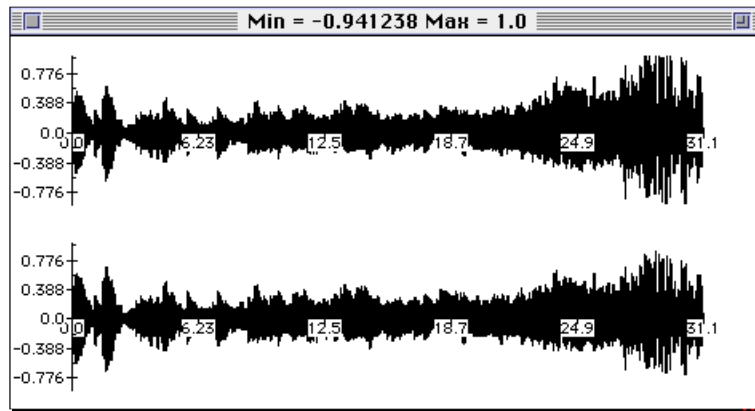
Environment applies only in the Sound editor. It shows the current mapping of Kyma variables to values. To clear that mapping, select **Reset environment**. See *Variables* on page 510 for more information.

Info menu: Reset environment

Reset environment clears the current mapping of Kyma variables to values. See *Variables* on page 510 for more information.

Info menu: Full waveform

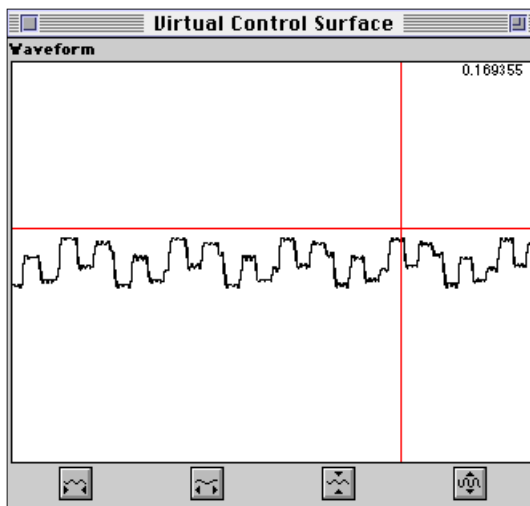
To see a graphic representation of the selected Sound as amplitude versus time, select **Full waveform**. The Cappybara will play the Sound, recording it into the Cappybara's sample memory. If the Sound's total duration is too long to fit into memory, Kyma will ask you to specify the length of time that you would like to plot. After performing the recording, Kyma reads the waveform out of the memory and displays it in a newly opened window. Also displayed are the minimum and maximum amplitudes for that Sound.



If the minimum and/or maximum are -1 or +1, respectively, and portions of the displayed waveform appear to be flattened on the top or bottom, it is likely that the Sound is clipping; correct this by attenuating the amplitudes in the Sound structure. Click in the close box when you are finished with the waveform.

Info menu: Oscilloscope

When **Oscilloscope** from the **Info** menu is chosen, Kyma compiles, loads, and starts the selected Sound and opens an oscilloscope display in the virtual control surface. The oscilloscope displays a mix of the left and right outputs of the selected Sound.



The buttons across the bottom shrink the time axis, expand the time axis, shrink the amplitude axis, and expand the amplitude axis, respectively. The amplitude value corresponding to the vertical position of the cursor is displayed in the upper right of the oscilloscope window as long as the mouse is inside the window.



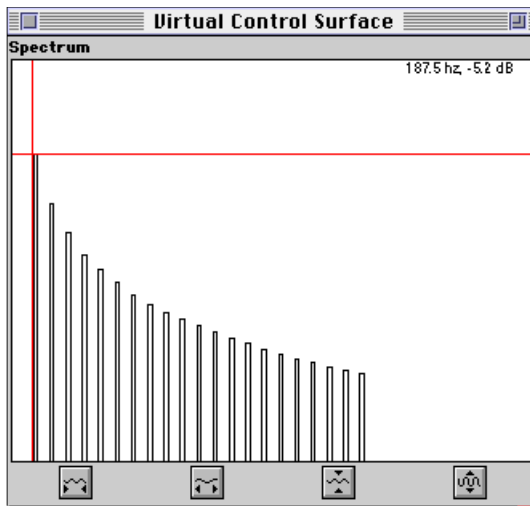
The oscilloscope always displays a power-of-two number of samples; a signal with a period that is 1, 2, 4, 8, 16, 32, 64, 128, 256, or 512 samples will be locked on the display.

Info menu: Spectrum analyzer

When **Spectrum analyzer** is chosen, Kyma compiles, loads, and starts the selected Sound and displays the spectrum of the selected Sound in real time in the virtual control surface window. The spectrum of a mix of the left and right outputs of the selected Sound is displayed.

The spectrum window displays the output of an FFT; the horizontal axis is linear frequency and the vertical axis is magnitude in dB. The buttons across the bottom shrink the frequency axis, expand the frequency axis, shrink the magnitude axis, and expand the magnitude axis, respectively. The length of the

FFT and the window function applied before the FFT are both specified in the preferences under **Spectrum analyzer...** (see *Spectrum Analyzer...* on page 432 for more information).



The image above shows the real-time spectrum analyzer display of a band-limited square wave whose fundamental frequency is 187.5 hz. The crosshairs are centered on the fundamental. The frequency and level information are displayed in the upper right.



Kyma also includes Sounds that put oscilloscopes and spectrum analyzers at any point in the signal path, so that the signal at that point will be displayed in the virtual control surface. See the *OscilloscopeDisplay* and *SpectrumAnalyzerDisplay* Sounds in the **Analysis** category of the system prototypes.

Tools Menu

Operations under the **Tools** menu are used to start up high level tools, such as those for designing alternate tunings and for recording and analyzing sample files.

Tools placed into the **Tools** folder within the **Program** folder of the **Kyma** folder are available directly from the **Tools** menu. Additionally, you can use a Tool by choosing **Open...** from the **File** menu, choosing **Use tool** as the file type, and selecting the Tool you want to use.

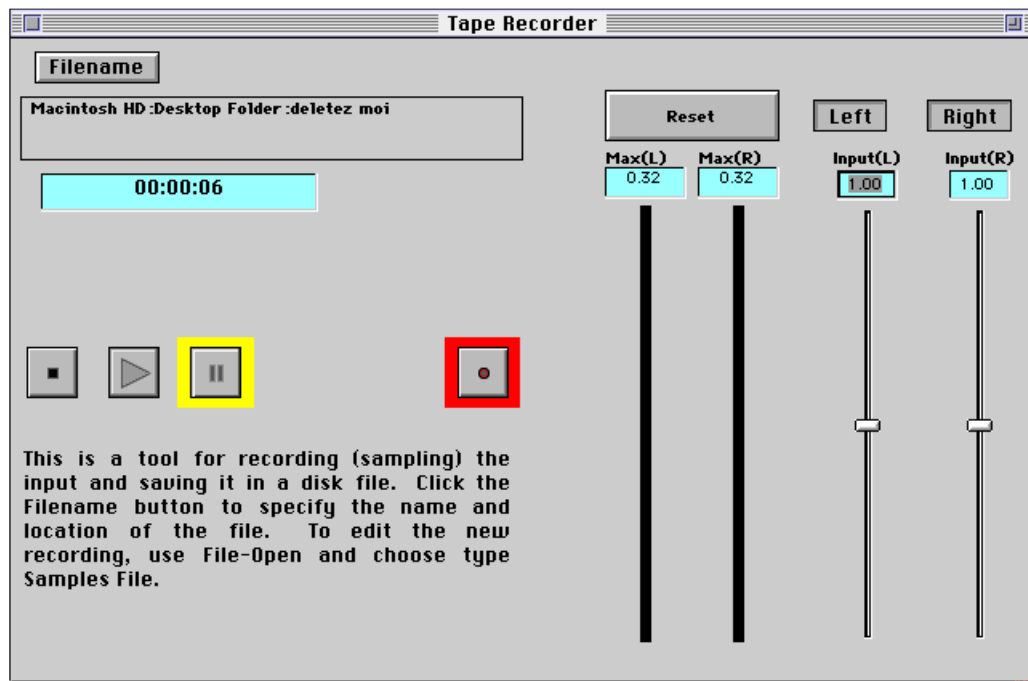
This section describes the Tools that come with Kyma Release 4.5; additional tools are available from third-party developers and from the Symbolic Sound *FTP* site.

Tools menu: Tape Recorder

The tape recorder tool provides a way to record the audio signal presented to the signal processor's analog or digital inputs. To record the output of a Kyma Sound, see *Action menu: Record to disk...* on page 437.

This tool mimics the operation of a tape recorder: the transport buttons (at the left center of the tool) control the recording and playback of audio, the faders control the level of the signal being recorded, the channel select buttons (above the level faders) select the input channels to record, and the metering section (to the left of the level faders) shows the current input level and the peak level since last reset.

The **Filename** button, at the upper left, is used to select the type of digital recording to make, including the file type (AIFF, WAV, *etc.*), the resolution (8, 16, or 24 bits), the number of channels in the file, and the name of the new file. The selected file name is shown in the box immediately below.



To record, click the record button (the rightmost transport control button). This automatically depresses the pause button. The recorder will begin monitoring the input signal. While monitoring, you can adjust the recording level and choose which input channels to record. When ready to record to the disk, click the pause button to un-pause the recording. During the recording, the time counter will display the approximate elapsed time of the recording. To pause the recording, click on the pause button again; to stop the recording, click on the stop button (the leftmost transport control button).

To play back your recording, click the play button. You can pause the playback at any time by pressing the pause button. Clicking the stop button will stop the recording. To edit the recording, choose **Open...** from the **File** menu, select **Sample file** as the file type, and locate the recording in using the file dialog.

Tools menu: Spectral Analysis

Use the Spectral Analysis Tool to create a spectrum file from a sample recorded at a 44.1 khz sample rate. The spectrum file, when used in conjunction with the *SumOfSines* Sound, can be used for morphing and other spectral manipulations. Spectrum files can be manipulated in a graphical editor, see *Spectrum Editor* on page 487 for more information.

The Spectral Analysis Tool can create two different kinds of spectrum files: straight spectrum files that contain the time-varying amplitude information at equally spaced frequencies (between 0 hz and one-half of the sample rate), and quasi-harmonic files that contain spectral information for the harmonic part of the sample only.

Both kinds of spectrum files can be used for time scaling, pitch shifting, and morphing,[§] however, you can get quite different results between the two kinds of spectrum files. Additionally, the results you get are very dependent on the source samples that you analyze: try to use recordings that have very little background noise and that are as free of reverberation as possible.

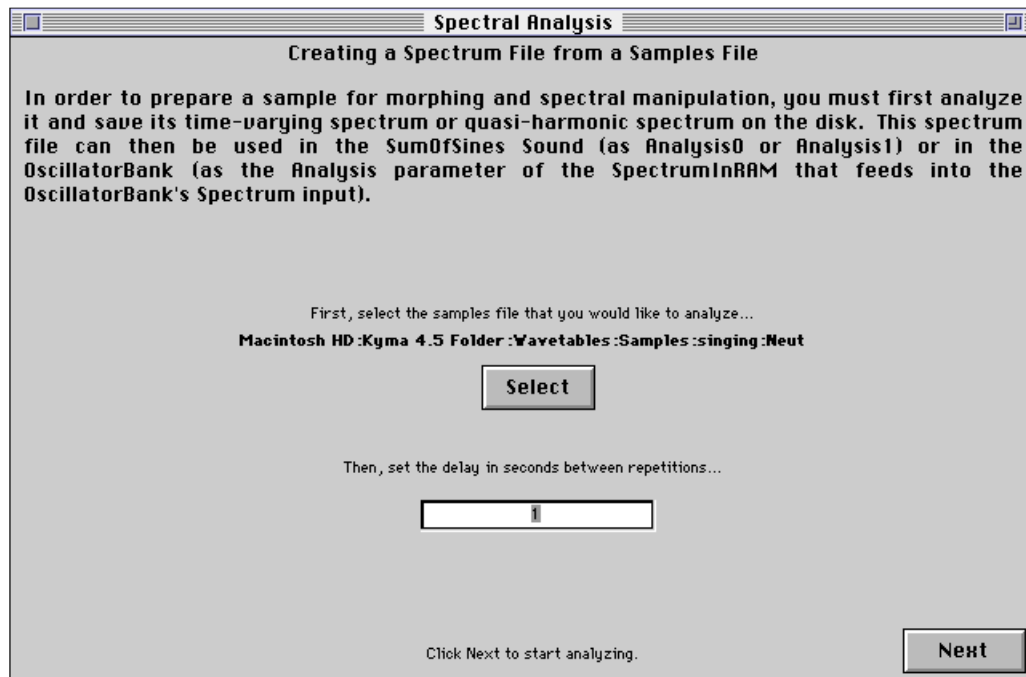
Quasi-harmonic files generally work best for morphing between nearly harmonic sources (such as voices or musical instruments). Because these files store only the harmonic part of the sample, non-harmonic or noisy sounds (such as church bells, car crashes, or jet fly-bys) and polyphonic or other overlapping sounds (such as a chorus, a conversation, or an outdoor ambiance) will work best as straight spectra.

Straight spectrum files generally have the fewest artifacts when used for time scaling and pitch shifting.

[§] You should always perform morphs between the same kinds of spectrum files: either both straight spectrum files or both quasi-harmonic files.

Using the Tool

Choose **Spectral Analysis** from the **Tools** menu to start analyzing a sample. The analysis has several steps to it, and the Tool guides you through them.



First, you must choose a sample file to be analyzed. Click the **Select** button and choose the sample from the file dialog.* The name of the file that you select will be displayed immediately above the button and you will hear the sample played back repeatedly, with a one second delay between repetitions.

In the text field below the button, you can change the number of seconds of silence between repetitions of the playback when auditioning the analysis. Click **Next** to proceed to the next step of the analysis procedure.

The next step is to specify the three analysis parameters that are common to both kinds of analysis files.

The first parameter setting is used to set the lowest frequency in the analysis. Choose one of the five settings based on the material you are analyzing. You should select the highest setting that is still below the fundamental frequency (or the lowest frequency) in the sample. For example, if you are analyzing a sample of singing and the pitch of the sample is middle C (4 c), then you should choose **Above 3F**.

The second setting concerns the tradeoff between time and frequency. Whenever a spectral analysis is performed, when you request high frequency resolution you get poor timing resolution and *vice versa*. These settings allow you to choose between favoring frequency resolution and favoring timing resolution in order: **BestFreq**, **BetterFreq**, **BetterTime**, and **BestTime**.

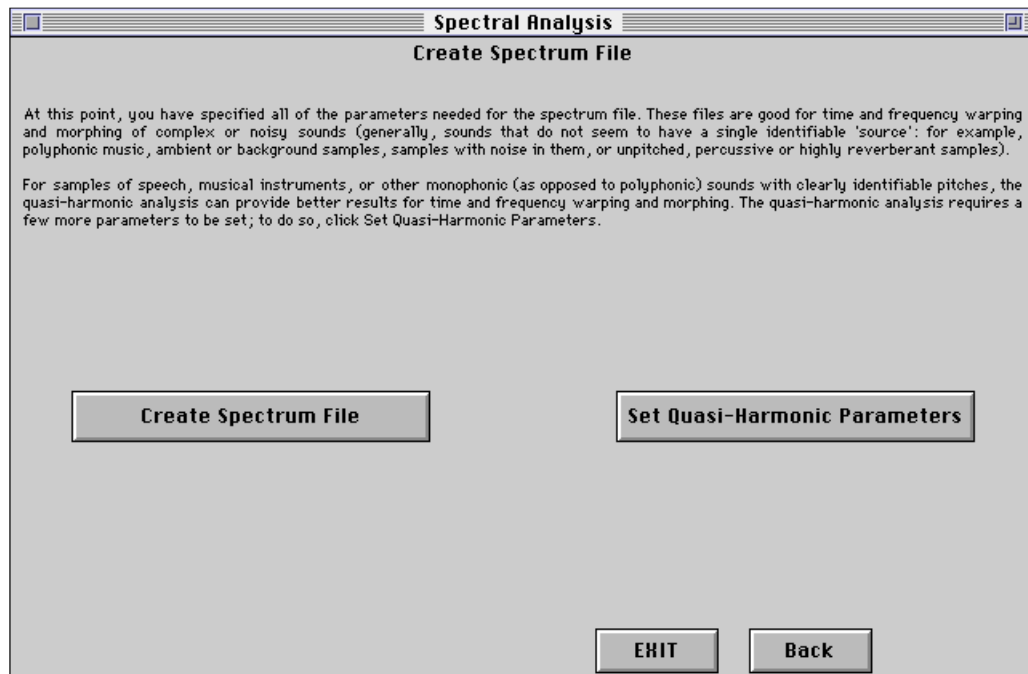
The level fader is used to control the overall amplitude level of the analysis.

* The Spectral Analysis Tool only works with monaural sample files recorded at a sample rate of 44.1 khz.



At any time during the setting of these analysis parameters, you can click the **Audition** button to hear a resynthesis based on an analysis with the current settings.^s Generally, you should try different combinations of these parameters and audition them, using your ear to judge which setting is best. Press **Next** when you are satisfied with the settings.

The next step requires a decision. If your goal is to create a straight spectrum file, click the **Create Spectrum File** button, and you are done. If you want to create a quasi-harmonic file, click the **Set Quasi-Harmonic Parameters** button.



^s The number of sine waves used in the resynthesis depends on the power of your signal processor, and is indicated immediately below the **Audition** button. Certain settings of the parameters may require more processing power than is available on your signal processor. Kyma will warn you when this occurs.

If you are unsure which type of file to create, here are some guidelines. Use a quasi-harmonic file for speech

monophonic (as opposed to polyphonic) musical sounds with clearly identifiable pitch

Use a straight spectrum file for

polyphonic musical sounds

ambient or background sounds

noisy or unpitched sounds

highly reverberant sounds

For straight spectrum files, this is as far as you have to go. The Tool will open up an untitled Sound file window with an example *SumOfSines* Sound set up to play the analysis file you just created.

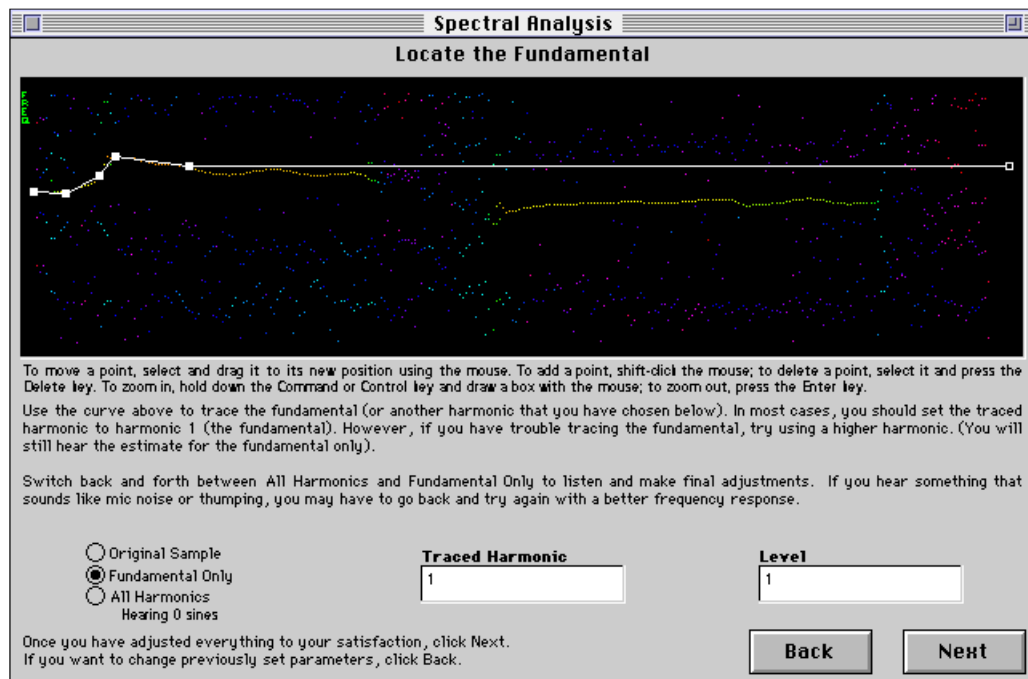
If you want to create a quasi-harmonic spectrum, you should click Set **Quasi-Harmonic Parameters** to continue.

There are two more steps involved in creating a quasi-harmonic analysis file. The first step is to identify the unpitched material in the sample.



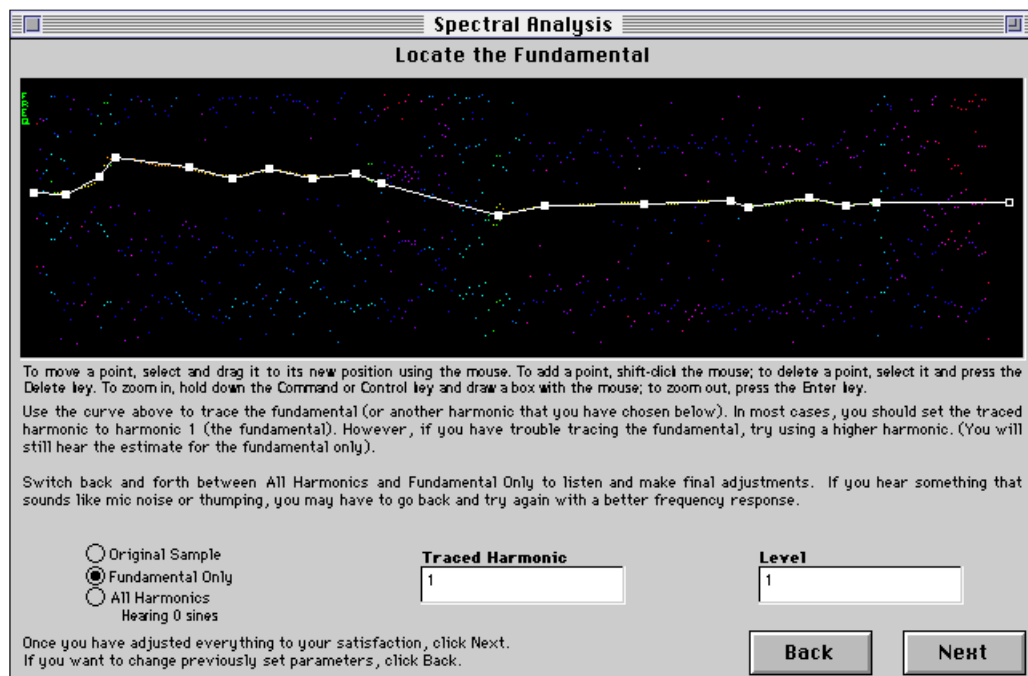
Adjust the fader slowly upwards while you listen to the resynthesis. As you move the fader up, more and more of the pitched material in the sample will be removed. You should adjust the fader until you hear only the unpitched part of the sample. These unpitched parts include the attack of a musical instrument note, the consonants (such as "t", "f", and "s") in speech or singing, or noisy parts of a sound effect sample. Adjust the fader such that you no longer hear any pitched sounds. If you hear some pitched segments of the signal breaking up, you should adjust the fader higher until the pitched segment disappears entirely. Press **Next** when you have finished setting the fader.

The last step is to trace the fundamental frequency envelope of the sample using a series of line segments. The Tool displays a tentative analysis of the sample. The fundamental frequency envelope is indicated by the white curve drawn in this display. Using the mouse, add points to this curve and adjust the locations of the points to closely follow the fundamental frequency of the sample.



To zoom in on an area, hold down the **Command** or **Control** key while clicking and dragging a box around the area to magnify. To return to a display of the entire spectrum press the **Return** or **Enter** key. To toggle between drawing tracks as unconnected frames or as frames connected by lines, press the **L** key (lowercase).

You should try to follow the fundamental with the white curve as closely as possible. You can add points by clicking the mouse while holding the **Shift** key down. To delete a point, select it using the mouse and then press the **Delete** key. You can move a point by clicking and dragging it.

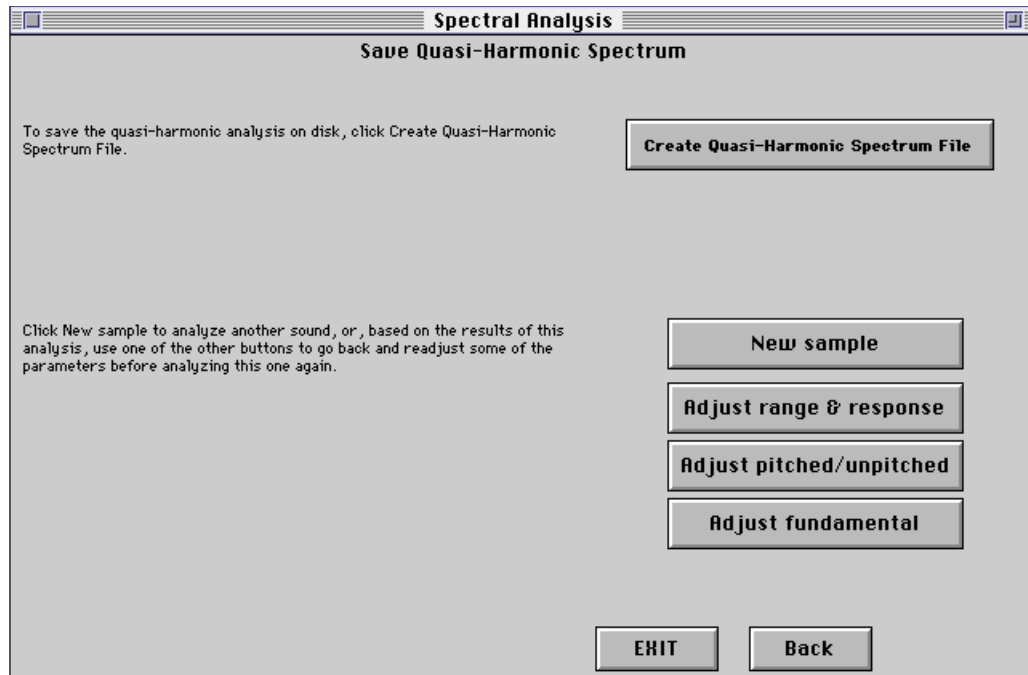


While you are making adjustments to the curve, you can use the radio buttons to choose between the original sample, a resynthesized fundamental, and a resynthesis of the sample using the number of partials shown.

If you hear clipping in the resynthesis, you can use the level field to reduce the amplitude of the analysis. If the fundamental frequency envelope of the sample being analyzed is difficult to follow, you can choose to trace different harmonic. Enter the harmonic number into the **Traced Harmonic** field, and adjust the white curve to track that harmonic.

When you have finished tracing the fundamental frequency curve, click **Next**.

At this point, you have provided all of the parameters needed to create a quasi-harmonic file. Click the **Create Quasi-Harmonic Spectrum File** button to do this. Kyma will prompt you for the name of the file, perform the analysis, and open an untitled Sound file window with a *SumOfSines* Sound set up to play the analysis just created.



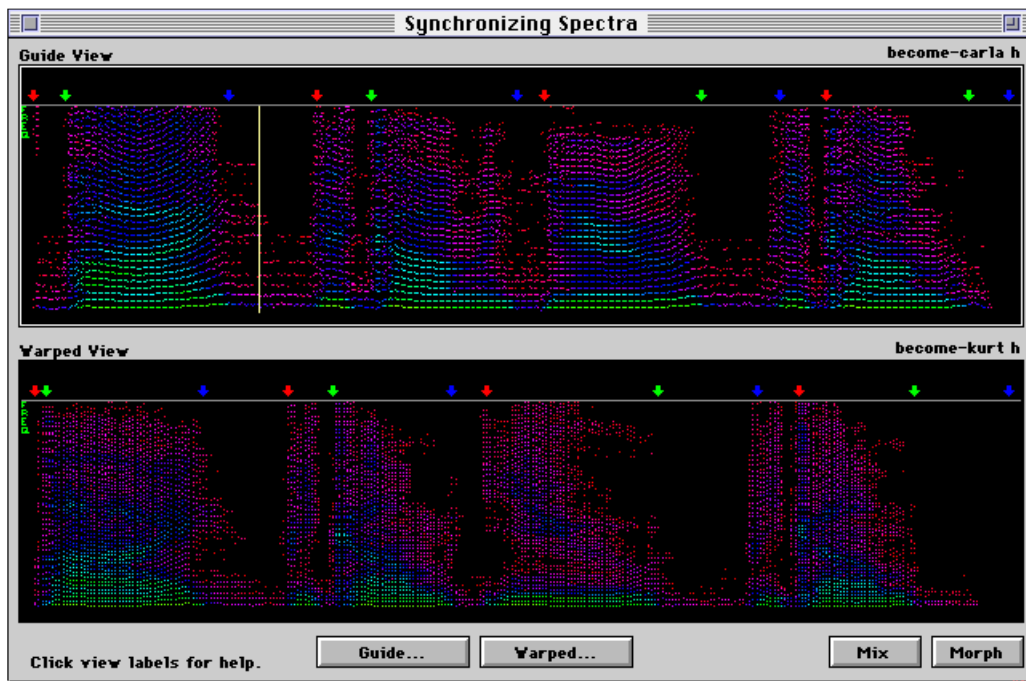
The other buttons on this page allow you to repeat any of the steps along the way to creating the analysis. **New sample** takes you to the first page, where you can specify a new sample to analyze. **Adjust range & response** takes you to the second page, where you can adjust the lowest analysis frequency parameter and the tradeoff between frequency and timing resolution. **Adjust pitched/unpitched** takes you to the fourth page, where you made an adjustment to leave only unpitched material audible. **Adjust fundamental** takes you to the penultimate page, where you traced the fundamental frequency envelope.

Tools menu: Synchronizing Spectra

It is important to tightly synchronize the spectra used in performing certain kinds of morphs. For example, morphing between the voices of two people speaking the same text has the fewest artifacts when the voices are very well synchronized in time.

The Synchronizing Spectra Tool provides a way to specify how two spectra should align in time. In this tool, you enter color coded markers at corresponding time points in two different spectra. The Tool can then produce an example of either a mix or a morph of the two spectra in which the corresponding points line up in time.

To open the Synchronizing Spectra Tool, choose **Synchronizing Spectra** from the **Tools** menu.



The two spectra to be synchronized are displayed in the **Guide View** and **Warped View** fields. Using markers, you specify the time points in the *warped* spectrum file that will be shifted to align with the corresponding points in the guide spectrum file. For example, you could mark the beginning and the ending of each feature in the guide spectrum and then mark the beginning and ending of those *same* features in the warped spectrum. “Features” would be things like the attack of an instrument tone, each phoneme in a sample of human speech, or each note in a musical passage.

Click the **Guide...** button and the **Warped...** button to choose these two spectra using file dialogs. If the files already contain markers, the markers will be displayed along with the spectrum.

Only one of the spectrum views can be active at a time — the one with the white border. To switch to the other spectrum view, you can either click the mouse in the view, or you can press the **Tab** key.

To zoom in on an area in the active spectrum view, hold down the **Command** or **Control** key while clicking and dragging a box around the area to magnify. To return to a display of the entire spectrum, click in the marker area to deselect the markers and press the **Return** or **Enter** key.

To audition the active spectrum file, press the **Space Bar**. Use either pitch bend on a MIDI controller or drag the yellow scrub bar to scrub through the spectrum.

To add a marker at the scrub bar position, press the **m** key. To let Kyma choose these markers automatically, press the **a** key. Then add or remove markers as necessary.

To make it easier to see the correspondence between markers in the two files, the markers are color coded in order from left to right: red, green, blue, red, green, *etc.* A red marker in the warped spectrum will be shifted ahead or behind in time to line up with the corresponding red marker in the guide spectrum.

To name a marker, select the marker with the mouse, and press **Enter**. To delete a marker, select it and press the **Delete** key. Press the **s** key to save the markers into the spectrum file in the active view.

Click the **Save markers** button to save the markers you have entered.

The **Mix** and **Morph** buttons construct example Sounds to perform a mix of the resynthesized spectra or a morph between the two spectrum files. In order for these examples to be generated, the two files must have the same number of markers.

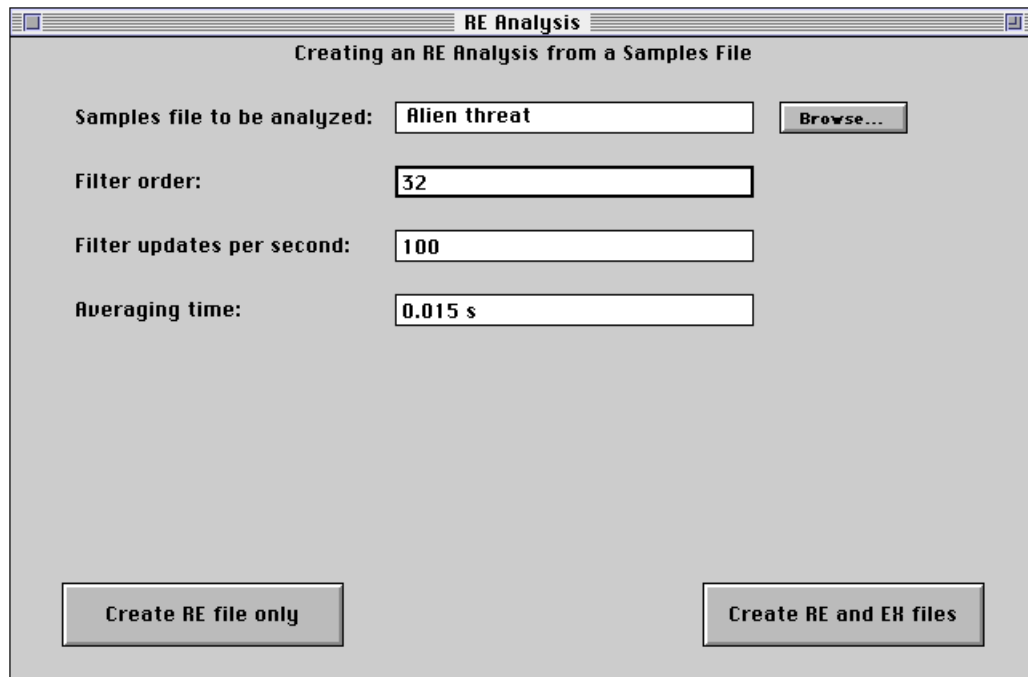
Tools menu: RE Analysis

The RE Analysis Tool is used to create files for resynthesis using the Resonator/Excitation synthesis method (RE synthesis) found in the *REResonators* Sound.

The Tool performs an analysis that breaks down the sample into a time-varying filter (the resonator) and an input to the filter (the excitation) that exactly recreates the sample. The description of the resonator is placed into an RE file, and the excitation is placed into an EX file.

The RE file can be used to control an *REResonators* filter operating on an entirely different input in order to perform cross synthesis. In this case, the *REResonators* will impose the formant (or filter characteristics) of the analyzed sample onto its input.

To use the RE Analysis Tool, choose **RE Analysis** from the **Tools** menu.



Click on the **Browse...** button to choose from a file dialog the sample to be analyzed. Kyma will play the sample that you selected.

The other parameters control the analysis. Try the default parameters first. Then you can try adjusting them:

Filter order is approximately twice the number of resonant peaks in the time-varying filter. The larger this number, the larger the number of formants or harmonics that can be tracked, but the longer it will take to complete the analysis.

Filter updates per second controls the number of individual filters that are created each second. For improved time resolution in the resynthesis, make this number larger.

Averaging time controls how much of the sample at any one time should be used by the analyzer to determine the filter. This number should be about twice the period of the input signal (in other words, twice the duration of a typical cycle of the sample). If you know the fundamental frequency of the sample, you can take the inverse of that and multiply by two to get an approximate averaging time.

Click **Create RE file only** to create just the file containing the description of the time-varying filter, or click **Create RE and EX files** to create both the filter description and the filter input files. In either case, after the analysis has completed, an example Sound will be placed into an untitled Sound file window for your use.

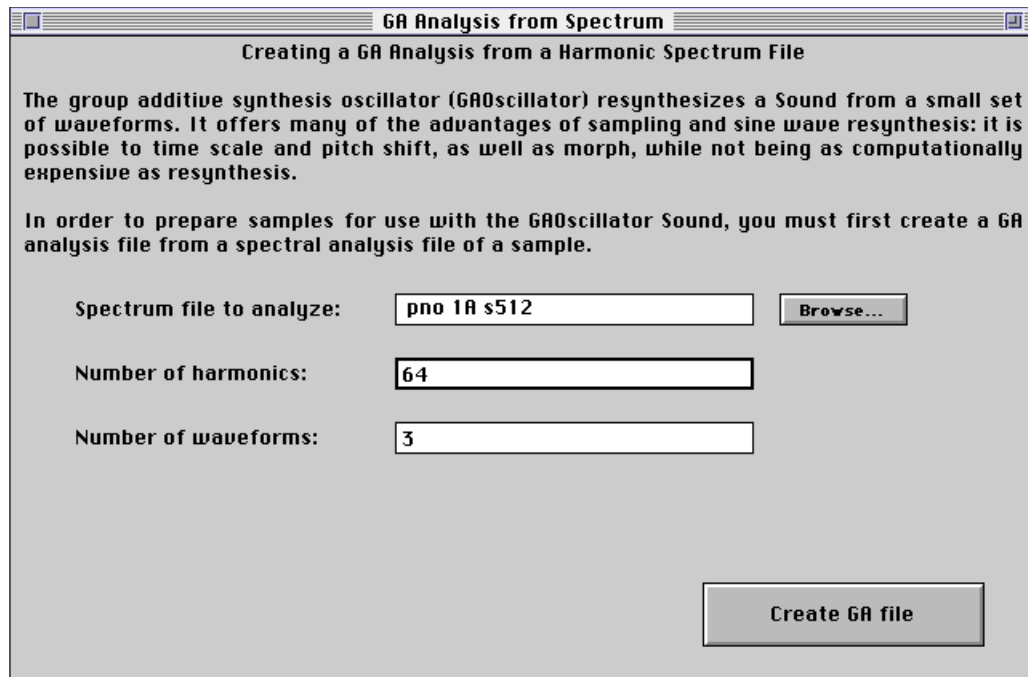


The resulting filter is *extremely* sensitive to overflow when not using the corresponding EX file as its input. You should use an *Attenuator* to attenuate the *REResonators Input* (sometimes by as much as 0.001).

Tools menu: GA Analysis from Spectrum

The group additive synthesis oscillator (called *GAOscillator* in Kyma) resynthesizes a sound from a small set of complex waveforms. It offers many of the advantages of both sampling and sine wave additive synthesis. Using GA synthesis you can independently scale time and frequency, and you can morph between two analyses. While not general enough to resynthesize speech or long musical passages, GA synthesis is more computationally efficient than full additive synthesis, and, for individual harmonic tones, can sound just as good.

Use the GA Analysis from Spectrum Tool (choose **GA Analysis from Spectrum** from the **Tools** menu) to create an analysis file for use with the *GAOscillator*.



Click on the **Browse...** button to choose the quasi-harmonic analysis from a file dialog. Kyma will play a resynthesis of the file that you selected.

The other parameters control the GA analysis algorithm. Try the other parameters first. Then you can try adjusting them:

Number of harmonics controls the number of harmonics to be included in the complex waveforms.

Number of waveforms controls the number of complex waveforms used in the group additive resynthesis.

When choosing the parameters for GA analysis, keep in mind that the larger the numbers, the longer the analysis will take.

Click **Create GA file** to create the file containing the complex waveforms and their corresponding amplitude envelopes. After the analysis has completed, an example Sound will be placed into an untitled Sound file window for your use.

Tools menu: Design Alternate Tunings

The Design Alternate Tunings Tool (choose **Design Alternate Tunings** from the **Tools** menu) lets you interactively design certain kinds of tuning systems. Throughout the use of this Tool, a MIDI keyboard can be used to try out the tuning in real time.[§]

[§] Thanks to Marcus Hobbs for motivating the work on this Tool. Marcus, Ervin Wilson, and Stephen Taylor are working on more advanced tuning systems and a special 500+ key keyboard for controlling Kyma. You can reach Marcus at: marcus@fa.disney.com.

The main page of the Tool contains a few global settings, as well as buttons to display pages for designing specific kinds of tunings.

Tonic Key Number and **Tonic Frequency** specify a MIDI key number and the corresponding frequency in hertz. These two parameters are used to specify a “fixed point” between all of the tunings; no matter which tuning is selected, the specified key number will always play at the specified frequency. The default value specifies middle C and its frequency in the conventional equal tempered tuning system.

In the middle of this page, you can specify the kind of sound to be used for auditioning the scale. Use the radio buttons to choose between a 21 harmonic sawtooth oscillator, a sine wave oscillator, or a sample. Click the **Choose...** button to choose the sample to be used from a file dialog. The **Volume** fader (active at all times through MIDI continuous controller 7) controls the volume of the sound being played.

The five buttons provide access to the five different ways to design tunings in this Tool:

Ratio Scale is used to design 12-note octave scales by specifying the ratio of each scale degree to the tonic.

Cents Scale is used to design 12-note octave scales by specifying the number of cents difference between each scale degree and the tonic.

Equal Tempered is used to design equal tempered scales with an arbitrary number of scale degrees per octave.

Two Interval is used to design scales with an arbitrary number of scale degrees per octave, where the scale is made up by raising an interval to successively larger powers. Also called a y^x scale.

From Text File is used to design completely arbitrary scales.

Design Alternate Tunings

Arbitrary Ratio Scales

This scale is generated from arbitrary ratios entered in the boxes below.

0	1	2	3	4	5	6	7	8	9	10	11
1	135	9	75	5	4	45	3	25	5	225	15
1	128	8	64	4	3	32	2	16	3	128	8

Results

Key	Freq	Ratio	Cents
60:	261.6 hz	1.0 : 1	0.0
61:	275.9 hz	1.055 : 1	92.2
62:	294.3 hz	1.125 : 1	203.9
63:	306.6 hz	1.172 : 1	274.6
64:	327.0 hz	1.25 : 1	386.3
65:	348.8 hz	1.333 : 1	498.0
66:	367.9 hz	1.406 : 1	590.2
67:	392.4 hz	1.5 : 1	702.0
68:	408.8 hz	1.562 : 1	772.6
69:	436.0 hz	1.667 : 1	884.4
70:	459.9 hz	1.758 : 1	976.5
71:	490.5 hz	1.875 : 1	1088.0

Create Example

Load

Save

Back

Equal

Just

Pythagorean

Clicking on **Ratio Scale** brings up this page. Across the top are twelve fields for the numerator and twelve fields for the denominator of the twelve ratios that define the scale. The buttons at the lower left are used to preset the ratios to three different example ratio scales.[†] To change a value, click the mouse in the field, and type in the new value. To cause the tuning to be updated with the new value, either press **Enter**, press **Tab** to go to the next field, or click in a different field.

The central view is common to all of the tuning pages in this Tool. It describes an octave starting on the tonic key number specified on the main page. It lists the MIDI key number, the absolute frequency of that key number in this scale, the ratio of the scale degree to the tonic, and the cents difference in pitch between the tonic and the scale degree.

The buttons at the lower right are also common to all of the tuning pages in this Tool:

Create Example opens an untitled Sound file window containing an example using the current tuning and currently selected instrument.

Load allows you to select a text file containing a tuning that you had saved earlier. The information in the file is then used to set the parameters in the current tuning method. The Tool either uses the information in the file directly (if the current tuning method is the same as the one used when saving the file), or it comes up with a set of parameters for this tuning method that approximates the tuning in the text file.

Save saves the parameters for the current tuning into a text file. Even though the Tool shows only one octave of the scale, the text file contains a listing of the frequencies in hertz of the 128 MIDI key numbers. You may find it useful to open this file with a text editor to view or edit that information.

Back takes you back to the main tuning page.

[†] Thanks to Brian Belet of the San Jose State University for providing the ratios of the “BB Sharp Chromatic (5-limit) Just Scale” used for the Just scale example.

Design Alternate Tunings

Arbitrary Scales in Cents

This scale is generated from arbitrary cent values (1/1200 of an octave) entered in the boxes below.

0	1	2	3	4	5	6	7	8	9	10	11
0.0	114.0	204.0	318.0	408.0	498.0	612.0	702.0	816.0	906.0	1020.0	1110.0

Results

Key	Freq	Ratio	Cents
60:	261.6 Hz	1.0 : 1	0.0
61:	279.4 Hz	1.068 : 1	114.0
62:	294.3 Hz	1.125 : 1	204.0
63:	314.4 Hz	1.202 : 1	318.0
64:	331.2 Hz	1.266 : 1	408.0
65:	348.8 Hz	1.333 : 1	498.0
66:	372.6 Hz	1.424 : 1	612.0
67:	392.4 Hz	1.5 : 1	702.0
68:	419.2 Hz	1.602 : 1	816.0
69:	441.5 Hz	1.688 : 1	906.0
70:	471.6 Hz	1.803 : 1	1020.0
71:	496.7 Hz	1.899 : 1	1110.0

Buttons: **Equal**, **Just**, **Pythagorean**, **Create Example**, **Load**, **Save**, **Back**

Clicking on **Cents Scale** brings up this page. Across the top are twelve fields for the pitch difference in cents between the tonic key number and each scale degree. The buttons at the lower left are used to preset the fields with a few different example scales. To change a value, click the mouse in the field, and type in the new value. To cause the tuning to be updated with the new value, either press **Enter**, press **Tab** to go to the next field, or click in a different field.

Design Alternate Tunings

Equal Tempered Scales

This scale is generated by dividing the octave into equal sized intervals.

Notes Per Octave

19

Results

Key	Freq	Ratio	Cents
60:	261.6 Hz	1.0 : 1	0.0
61:	271.3 Hz	1.037 : 1	63.2
62:	281.4 Hz	1.076 : 1	126.3
63:	291.9 Hz	1.116 : 1	189.5
64:	302.7 Hz	1.157 : 1	252.6
65:	314.0 Hz	1.2 : 1	315.8
66:	325.6 Hz	1.245 : 1	378.9
67:	337.7 Hz	1.291 : 1	442.1
68:	350.3 Hz	1.339 : 1	505.3
69:	363.3 Hz	1.389 : 1	568.4
70:	376.8 Hz	1.44 : 1	631.6
71:	390.8 Hz	1.494 : 1	694.7
72:	405.3 Hz	1.549 : 1	757.9
73:	420.4 Hz	1.607 : 1	821.1
74:	436.0 Hz	1.667 : 1	884.2
75:	452.2 Hz	1.728 : 1	947.4
76:	469.0 Hz	1.793 : 1	1010.0
77:	486.4 Hz	1.859 : 1	1074.0
78:	504.5 Hz	1.928 : 1	1137.0

Buttons: **Create Example**, **Load**, **Save**, **Back**

Clicking on **Equal Tempered** brings up this page. The only parameter for an octave-based equal tempered scale is the number of scale degrees per octave.

Design Alternate Tunings

Two Interval Scales

This scale is generated from powers of a "generating ratio" that have been octave reduced. This means that the scale will have two different intervals: the "generating ratio" and half the "generating ratio".

Notes Per Octave
12

Generating Interval
1.25

Results

Key	Freq	Ratio	Cents
60:	261.6 hz	1.0 : 1	0.0
61:	304.6 hz	1.164 : 1	263.1
62:	311.9 hz	1.192 : 1	304.2
63:	319.4 hz	1.221 : 1	345.3
64:	327.0 hz	1.25 : 1	386.3
65:	380.7 hz	1.455 : 1	649.5
66:	389.9 hz	1.49 : 1	690.5
67:	399.2 hz	1.526 : 1	731.6
68:	408.8 hz	1.562 : 1	772.6
69:	487.3 hz	1.863 : 1	1077.0
70:	499.0 hz	1.907 : 1	1118.0
71:	511.0 hz	1.953 : 1	1159.0

Create Example

Load

Save

Back

Clicking on **Two Interval** brings up this page.^s There are two parameters for this tuning: the number of scale degrees per octave, and the generating interval. This scale is constructed by raising the generating interval to successively higher powers. Whenever the value exceeds two, it is reduced by a factor of two so that it falls within the same octave as the other scale degrees.

Design Alternate Tunings

Read a Scale from a Text File

A completely arbitrary scale is read directly from a text file. The file should contain one line for each MIDI key number defined in the scale. Any undefined key numbers have a frequency of 0 hertz. Each line in file should specify the MIDI key number followed by the frequency in hertz for that key.

Unlike the other displays, the listing below indicates the ratios and cents between adjacent MIDI note numbers, rather than from some tonic key number.

Results

41:	87.21 hz	1.067 : 1	111.7
42:	91.98 hz	1.055 : 1	92.2
43:	98.11 hz	1.067 : 1	111.7
44:	102.2 hz	1.042 : 1	70.7
45:	109.0 hz	1.067 : 1	111.7
46:	115.0 hz	1.055 : 1	92.2
47:	122.6 hz	1.067 : 1	111.7
48:	130.8 hz	1.067 : 1	111.7
49:	138.0 hz	1.055 : 1	92.2
50:	147.2 hz	1.067 : 1	111.7
51:	153.3 hz	1.042 : 1	70.7
52:	163.5 hz	1.067 : 1	111.7
53:	174.4 hz	1.067 : 1	111.7
54:	184.0 hz	1.055 : 1	92.2
55:	196.2 hz	1.067 : 1	111.7
56:	204.4 hz	1.042 : 1	70.7
57:	218.0 hz	1.067 : 1	111.7
58:	229.9 hz	1.055 : 1	92.2
59:	245.3 hz	1.067 : 1	111.7
60:	261.6 hz	1.067 : 1	111.7
61:	275.9 hz	1.055 : 1	92.2

Create Example

Load

Back

Clicking on **From Text File** brings up this page. There are no parameters for this tuning; all pitches come directly from the file. Scales made with this part of the Tool can have completely arbitrary tunings.

^s The name **Two Interval** comes from the fact that each scale degree is generated by multiplying the previous scale degree by one of two intervals: either the generating interval or one-half of the generating interval. Thanks to tuning theorist Ervin Wilson for introducing us to this kind of scale.

This part of the tool reads lines from the file consisting of the MIDI key number and the corresponding frequency in hertz. The last line has a key number of 999. The easiest way to prepare a file for use here is to save a tuning file from any of the other methods, and edit it using a text editor.

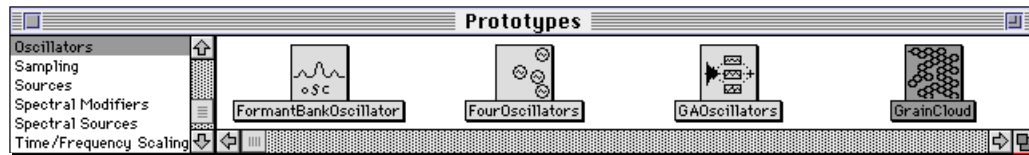
Here is an example of one such prepared file:

0 8.17581 hz
1 9.51787 hz
2 9.7463 hz
3 9.98021 hz
4 10.2198 hz
5 11.8974 hz
6 12.1829 hz
7 12.4753 hz
8 12.7747 hz
9 15.2287 hz
10 15.5941 hz
11 15.9683 hz
12 16.3516 hz
13 19.0357 hz
14 19.4926 hz
15 19.9604 hz
16 20.4395 hz
17 23.7947 hz
18 24.3659 hz
19 24.9506 hz
20 25.5494 hz
21 30.4574 hz
22 31.1883 hz
23 31.9367 hz
24 32.7033 hz
25 38.0715 hz
26 38.9852 hz
27 39.9209 hz
28 40.8791 hz
29 47.5894 hz
30 48.7318 hz
31 49.9012 hz
32 51.0988 hz
33 60.9147 hz
34 62.3765 hz
35 63.8734 hz
36 65.4065 hz
37 76.143 hz
38 77.9704 hz
39 79.8417 hz
40 81.7581 hz
41 95.1789 hz
42 97.4635 hz
43 99.8025 hz
44 102.198 hz
45 121.829 hz
46 124.753 hz
47 127.747 hz
48 130.813 hz
49 152.286 hz
50 155.941 hz
51 159.683 hz
52 163.516 hz
53 190.358 hz
54 194.927 hz
55 199.605 hz
56 204.395 hz
57 243.659 hz
58 249.506 hz
59 255.493 hz
60 261.626 hz
61 304.572 hz
62 311.882 hz
63 319.367 hz
64 327.032 hz
65 380.716 hz
66 389.854 hz
67 399.21 hz
68 408.791 hz
69 487.318 hz
70 499.012 hz
71 510.987 hz
72 523.252 hz
73 609.144 hz
74 623.763 hz
75 638.734 hz

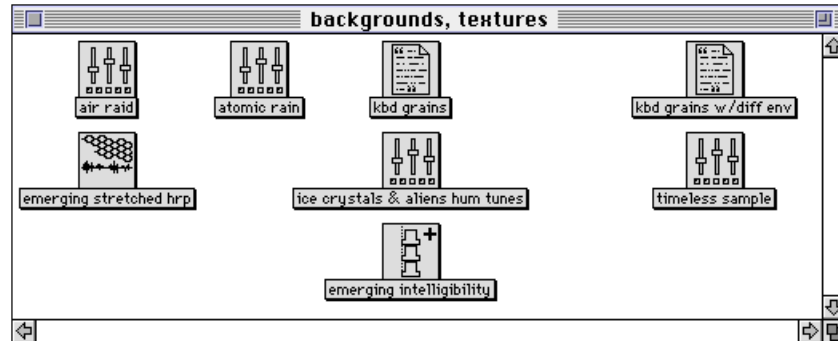
76 654.065 hz
77 761.431 hz
78 779.708 hz
79 798.42 hz
80 817.581 hz
81 974.635 hz
82 998.025 hz
83 1021.97 hz
84 1046.5 hz
85 1218.29 hz
86 1247.53 hz
87 1277.47 hz
88 1308.13 hz
89 1522.86 hz
90 1559.42 hz
91 1596.84 hz
92 1635.16 hz
93 1949.27 hz
94 1996.05 hz
95 2043.95 hz
96 2093.01 hz
97 2436.58 hz
98 2495.05 hz
99 2554.93 hz
100 2616.26 hz
101 3045.72 hz
102 3118.83 hz
103 3193.68 hz
104 3270.33 hz
105 3898.54 hz
106 3992.1 hz
107 4087.9 hz
108 4186.02 hz
109 4873.15 hz
110 4990.11 hz
111 5109.87 hz
112 5232.52 hz
113 6091.45 hz
114 6237.67 hz
115 6387.36 hz
116 6540.65 hz
117 7797.08 hz
118 7984.2 hz
119 8175.79 hz
120 8372.03 hz
121 9746.3 hz
122 9980.22 hz
123 10219.7 hz
124 10465.0 hz
125 12182.9 hz
126 12475.3 hz
127 12774.7 hz
999 END

System Prototypes and the Sound File Window

The *system prototypes window* contains an example of each kind of Sound included with the Kyma system. The system prototypes can be used to supply the components you need to build new sound designs.



A *Sound file window* allows you to examine and manipulate the Sounds contained in a Sound file. It is used as a workspace for creating new Sounds and editing old Sounds.



Prototype Strips

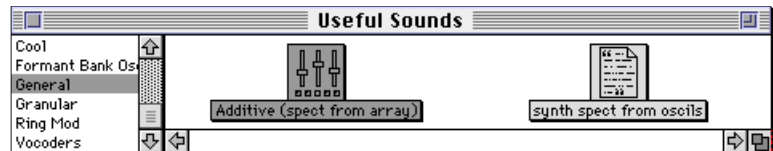
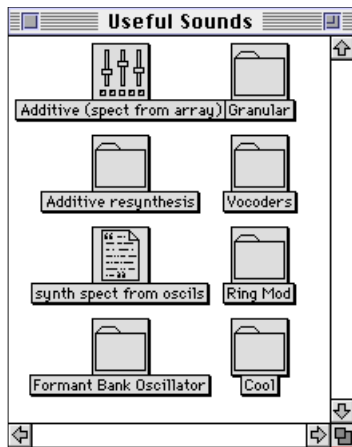
A *prototype strip* (or prototypes window) is a collection of Sounds that you can use as templates for new Sounds. There are two types of prototype strips in Kyma: the system prototypes and custom prototype strips.

The system prototypes contains an example of each kind of Sound included with the Kyma system. The system prototypes are organized into categories. The categories are identified in the scrolling list at the left of the system prototype strip; the Sounds in the selected category are displayed in the horizontal scrolling strip. See *Prototypes Reference* beginning on page 218 for descriptions of each system prototype Sound. To open the system prototypes, choose **System prototypes** from the **File** menu.



By exploring the system prototypes using **Describe Sound** from the **Info** menu and **Compile, load, start** from the **Action** menu, you can learn about the different prototypes provided in Kyma. Each prototype serves as an example of a Kyma Sound, so experimenting with the prototypes is one way to learn about the Kyma System.

As you start to design your own Sounds, you will probably want to use some of them as templates. To use one of your customized Sounds as a template, open the Sound file that contains the template as a custom prototype strip. To open a Sound file as a custom prototype strip, choose **Open...** from the **File** menu, select **Custom prototypes** as the file type, and select the Sound file to be opened. Any Sound collections in the Sound file will show up as categories in the custom prototype strip; other Sounds will show up under the **General** category in the custom prototype strip.

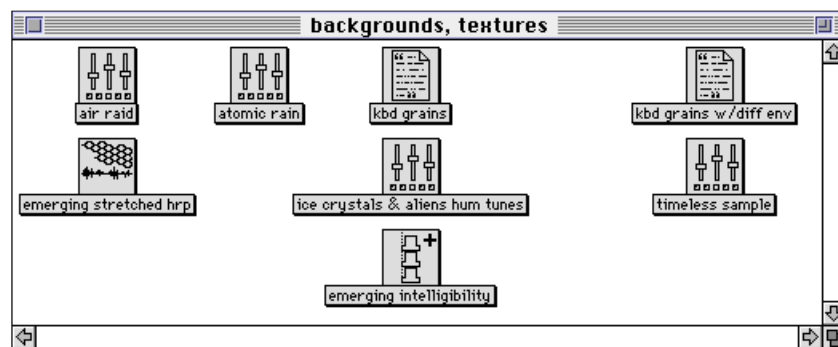


Sound File Windows

While you are in Kyma, a Sound file window allows you to examine and manipulate the Sounds contained in a Sound file. It serves as a workspace for creating new Sounds and editing old Sounds.

When you leave Kyma, you have the option of saving your work in the Sound file on the disk. Sounds stored in a Sound file are not sample files or digital recordings but instructions for producing sound on the digital signal processor (DSP). This representation is much more compact than a digital recording.

A Sound file can contain individual Sounds and/or Sound collections. A Sound collection is itself a collection of individual Sounds and/or other Sound collections. A Sound collection is a convenient way to categorize or organize the objects in a Sound file.



The operations in the **Action**, **Info** and **Edit** menus (see pages 436, 439, and 425) affect the selected Sound(s). Click a Sound to select it. Press **Enter** to change the name of a selected Sound. Use standard click and drag techniques to select and move Sounds in the Sound file window or between windows.

Clicking and dragging the mouse in the background of the Sound file window lets you draw a box around many Sounds at once, selecting all Sounds within the box. If you hold down the shift key while performing any of these mouse operations, you will add to or remove from the current selection, rather than starting a new selection.

To move a Sound, hold down the **Control** or **Option** key as you click and drag. Normally, you will not want to move a Sound; it is safer to move a copy. To move a copy of a Sound, click on the Sound (without pressing the **Option** or **Control** keys) and drag the Sound's icon.

Creating and Editing Sounds

In Kyma, you do not start from scratch when designing a new Sound. Instead, you use an existing Sound as a template for your new Sound, substituting new components for the old. Any existing Sound can function as a template for a new Sound.

To create a new Sound, first choose an existing Sound to serve as a template. Drag the template's icon to the Sound file window from either a prototype strip or another Sound file window. In Kyma, when you drag a Sound's icon from one window to another, you are actually dragging a duplicate of that Sound. The original Sound remains unchanged, ready to be reused.

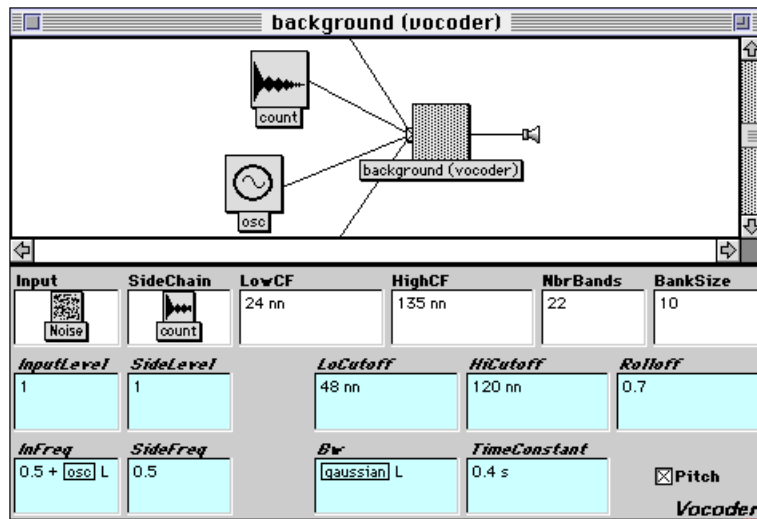


Whenever you drag a Sound from a prototype strip, you are really creating and dragging a duplicate of the Sound. Kyma automatically appends a number to the name of the Sound that you drag from the prototype strip to indicate that you have a copy of the original. The number Kyma appends to the Sound's name corresponds to the number of times you have dragged that Sound from a prototype strip.

Once the Sound you have chosen as a template is in a Sound file window, double-click on the Sound to begin editing it. A Sound editor window will open, allowing you to make changes to the Sound's parameters. When you close the Sound editor window after making changes, a dialog will allow you to choose between saving or discarding your changes to the Sound.

Sound Editor Window

In Kyma you construct new Sounds by combining and modifying other Sounds in a Sound editor window. To open a Sound editor, go to the Sound file window and double-click on the Sound you would like to edit. A Sound editor window will appear. Only one editor at a time can be open on any particular Sound, but any number of editors can be open at once. Opening or double-clicking a Sound that is already being edited will bring its Sound editor to the front.



The Sound editor is divided into two parts; the top section shows the signal flow diagram, and the lower section shows the parameter settings.



Select a Sound in the signal flow diagram by clicking on it once; the Sound will be highlighted in gray to indicate that it has been selected. Operations in the **Action** and **Info** menus apply to the Sound that is *selected* in the signal flow diagram.

You can edit the parameters of a Sound in the signal flow diagram by double-clicking on it. The parameter area of the Sound editor will change to the parameters of the double-clicked Sound and the double-clicked Sound's icon will be replaced with a gray pattern.



Operations in the **Edit** menu apply to the *active* field in the Sound editor. The active field is the field with a border drawn just inside its editing area. You can click inside a field to make it active, or press **Tab** to cycle through all of the editable fields.

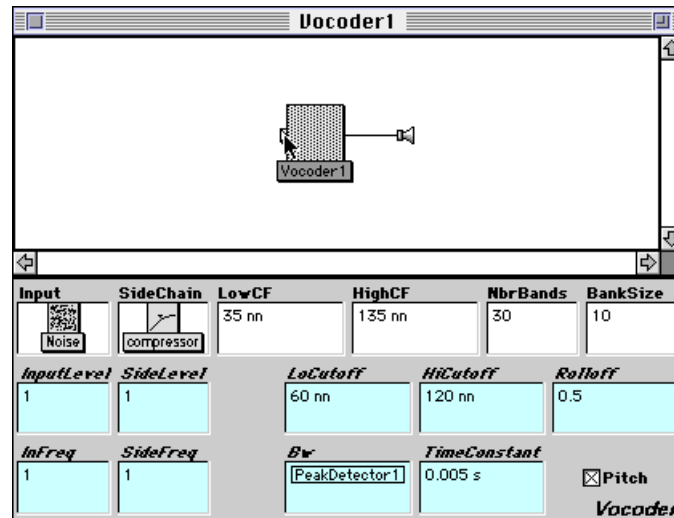
Signal Flow Diagram

The top half of the Sound editor window is the signal flow diagram. The signal flows from left to right. The rightmost Sound is the "output" Sound, the one you double-clicked in the Sound file window to open the Sound editor. The output Sound's immediate inputs appear to its left.

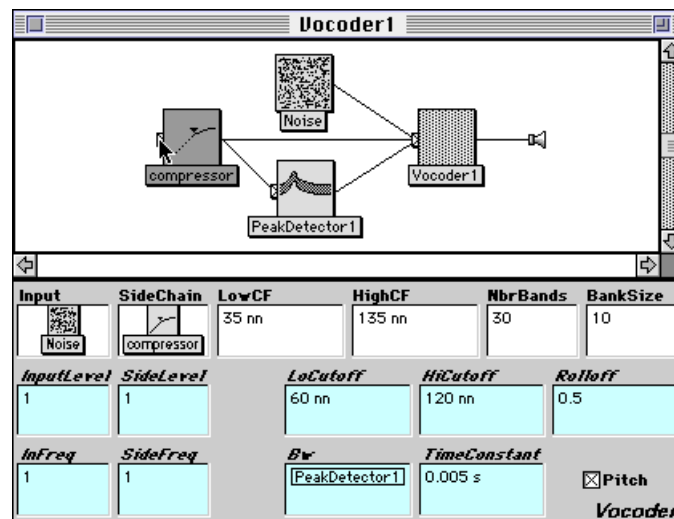
Viewing Inputs

In the Sound editor, an arrow tab on the left side of a Sound's icon indicates that the Sound has inputs. The arrow tab acts as a toggle — click once to see the input(s) of a Sound, and click again to hide the inputs. Clicking on the arrow tab while holding down the **Control** or **Command** key displays the entire signal flow path leading into that Sound.

Clicking the arrow tab on *Vocoder1*, shown here:

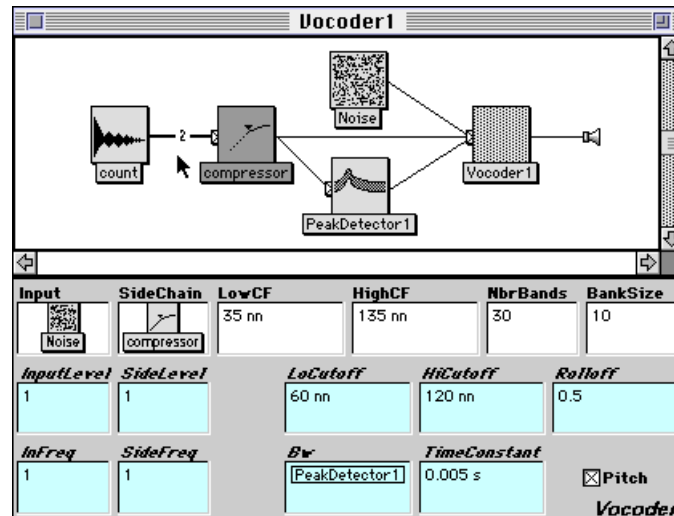


reveals that it has several inputs:



Several Sounds can share an input. In this case, there is a connection drawn from each Sound to the shared input. In the example above, *compressor* is an input to both *Vocoder1* and *PeakDetector1*.

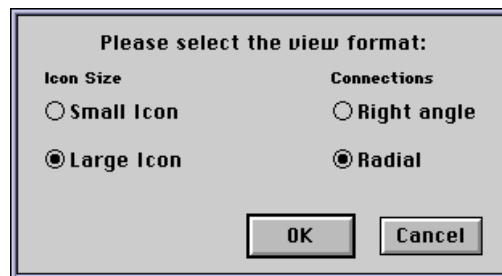
Clicking on the arrow tab of *compressor* reveals its inputs:



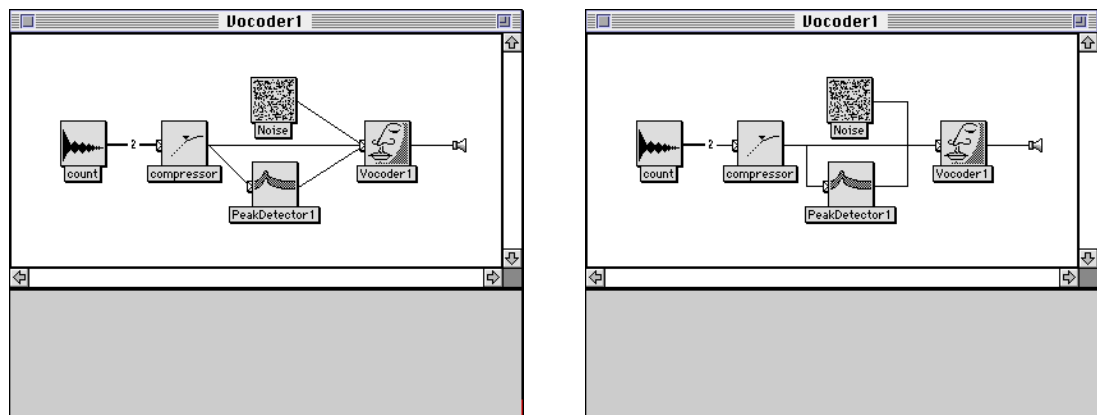
If a Sound and its input are connected with a thick line labeled with a number, that means the input is used more than once within that Sound. The number of times the input is used is indicated by the label displayed on the connection. In the example above, *count* is used twice as an input to *compressor*.

Signal Flow Diagram Layout

You can change the appearance of the signal flow diagram by selecting **View...** from the **Edit** menu and selecting the desired layout attributes.



Sounds can be displayed as large or small icons. Small icons take up one quarter the area of large icons. The radial and right angle attributes refer to the contour of the connections between the Sounds. Radial edges fan out radially from the left edge of a Sound to points at the right edges of its inputs. Right angle connections use only straight lines and right angles.



You can adjust the layout of the signal flow diagram by dragging the Sound icons around with the mouse. Dragging an icon moves its position relative to the rest of the signal flow diagram; dragging an icon while holding down the **Shift** key moves the Sound and all of its inputs simultaneously.

When you move a Sound to a new position within the same signal flow diagram, you are producing only a cosmetic change not a change to the Sound. **Clean up** (in the **Edit** menu) restores the signal flow diagram to its original layout.

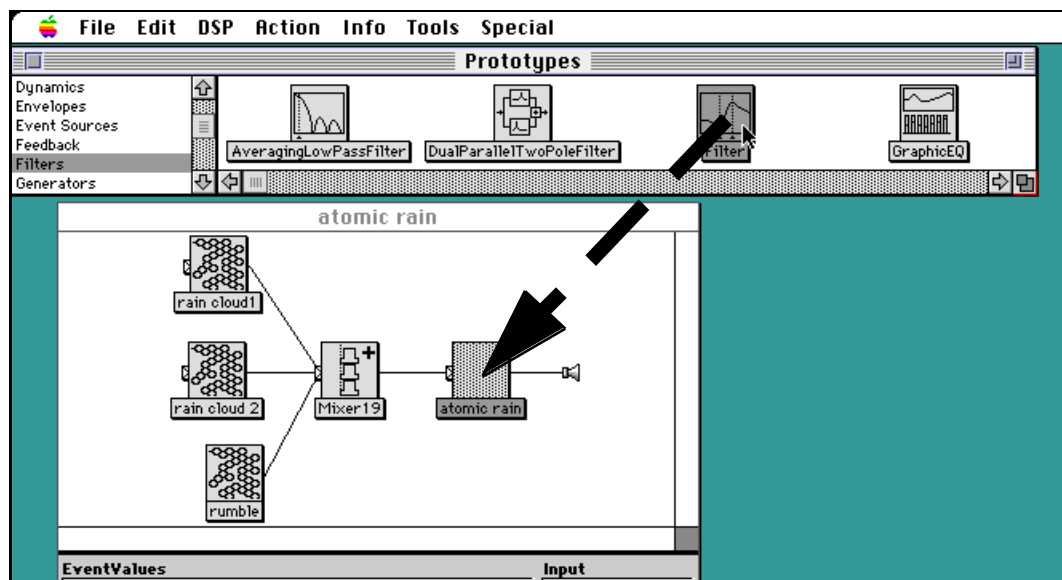
Editing the Signal Flow Diagram

You can edit the signal flow diagram by replacing Sounds in the diagram with other Sounds and by inserting new Sounds into the diagram. You can replace a Sound in the signal flow diagram by dragging another Sound over its icon. To insert a Sound between two Sounds in the signal flow, drag and drop the Sound onto a line in the signal flow diagram. To remove a Sound that is between two other Sounds, select the Sound and press **Delete** or **Backspace**. To change the name of the selected Sound, press **Enter**, then type the new name in the dialog box.

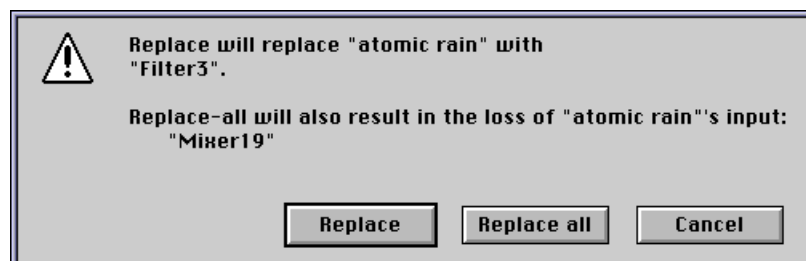
Replacing Sounds in the Signal Flow Diagram

To replace a Sound in the signal flow diagram:

1. select the new Sound with the mouse
2. while holding the mouse button down, drag the icon on top of the Sound that you want to replace
3. release the mouse button



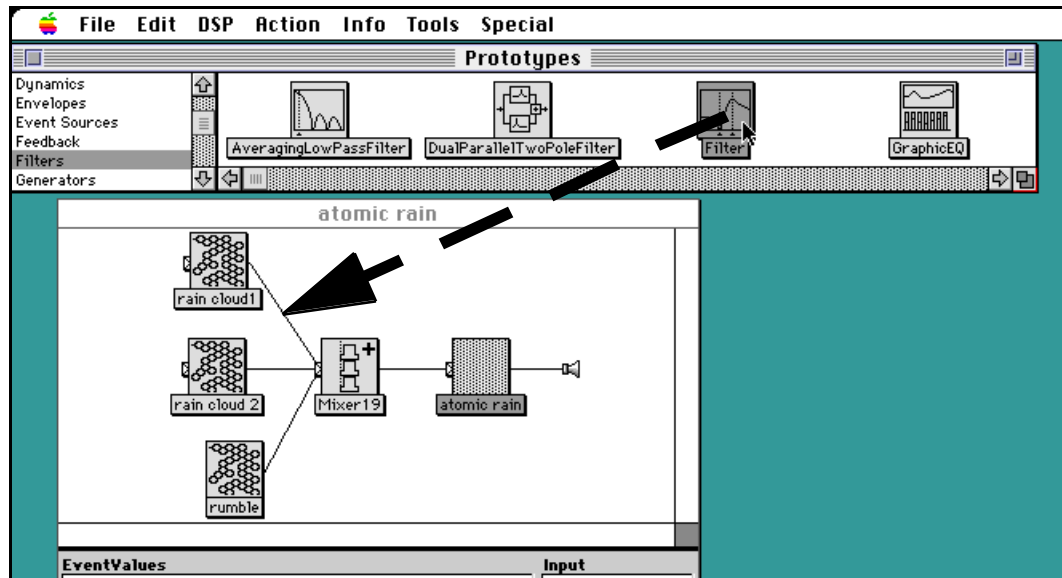
You will be prompted to confirm the replacement. If the Sound has inputs, you will be given a choice between replacing the Sound and keeping any inputs with the same name (**Replace**), or to replace the Sound *and* its inputs (**Replace all**).



Inserting Sounds to the Signal Flow Diagram

To insert a Sound between two others in the signal flow diagram:

1. select the new Sound with the mouse
2. while holding the mouse button down, drag the icon on top of the line connecting the two Sounds
3. release the mouse button



Only Sounds that have input fields can be inserted into the signal flow diagram in this way. If the new Sound has several inputs, you will be asked to choose which input the existing Sound should be placed into.[‡]

Removing Sounds from the Signal Flow Diagram

To remove a Sound from the signal flow diagram, select the Sound and press the Delete or Backspace key. Only Sounds with visible inputs can be removed in this way.

Internally, Kyma implements this by replacing the Sound to be deleted with one of its inputs, so you will be asked to confirm the replacement and to decide which input should be used as the replacement.



Renaming Sounds in the Signal Flow Diagram

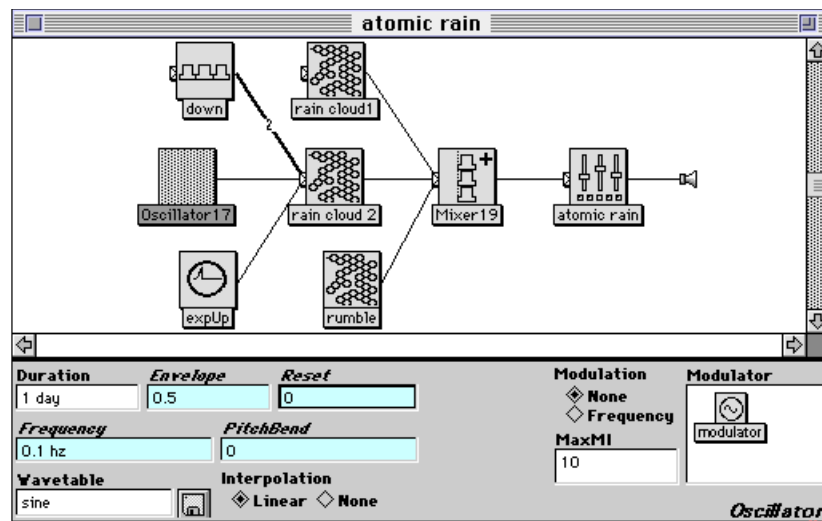
To rename a Sound in the signal flow diagram, select the Sound and press **Enter**. You will be asked for the new name.

Parameter Settings

In addition to editing a Sound by changing its signal flow diagram, the Sound editor window also allows you to edit the Sound's parameters.

[‡] To change the number of inputs to a Sound that takes multiple input Sounds, edit the **Inputs** field of that Sound; you can drag additional Sounds in the **Inputs** field or delete Sounds that are already in the **Inputs** field.

Double-click a Sound in the signal flow diagram to edit its parameters. While a Sound's parameter fields are being edited, the Sound's icon is gray, and its parameters are shown in the lower section of the Sound editor window. You can edit the parameters of only one Sound at a time. To close the parameters of a Sound, double-click on either an empty area of the signal flow diagram or double-click on another Sound in the signal flow diagram.



Each parameter field has the parameter name above it. To see a description of the parameter, click on the parameter name. To see a complete description of the Sound, click on the Sound class name in the lower right of the window (*Oscillator* in the image above). To expand the active parameter field to the full size of the screen, choose **Large window...** from the **Edit** menu. Alternatively, to make more space for the parameter fields, you can either resize the window (by clicking and dragging the mouse in the lower right corner of the window), or you can click and drag the thick line that divides the upper and lower halves of the Sound editor window.



Operations in the **Edit** menu apply to the *active* field in the Sound editor. The active field is the field with a border drawn just inside its editing area (**Reset** in the image above). You can click inside a field to make it active, or press **Tab** to cycle through all of the editable fields.

In parameter fields that contain text, all of the standard text editing operations are available, including the keyboard arrow keys (, , ,) for positioning the cursor and the operations listed in the **Edit** menu. To select text that is enclosed within parentheses, brackets, or quotes, place the blinking insertion point just within the enclosing punctuation, and double-click the mouse.



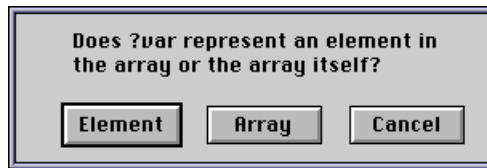
To set a parameter field to the same value it has in another Sound, drag the other Sound into the parameter field. (If the dragged Sound doesn't have a parameter of that name, the Sound is rejected and the parameter field is unchanged.)

Variables as Parameter Values

Variables are names preceded by question marks; they are green on color monitors and bold on other monitors. A variable is a kind of place holder in a parameter field of a Sound. The variable must be set to a real value before the Sound can be compiled; if you try to load a Sound that still has some free variables in it, a series of dialog boxes will prompt you to enter a value for each variable. Once the variable has been assigned a value, that variable takes on that value for the full duration of the Sound.

A variable or an expression containing variables can appear in any parameter field. For more information about variables, see *Variables* on page 510.

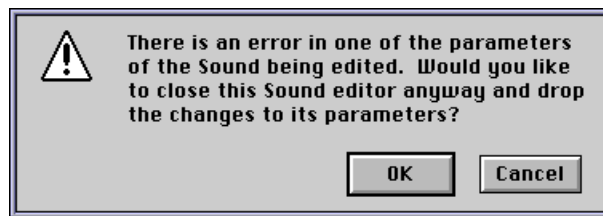
If you set an array parameter to a variable, Kyma will ask whether you want an array containing that one variable or whether you want the variable to represent the entire array:



Evaluating the Parameters

When you change a parameter of a Sound, Kyma will test the new value to make sure that it is a legal value. This test will not occur until you double click a different Sound, or until you select one of the operations in the **Action** or **Info** menus. If a parameter value is illegal, the field will flash and the computer will beep twice. The most common problems with parameters are leaving the units off frequency or duration parameters, specifying a negative or zero value where it does not make sense, and introducing a syntax error (such as an extra space or a hidden character).

As long as there is an illegal parameter value, you cannot edit the parameters of another Sound. If you click in the close box of the Sound editor while there is a problem in a parameter field, Kyma will ask you if you would like to drop the changes made to the Sound being edited.



If there is trouble with a value entered into a parameter, you can either fix the parameter or use **Revert** from the **Action** menu. **Revert** drops any changes that have been made to the parameters since the last time the Sound was double-clicked.

Working with the Parameter Fields

There are several different types of parameter fields: file name, disk file segment, envelope breakpoint, radio button, check box, input, text, script, value, array, and others.[§] You are probably familiar with how to use many of these fields, so only the unfamiliar field types will be explained here.

A *file name field* consists of a text field together with a disk button:

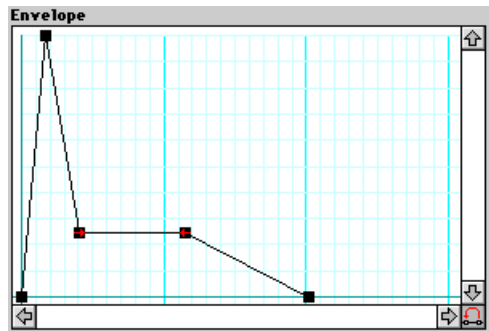


You can either type in a file name (with or without the path information, see *How Kyma locates files* on page 431), or click the disk button to choose the file from a file list. Hold down **Command** or **Control** while clicking the disk button to open an editor on the file.

A *disk file segment field* is used in the *DiskSplicer* Sound, see *DiskSplicer* on page 495 for more information.

[§] A complete list of parameter field types can be found in *Parameter Types and Ranges* on page 540.

An *envelope breakpoint field* is used in the *GraphicalEnvelope* Sound. Time is displayed horizontally, and the envelope value (between 0 and 1) is displayed vertically. The heavy lines indicate one second boundaries, the thin lines are one-tenth of a second apart. The duration of each segment is scaled by the inverse of the value in the **Rate** field. For example, if **Rate** is 0.5, then the heavy lines are two seconds apart. Use **Zoom in** and **Zoom out** from the **Edit** menu to change the magnification of the display.

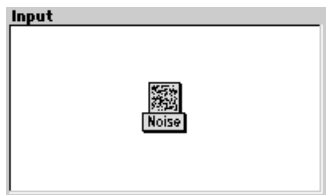


To select a breakpoint, click the mouse on a square marker; the selected breakpoint is always shown as a hollow square, and its location is shown in the upper right. To add a breakpoint, click the mouse at the new breakpoint location while holding down the **Shift** key. To remove a breakpoint, select the breakpoint and press the **Delete** or **Backspace** key. To move a breakpoint, click on the breakpoint and drag it to its new location; the coordinates of the breakpoint will be displayed as you move it around.

Breakpoints with arrows within them indicate loop points for the envelope. To add or remove a loop point, select a breakpoint and click on the loop button at the lower right of the field. The behavior of the envelope when there are right and left loop points is as follows

Right	Left	Action
no	no	no looping
no	yes	illegal
yes	no	on trigger off, jumps to right-arrow breakpoint
yes	yes	if trigger on, jumps to right-arrow breakpoint upon reaching left-arrow breakpoint; on trigger off, jumps to left-arrow breakpoint

An *input field* accepts Sounds that are to be used as signal input or inputs to the edited Sound. Some input fields will accept an arbitrary number of Sounds. Other Sound fields place some restrictions on the number or kinds of Sounds they will accept.



You can drag Sounds into and out of input fields, as well as use any of the operations in the **Edit** menu (when the input field is active, that is, when it has the black border drawn within it). Selecting one or more Sounds and pressing **Delete** or **Backspace** is a quick way to remove Sounds from an input field.

In Sounds where the ordering of inputs is important (*Concatenation*, for example), the inputs are arranged within the parameter field from left to right and from top to bottom. To change the order of the inputs, change their positions in the parameter field.



Sounds in input fields (as well as Sounds used as control signals in other parameter fields) also show up in the signal flow diagram. Remember to double click a different Sound (or double click in the background of the signal flow diagram) to update the signal flow diagram after making changes to an input parameter field.

You can create shared inputs between Sounds in several ways using an input field:

Use **Copy** from the **Edit** menu to place the input Sound into the clipboard. Then repeatedly use **Paste special...** from the **Edit** menu to place the input Sound into each of the parameter fields that need that input Sound.

Hold the **Control** or **Option** key while dragging the input Sound from the parameter field where the Sound is used on top of the Sound you want to replace in the signal flow diagram.

Hold the **Control** or **Option** key while dragging the input Sound from the signal flow diagram into the parameter field where you want to share that Sound.

Script fields are text fields that contain a script for Kyma to evaluate when it is compiling a Sound. For more information about scripts, see *Scripting* on page 522.

Value fields are text fields that contain a numeric value. The numeric value may be given directly, or it may be the result of a Smalltalk-80 calculation. Certain value fields (with cyan colored backgrounds) are *hot*; hot value fields can be set to a time-varying numeric value. Some value fields require units to be supplied along with the numeric value. You can add explanatory comments to value fields by enclosing the comments within double quotes.

Array fields are value fields that contain a list of numeric values. The numeric values are simply listed one after another, separated by spaces. If the value is complex or the result of a calculation, it must be enclosed in curly braces so that Kyma can find where one value ends and the next begins. For example,

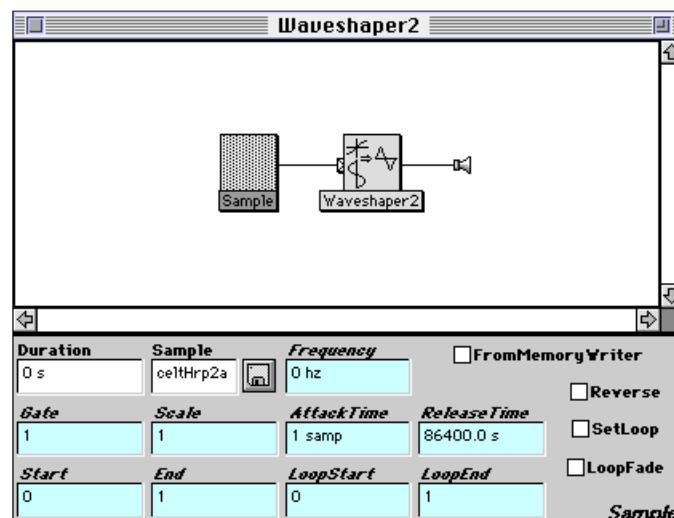
```
1 !Fader1 {2 + 3} {!Frequency * 2}
```

could be used in an array parameter to specify four different values.

More about Value Fields

A value field is a text field that contains a numeric value. These fields can contain anything from a simple constant value to a complete Smalltalk program. The value of the field is calculated when the Sound that contains the field is compiled.

There are two types of value parameters: hot and cold. *Cold* parameters are fixed values, set when you first load the Sound and never altered again. Cold parameter values can be constants, variables, or Smalltalk expressions that evaluate to constant values. *Hot* parameters can be continuously updated even after the Sound has been loaded into the signal processor. Using hot parameters, you can control Kyma Sound parameters in real time. Hot parameter fields can be identified by their light cyan background color and italicized parameter names. Hot parameter fields can take all the same kinds of values that cold parameters can; in addition, they can be controlled by Event Values or the output of other Kyma Sounds. See *Event Values*, *Virtual Control Surface*, and *Global Map* on page 472 for more information.



Duration and Sample are cold parameters; Frequency, Gate, etc. are hot parameters.

Specifying Numbers

You specify numbers in Kyma much as you would in any other application. The particulars of the syntax are as follows:

Whenever you enter a number between 1 and -1 that has a decimal point, you should include the leading zero. For example, you should use 0.1, or -0.005 not .1 or .005.

Floating point numbers can be entered as a mantissa and an exponent. For example, 44.1E3 = 44100, or 1.0E-6 = 0.000001. This notation can also be used to specify large integers. For example, 2E32 or 1e6.

In general, an integer can be expressed in any radix (or base) by preceding it with the radix and the letter r. For example, hexadecimal (i.e., base 16) numbers are preceded by 16r. Similarly, to specify an integer as a binary number, precede it with 2r as in 2r1011. Numerals greater than nine are represented by the letters A, B, C, etc. Some examples are 16rFFFF, 12rBAABAA and 3r211.

If you enter a number as the ratio of two integers, the number will be retained as a fraction, or, if the numerator is evenly divisible by the denominator, as an integer; for example 3 / 2 will remain as the fraction 3/2, whereas 4 / 2 will become 2.

Specifying Units

You must specify units in any parameter setting that involves frequency or duration, using one of the following abbreviations:

Abbreviation	Meaning
days	days
h	hours
m	minutes
s	seconds
ms	milliseconds
usec	microseconds
samp	samples
beat	(default BPM setting is quarter note = 60)
SMPTE	SMPTE time code
hz	hertz
nn	MIDI note number

For example,

```
440 hz
60 nn
1 s
1 day + 3 m + 17 s
```

To specify frequency, you can also type the octave followed by the lettername of the pitch class. For example,

```
4 c sharp
```

indicates middle C sharp (MIDI note number 61 nn). The *solfege* names for the pitch class are also accepted: do, re, mi, fa, so or sol, la, ti or si.

To specify an arbitrarily long duration, use on. ON is equivalent to 730 days or 2 years.

Using Values with Units in Calculations

The rules of regular arithmetic with units apply to parameters that have units of time or frequency. For instance, you can multiply a value with units by a number with no units, but if you are adding two values, one of the following conditions must apply: both must be in units of time, both must be in units of frequency, or both must be without units.

If the values being combined are of the same type (both in units of time or both in units of frequency), but are not the same units, the calculation will be performed in units of hertz or seconds:

Example	Value
60 nn hz	261.6256 hz
100 hz nn	43.35 nn
60 nn + 100 hz	361.6256 hz
60 nn + 100 hz nn	103.35 nn
60 nn hz + 100 hz	361.6256 hz

If you want to remove the units after performing a conversion or calculation, use `removeUnits`. For example, if you needed to use a pitch in units of hertz in computing the value of a non-frequency field, you would need to remove the units, as in the following code:

```
4 c hz removeUnits / SignalProcessor sampleRate.
```

A SMPTE time or duration can be specified as

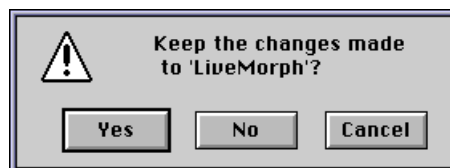
```
hours:minutes:seconds.frames SMPTE
```

as in the following example:

```
1:04:32.2 SMPTE
```

Closing the Sound Editor and Saving Changes

To close the Sound editor, click in the close box at the upper left. Kyma will ask whether or not you want to keep the changes you have made. At this point you have a choice of replacing the old Sound with the edited version, dropping the changes that you just made, or changing your mind and canceling.



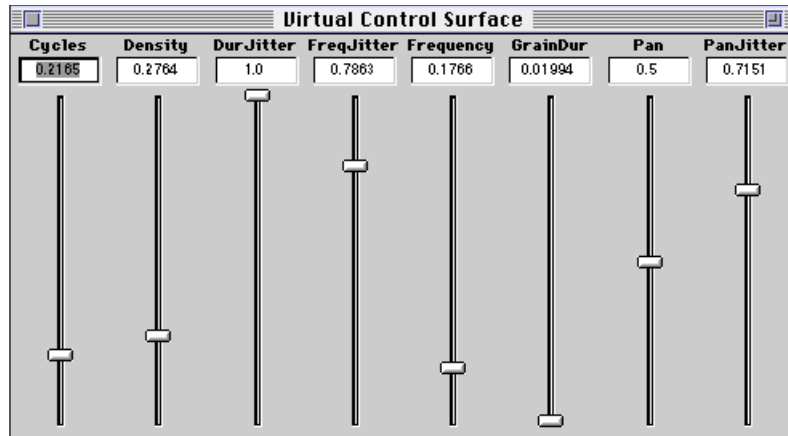
If you want to save the edited Sound as a new Sound while keeping the original Sound unchanged, drag the rightmost Sound from the signal flow diagram into a Sound file window; then close the Sound editor, and choose **No** when it asks if you want to save the changes.

Note that changes made to a Sound are not saved onto the disk until you have saved the entire Sound file window. To do this, choose **Save** or **Save as...** from the **File** menu.

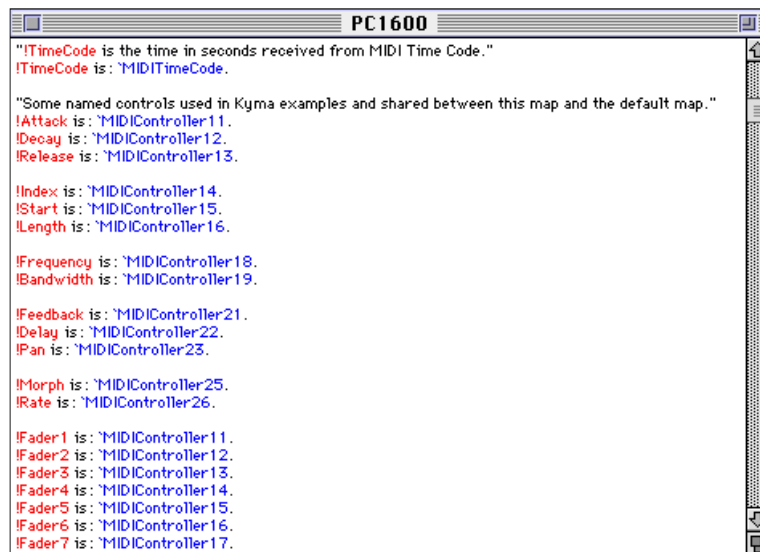
Event Values, Virtual Control Surface, and Global Map

Placing Event Values and Sounds into hot parameter fields of a Sound lets you control the parameters of the Sound in real time. You can exercise this real-time control when you link the Event Values to the virtual control surface, external MIDI controllers, or programmed events. In effect, you create custom, real-time controls of your patch by using Event Values in the parameter fields of your Sounds.

The *virtual control surface* allows you to monitor and control Event Values in the currently loaded Sound.



Event Values are like hardware controls (a volume control, for instance) in that they have initial positions that you can adjust within specified ranges. Event Values are also like most hardware controls in that they remain at the value where you last set them until you reset them. But unlike hardware knobs and buttons, these software controls are easily remapped. This mapping is contained in the *global map*.



Event Values

Event Values can be used as real-time “hot-links” between your Kyma Sound and the outside world. You can set an Event Value to be updated by external event sources such as MIDI, the virtual control surface, or programs other than Kyma. Event Values can be linked to sources of events by *MIDIVoice*, *MIDIMapper*, *SoundToEvent* or *AnalogSequencer* Sounds. If your Sound does not include any of these Sounds, the Event Value will be associated with a source in the global map. This global map is initially set to a default map. Later, you may want to devise your own maps that correspond to the controllers in your studio.

An Event Value is displayed as a name preceded by an exclamation point; Event Values are red on color monitors and bold on other monitors. If you use an Event Value that is defined in the global map or the **Map** of a *MIDIMapper*, the map transparently connects the source of events as defined in the map to that Event Value. If you use an Event Value that is not defined in any map, Kyma will only allow that Event Value to be controlled from the virtual control surface.

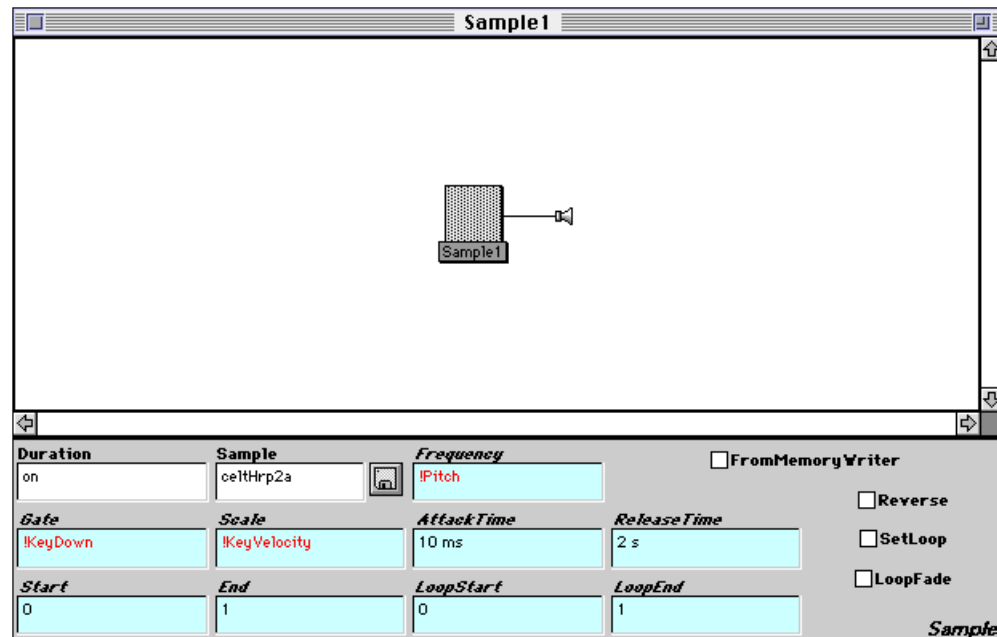
Events Values and expressions involving Event Values can appear only in hot parameter fields.* There are several ways to enter Event Values:

Press the **Escape** key once and play a chord of one, two, or three keys on a MIDI keyboard to obtain !Pitch, !KeyDown, or !KeyVelocity.

Press the **Escape** key once and move a MIDI continuous controller to enter the name of that continuous controller.

Choose **Paste hot...** from the **Edit** menu to select an Event Value or expression from a list and paste it into the parameter field.

Type the name of the Event Value (preceded by an exclamation point) directly into the field.



Sources of Event Values

All Event Values are made up of combinations of Event Sources. The mapping specified in the global map or in a *MIDIMapper* Sound (see *Global and Local Maps* on page 482) provides a memorable name (the Event Value) for a combination of hardware and software sources (the Event Sources).

An Event Source is one of the following

- a value derived from MIDI continuous controller events
- a value derived from MIDI keyboard (note) events
- a value derived from other MIDI events (for example, MIDI time code)
- a value derived from the signal processor (such as the current time)
- a value derived from the output of a Sound (the **Value** of a *SoundToEvent*)
- a value derived from the virtual control surface or other software running on the host computer

* A hot parameter field has a cyan colored background and an italics label. The **Frequency** field of a *Sample* is an example of a hot parameter field.

Event Sources are specified by preceding the name of the Event Source by an accent grave ('). They are displayed in blue in the user interface.

The only way to control an Event Value externally (for example, by a MIDI keyboard, fader, sequencer, etc.) is to map it to one of the following Event Sources:

```
`MIDIKeyDown          `MIDIKeyNumber    `MIDIKeyVelocity
`MIDIController00      thru              `MIDIController127
`MIDIPitchBend         `MIDITimeCode     `MIDIChannelPressure
`MIDIProgramNumber
```

MIDI Continuous Controller Sources

MIDI continuous controller Event Sources are designated as `MIDIController00 through `MIDIController127. The signal processor automatically scales the continuous controller values received *via* MIDI control change messages (which are in the range of (0, 127)) to the range of (0, 1).

MIDI Keyboard Sources

MIDI note on and off messages control the Event Sources `MIDIKeyDown, `MIDIKeyNumber, and `MIDIKeyVelocity. Kyma treats MIDI note on messages with zero velocity in the same way as MIDI note off messages with a velocity equal to the velocity of the original note on message.

`MIDIKeyDown is set in the following way:

- a note off message causes it to be set to 0

- a note on message causes it to first be set to -1, and then, five milliseconds later, to be set to +1

A value of -1 in the trigger of an envelope (for example, *ADSR*) tells the envelope to rapidly decay the envelope to zero. The later value of +1 tells the envelope to begin its attack.

`MIDIKeyNumber contains the key number from the note on message. It is set at the same time that `MIDIKeyDown is set to +1.

The velocity part of a MIDI note message is scaled from its range of (0, 127) to the range of (0, 1). `MIDIKeyVelocity is set to this scaled value at the same time that `MIDIKeyDown is set to +1.



MIDI note events are allocated to individual voices within a *MIDIMapper* or *MIDIvoice* according to the polyphony, MIDI channel and pitch range specified in the mapping Sound. A note is allocated to the oldest voice that is not in use, or, if all voices are currently in use, the voice that has been on for the longest amount of time.

Other MIDI Sources

The remaining MIDI messages are mapped in the following way:

- MIDI program change messages cause the Event Source `MIDIProgramNumber to be set to the new program number. Program numbers are in the range of 1 to 128.

- MIDI channel aftertouch messages cause the Event Source `MIDIChannelPressure to be set to the channel pressure value (scaled from (0, 127) to (0, 1)).

- MIDI pitch bend messages set `MIDIPitchBend. Its range of (-2048, 2048) is scaled to the range of (-1, 1).

- MIDI time code messages cause `MIDITimeCode to be set to the time received. Additionally, `RealTime (the time in seconds since the Sound was started) is updated whenever these messages are received.

Signal Processor Sources

The signal processor itself is the source of some Event Sources:

- `Time is the time in seconds since the entire Sound structure was started.

- `LocalTime is the time in seconds since the Sound using `LocalTime was started.

Other Event Source names can be used, but can only be controlled through Kyma:

Kyma Event Sources

SoundToEvent acts as a source of Event Values. Changes to its **Value** parameter cause the **GeneratedEvent** Event Value to change in response. *AnalogSequencer* can also act as a source of the values of the Event Values listed in its **ExtraValues** field, as can *MIDIVoice* and *MIDIMapper* when you have **FromScript** checked.

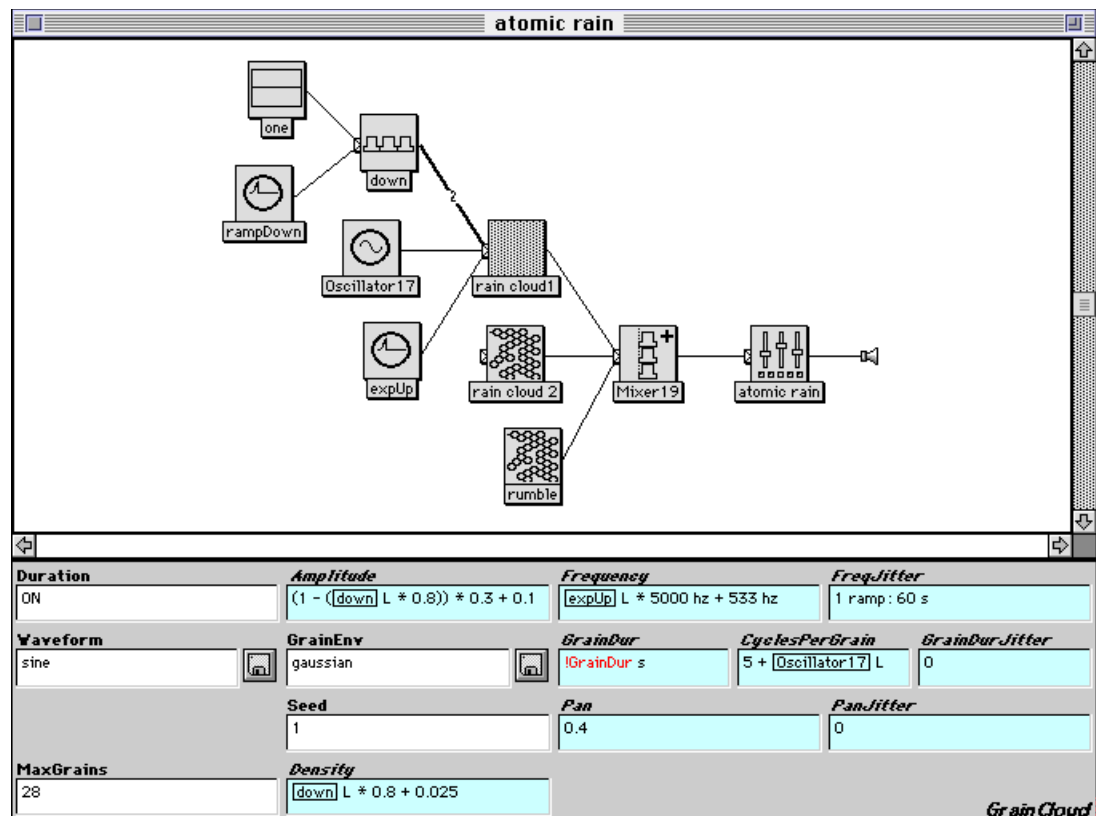
Other Sources

All sources (except ``MIDIKeyDown`, ``MIDIKeyNumber`, and ``MIDIKeyVelocity`) can additionally obtain their value from the virtual control surface, a tool, or a third party program that uses the Capybara driver. Any source not listed above can *only* obtain its value in this way.

Using Sounds in Hot Parameter Fields

Hot parameter fields can also use Sounds as their values. This is like connecting the output of one module to the input of another in a voltage-controlled analog synthesizer. As with the analog synthesizer, the control signal is never actually heard; it just supplies a constantly changing value for the parameter of a Sound which is heard.

When you paste a Sound into a hot parameter field, it appears in the field as the Sound's name enclosed in a box followed by an `L` to indicate the left channel. Once you update the signal flow diagram (by double-clicking in a white space), the Sound will also show up in the signal flow diagram.



You can combine Sounds, Event Values, and numbers in arithmetic expressions. Since the output of a Sound is a stereo pair, when you use Sounds with numbers and Event Values, you must specify which channel to use as the source of the control signal. Use `L` to indicate the left channel of the Sound, `R` to indicate the right channel, and `M` to indicate the monophonic mix of the two channels.

By default, Sounds in hot parameter fields are treated as control signals and update once every millisecond (so the maximum frequency *without aliasing* of an LFO is 500 hz). To use a lower update rate, append a colon and a number to indicate the number of milliseconds between updates. For example,

Oscillator

 L: 5

updates once every five milliseconds (200 hz sample rate).

Arithmetic with Event Values and Sounds

The signal processor contains a *real-time evaluator* that can calculate the value of expressions involving Event Values and/or Sounds in “hot” parameter fields. The real-time evaluator calculates the value of an expression whenever it receives a new value for a Sound or Event Value used in the expression.

The real-time evaluator implements a subset of the operations available to the Smalltalk evaluator on the host computer. The following is a list of all operations available to the real-time evaluator. Examples are given using constants, to make it easier to see exactly what the operation does, but any of these operations can also be used on Sounds or Event Values.

Arithmetic Operators

Plus, minus, times, and divide-by are standard arithmetic operators and behave just as you would expect them to behave.

Message	Example	Result
+	3 + 2, -3 + 2	5, -1
-	3 - 2, -3 - 2	1, -5
*	3 * 2, -3 * 2	6, -6
/	3 / 2, -3 / 2, 3.0 / 2	3/2, -3/2, 1.5

****** means raise to a power, the **//** takes the floor of the result of a division (rounds toward negative infinity), the **** is the first argument modulo the second, and **negated** toggles the sign of its argument. **mod:** is the same as the double-backslash operator.

Message	Example	Result
**	3 ** 2, -3 ** 2	9, ERROR
//	3 // 2, -3 // 2	1, -2
\	3 \ 2, -3 \ 2, 3 \ -2	1, 1, -1
mod:	3 mod: 2, -3 mod: 2, 3 mod: -2	1, 1, -1
negated	3 negated, -3 negated	-3, 3

inverse is one divided by its argument. **abs** gives the absolute value (the magnitude) of its argument. **sqrt** is the square root of its argument.

Message	Example	Result
inverse	3 inverse, -3 inverse, 3.0 inverse	(1/3), (-1/3), 0.333333
abs	3 abs, -3 abs	3, 3
sqrt	3 sqrt, -3 sqrt	1.73205, ERROR

truncated drops any fractional part of the number (towards zero, not minus infinity). **rounded** rounds to the nearest integer. **sign** returns -1 for negative numbers, 0 for zero, and 1 for positive arguments. **clipTo01** returns 0 for all arguments less than or equal to zero, and it returns 1 for all arguments greater than or equal to one; arguments between zero and one are unaffected.

Message	Example	Result
truncated	3.1 truncated, -2.1 truncated	3, -2
rounded	3.5 rounded, -2.1 rounded	4, -2
sign	-3 sign, 0 sign, 2 sign	-1, 0, 1
clipTo01	-3 clipTo01, 0.5 clipTo01, 2 clipTo01	0, 0.5, 1

vmin: returns the smaller of two numbers and **vmax**: returns the larger of two numbers.

Message	Example	Result
vmin :	3 vmin: 2, 2 vmin: 3	2
vmax :	3 vmax: 2, 2 vmax: 3	3

Transcendental Functions

cos and **sin** return the cosine and sine of a number. **normCos** returns the cosine of the number multiplied by π (and similarly **normSin** returns the sine of a number multiplied by π). **normCos** and **normSin** come in handy because Kyma controllers have a range of $(0, 1)$, but typically you would like to take the sine or cosine of values between 0 and π . **normCos** and **normSin** scale your fader value to between $(0, \pi)$ before taking the cosine or sine.

exp gives you e raised to the power of the argument. **twoExp** gives you 2 raised to the power of the argument. **log** is the power of 10 which would evaluate to the argument. **twoLog** is the power of 2 which would evaluate to the argument.

Message	Example	Result
cos	(Float pi) cos, 0 cos	-1.0, 1.0
sin	(Float pi * -0.5) sin, (Float pi * 0.5) sin	-1.0, 1.0
normCos	0 normCos, 1 normCos	1.0, -1.0
normSin	0.5 normSin, -0.5 normSin	1.0, -1.0
exp	0, 1.0	1.0, 2.71828
twoExp	0, 9 twoExp, 10 twoExp	1.0, 512.0, 1024.0
log	1.0 log, 10 log, 100 log, 0.1 log	0.0, 1.0, 2.0, -1.0
twoLog	1.0 twoLog, 2 twoLog, 0.5 twoLog	0.0, 1.0, -1.0

Array Accessing

The first argument of the **of**: message is an index, and the second argument is the array. The first element of the array is considered to be at index zero, not at index one. The index can be a real-time expression, but the array must consist of numbers. Since the index is typically a continuous controller whose range is $(0, 1)$, it should in most cases be scaled by the size of the array less one (since the indexing goes from $(0, \text{size}-1)$).

Message	Example	Result
of :	1 of: #(8 9 10 11), (0 of: #(1 3)) of: #(4 3 2 1)	9, 3

Booleans

The real-time evaluator can perform all of the standard comparison operators. They return 0 for false and 1 for true.

Message	Example	Result
ne: (not equal)	3 ne: 2, 3 ne: 3	1, 0
eq: (equal)	3 eq: 2, 3 eq: 3	0, 1
gt: (greater than)	3 gt: 2, 3 gt: 3	1, 0
lt: (less than)	3 lt: 2, 3 lt: 3	0, 0
ge: (greater or equal)	3 ge: 2, 3 ge: 3	1, 1
le: (less or equal)	3 le: 2, 3 le: 3	0, 1

Conditionals

neg:zero:pos: takes on one of three values, depending upon the value of the message receiver. All expressions are evaluated, but only one of the results is used, depending on whether the first expression has a negative, zero, or positive value. **neg:zero:pos:initially:** is the same as **neg:zero:pos:** except that you can specify the initial value of the entire expression. In **neg:zero:pos:**, the initial value of the entire expression is always 0. The expression takes on the initial value only for as long as the message receiver does not yet have a value (e.g. if it depends on an event that has not yet occurred).

true:false: or **false:true:** does conditional evaluation of either the true argument or the false argument depending upon the value of a test expression. It evaluates only one, not both of the true and false arguments. For this conditional, true is considered to be any positive number, false is any zero or negative number.

Message	Example	Result
neg:zero:pos:	-1 neg: 4 zero: 3 pos: 2	4
neg:zero:pos:initially:	1 neg: 4 zero: 3 pos: 2 initially: 8	2
true:false:	0 true: 3 false: 2, 0.5 true: 5 false: 6	2, 5
false:true	-2 false: 0 true: 100, 10 false: 40 true: 60	0, 60

Conversions

Kyma considers positive numbers as boolean true values and zero or negative numbers as boolean false values. **asLogicValue** maps all true values to 1 and all false values to 0.

You can use **db** to convert a decibel value into its linear equivalent. 0 db is equal to 1 or full amplitude, and increases of 6 db double the amplitude, and decreases of 6 db halve the amplitude.

inSOSPitch converts a frequency in hertz or pitch in note numbers to a pitch in the same format and range as the output of the spectral sources (e.g. *SpectrumInRAM*, *SyntheticSpectrumFromArray*, etc.). These pitches map the range from 0 hz through 22050 hz to a log curve drawn between 0 and 1. Every doubling of the frequency in hertz corresponds to adding 1/15 to the number in SOS pitch, so at 22050 hz, the SOS pitch is 1, at 11025 hz it is $1 - (1/15)$, and so on.

Message	Example	Result
asLogicValue	-2 asLogicValue, 0 asLogicValue, 3 asLogicValue	0, 0, 1
db	0 db, 6 db, -6 db, -12 db	1.0, 1.99, 0.5, 0.25
inSOSPitch	22050.0 hz inSOSPitch, 1378.12 hz inSOSPitch	1.0d, 0.73

You can use **asLogicValue** to perform logic operations using arithmetic:

Logic	Kyma Equivalent
a AND b	a asLogicValue * b asLogicValue
a OR b	(a asLogicValue + b asLogicValue) asLogicValue
NOT a	1 - a asLogicValue
a XOR b	a asLogicValue ne: b asLogicValue
a EQV b	a asLogicValue eq: b asLogicValue

Units

Units of frequency, amplitude and time are also understood by the real-time evaluator. The following are units of time for seconds, minutes, hours, days, milliseconds, samples, and microseconds:

Message	Example	Result
s	3 s	3 s
m	3 m	180.0 s
h	3 h	10800.0 s
days	3 days	259200.0 s
ms	3 ms	0.003 s
samp	3 samp	3 samp
usec	3 usec	3.0d-6 s

The following are the messages for specifying hertz (cycles per second), MIDI note number (where middle C is note number 60), and octave pitch class (where 4 c is middle C).

Message	Example	Result
hz	440 hz	440 hz
nn	69 nn	69 nn
	4 a	69 nn
	4 mi flat	63 nn

Unit Conversions

To convert between seconds and samples or hertz and note number, cascade the unit messages. To strip off the units (e.g. for parameter fields like **Scale** that do not expect a number with units), use **removeUnits**.

Example	Result
1 s samp	44100 samp
4410 samps	0.1 s
4 c hz	261.6256 hz
269.2918152432d hz nn	60.5 nn
4 a removeUnits	69

Symbolic Times and Frequencies

Use `on` in a **Duration** field to indicate (virtually) infinite duration. Use `default` in a **Duration** field of a *Sample*, *DiskPlayer*, or *GenericSource* to indicate the “natural” duration, *i.e.* the duration of the original recording. If used in the **Frequency** field, it represents the “natural” frequency or the originally recorded frequency as stored in the header of the AIFF or WAV file (if no base pitch has been indicated in the header, it will be set to 60 `nn` for you). The symbolic durations and frequencies can be used only in the context of a Sound that is playing back a digital recording.

Functions of time

ramp has a resting value of 1. When the message receiver becomes positive, this expression will grow linearly from 0 up to 1 over the course of one second, and then stick at its resting value of 1. It cannot be triggered again until the message receiver becomes zero or negative. For example

```
1 ramp
```

will immediately go from 0 to 1 in one second and stick at 1 forever, and

```
!KeyDown ramp
```

will start at 1, and go from 0 to 1 over the course of one second whenever a MIDI key down event is received.

ramp: is like **ramp** except that you can specify the amount of time it takes to get from 0 to 1. For example

```
1 ramp: 3 s
```

will take three seconds to go from 0 to 1 and then stick at the value of 1 forever.

fullRamp is like **ramp** except that it goes from -1 up to 1. **fullRamp:** is like **ramp:** except it goes from -1 to 1 in the amount of time specified after the colon.

repeatingRamp: is like **ramp:** except that instead of sticking at its resting value at the end, it continuously repeats the ramp for as long as it is triggered (*i.e.* for as long as its message receiver is positive). There exist repeating versions of all of the ramp-like functions: **repeatingRamp**, **repeatingRamp:**, **repeatingFullRamp**, and **repeatingFullRamp:**.

bpm: has a resting value of 0 and changes to 1 periodically, at a rate specified in beats-per-minute (the same way you would specify a tempo using a metronome marking). It remains at 1 for half the time between beats and then returns to 0 before the next beat. These expressions are typically used in **Gate** or **Trigger** fields in order to periodically trigger the Sound. For example,

```
!KeyDown bpm: 60
```

would become 1 once per second whenever and for as long as a MIDI key is held down, and

```
1 bpm: (!Rate * 208)
```

would repeat at an adjustable rate from 0 up to 208 beats per minute. To convert a rate in terms the number of seconds between beats into a value in beats per minute, divide 60 by the number of seconds between beats. For example

```
1 bpm: (60.0/3)
```

would trigger your Sound once every three seconds. You can use this in a *DiskPlayer* to “loop” a sample being read from disk by putting the natural duration (without units) of your recording in as the number of seconds between repetitions,

```
1 bpm: (60.0 / durationOfSample)
```

bpm:dutyCycle: is like **bpm:** except that you can control the amount of time that it remains at a value of 1. A duty cycle of 0.5 means that it stays at 1 for half of the duration between beats. A smaller duty cycle means that it will stay at 1 for less time, and a larger duty cycle means it will stay at 1 for a longer amount of time. A duty cycle of 0 means that it will never go to 1 and a duty cycle of 1 would mean that it sticks at 1 forever.

```
1 bpm: 120 dutyCycle: 0.25
```


Random Numbers

random is a message sent to a time. It generates a new random number between -1 and 1 at the periodic rate. For example,

```
2 s random
```

would generate a new number once every 2 seconds.

nextRandom is typically sent to an Event Value that is acting as a trigger. It generates a random number between -1 and 1 each time the trigger becomes positive. For example,

```
!KeyDown nextRandom
```

would generate a new random number each time you pressed a key on the MIDI keyboard.

Smoothing

There are two operations that smooth out or filter the changes between sequential real-time expression values.

```
!Fader1 smooth: 2 s
```

will take two seconds to get from the old value of `!Fader1` to its new value, rather than jumping instantly to its new value.

```
!Fader1 smoothed
```

will use a default value of 100 milliseconds as the time it takes to get from the old value to the new value.

For example, to add portamento to a Sound, you can smooth its frequency parameter using

```
!Pitch smoothed
```

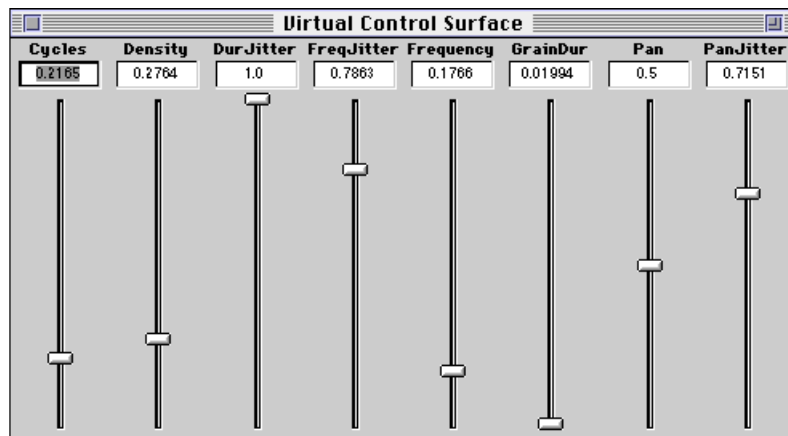
or,

```
!Pitch smooth: 25 ms
```

for a quicker portamento.

Virtual Control Surface

To open the virtual control surface, choose **Virtual control surface** from the **File** menu.[§] The virtual control surface allows you to monitor and control Event Values in the currently loaded Sound. The virtual control surface is global; it will change whenever you load a different Sound.



The faders, toggles and gates in the virtual control surface serve two functions: they can be used to control Event Values directly from the virtual control surface; they also display any changes made to the

[§] By default, Kyma opens the virtual control surface automatically for you whenever a Sound is played, see *Performance...* on page 430.

Event Value via the MIDI input of the signal processor or by some other software running on your computer.[‡]

The Event Values displayed in the virtual control surface are organized alphabetically from left to right. Each type of control: annotations, oscilloscope and spectrum analyzer displays, faders, gates, small faders, and toggles occupies a separate row on the virtual control surface. The type of display a particular Event Value will have depends on the settings in the local or global map that defines the Event Value; see *Virtual Control Surface and Mappings* on page 483. Any Event Value not defined in a map will be displayed as a fader. See the quick reference on page 212 for information on how to use the controls within the virtual control surface.

When you use an Event Value that is defined in the global map, the specified Event Source will supply values to that Event Value. If you use an Event Value not defined in the global map, Kyma will create a new Event Source that gets its value from the virtual control surface.

Global and Local Maps

Unless you specify otherwise, Event Values in your Sounds are evaluated in the context of a global map that associates Event Values with Event Sources. You can override the global map by using a local map specified in the **Map** field of a *MIDIMapper* Sound.

The Global Map

To edit a global map, choose **Open...** from the **File** menu, select **Global map** as the file type and select the global map file to edit. You can create a new global map by choosing **New...** from the **File** menu and by selecting **Global map** as the new file type.

The global map editor is simply a text editor. Make changes to the global map as you would to any text file, choosing **Save**, **Save as...**, or **Close** from the **File** menu to save any changes that you have made. Use **Choose global map...** from the **File** menu to tell Kyma which global map to use when compiling Sounds.

Your system arrives with a default global map, but you may find that a customized global map is helpful when working with the MIDI controllers found in your studio. To see the default map on-line, choose **Open...** from the **File** menu, select **Global map** as the file type, then select the file named **Default** (in the **Global Maps** folder) from the file list.

You can create your own customized global map that corresponds to the specific MIDI devices that you use with Kyma. For example, Kyma comes with example global maps for use with the Peavey PC 1600 MIDI controller (called **PC1600**) and the Buchla Lighting wand-based controller (called **Lighting**).

It is a good idea to create a new map rather than modifying the default map and saving it; that way you always have the original map to use as a reference.

You can also override the global map locally. To override or extend the global map, use a *MIDIMapper* Sound. In the **Map** parameter of the *MIDIMapper*, you only need to enter the associations that are different from the associations in the global map. When you use a *MIDIMapper* to change the global map, the changes you make affect only Sounds to the left of the *MIDIMapper* in the signal flow diagram and only for the duration of those Sounds — not in any permanent way.

Specifying the Map

A map (whether in the global map file or in the **Map** field of a *MIDIMapper* Sound) associates Event Values with Event Sources.

There are several ways to specify this association, all variants of the following syntax:

```
!anEventValue is: `anEventSourceOrExpression.
```

[‡] To send MIDI data to Kyma from another program, connect your keyboard controller to the MIDI input on your computer or MIDI interface, connect the MIDI output of your MIDI interface to the MIDI input of the signal processor; then run Kyma and the other program simultaneously. (It is recommended that you switch off your computer when connecting MIDI cables.) Remember that when you want to use Kyma alone you might have to explicitly patch a MIDI thru connection using a MIDI patcher on your host computer.

For example,

```
!Volume is: `MIDIController07.
```

You could enter `!Volume` as the value of the **Scale** parameter in a *Sample*; the map would then associate the **Scale** parameter with MIDI controller 7 data on the default MIDI channel (as specified in **Configure MIDI...** under the **DSP** menu).

Event Values can be combinations of Event Sources and constants. For example,

```
!Pitch is: (`MIDIKeyNumber + (`MIDIPitchBend displayAs: #nothing)) nn.
```

Defines `!Pitch` to be the key number found in incoming MIDI note-on events offset by the incoming value of the pitch bend wheel (which will not show up in the virtual control surface). In this example, the pitch bend can contribute up to a half-step variation in the pitch. You can change the range of pitch bend to an octave variation by using

```
!Pitch is: (`MIDIKeyNumber + (12 * `MIDIPitchBend)) nn.
```

in the global map.

When the global map is used to supply Event Values, Kyma uses the default MIDI channel number given in **Configure MIDI...** under the **DSP** menu. A *MIDIVoice* or *MIDIMapper* Sound can override the default MIDI channel for its input Sound by specifying the channel number in the **Channel** field.

Event Source Options

The Event Sources on the right side of each association in the map can have additional options controlling MIDI channel, value scaling, and presentation in the virtual control surface.

You can override the default MIDI channel specification by adding a `channel` tag to the specification of the Event Source:

```
!Volume is: (`MIDIController07 channel: 5).
```

This tells Kyma that `!Volume` is associated with continuous controller number 7 on MIDI channel 5. Notice that parentheses must enclose the Event Source whenever a tag is used.

You can specify how the Event Source should be scaled prior to being used as an Event Source:

```
!Volume is: (`MIDIController07 min: 0.5 max: 0.75).
```

Kyma will scale the normal (0,1) range of the MIDI continuous controller to the specified range of (0.5,0.75).

You can also restrict the values to fall on a grid:

```
!Volume is: (`MIDIController07 min: 0.5 max: 0.75 grid: 0.05).
```

In this case, `!Volume` can only be one of the values 0.5, 0.55, 0.6, 0.65, 0.7, or 0.75.

You can specify whether changes in the Event Source result in linear or logarithmic changes in the Event Value. By default, a change in the Event Source results in a linear change to the Event Value. To cause logarithmic changes, use:

```
!Volume is: (`MIDIController07 taper: #log).
```

You can combine these options by separating them by semicolons, as in:

```
!Volume is: (`MIDIController07 channel: 5; min: 100 max: 1000 grid: 50; taper: #log).
```

Virtual Control Surface and Mappings

By defining the controller type in the map, you determine how controllers associated with Event Values will appear in the virtual control surface. Controllers can be one of three types: faders, toggles, or gates. Faders are commonly associated with continuous controllers. Toggles are typically used to switch from one state to another (*i.e.*, you can use a toggle to switch something on until you want to switch it off). Gates are like momentary buttons; they are on for as long as they are held down, and they turn off as soon as the button is released. (For example, the damper pedal is typically interpreted as a gate.)

Defining a controller type in the global map affects only the way in which the control appears in the virtual control surface; faders show up as (you guessed it) faders, gates show up as buttons, and toggles show up as check boxes.

If you want a controller to show up as a fader in the virtual control surface, you would use

```
!Brightness is : (`MIDIController64 displayAs: #fader).
```

If you want a controller to show up as a numeric entry box in the virtual control surface, you would use

```
!Brightness is : (`MIDIController64 displayAs: #smallFader).
```

You can use this kind of screen control as a fader as well. Hold down the **Command** or **Control** key while pressing the mouse button in the numeric entry box; vertical mouse movements will then be interpreted as if you were moving an invisible fader.

If you want the controller to show up in the virtual control surface as a check box (allowing only values of 0 or 1 for brightness), use

```
!Brightness is : (`MIDIController64 displayAs: #toggle).
```

If you want it to show up in the virtual control surface as a button (*i.e.*, so that its output is 1 when you hold down the button and 0 otherwise), use

```
!Brightness is : (`MIDIController64 displayAs: #gate).
```

If you do not want the controller to be displayed in the virtual control surface, use

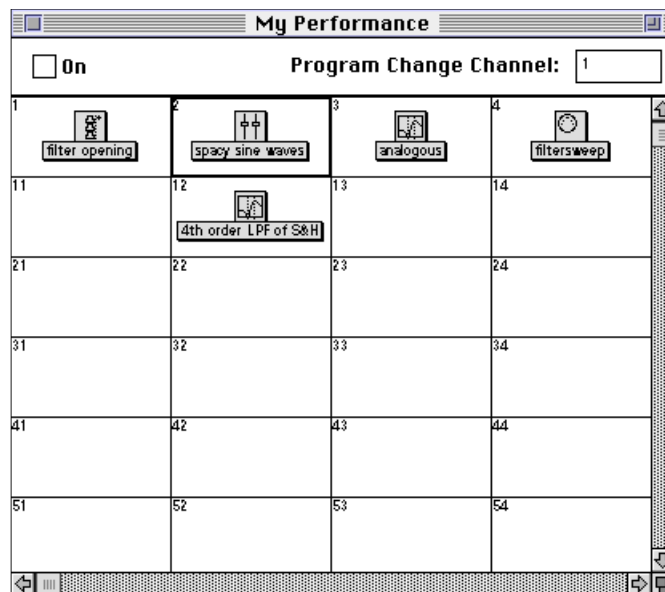
```
!Brightness is : (`MIDIController64 displayAs: #nothing).
```

Compiled Sound Grid

The Compiled Sound grid is useful for organizing Sounds to be used in an interactive performance environment (which could include musical performances, psychoacoustic tests, classroom demonstrations, or any other environment where a user selectively loads Sounds whose ordering and timing are not predetermined). The grid provides a means for quickly loading and starting Sounds that have already been compiled, simply by clicking in a box in the grid or by sending a MIDI program change with the number indicated in the box.

Creating a Compiled Sound Grid

To create a compiled Sound grid, select **New...** from the **File** menu, then choose **Compiled Sound grid** as the file type. The interface for a compiled Sound grid consists of a grid of 128 boxes, a program channel field, and an **On** check box. Each numbered box in the grid corresponds to a MIDI program number. To associate a Kyma Sound with a specific MIDI program number, drag the Sound into the corresponding box. To edit a Sound in the grid, make sure the **On** box is unchecked and then double-click the Sound in the grid.



The Program Channel Field

The value you enter in the program channel field determines the MIDI channel on which Kyma looks for program changes. To set the channel, type in the number of a MIDI channel and press **Enter**. This setting does not affect the MIDI channels on which the Sound(s) can receive or send MIDI information other than program changes. In other words, all other MIDI events are received on the default MIDI channel. This default channel need not be the same channel on which you choose to receive program changes.

Using the Compiled Sound Grid

When you have finished making the associations between Sounds and program numbers, choose **Compile to disk...** from the **Action** menu; this compiles each Sound, stores the compiled Sound on the disk in Kyma's temporary folder, and names each file using a unique number. Compiling also stores copies of the samples that would be loaded into sample RAM and copies of any MIDI files needed to play the Sounds, as well as saving the global map, the current number of expansion cards, the available wavetable memory, and the full file names of any sample files (such as digital recordings) used in the Sound.



Because the compiled Sounds do not automatically update if you make changes to the original Sounds or any of the samples or MIDI files the Sounds may depend upon, you should recompile the compiled Sound grid if you make changes to these files or any time you update your Kyma software or hardware to a different version. To force all Sounds in the grid to compile, choose **Compile to disk...** from the **Action** menu and click on the **All Sounds** button.

Once you have compiled the Sounds in the grid, you can use the grid by clicking in the **On** check box. When the **On** box is checked, an incoming MIDI program change causes Kyma to load the corresponding Sound. You can also load any Sound in the grid by clicking once in its box in the grid.

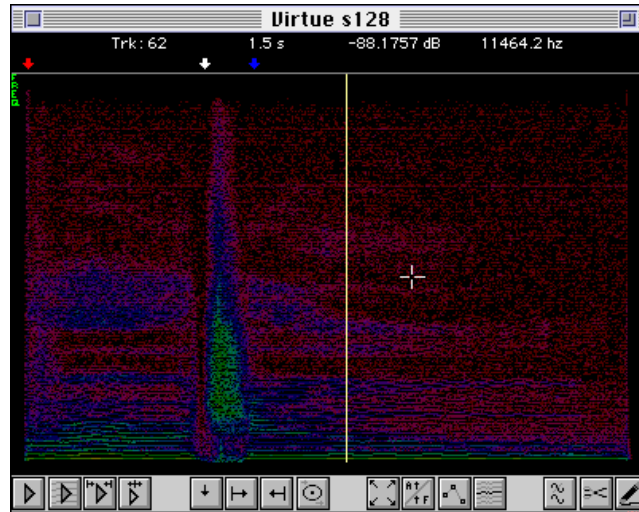
Since you can play Sounds, save Sounds, edit Sounds, and organize Sounds in a compiled Sound grid, you may prefer to use these grids in place of Sound file windows as your basic workspace in Kyma.

Spectrum Editor

A *spectrum file* contains a collection of tracks; each *track* consists of an amplitude and a frequency envelope that describe the change over time of the amplitude and frequency of a single sine wave oscillator. The amplitude and frequency values at one specific point in time for all of the tracks is called a *frame*. Spectrum files are typically created using the Spectrum Analysis Tool or a third-party program such as Lemur.[§]

Opening a Spectrum File

Open the spectrum editor on a file using **Open...** from the **File** menu or by double-clicking a spectrum file in the file organizer.



The band across the top is reserved for showing marker positions and track information. The large color area in the center is for displaying the spectrum.* The buttons across the bottom are for editing and display operations. They are grouped to correspond with the function keys on your computer keyboard.

Spectrum Display

In the center area of the editor, time is shown from left to right, frequency from bottom to top, and amplitude is mapped to color: the bright colors like yellow and green are the largest amplitude, then the blues and purples, with red (and black) showing the smallest amplitudes.

The horizontal lines are the tracks. Each track corresponds to an amplitude and frequency envelope for a sine wave oscillator. The change in the color of a track as you trace it from left to right corresponds to the amplitude envelope on that oscillator. The change in vertical position from left to right corresponds to the frequency envelope.

To display information on a particular track at a particular point in time, place the mouse over that track (without clicking down). The track number and the time in seconds corresponding to the mouse location, the amplitude in dB (where 0 dB is the maximum), the raw amplitude value[§] (where 127 is the maximum), and the frequency in hertz are displayed in the upper part of the editor.

[§] For more information on the Spectrum Analysis Tool, see *Tools menu: Spectral Analysis* on page 443. Lemur is available from <http://datura.cerl.uiuc.edu>.

* Since the spectrum editor uses color extensively, some of the screen images in this section may not reproduce well in the printed version of this manual.

[§] Raw amplitude values are used by the track selection and track filtering options.

Display Amplitude Envelopes



F10

By default, the spectrum editor displays frequency on the vertical axis and time on the horizontal axis. To switch to showing amplitude on the vertical axis, use the button that toggles between amplitude and frequency. In this mode, frequency is not shown at all (other than the general tendency that the higher numbered tracks are centered at higher frequencies). This mode makes it possible for you to change an amplitude envelope by drawing all or part of it in by hand.

When you are displaying in amplitude mode, the vertical axis is labeled **AMP** in red (rather than **FREQ** in green).

Join the Dots



F11

Pressing the join-the-dots button toggles between displaying each time point (frame) for each track as an isolated dot, or by connecting the time points (frames) for each track by lines. Join-the-dots mode only takes effect at higher magnifications where there are large gaps between the frames (and thus, between the dots).

Selecting and Deselecting

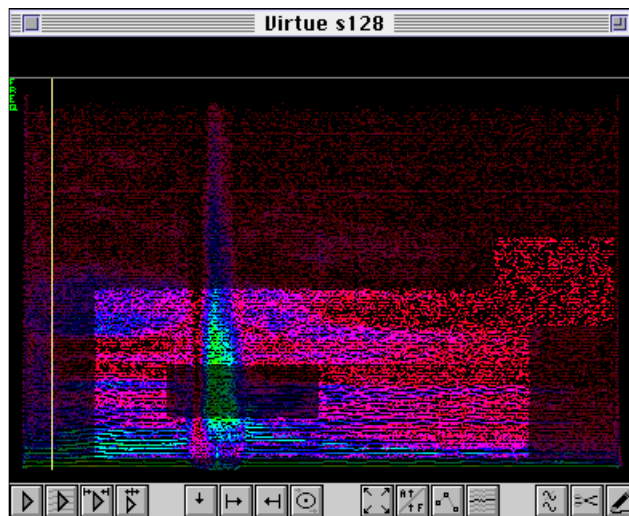
To select a single track, click on it. You will hear that track and its color will become brighter to indicate that it is selected. To extend the selection, hold the **Shift** key down while making another selection. Otherwise, the current selection will be deselected before a new selection is made. To select any of tracks 1 through 10, type the corresponding number on the computer keyboard (use **0** to select track ten). To select the track above the currently selected one, use the up-arrow key (and similarly to select the track below the selected one, use the down-arrow key).

Anytime you make a selection, the selected tracks or portions of tracks are immediately played so you can hear what you just selected.

Selecting Regions

To select a region that includes several tracks, use the mouse to draw a box around the region. You can extend boxed selections in the same way that you extend single track selections, by holding the **Shift** key down when you make the new selection.

To deselect a smaller box contained *within* a selected region, hold the shift-key down and draw a box around the region to be deselected. This will leave a “hole” in the middle of the selected area. As long as you begin drawing the new box inside a selected region, the new box will be deselected and leave a hole in the selected area. If you begin drawing a new box from a point outside the selected area, the new box will extend the current selection (even if it intersects the selected region).



Using Selection Criteria



To select tracks according to some criteria, press the selection criteria button. This presents you with a list of options for selecting tracks:

<i>Amplitudes within a range</i>	selects only those tracks and frames whose raw amplitude values [§] fall within the specified range
<i>Above amplitude threshold</i>	selects only those tracks and frames whose raw amplitude value exceeds the given threshold
<i>Complement selection</i>	selects all tracks and frames that are not currently selected
<i>Even tracks</i>	selects the even numbered tracks only
<i>Odd tracks</i>	selects the odd numbered tracks only
<i>Octave harmonics</i>	selects tracks whose numbers are a power-of-two
<i>Fibonacci tracks</i>	selects tracks numbered (1, 2, 3, 5, 8, 13...)
<i>Prime numbered tracks</i>	selects tracks whose numbers are prime (1, 2, 3, 5, 7, 11, 13...)

Playing

The leftmost play button plays all tracks. Pressing the **F1** key also plays all tracks. The button to the right of that one is for playing the selected regions only. Pressing **F2** also plays the selected tracks only. To play the segment marked by begin and end loop markers, use the button third from the left, and the button fourth from the left is used for playing everything *but* the segment marked by the begin and end loop markers.



When you move the scrub bar with the mouse, with MIDI pitch bend, or using the left/right arrow keys, you will hear either all tracks or the selected tracks only, depending upon which of the play buttons was used last.

Scrubbing

To hear different sections of the file, use the mouse to drag the vertical yellow scrub bar over the regions you would like to hear. If you hold down the **Shift** key while dragging the scrub bar, the scrub bar will always jump to the mouse location rather than smoothly approach the mouse location.

You can also use MIDI pitch bend to move the scrub bar. To step the scrub bar one frame to the left or one frame to the right use the left and right arrow keys.

Zooming in/out

To zoom closer, that is, to look in more detail at a smaller region of the display, use **Zoom in (Ctrl+I)** from the **Edit** menu.

Alternatively, you can zoom in on a region of interest by holding down the **Control** or **Command** key while drawing a box around the area of interest. The editor will magnify the boxed region so that it fills the entire area of the display.



To zoom back out again use **Zoom out (Ctrl+O)** to zoom out in steps, or press the full display button to zoom all the way out.

[§] The raw amplitude value is the value shown in parenthesis in the information area when you pass the mouse over a track.

Markers



F5

You can use markers to label different time points in the spectrum file. Click the marker button to place a marker at the current location of the scrub bar. The markers are associated with labels (strings) and stored in the header of the spectrum file the next time you save the file. By default the name of a marker is 'm' followed by the time in frames. To change the label, select the marker and hit **Enter** or **Return**.



F6 F7

The next two buttons place special markers for the beginning and ending points of a loop. The *SumOfSines* Sound uses the begin and end loop markers to loop the analysis file during resynthesis.



F8

The button with a circle on it lets you set a pre- and post-roll time. This is the amount of time prior to the loop start and after the loop end that you will hear when you hit the play-between-loop-markers button.

Clearing and Deleting

There are two ways to “remove” material from the spectrum. One is to simply set the amplitude of the track to zero for all or part of its duration. The other is to actually delete the material from the analysis file.

Clearing

To zero the amplitude of the selected tracks or portions of tracks, use **Clear** from the **Edit** menu. To zero the amplitudes of everything *but* the selected region, use **Trim** from the **Edit** menu. Zeroed tracks are displayed in dark gray and no longer contribute to the resynthesis.

To restore a grayed-out track to its original amplitude, select it. To permanently zero the amplitude in such a way that it can no longer be restored through selection, save the file to disk (choose **Save** or **Save as...** from the **File** menu).

Deleting

Deleting time frames from a file is done in two steps.



F6

The first step is to mark the beginning and ending of the segment of time that is to be deleted using the begin loop and end loop markers. To place the start marker at the current position of the scrub bar, press the start loop button.



F7

Similarly to place an end loop marker at the current position of the scrub bar, press the end loop button.



F3

To listen to the marked time segment, use the play-between-markers button.



F14

To delete the marked time frames use the scissors button. This will remove the marked time frames. To effect a permanent removal of the time frames, choose **Save** or a **Save as...** from the **File** menu.

Copy and Paste

To create a new spectrum file based only on the tracks and times within the selected region, use **Copy (Ctrl+C)** from the **Edit** menu. Kyma will prompt you for a name for the new spectrum file. To create a Kyma Sound based on this new spectrum file, click in a Sound file window and **Paste (Ctrl+V)**.

Modifying

The amplitude and frequency envelopes associated with each track can be modified by redrawing them or by applying a “filter” or algorithm to all tracks within the selected region.

Drawing Mode



F9



F15

You can redraw any part of a single amplitude or frequency envelope using the mouse. To redraw part of an envelope, first choose whether you want to edit amplitude or frequency envelope by toggling the A/F button. Then select the envelope to be edited and zoom in on it so you can see it in detail. When you are ready to start drawing, click on the draw button.

While in draw-mode, you can use the mouse as a pen, redrawing sections of the selected envelope. You will also hear the results of your redrawing as you edit. Be careful to turn off draw-mode by hitting the pencil button again before you use the mouse for anything else (because you can inadvertently alter the envelope by clicking anywhere in the display while in draw mode).

When you are in drawing mode, you hear exactly what you are drawing in either the amplitude or the frequency envelope.

Algorithmic Modifications to Tracks



F13

By pressing the track-filters button, you can get a list of algorithmic modifications that can be applied to the selected tracks. These are filters, not in the audio sense, but in the graphical sense that they smooth or otherwise modify the graphical representation of the amplitude or frequency envelope.

<i>Scale amplitudes</i>	Scale the amplitudes of the selected by the given amount.
<i>Replace amplitude with average</i>	For each selected track, replace the amplitude envelope with a constant envelope of the average amplitude value.
<i>Smooth amplitude</i>	For each selected track, replace each single amplitude envelope value with the average (mean) of the neighboring group of values. You are prompted for the size of the neighborhood.
<i>Remove outliers in amplitude</i>	For each selected track, replace each single amplitude envelope value with the middle value (median) of the neighboring group of values. You are prompted for the size of the neighborhood.
<i>Force to harmonic frequency</i>	<p>You are prompted for the number of a “harmonic” guide track. The frequency envelope of each track in the selection is replaced with a copy of the guide track frequency envelope that has been scaled according to the ratio of the track number to the guide track number.</p> <p>For example, if track 4 is selected, and track 2 is the guide track, track 4’s frequency envelope will be set to a copy of track 2’s frequency envelope moved an octave higher (two times the frequency of track 2).</p>
<i>Replace frequency with average</i>	For each selected track, replace the frequency envelope with a constant envelope of the average frequency value.
<i>Smooth frequency</i>	For each selected track, replace each single frequency envelope value with the average (mean) of the neighboring group of values. You are prompted for the size of the neighborhood.
<i>Remove outliers in frequency</i>	For each selected track, replace each single frequency envelope value with the middle value (median) of the neighboring group of values. You are prompted for the size of the neighborhood.
<i>Add noise to frequency</i>	You are prompted for the amount of frequency deviation and the rate of change of the deviation. For each selected track, this track filter adds a random amount of frequency deviation.
<i>Arpeggiate</i>	Stagger the selected tracks in time, with the higher numbered tracks delayed by proportionally more time. You are prompted for the amount of time to delay each track relative to the previous track.

Samples

A *sample file* is a digital recording of a signal; it stores the instantaneous amplitude versus time of the recorded signal. A sample file usually contains a *header* in a specific *format*; the header contains information about the way the sample was recorded (for example, the sample rate of the recording, the number of channels recorded, and the resolution of the recording). Kyma can work with sample files in the AIFF, SD-I, SD-II, SF/IRCAM/MTU or WAV formats.

The *sample memory* (also called sample RAM) on the signal processor is like the RAM on your computer; it is volatile memory that provides only temporary storage, but offers faster access times and more flexibility than the computer's hard disk. Kyma uses sample memory for delay lines, samples, and *wavetables* (single cycles of standard waveforms, such as sine waves or envelope functions). Sample memory can either be automatically loaded from a sample file stored on the host computer's hard disk, or it can be written in real time by a Sound.

The *sample editor* provides basic cutting and splicing tools for manipulating sample files.

What is a Wavetable?

There are two uses for the word "wavetable" in Kyma. One use refers to the size of a block of memory in the Capybara sample RAM and the other use refers to how a sample file is intended to be used.

When Kyma uses the Capybara sample memory, it allocates the memory in chunks of 4096 samples. These 4096 sample long chunks are called *wavetables*. Standard configurations have either 255 or 1023 wavetables. Sometimes you will see messages referring to the amount of sample RAM in terms of these wavetables.

The parameter names of the Sound modules have been chosen to assist you in understanding how Kyma will use the recording stored in the sample file. Typically, a **Wavetable** field is used to specify a monophonic sample file that contains exactly one cycle of a waveform of exactly 4096 sample points (one wavetable long). Other fields (for example, **Sample**) in which you specify sample files generally assume that the file contains an arbitrary digital recording of arbitrary length. To be sure of the kind of digital recording a specific Sound requires, see the on-line help for that parameter by clicking on the name of the parameter in the Sound editor.

Using Sample Files

Kyma Sounds are not digital recordings. The typical Kyma Sound is a set of instructions telling the Capybara how to generate an audio signal in real time. There are two fundamental advantages to this approach: an instruction list takes up much less memory than the equivalent digital recording, and instructions can be more flexibly modified than digital recordings.

There are times, however, when you will want to create, store and play digital recordings stored in sample files. In Kyma, recording to the hard disk allows you to

- sample live input from the A/D converter or other sources
- record any Kyma Sound as a sample
- export samples to other programs
- build up complex textures through multi-tracking, or
- pre-compute subsections of a Sound that is too complex to compute in real time

Kyma also has the complementary ability to play digital recordings stored on the hard disk, so you can listen to Sounds you have recorded in previous sessions. Because Kyma can recognize sample files in a variety of formats, the program's ability to play disk recordings also allows you to import samples from other sources (CD-ROM sample collections and hard disk recording software, for example).

Exporting, Importing and Post-Processing

Kyma's ability to record and play sample files allows you to exchange files between Kyma and other synthesis and editing environments. For example, you can use Kyma to design Sounds, record the Sounds into sample files, then edit, mix and sequence the sample files using a different program (like a DAW or an audio sequencer). You could also exchange Sounds with a colleague who doesn't have Kyma.

Kyma can exchange sample files with any program that recognizes AIFF (Audio Interchange File Format), WAV (Microsoft Wave), SF/IRCAM/MTU, or SD-I and SD-II (Sound Designer I and II) files.

Regardless of whether your sample files were generated in Kyma or another program, Kyma lets you play them directly from the hard disk using a *DiskPlayer*, *MultifileDiskPlayer*, *KeyMappedMultisample*, or a *DiskSplicer* (all found in the **Disk** category of the system prototypes), or from sample memory using *Sample*, *MultiSample*, or *KeyMappedMultiSample* (all found in the **Sampling** category), or from either the hard disk or sample memory using the *GenericSource* (found in the **Sources** category). You can add reverberation, mix it with Sounds generated in Kyma, do filtering; in short, you can treat these Sounds like any other Sound in Kyma. Then, once you have processed the sample file to your satisfaction, you can record the processed version to the hard disk as a new file.

Playing and Editing

You can play back sample files by choosing **Play...** from the **File** menu and then selecting the sample file you wish to play from the file dialog. To see the files graphically and make minor edits, select **Open...** from the **File** menu and choose **Sample file** as the file type (see *Sample Editor* on page 501 for more information). You can also play sample files using the *DiskPlayer*, *MultifileDiskPlayer*, *DiskSplicer*, *Sample*, *MultiSample*, *KeyMappedMultiSample*, and *GenericSource* Sounds. Alternatively, you can play and edit sample files from the file organizer, see *File menu: File organizer* on page 424.



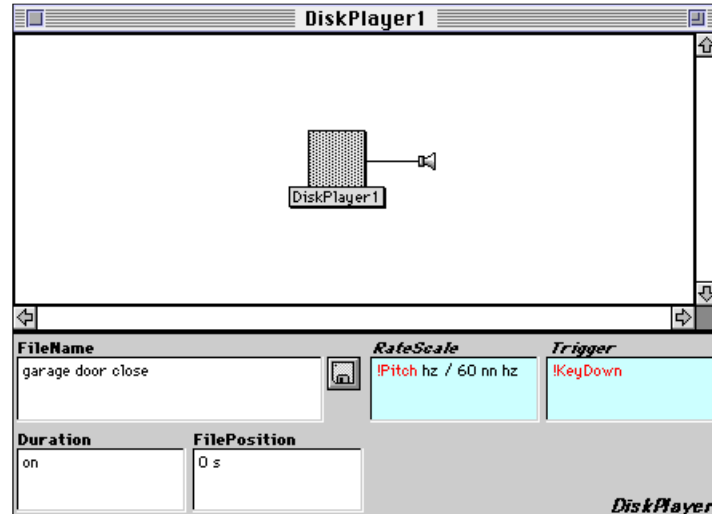
DiskPlayer, *MultifileDiskPlayer*, *DiskSplicer*, and *KeyMappedMultisample* and *GenericSource* (if set to play from disk) play sample files directly from the hard disk of your computer. The number of sample files you can play directly from disk at any one time depends on the speed of your computer's hard disk and disk driver software. (At a 44.1 khz sample rate, you should be able to play three or four simultaneous monaural sample files.) You can, however, build up arbitrarily complex Sounds by recording to disk, playing back from the disk with modifications and additions, and recording the modified Sound into another disk file, ad infinitum (or at least until you run out of disk space).



Sample, *MultiSample*, *KeyMappedMultiSample*, and *KeyMappedMultisample* and *GenericSource* (if set to play from RAM) play sample files from the RAM of the signal processor. Because of this, these Sounds are restricted to playing sample files that can fit in the sample memory. However, when the sample is stored in sample memory, it is possible to play through the sample in different ways, including in reverse and with arbitrary, moveable loop points.

DiskPlayer

You can play back a recording from the hard disk by using a *DiskPlayer* Sound. A *DiskPlayer* can be treated like any other Kyma Sound.

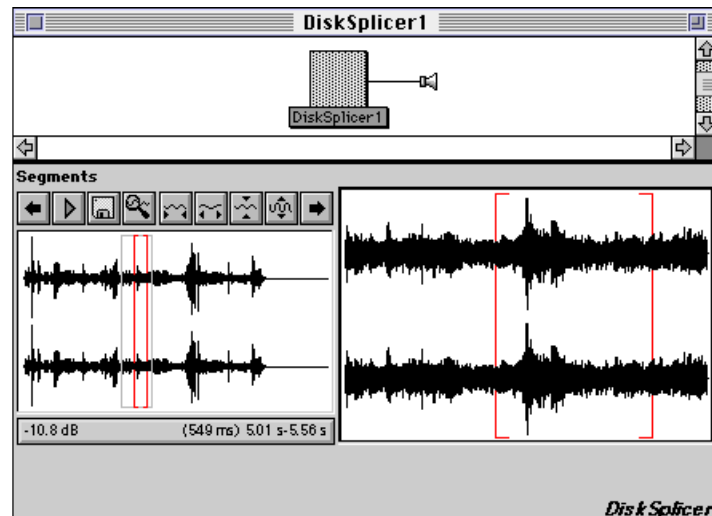


MultifileDiskPlayer

The *MultifileDiskPlayer* is similar to the *DiskPlayer*, except that you provide it with a list of sample files. The **Index** field provides a real time way of choosing which sample file to play when the *MultifileDiskPlayer* is triggered. See the *Prototypes Reference* beginning on page 218, the on-line help, and the examples that come with Kyma for more information about this Sound.

DiskSplicer










Another way to play back sample files is to use a *DiskSplicer*. A *DiskSplicer* Sound allows you to play all or part of a samples file; it also lets you splice together segments from one or more samples files. With a *DiskSplicer* Sound you can perform non-destructive graphic edits on multiple samples files.



Segments Fields and Controls

The left half of the **Segments** field provides an overview of all of the sample files as they are currently spliced together. Click and drag in this part of the field to select the portion to be displayed in the right half of the **Segments** field. Click and drag in the right half of the **Segments** field to make a selection. **Cut**, **Copy**, **Paste**, **Clear**, and **Trim** from the **Edit** menu operate on this selection.

The icon buttons perform actions that apply to the selection (with the exception of the left and right arrow buttons). From left to right, the buttons perform the following actions:

-  scrolls one page of samples to the left
-  plays the currently selected samples
-  pastes a disk file over the selection
-  causes selection to exactly fill the display area
-  contracts the display in the time direction
-  expands the display in the time direction
-  contracts the display in the amplitude direction
-  expands the display in the amplitude direction
-  scrolls one page of samples to the right

Information on the selection is also shown in the field below the overview display. The value on the left displays the magnification of the selection. The values on the right displays the size and range of the selection. Click these fields to change their units.

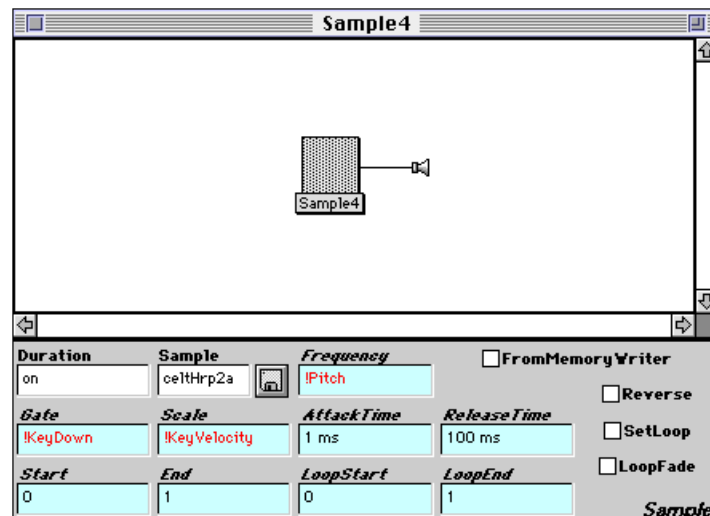
To play a sample file using a *DiskSplicer*, drag one from the system prototypes into a Sound file window and open it. To replace the default samples provided in the prototype with the sample file you wish to play, select the entire right half of the **Segments** field using **Select all** from the **Edit** menu. Click the disk button and select the sample file you want to play from the file list. The samples from this sample file will appear in the **Segments** field.



Unlike a *DiskPlayer*, a *DiskSplicer* does not automatically change the sample rate of a sample file to match that of the current sample rate or to match the sample rates of the other files in the same *DiskSplicer*. For example, if your signal processor is set to a sample rate of 44.1 khz and you splice in part of a file that was originally recorded at a sample rate of 22.05 khz, the spliced section will play back at twice the speed and frequency of the original recording.

Sample, MultiSample and KeyMappedMultiSample

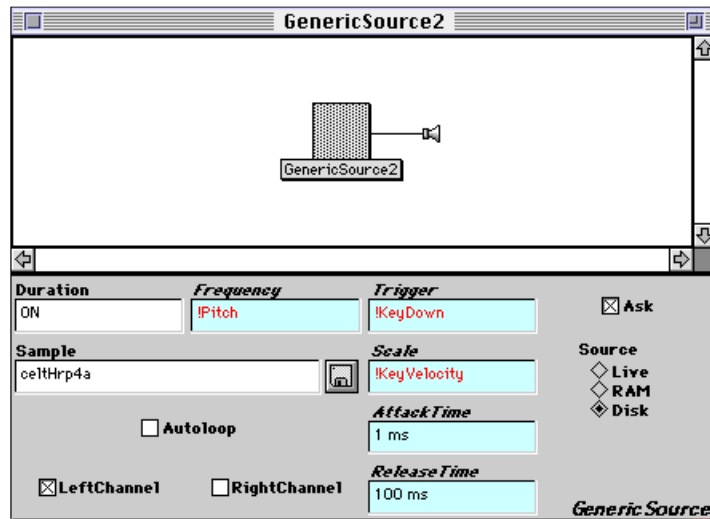
A *Sample* plays back the sample file after first loading it into the sample memory. It can play its sample back in reverse, and it has real-time adjustable loop points.



The *MultiSample* and the *KeyMappedMultiSample* are similar to the *Sample*, except that you can provide a list of sample files. See the *Prototypes Reference* beginning on page 218, the on-line help, and the examples that come with Kyma for more information about these Sounds.

GenericSource

A *GenericSource* can play back a sample file either directly from the hard disk or from RAM after the sample file has been loaded into the sample memory. It can play either or both channels of the sample file, and includes an attack/release envelope.

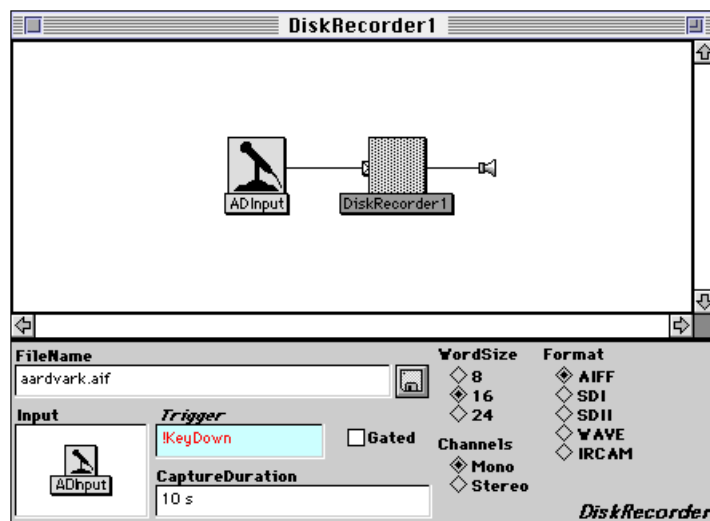


Recording to the Hard Disk

You can make recordings to the hard disk using the tape recorder tool (see *Tools menu: Tape Recorder* on page 442), **Record to disk...** from the **Action** menu (see *Action menu: Record to disk...* on page 437), the *DiskRecorder* Sound, or the *DiskCache* Sound. Like other Kyma Sounds, *DiskRecorder* and *DiskCache* can be placed anywhere in a Sound's structure, allowing you to schedule a recording to disk at a certain time during a Sound. You can also have multiple and simultaneous *DiskRecording* and *DiskCache* Sounds within the same Sound structure.

DiskRecorder Sound

You can use a *DiskRecorder* Sound to make digital recordings. Using a *DiskRecorder* as an input in a larger Sound structure allows you to record to disk at a specific point in time and a specific point in the signal flow diagram within a complex Sound.



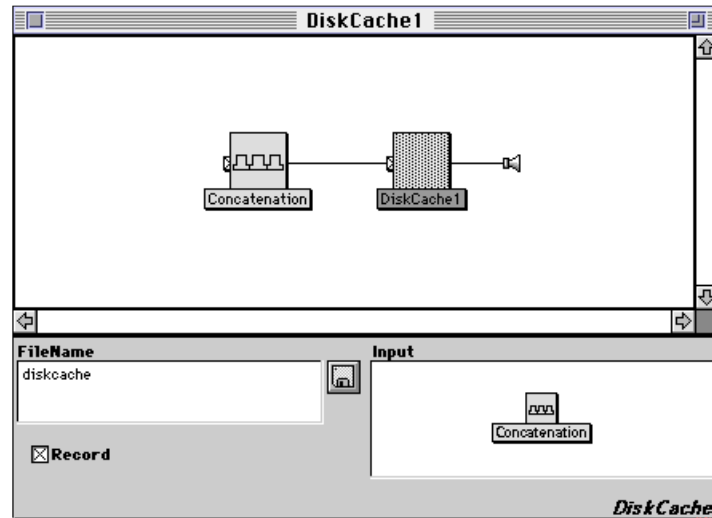
To use a *DiskRecorder*, edit its parameters, make your recording format choices, name the destination sample file, and drag the Sound you want to record into the *DiskRecorder's* **Input** field. Choose 0 s for the **CaptureDuration** if you want to record for the duration of the *DiskRecorder's* input, otherwise,

enter the length of time you want to record. The recording will take place when the **Trigger** becomes positive; if **Gated** is checked, the recording will stop when **Trigger** becomes zero or negative, and will resume when **Trigger** becomes positive again.

DiskCache Sound

If your Sound is too complex to be played in real time, you can ease the computational burden on your Cappybara by replacing part of the Sound with a *DiskCache* Sound. A *DiskCache* computes its input and records it to disk only if **Record** is checked. When the **Record** parameter is unchecked, the *DiskCache* reads the recording stored on the disk.

In the following example, everything to the left of the *DiskCache* has been computed ahead of time and recorded in a disk file. When you play the Sound, the cached part of the structure can be read from the disk rather than generated in real time — thus easing the computational load on the signal processor.



Traditionally, when you make a recording you lose the history of how the recorded sound was created. Using the *DiskCache* in Kyma, you can preserve the structure that you used to generate the samples, making it easier for you to come back later and make changes to the parameters of the generating Sound.

Using Sample Memory

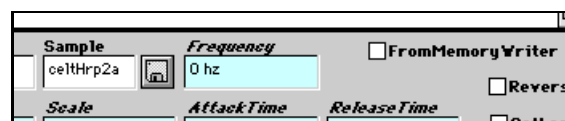
Many Sound modules in Kyma use sample memory in order to carry out their functions. For example, *Sample* plays a sample out of sample memory, and *GrainCloud* uses the grain waveform and the grain envelope stored in sample memory to create its output.

There are several different ways that the Cappybara uses sample memory:

- for storing an entire digital recording
- for storing one cycle of periodic functions
- for storing control parameters
- for delay lines
- to hold the intermediate results of a calculation

In all cases, Kyma manages the memory allocation automatically.

To use a sample file in a Sound, click on the disk button in the parameter field and select the name of that sample from the file list, or simply type the name into the field.



The Kyma Sound compiler allocates a portion of sample memory for every parameter field in which you specify a sample file. The compiler can then do one of several different things:

If the Sound has **FromMemoryWriter** checked, the compiler tries to make this Sound use the same sample memory that a *MemoryWriter* in the same Sound structure and with the same name for its **RecordingName** parameter has written, see *Writing Sample Memory In Real Time*, below.

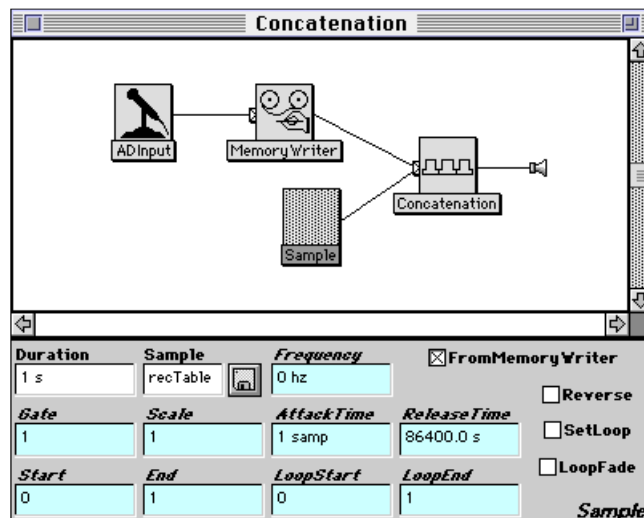
If the parameter field is set to *Private*, the compiler will not allow the sample memory to be shared. Private memory is typically used as delay lines for filters or reverberators, or in other situations where the sample memory is only needed internally for the duration of the Sound.

Otherwise, Kyma assumes the parameter field names a sample file on the hard disk and tries to locate it. (See *How Kyma locates files* on page 431 for more information.) If **Optimize Sample RAM Use** (see *Performance...* on page 430) is on, Kyma will load the sample file onto only the expansion cards that require it; otherwise, Kyma will load the sample file onto all expansion cards. Note that all Sounds that use a sample file on a particular expansion card will share the same sample.

Writing Sample Memory In Real Time

If you are using Kyma as a real-time performance tool, there can be situations in which you might want to capture a sound and play it back later during the performance. You can use a *MemoryWriter* Sound to write the output of any other Sound (including *ADInput*) into sample memory in real time. Once written, that section of sample memory can be read by any Sound that can access sample memory as long as that Sound has **FromMemoryWriter** checked.

In the following example (a *Concatenation* of a *MemoryWriter* followed by a *Sample*), the inputs write to and read from sample memory:



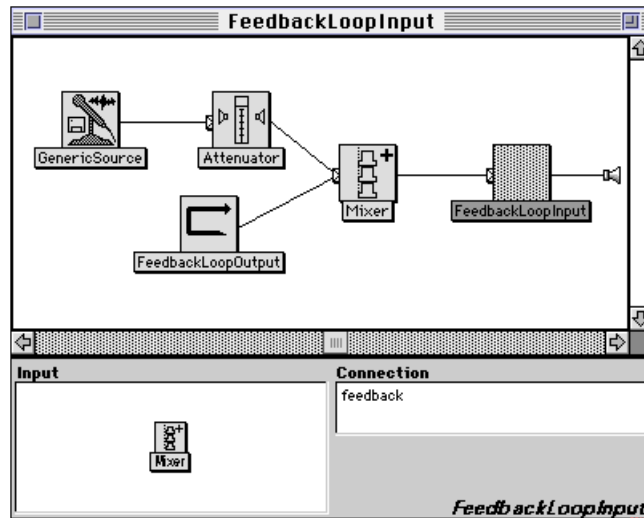
MemoryWriter captures one second of the *ADInput* in a segment of sample memory named *recTable*. After that the *Sample* reads *recTable* for a duration of one second, thus playing back the recording that was just made. When you load this Sound, Kyma reserves one second of time in the sample memory, and it protects this memory until the last time it is read by the *Sample* at time 2 s. Then it releases the memory for use by other Sounds.



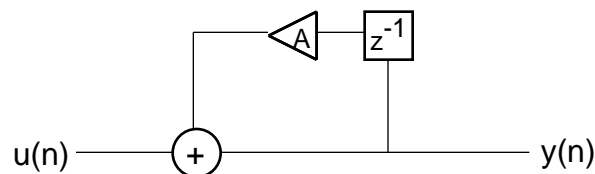
A sample written by *MemoryWriter* Sounds is ephemeral by nature; it is protected in sample memory only for the duration of the Sound(s) for which it is a parameter. When the last Sound to reference that sample finishes playing, the sample memory used by the recording is made available for recycling. To guarantee that a sample persists for longer than the Sound in which it is recorded, use a *DiskRecorder* or a *DiskCache* Sound to write the sample to disk rather than to sample memory.

Feedback

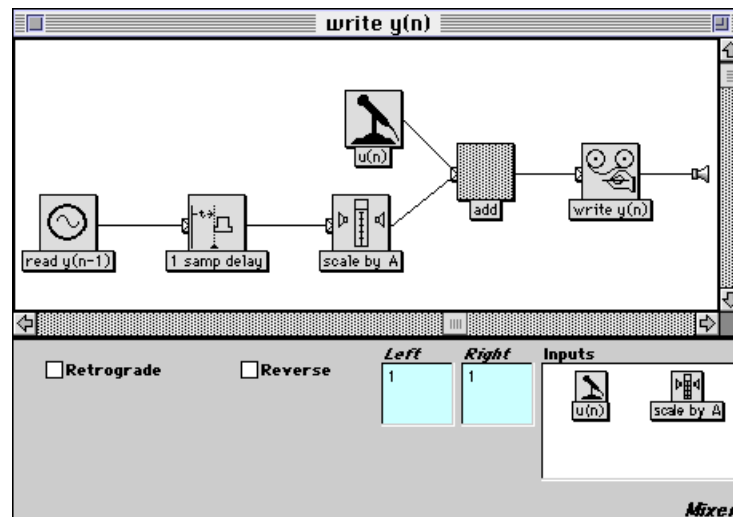
Typically, you use the *FeedbackLoopInput* and *FeedbackLoopOutput* Sounds to feed back the output of a Sound back into its input.



You can also use *MemoryWriters* to create regeneration effects and other types of feedback. For example, to specify the following system:



as a Sound, you could use a *MemoryWriter* to write into the sample memory and an *Oscillator* to read out of that same memory after a one-sample delay:



Sample Memory Warning Messages

Each of the expansion cards in your Copybara has its own sample memory. If a *MemoryWriter* writes into the sample memory on one expansion card and the Sound that reads out of that sample memory cannot be scheduled on the same expansion card, you will get an error message. The easiest solution to

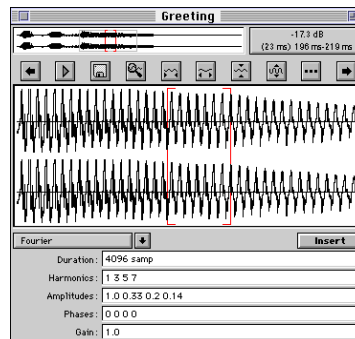
this problem is to write the samples to the sample memory of all the expansion cards by checking **Global** in the *MemoryWriter*.

If you inadvertently try to read from memory before you have written anything there (or if you write to memory and never read what you have written), you will also get a warning message. You can usually solve this problem by putting a *TimeOffset* on the Sound that *reads* the memory.

Sample Editor

The sample editor can be used to edit or create AIFF, SD-I, SD-II, SF/IRCAM/MTU or WAV format sample files stored on the hard disk of your computer.

Select **New...** from the **File** menu, choose **Sample file** as the file type, and press **Enter** or click **OK** to create a new sample file. Select **Open...** from the **File** menu and choose **Sample file** as the file type to open a sample editor on an existing sample file. Alternatively, double-click a sample file name in the file organizer, or click the **Disk** button while holding down the **Command** or **Control** key in a parameter field of the Sound editor.



There are two parts to the editor: the graphic editor (top half) and the generator templates (bottom half). The graphic editor displays a graphic view of the samples versus time and is primarily used for editing what is already there. The generator templates section provides both templates and short Smalltalk programs for generating new samples.






Graphic Editor






The top half of the graphic editor provides an overview of the entire sample file. Click and drag in this part of the editor to choose the portion of the sample file to be displayed in the lower half of the graphic editor. Click and drag in the lower half of the graphic editor to select the samples you wish to manipulate or play. **Cut**, **Copy**, **Paste**, **Clear**, and **Trim** from the **Edit** menu all operate on this selection.

The numbers to the right of the overview provide information on the selection. The upper number tells you the magnification of the waveform shown in the lower half of the graphic editor. The lower numbers tell you the beginning and ending times of the current selection, with the total duration indicated in parentheses. Click on the numbers to switch between dB and linear scales and between seconds and samples.

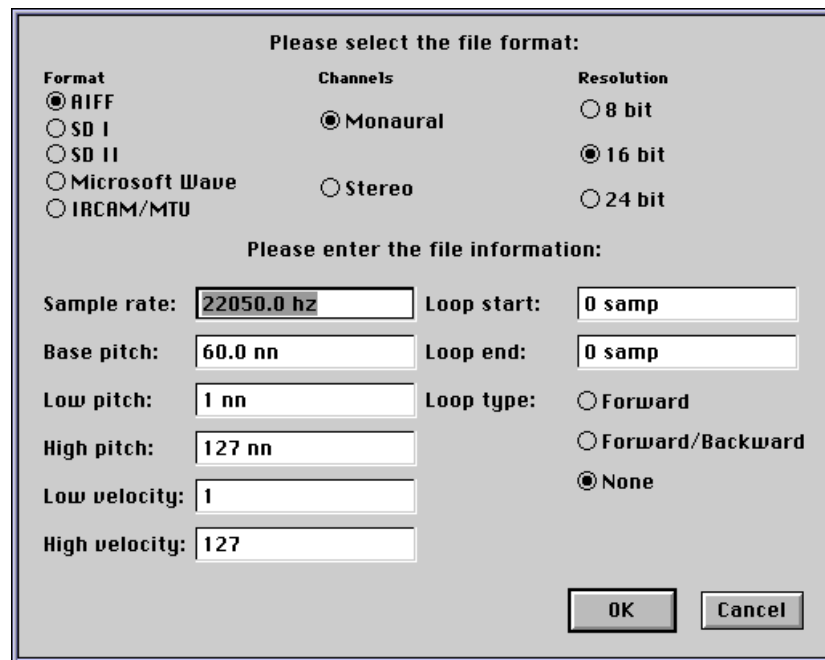
The buttons between the top and bottom sections of the graphic editor perform operations on the selection and control the placement and size of the detailed view in the lower half of the editor.

From left to right, the buttons perform the following actions:

-  scrolls one page of samples to the left
-  plays the currently selected samples
-  pastes a disk file over the selection
-  causes selection to exactly fill the display area
-  contracts the display in the time direction

-  expands the display in the time direction
-  contracts the display in the amplitude direction
-  expands the display in the amplitude direction
-  additional options (e.g. edit header information)
-  scrolls one page of samples to the right

To edit the information in the file's header, click on the additional options button (...) and select **Edit info**.



The dialog box is titled "Please select the file format:". It contains three columns of radio buttons for selection:

Format	Channels	Resolution
<input checked="" type="radio"/> AIFF	<input checked="" type="radio"/> Monaural	<input type="radio"/> 8 bit
<input type="radio"/> SD I		<input checked="" type="radio"/> 16 bit
<input type="radio"/> SD II	<input type="radio"/> Stereo	<input type="radio"/> 24 bit
<input type="radio"/> Microsoft Wave		
<input type="radio"/> IRCAM/MTU		

Below this is a section titled "Please enter the file information:" with several input fields:

Sample rate:	22050.0 hz	Loop start:	0 samp
Base pitch:	60.0 nn	Loop end:	0 samp
Low pitch:	1 nn	Loop type:	<input type="radio"/> Forward
High pitch:	127 nn		<input type="radio"/> Forward/Backward
Low velocity:	1		<input checked="" type="radio"/> None
High velocity:	127		

At the bottom right are "OK" and "Cancel" buttons.

In this dialog, you can change the file's format and the information stored in its header. For example, you can select the start and end times of a sustain loop (this loop information is used by the *Sample Sound*).

Generator Templates

The generator templates generate new waveform segments that replace the selection in the graphic editor when you click the **Insert** button. If nothing is selected in the graphic editor, the new segment will be inserted into the file at the insertion point when you click **Insert**.

The pop-up menu in the upper left of the generator section includes several templates for generating standard shapes or functions. The pop-up list also includes a short Smalltalk program corresponding to each of the templates, so you can modify the code or use the template code as an example or starting point for writing your own function-generating program.

To select a template, click on the arrow to the right of the list. You will see a hierarchical pop-up list. Select the desired template.

The following templates are available in the sample editor.

Buzz

Generates the specified number of harmonic sine waves, starting with the **Lowest Harmonic**. The amplitude of each harmonic is the **AmpBase** raised to the power of the harmonic number. For example, if you specified an **AmpBase** of 0.5, then the amplitude of the third harmonic would be 0.125.

Use **Gain** to adjust the overall amplitude of the waveform until it is close as possible to a range of -1 to 1 without clipping. Test the **Gain** setting by using the **Insert** button; if the inserted segment looks clipped, adjust **Gain** and press **Insert** again.

Cubic Spline

Indicate the total length of the segment in the **Duration** field (4096 samp is the length of a standard wavetable). Enter a list of points that the spline should pass through (range is -1 to 1). Then indicate the proportion of the total duration taken up by each segment. The number of segments should always be one less than the number of end points.

Exponential Segments

Indicate the total length of the waveform in the **Duration** field (4096 samp is the length of a standard wavetable). Enter a list of end points for the exponential segments (remember that exponential functions are undefined at 0, so keep the values between 0 and 1). Then indicate the proportion of the total duration taken up by each segment. The number of segments should be one less than the number of end points.

Fourier

This lets you specify a waveform by specifying its component sine waves. Enter the list of desired harmonics by number (where 1 is the fundamental, 3 is the third harmonic, etc.). Then enter the relative amplitudes of each harmonic (range of -1 to 1), and the relative phase (in radians) of each harmonic. To specify π , use `{Float pi}`. Do any arithmetic on `pi` within curly braces, e.g., `{2 * Float pi}`. For waveforms requiring a large number of partials and/or to get inharmonic partials, modify the Fourier Program template (found in the hierarchical list under Programs).

Impulse Response

This generates the impulse response of a filter that you specify. Set the number of formants and the center frequency, bandwidth, and relative amplitude of each formant of the filter. Use **Gain** to adjust the overall amplitude to be as close as possible to the full range (-1 to 1) without clipping. You may have to try **Insert** several times to adjust the **Gain** to the optimum level.

Interpolate Selection

This is used to time-stretch or time-compress the selection. It can be used, for example, to take a single cycle from a recording and stretch or compress it to exactly 4096 samples so that it can be used as the wavetable of an *Oscillator*. You can also use this to shift a sample down in frequency without aliasing (by time-stretching it).

The selection will be *sinc*-interpolated to the length specified in the template parameters. **Gain** is used to scale the samples. **Interpolation Points** controls how many points are used in the *sinc* interpolation; use larger numbers than the default for better results, especially when changing the length drastically. (This operation can take a long time).

Line Segments

Indicate the total length of the waveform or function in the **Duration** field (4096 samp is the length of a standard wavetable). Enter a list of end points for the linear segments (in the range of -1 to 1). Then indicate the proportion of the total duration taken up by each segment. The number of segments should be one less than the number of end points.

Normalize Selection

This scales all amplitudes in the selected portion of the sample to the maximum indicated in the **Amplitude** field.

Polynomial

This generates the polynomial whose x values range over the interval you provide and whose coefficients are supplied in the **Coefficients** field. **Polynomial** is particularly useful for generating wavetables

that can be used as waveshapers by the *Waveshaper* Sound. For example, if you typed in the coefficients 5, 4, and -2.5 you would be specifying the polynomial

$$4 + 4x - 2.5x^2$$

If the interval you enter is -1 to 1, then the editor will evaluate this polynomial for x values ranging from -1 to 1.

Random

This generates a random waveform whose amplitude ranges from -1 to 1. Specify a seed so that you can get repeatable results. Change the seed when you want to generate a different random waveform. To generate different kinds of noise, modify the **Random Program** template.

Window Selection

This template applies a fade-in and fade-out curve to the selection in the graphic editor. The **Transition Duration** is the duration of the fade in and fade out.

Viewing the Template Code

The hierarchical list of templates also contains the subheading **Programs**, which contains Smalltalk code for each of the template types. If you like a particular template but want to make a minor change in the algorithm, you can modify (but not save) its Smalltalk implementation. If you are not sure you understand one of the templates, you can read its Smalltalk code to see what it does.

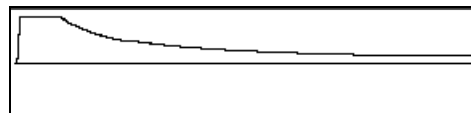
If you want to write your own Smalltalk code for generating a wavetable, you can use the existing Smalltalk programs as models or starting points for your own code. To save your modified wavetable generator, copy the text and paste it into a newly created text file.

Wavetables Provided with Kyma

Compress

When used in a *MultiplyingWaveshaper*, this waveform can be used to do computationally inexpensive compression and expansion. Think of the curves as output-attenuation-as-a-function-of-input-amplitude. To create new curves, use the `InputOutputCharacteristic` Program template in the sample editor.

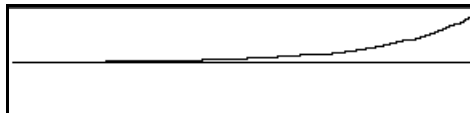
c5_01_1



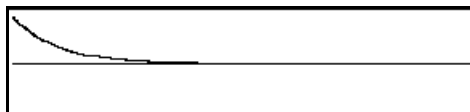
Control

These are typically used in *FunctionGenerators* to generate envelopes or “control signals” for hot parameters. If you want a periodically repeating control signal, use one of these samples in a low frequency oscillator. To generate your own, use any of the templates in the sample editor or modify template code to compute an arbitrary mathematical function and store it as a sample.

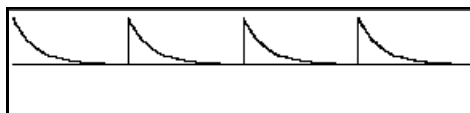
Exponential

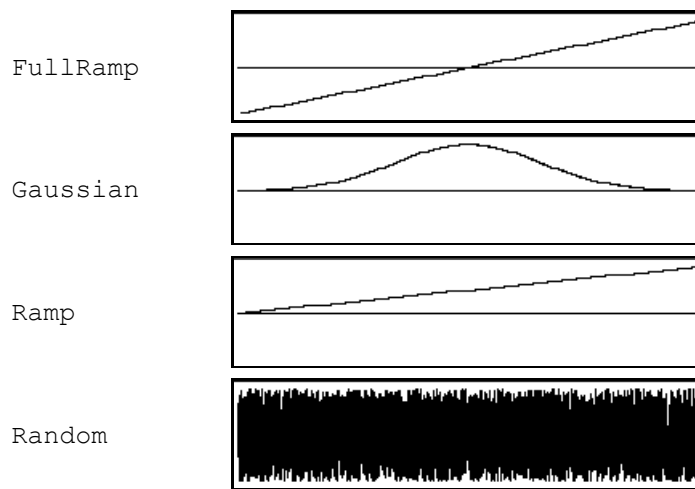


ExponRev



FourExpons



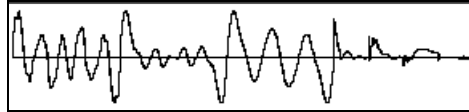


EX

The RE analysis tool separates a sample into two parts: a time-varying filter (the resonator) and the signal (the excitation) to feed through this filter to recreate the original sample. These files are the excitations from various samples, and can be used as an input to the *REResonator* Sound.

GA

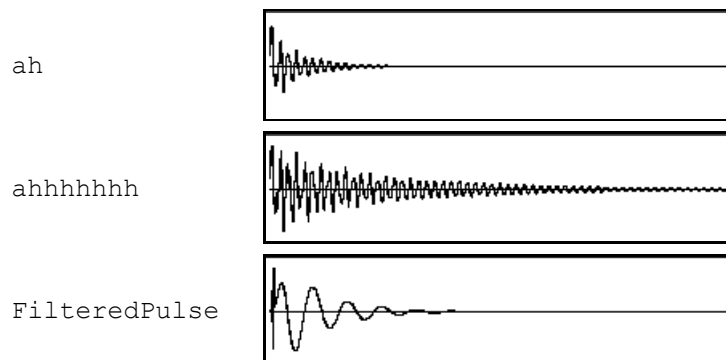
The GA (group additive) analysis tool reduces a spectral analysis file (consisting of potentially hundreds of sine waves with different frequency and amplitude envelopes) into a smaller set of more complex waveforms with corresponding amplitude envelopes. These files are derived from various musical instrument samples, and can be used in the **Analysis0** or **Analysis1** parameter in the *GAOscillators* Sound.



This is a display of a GA file for a harp pluck. The GA analysis tool reduced a spectral analysis file with 128 partials into three complex waveforms (shown above in the left two-thirds of the display), three corresponding amplitude envelopes, and a frequency deviation envelope (all shown in the right one-third of the display above).

Impulse Responses

These files contain the responses of various filters to an impulsive input. They are typically used in *FunctionGenerators*, *Samples*, or *Oscillators*.



oo



Internal

These are used internally by Kyma Sounds. You don't want to delete or modify any of these tables or some of the Sounds will stop working!

Cosine	Inverse	Kym3	SpecLPF2	SpecLPF8
FTrkrTbl	Kym1	MiscTbls	SpecLPF4	VocodExp
Greeting	Kym2	SOSExp	SpecLPF6	Zero

Iterated Functions

These files can be used with the *IteratedWaveshaper* Sound to perform Functional Iteration Synthesis.[§] These files can also be used with *Waveshaper* to create variable-width oscillators.

RE

The RE analysis tool separates a sample into two parts: a time-varying filter (the resonator) and the signal (the excitation) to feed through this filter to recreate the original sample. These files are the time-varying filters from various samples, and can be used as the **Wavetable** for the *REResonator* Sound.

Samples

These digital recordings are typically used in a *Sample* or in a *DiskPlayer* Sound. Some of these files are grouped in folders for use directly in *KeyMappedMultiSample*.

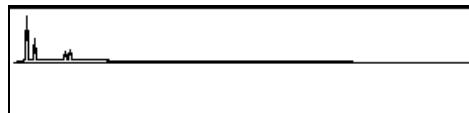
Spectra

These files are used by the *SpectrumInRAM*, *SpectrumOnDisk*, and *SumOfSines* Sounds. Each of these files represents a spectral analysis of a sample. When you use one of them to control an *OscillatorBank*, you can resynthesize the original signal through additive synthesis. To generate your own spectral analyses, see *Tools menu: Spectral Analysis* on page 443.

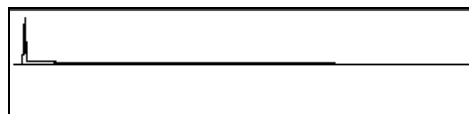
Spectral Shapes

These are typically used in *SpectralShape* Sounds and then used to control the spectrum of an *OscillatorBank*. Think of these as showing amplitude as a function of frequency.

Aformnts



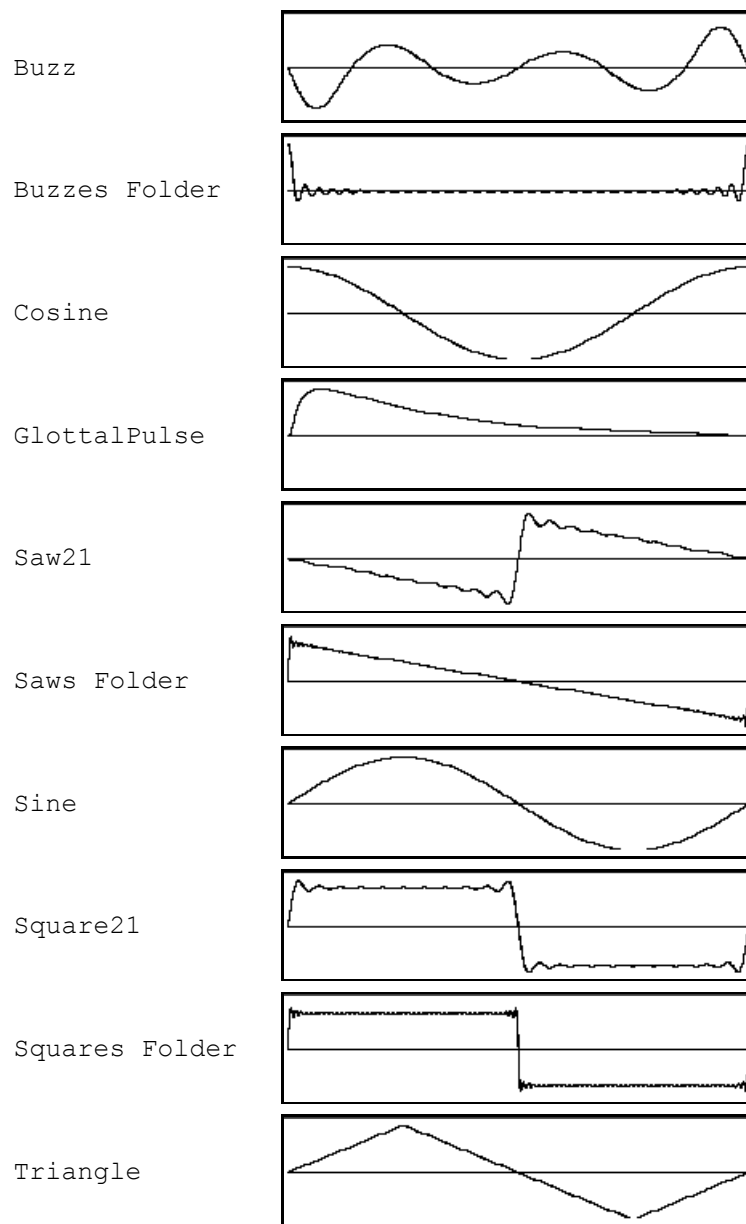
BPF500



Waveforms

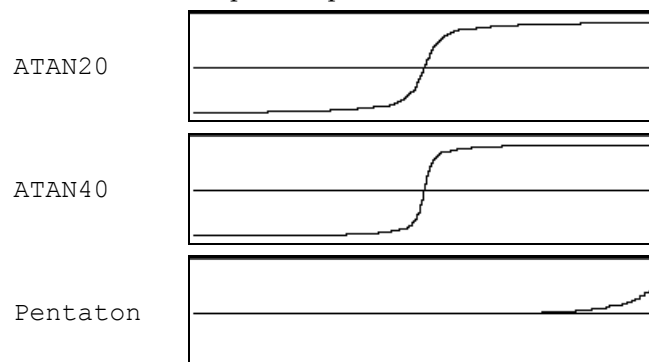
These are typically used in *Oscillators* but can also be used in *Samples* with looping turned on. Think of them as single cycles of periodic waveforms. The files in the *Buzzes*, *Saws*, and *Squares* folders contain band-limited approximations to those waveforms, each constructed using a different number of harmonics; these files are appropriate for use with the *KeyMappedMultiSample*.

[§] See Agostino Di Scipio's paper for information about Functional Iteration Synthesis.

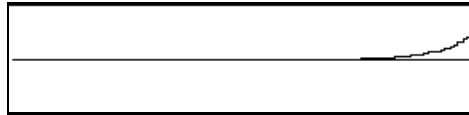


Waveshapers

These are typically used a *Waveshaper* Sound as the shaping function. Think of these shapes as output-sample-as-a-function-of-input-sample.



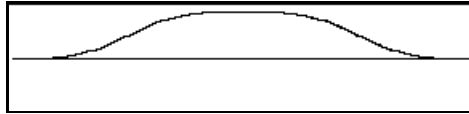
Pythag



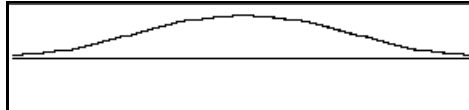
Windows

These are symmetric waveshapes that can be used in *FunctionGenerators* or *Oscillators* as control signals or amplitude envelopes. They can also be useful in other Sounds that have window parameters like *SampleCloud*, *GrainCloud*, *FormantBankOscillator*, and the *SpectrumAnalyzerDisplay*.

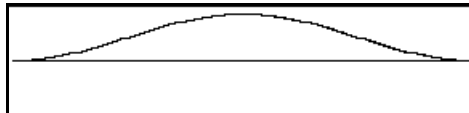
BristowJohnson



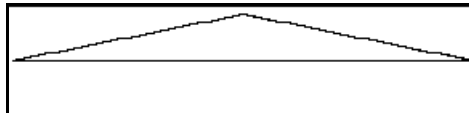
Hamming



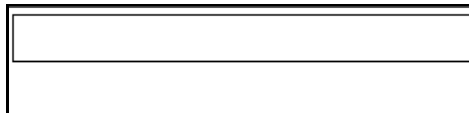
Hann



LinearEnvelope



Rectangular



Advanced Topics

Variables

By introducing variables, we can generalize one particular Sound to describe a whole class of Sounds. Such an abstracted Sound is called a *lifted* Sound, so-called because it is at a “higher” level of abstraction than is any specific Sound.

A single lifted Sound represents an infinite number of real Sounds. Lifting a Sound by introducing variables is similar to introducing variables into an arithmetic expression. While $4 + 5$ is just one expression, $x + y$ represents a whole class of expressions — all possible sums of two numbers.

There are two places you might want to use variables: when encapsulating a complex Sound (see *The Class Editor* on page 536), or in a *Script* or in one of the various *FileInterpreters* (see *Scripting* on page 522).

Kyma Variables

In Kyma there are three kinds of variables: variables, *Variable* Sounds, and *SoundCollectionVariables*. A variable is used to represent values you can type into parameter fields, a *Variable* Sound acts as a place holder for Sounds, and a *SoundCollectionVariable* represents a collection of Sounds.

Variables

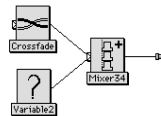
The typeable parameters of a Sound can be represented by variables or expressions involving variables. To assign a variable to a parameter, type a name for the variable preceded by a question mark:



The variable will be displayed in green in the parameter field.

Variable Sounds

A *Variable* functions as a place holder for an actual Sound that may be assigned to it later on. You can use a *Variable* anywhere you would use any other Sound.

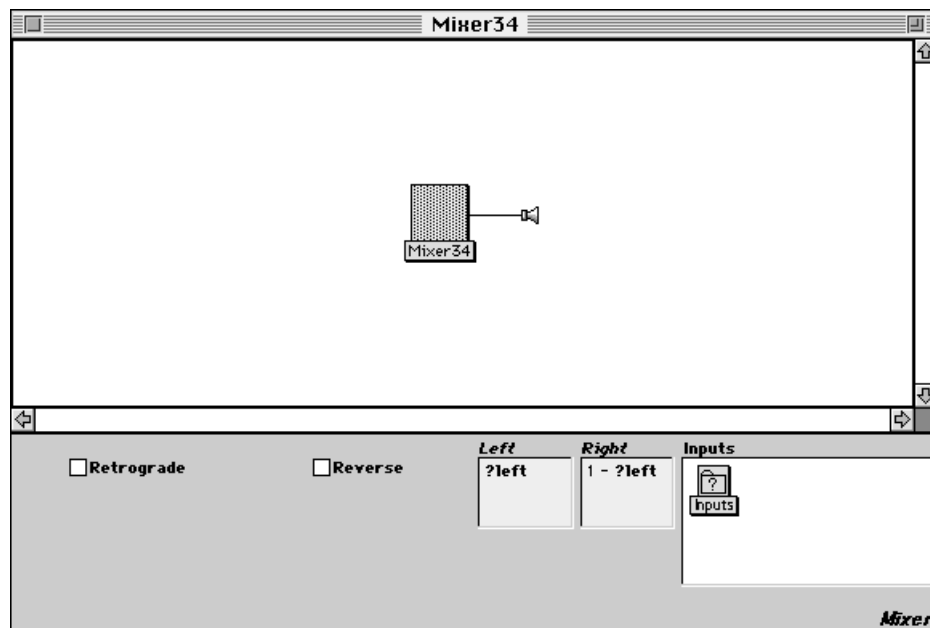


SoundCollectionVariables

Sometimes you may want to set an entire collection of Sounds (rather than the individual Sounds of a collection) to a variable — for example, in the **Inputs** field of a *Concatenation* or a *Sum*. To do this, drag *SoundCollectionVariable* from the system prototypes into the **Inputs** parameter field. The *SoundCollectionVariable* acts as a place holder for a collection of Sounds. It can be placed only in fields that accept more than one Sound.

You can use *SoundCollectionVariables* in defining a new kind of Sound that can take multiple inputs (see *The Class Editor* on page 536 for more information). You can also use a *Script* to set the value of a *SoundCollectionVariable*. Its value should be a collection of Sounds (for example, an Array). See *Scripting* on page 522 and *The Smalltalk-80 Language* on page 513 for more information.

Like the *Variable* Sound, you can refer to a *SoundCollectionVariable* by name elsewhere in the structure. See *The Smalltalk-80 Language* on page 513 or one of the recommended Smalltalk texts listed there for messages that you can send to collections.



Despite the fact that a *SoundCollectionVariable* is not a Sound (it represents a collection of Sounds), it will appear as a Sound in the Sound structure, and you can edit its name just as you would edit a Sound's name.

Syntax

Variables typed into parameter fields are indicated by a leading ? character followed by a string of numbers or letters; the first character in the string must be a letter. Variable names should contain no internal spaces. Variables are displayed with a green color in the Kyma user interface.

The names of *Variables* and *SoundCollectionVariables* are not preceded by question marks, since it is clear from the context that these are variables. Otherwise, they should follow the same syntactical guidelines as typed variables.

Some examples of legal variables are:

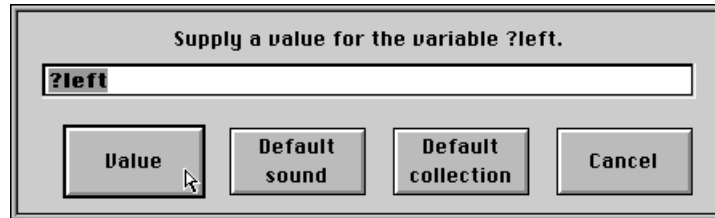
```
?freq
?formant2
?theDaysAreAhead123456789
?syntheseVirtuelle
?fumoBianco
```

and some illegal examples would be:

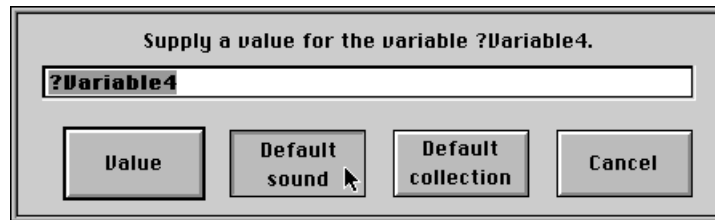
```
gaaf
?20Jahrhunderts
?table Lookup
```

Playing a Lifted Sound

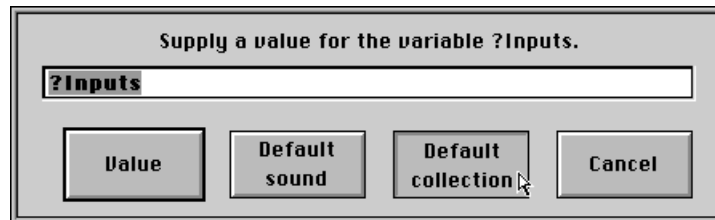
When you play a lifted Sound, Kyma prompts you to supply a value for each of the variable parameters. Before you can hear the lifted Sound, a specific value must be assigned to each variable.



If the requested value is a number, string, symbol, or array, enter the value and click **Value** or press **Enter**.



If the requested value is a Sound, click the button labeled **Default Sound**.



If the requested value is a Sound collection, click the button labeled **Default Collection**. Use **Set default Sound** and **Set default collection** from the **Action** menu to set the selected Sound or collection of Sounds to be the defaults used by these dialogs. The default Sound should always be a single Sound. The default Sound collection can be a set of one or more Sounds or collections.

Environments

In the Sound file window, you will be asked to supply values for the variables each time you play a lifted Sound. In the Sound editor, you only have to supply values for the variables once. The Sound editor remembers the associations of variables to values in what is called the *environment*. To see the state of the current environment, select **Environment** from the **Info** menu.

If you want to change the values associated with the variables, select **Reset environment** from the **Info** menu. The next time you play the Sound, Kyma will ask you to supply values for the variables again.

The Smalltalk-80 Language

This chapter is intended to provide you with just enough background to get you started in writing your own algorithmically constructed Sounds. Once you start using Smalltalk extensively, you should keep the book *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg and David Robson around for reference.

Learning Smalltalk

If you have never programmed before, Smalltalk is an excellent first programming language. It has a simple syntax and reflects a natural way of thinking about problems, the properties of objects, and the interactions between objects. If you are already familiar with a procedural programming language like C or Pascal, you will find that, once you spend a little time familiarizing yourself with the syntax, Smalltalk is not very different from the languages you are using now.

Smalltalk has its roots in the work of the Learning Research Group at the Xerox Palo Alto Research Center in the early 1970's. The mandate of this group was to come up with ways in which people of different backgrounds might effectively and joyfully make use of computers. This group later evolved into the Software Concepts Group (SCG). The goal of the SCG was to design a system that could grow as its user's knowledge of the system grew.

Sending Messages to Objects

You can think of a Smalltalk program as a miniature universe of your own creation. The universe contains objects that you, as the master of the universe, can order around. For example, suppose the universe contains the object 10 and you wanted that object to negate itself. You would evaluate the following statement:

```
10 negated
```

This is an example of a *message-send*; the message is `negated` and the receiver of the message is the object 10. This particular kind of message-send is called a unary message send, since there is only one receiver and one message. You can test the effects of this statement by typing it into the **Script** field of a **Script**, highlighting it, and then selecting **Evaluate** from the **Edit** menu.

If a message has particular values associated with it, it is called a keyword message and consists of pairs of keywords and values separated by colons. For example, if you wanted the string object 'smalltalk' to tell you the value of its third character, you would type

```
'smalltalk' at: 3.
```

Try evaluating this in the script field of a **Script**.

A keyword message can consist of any number of *keyword:value* pairs, for example

```
'smalltalk' copyFrom: 3 to: 5.
```

The simplest examples of *binary messages* are the binary operations of arithmetic. For example

```
3 + 5
```

is technically a keyword `+` with the value 5 sent to the receiver 3. This would be an awkward way to describe something as straightforward as the addition of two numbers, so it is simply referred to as a binary message instead.

Precedence

Messages are cumulative. For example the message-send,

```
60 nn hz
```

would first send the message `nn` to the object 60. The result would be a **FrequencyInPitch** object. Then the message `hz` is sent to the **FrequencyInPitch** object. The result is a **FrequencyInHertz** object. Try evaluating this in the script field of a **Script**.

Binary messages and keyword messages are also cumulative, but they are evaluated in an order defined by the following precedence rules:

Unary

Binary

Keyword

You can override the precedence rules by placing an expression within parentheses. For example

```
2.0 raisedTo: 13 + 3 negated
```

evaluates to 1024, whereas

```
((2.0 raisedTo: 13) + 3) negated
```

evaluates to -8195.

When in doubt, use parentheses to make your intentions clear.

Comments

A comment is some explanatory text that is not evaluated as part of the program. In Smalltalk, comments are delineated by enclosing them within double quotes. It is important to include comments explaining each logical step of your program. Not only does this make it easier for others to understand your code, but it makes it easier for you to remember why you chose to do things the way you did when you look at your programs later.

Variables

Smalltalk variables must begin with a letter and contain only alphanumeric characters. It is common practice to begin a variable name with a lower-case letter and to indicate multi-word variable names by capitalizing the first letter of each new word. For example,

```
r
x2
tacuma
mrJavelina
theMainEvent
darthBara
tryToEnjoyTheDaylight
```

are all typical variable names in Smalltalk.

Smalltalk variables are distinct from the Kyma variables described in *Variables*. The scope of a Smalltalk variable extends only as far as the parameter field it is contained in. For example, variables declared in a **Script** field of a *Script* Sound are valid only within that **Script** field.

Variables are declared at the beginning of a program by listing them between vertical lines, for example

```
| r x2 hybris terminalLight gaaf |
```

is a variable declaration.

To assign a value to a variable, type the variable name, a space, the assignment symbol `:=`, another space, and then the expression whose value is to be assigned to the variable, followed by a period. For example,

```
| r x2 aSound darthBara |

r := Random newForKymaWithSeed: 1032.
x2 := r next.
darthBara := aSound frequency: x2 * 1000 hz.
```

Numbers

Integers, for example,

7 -9

are written as numerals with no decimal points.

To express an integer in another radix, for example as a binary or hexadecimal number

2r1010 16rEFFF

type the base, followed by the letter `r`, followed by the number expressed in that radix. For example, a number expressed in base 2 is a binary number and contains only 1s and 0s. For radix numbers greater than 10, any numbers represented by letters can be in upper or lower case (e.g. 16rFFF or 16rfff).

Sending the message `printStringRadix:` to a decimal integer will give a string that contains that number expressed in the given base. For example:

17 printStringRadix: 2

would evaluate to `'10001'`, which is a string that contains the binary representation of 17.

Floating point numbers are expressed as numerals with decimal points, for example,

1.3 0.1611

Floating point numbers can also be expressed in scientific notation (the following example expresses 1.14×10^{10} in scientific notation):

1.14e10

Like integers, a floating point number in another radix is preceded by the radix and an `r`, for example,

2r10.1

There must be a numeral before the radix point, even if that number is a zero. For example, the number

.5

must be written as

0.5

or it will not be interpreted correctly. In Smalltalk, a period indicates the end of a sentence. A decimal point that is not preceded by a zero is ambiguous; it could be the period terminating a previous statement, rather than a decimal point. By always making the leading zero explicit, you remove this ambiguity.

The number

0.991 % 1

is an example of a complex number in Smalltalk; the first number is the real part, and the second number is the imaginary part.

Arithmetic Operations

Besides the usual operations of addition, subtraction, multiplication, and division (+, -, *, and /), here are some of the arithmetic operators you can use in Smalltalk:

//	divide and truncate the result towards negative infinity
\	mod
**	raise to a power
abs	absolute value
inverse	1 divided by receiver
quo:	divide and truncate the result towards zero
rem:	like mod except truncated towards zero
rounded	add 0.5 and truncate
truncated	round towards negative infinity
vmax:	take the maximum of receiver and argument
vmin:	take the minimum of receiver and argument

For example,

-7 //	3	-3
7 //	3	2
-7 \	3	2
7 \	3	1
2 inverse		1/2
-7 quo:	3	-2
7 quo:	3	2
-7 rem:	3	-1
7 rem:	3	1
-2.5 rounded		-2.0
-2.5 truncated		-3
5 vmax:	2	5
5 vmin:	2	2

Mathematical Functions

The following are some of the more common mathematical functions that are available in Smalltalk:

ln	natural log
exp	exponent
sqrt	square root
sin	sine
cos	cosine
normCos	(receiver *) cos
normSin	(receiver *) sin
log	log of receiver base 10
twoLog	log of receiver base 2
twoExp	two raised to the receiver

The expression

```
2 raisedTo: 10
```

means two raised to the tenth power.

For more arithmetic operations, mathematical functions, and bit-level operations, see one of the referenced texts.

Other Literals

Smalltalk constants are called *literals*. Numbers, symbols, characters, strings, and arrays are all literals.

A single character is prefaced by a dollar sign, for example

```
$T
```

Strings of characters are delimited by single quotes, for example,

```
'this is a string'
```

You can concatenate two strings by typing a comma between them, for example,

```
'first half', 'second half'
```

evaluates to the string

```
'first half second half'
```

A symbol is prefaced by a number sign and cannot include any internal blanks, for example,

```
#thisIsASymbol
```

You can create a symbol from a string by sending it the message `asSymbol`, for instance,

```
'thisIsAString' asSymbol
```

Two strings of the same characters are not necessarily the same object. However, two symbols with the same name are actually the same object. Each symbol is unique in the system.

Arrays are enclosed within parentheses and prefaced by a number sign, as in

```
 #(3 5 6 #anArray) .
```

Each object within the array is interpreted as a literal, thus the contents of the array

```
 #(2 $+ 5 #anArray)
```

would be the number

```
 2
```

followed by the character

```
 $+
```

the number

```
 5
```

and the symbol

```
 #anArray
```

Leading number and dollar signs are not required within an array literal. For instance, the array defined earlier could be equivalently written as

```
 #(2 + 5 anArray)
```

Suppose you wanted to evaluate the expression `2 + 5` before putting it into the array and that `anArray` was actually a variable. Then you would need to create the array as in the following example:

```
| anArray |
anArray := #(a b c) .
anotherArray := Array with: 2 + 5 with: anArray.
```

The value of `anotherArray` is

```
 #(7 (a b c))
```

Alternatively, you can place expressions to be evaluated within curly braces:

```
#({2+5} # (abc) )
```

Booleans

A boolean is one of two values: `true` or `false`. There are two boolean literals:

```
true
```

and

```
false
```

Some examples of boolean expressions include:

```
(2 + 2) = 5
```

```
i >= 2
```

```
aSet isEmpty
```

```
file atEnd
```

Logical Operations

Boolean expressions can be combined using the logical operators AND and OR.

There are two different kinds of logical operations for both AND and OR. One type evaluates only as much of the expression as is necessary to determine whether the entire expression is true or false; in this type of expression, the second alternative is placed within a block, for example:

```
anArray isEmpty or: [(anArray at: 1) = 0]
```

```
aStream atEnd not and: [aStream next == 'a']
```

The other type always evaluates both arguments of the expression; in this type of expression, the two arguments are separated by an ampersand for AND or a vertical bar for OR, for example:

```
t > 2.5 & (t < 550.0)
```

```
r next > 0.5 | (r next = 0.0)
```

The following expression is true if `anArray` is empty *or* if its first entry is 0. If the first expression is true, then the expression within square brackets is not evaluated, since the entire expression is true if either of the subexpressions is true. By using this form of the OR, you are protected against trying to access an empty array.

```
anArray isEmpty or: [(anArray at: 1) = 0]
```

In the next example, the OR operation is represented by a vertical bar. Both expressions are evaluated no matter what the value of the first expression. In this example, the message `next` returns the next value in a stream of values. It also has the side effect of advancing the pointer into the stream. In cases where an expression has a side-effect, you may want to evaluate both expressions even when the first is true.

```
anArray isEmpty | (aStream next = 0)
```

In the following example of the AND operator, the second expression is evaluated only if the first expression is true. In the case of this example, you wouldn't want to try accessing the `i`th position of `anArray` unless you were sure that `anArray` had at least `i` positions in it.

```
(anArray size >= i) and: [(anArray at: i) = 10]
```

The ampersand operator is an AND in which both expressions are evaluated even if the first is false. In this particular case, the message `removeFirst` returns the first element of `aCollection`, but it also has the side effect of *changing* `aCollection` by removing the first element.

```
i > 10 & (aCollection removeFirst = 0)
```

Blocks

A *block* is a *deferred* sequence of instructions having the form

```
[ {blkArg}* |
    | {blkVar}+ |
    {blkStmnt}*]
```

where `blkArg` is an optional list of formal arguments to the block, `blkVar` is an optional declaration of variables local to the block and `blkStmnt` is a series of Smalltalk statements. The Smalltalk statements within a block are *not executed* until you send it one of the variants of the message `value`. In the following code

```
| incrBlk i result |

i := 0.
incrBlk := [i := i + 1].
result := incrBlk value.
```

the variable `result` ends up with the value 1. Block arguments are declared by typing them preceded by colons at the beginning of the block and separating the declaration from the rest of the code by a vertical bar. If the block has arguments, each one must be assigned a value in the evaluation message. In the following code,

```
| sumBlk result |

sumBlk := [ :a :b | a + b].
result := sumBlk value: 3 value: 2.5.
```

the variable `sumBlk` contains a block of code that will return the sum of its two arguments (`a` and `b`) when it receives the `value:value: message`. The variable `result` ends up with the value 5.5.

Loop Structures

The most common use of blocks is to delineate sequences of instructions that are to be executed conditionally or repetitively.

To simply repeat a sequence of instructions, send the message `timesRepeat:` to the integer number of repetitions desired. For example,

```
4 timesRepeat: [
    aSound
    start: 0 s
    freq: 100 hz
    dur: 1 s].
```

would start up four copies of *aSound*, all at the same time and having the same frequency and duration. (Since *aSound* is being played four times with the same start time and parameters, the result will sound like *aSound* being played with an amplitude four times larger.)

To iterate over an collection of values, send the message `do:` to the collection of values. For example, in

```
0 to: 3 do: [ :i |
    aSound
    start: i s
    freq: ((i + 1) * 100) hz
    dur: 1 s].
```

the block argument `i` takes on the values 0, 1, 2, and 3. This would start up four copies of *aSound* with the respective frequencies 100, 200, 300, and 400 hz, each starting one second after the other. (Refer to one of the books listed at the end of this appendix for more examples of what you can do with loop structures and collections in Smalltalk.)

To iterate as long as a condition holds true, send the message `whileTrue:` to a block that evaluates to true or false. For example,

```
| i |
```

```

i := 0.
[i <= 3] whileTrue: [
    aSound
        start: i s
        freq: ((i + 1) * 100) hz
        dur: 1 s.
    i := i + 1].

```

is functionally equivalent to the previous example.

Conditionals

To conditionally evaluate a block, send the message `ifTrue:` and/or `ifFalse:` to a boolean expression. For example, the statements

```

0 to: 5 do: [ :i |
    (i \% 2 = 0)
        ifTrue: [evenSound start: i s]
        ifFalse: [oddSound start: i s]].

```

sets `i` to the values 0 through 5. Whenever `i` modulo two is 0, `i` must be an even number. On iterations when `i` is an even number, *evenSound* starts at `i s`, and on iterations when `i` is an odd number, *oddSound* starts at `i s`. The result is an alternation between *evenSound* and *oddSound*.

If the boolean expression includes Kyma variables, use the messages `varIfTrue:ifFalse:`, `varIfFalse:ifTrue:`, `varIfTrue:`, or `varIfFalse:`. For example, if you wanted the attack time of an envelope to depend upon a variable `?dur`, you might use

```

(?dur > 1 s)
    varIfTrue: [1 s]
    ifFalse: [0.1 * ?dur]

```

as the value of the attack parameter.

Random Numbers

To obtain random numbers in Smalltalk, you create an object that will supply you with an endless stream of random numbers. To create a random number generator use

```
Random newForKymaWithSeed: 3413.
```

If you assign the random number generator to a variable, you can ask it for the next random value by sending it the message `next`.

```

| r |

r := Random newForKymaWithSeed: 1002.
4 timesRepeat: [r next].

```

generates four random numbers between zero and one. The following statements,

```

| r |

r := Random newForKymaWithSeed: 2530.
1 to: 100 do: [ :i |
    aSound
        start: i s
        freq: ((r next * 100) + 500) hz].

```

creates a sequence of 100 instances of *aSound*, each with a randomly selected frequency between 500 and 600 hz.

Set the seed to an integer value from 0 to 65535, so the Sound will be repeatable.

For Further Information

This should give you enough of the basics so that you can get started. For more depth of information on Smalltalk, consult the following texts:

Smalltalk-80: The Language and Its Implementation by Adele Goldberg and David Robinson, published by Addison-Wesley.

Inside Smalltalk, Volume I by Wilf R. LaLonde and John R. Pugh, published by Prentice-Hall.

An Introduction to Object-Oriented Programming and Smalltalk by Lewis J. Pinson and Richard S. Wiener, published by Addison-Wesley.

If you want to dive into Smalltalk as a general purpose programming language, you may want to obtain a Smalltalk environment. A freely available version of the Smalltalk-80 language is available on the Internet at <http://www.create.ucsb.edu>.

Scripting

There are two different kinds of scripts in Kyma: Event Value-generating scripts (found in the *MIDIVoice* and the *MIDIMapper* Sounds), and Sound-constructing scripts (*Scripts* and *FileInterpreters*).

MIDIVoice and *MIDIMapper* scripts provide an algorithmic way to generate events indistinguishable from events coming from a MIDI file or a MIDI device. You can use these scripts to generate control signals or note events algorithmically. MIDI scripts are intended to replace the older scripting Sounds in applications that model an “instrument” playing from a “score”.

MIDIVoice and *MIDIMapper* scripts do not run in real time. When you compile a MIDI script, the script is executed first and the generated events are saved. Then the remainder of the Sound is compiled and loaded along with the events saved from the script. When you start the Sound, the generated events will be started at the appointed time.

The *Script* and *FileInterpreter* Sounds construct new Sounds that are combinations of *Mixers* and *TimeOffsets*. Scripting Sounds do not generate events. Instead, they assist you in the process of building Sounds the way you would build them in the graphic Sound editor. When there is a describable pattern to the Sound parameters and the way the Sounds are put together, you can use a scripting Sound to construct complex Sound structures automatically. For example, say you wanted to construct a reverberation algorithm out of six *ReverbSection* Sounds, each with a slightly different **DecayTime** and **Delay**. Using a scripting Sound, you could use a loop to construct a *Mixer* of *ReverbSection* Sounds, each with its own **DecayTime** and **Delay**. That way, you could save time by writing a single loop instead of setting all twelve parameters by hand.

When you compile a scripting Sound, it first constructs the equivalent combination of *Mixers* and *TimeOffsets*; then it compiles, loads and starts that automatically-constructed Sound.

MIDI Scripts

The *MIDIVoice* and *MIDIMapper* modules give you three choices for the source of MIDI events: the live MIDI input stream (originating from external controllers, or from a sequencer or other software running in parallel with Kyma), a MIDI file stored on disk, or a script (which generates events algorithmically and then emits them). To the Kyma Sound, events coming from a script are indistinguishable from events coming from a MIDI file.

There are two main ways of working with the MIDI scripts:

1. Creating low-level notes and controller events at specific times
2. Creating and manipulating high level **EventCollections**: combinations of **Note** and **Rest** objects

Creating Notes and Controller Events

The syntax for a basic note event is:

```
self keyDownAt: aTime duration: aDur frequency: aFreq velocity: aVel.
```

Where *aTime* is a time specified with units (for example, 1 s or 2 beats), *aDur* is a duration specified with units, *aFreq* is a frequency specified with units (for example, 440 hz, 60 nn, or 4c), and *aVel* is a velocity value between 0 and 1. The `duration:`, `frequency:`, and `velocity:` tags are optional, the default values are 10 ms, 0 nn, and 1, respectively.

For example, to send a note event for a middle C lasting for one second, you would use:

```
self keyDownAt: 0 s duration: 1 s frequency: 4 c.
```

Note events cause the following to happen:

At the given time, `!KeyDown` is set to -1.

After 5 ms, `!KeyDown` is set to 1, `!KeyNumber` is set to the pitch equivalent of the frequency, and `!KeyVelocity` is set to the velocity value.

After the given duration has passed, `!KeyDown` is set to 0.



Note that `!KeyNumber` can take on non-integer values and has 16 fractional bits, giving a pitch accuracy of 0.0015 cents for events generated by *MIDVoice* and *MIDIMapper* scripts. `!PitchBend` is not used to get these non-integer values.

To specify a value for a continuous controller in a MIDI script, use either:

```
self controller: aName setTo: aValue atTime: aTime.
```

to cause the controller to jump immediately to a specified value at a specified time, or

```
self controller: aName slideTo: aValue steps: nbrSteps byTime: aTime.
```

to cause the controller to slide from its previous value to the specified value by the specified time in the specified number of steps, or

```
self controller: aName slideTo: aValue stepSize: step byTime: aTime.
```

if you prefer to specify the size of each step, rather than the total number of steps.

For example, to cause `!Frequency` to move from 100 to 1000 over 10 seconds, starting at 30 seconds, you could use:

```
self controller: !Frequency setTo: 100 atTime: 30 s.
self controller: !Frequency slideTo: 1000 steps: 100 byTime: 40 s.
```

Event Collections

Sometimes it is more convenient to specify the MIDI events as collections of notes and rests, without having to specify start times for each event. The actual start times can be inferred from the duration of the **Note** or **Rest** and where it occurs in the collection of events. This corresponds more closely to written music notation, where note and rest symbols arranged horizontally are interpreted as a sequence of events in time (where each event's start time occurs immediately after the previous event's duration has expired), and notes or rests arranged vertically are interpreted as all starting at the same time.

In the MIDI script language, an **EventSequence** is a collection of **Notes**, **Rests**, or other **EventCollections** that occur one after another in time (corresponding to horizontal placement in music notation). Since you can also construct sequences of other collections, you can create higher level structures as well. For example, sequences of **Notes** and **Rests** could be collected in a measure; sequences of measures could be collected into a phrase; sequences of phrases could be collected into sections; sequences of sections could be collected into movements, *etc.* until you run out of traditional musical names for the structures(!)

An **EventMix** is a collection of **Notes**, **Rests**, or other **EventCollections** that occur all at the same time (corresponding to vertical placement in music notation). Like an **EventSequence**, the **EventMix** is recursively defined (*i.e.* you could have an **EventMix** of **EventMixes**), allowing you to define hierarchical structures, somewhat analogously to the way you can define Sounds.

You can also create a generic **EventCollection** of **Notes**, **Rests**, or other **EventCollections**, specifying that you haven't yet decided whether the events should be simultaneous or sequential but will send a message to the object later to specify actual start times and turn it into a sequence or a mix of other events.

Notes and Rests

The syntax for creating a single **Rest** is:

```
Rest durationInBeats: aDurInBeatsWithNoUnits.
```

To create a **Note**, you can use:

```
Note durationInBeats: aDur frequency: freqWithUnits velocity: aVel.
```

You can leave off any of the tags if you don't mind using the default values for those parameters.

Examples of creating **Rests** and **Notes**:

```
| n1 n2 n3 |
n1 := Note durationInBeats: 2 frequency: 2 g.
r2 := Rest durationInBeats: 1.
n3 := Note durationInBeats: 2 frequency: 2 d.
```

Sequences and Mixes

An **EventSequence** is a collection of events that occur in time order. The syntax for creating an **EventSequence** is:

```
EventSequence events: anArray.
```

An **EventMix** is a collection of events that occur at the same time. The syntax for creating an **EventMix** is:

```
EventMix events: anArray.
```

There are several ways to create the **Array**. You can specify each item in the array:

```
EventSequence events:
  (Array
    with: (Note durationInBeats: 1 frequency: 3 g)
    with: (Note durationInBeats: 1 frequency: 4 e)
    with: (Note durationInBeats: 2 frequency: 4 c))
```

You can collect items that you create within a loop:

```
EventSequence events:
  ((1 to: 100) collect: [ :i |
    Note durationInBeats: 2 frequency: 4 c + (i * 2) nn])
```

Or you can put literal values within parentheses preceded by a sharp sign. If you create the **Array** this way, remember to enclose the **Notes** and **Rests** within curly braces to make them literal values:

```
EventMix events:
  #( {Note durationInBeats: 1 frequency: 3 g}
    {Note durationInBeats: 1 frequency: 4 e}
    {Note durationInBeats: 2 frequency: 4 c} )
```

TimedEventCollections

A **TimedEventCollection** is a collection of events together with the starting beat for each event. One way to create a **TimedEventCollection** is:

```
TimedEventCollection startingBeats: beats events: events.
```

where *beats* is a collection of the starting beat numbers for each event in the *events* collection.

You can also create a **TimedEventCollection** by reading in the events and starting beats from a MIDI file. If all you want to do is play exactly what's in the MIDI file, then it would be quicker and more straightforward to simply select MIDI file as the **Source** in your **MIDIVoice**. However, once you have read the MIDI file into a **TimedEventCollection** you can transform it, twist it, distort it, in short, do all the kinds of things composers love to do!

The syntax for reading a MIDI file is:

```
TimedEventCollection timesAndEventsFromMIDIFile: 'file.mid' channel: 1.
```

Generic EventCollections

An **EventCollection** is a collection of **Notes** and **Rests** without any ordering. To create an **EventCollection**, use:

```
EventCollection events: anArray.
```

Where *anArray* is a collection of **Notes**, **Rests** and other collections of events.

The message `randomizeUsing:totalBeats:quantizeTo:maxSpacing:` forces an **EventCollection** to become a **TimedEventCollection** by generating a set of random starting beats and randomly picking a set of events from the events supplied:

```
(EventCollection events: anArray)
  randomizeUsing: (Random newForKymaWithSeed: 92)
  totalBeats: 16
```

```
quantizeTo: 0.5  
maxSpacing: 1.
```

You specify the total duration in beats of the result, the smallest distance between beats (with the `quantizeTo:` argument) and the longest time between beats (the `maxSpacing:` argument).

Alternatively, you can choose the starting beats with one random number generator and the events using another, for example:

```
(EventCollection events: anArray)  
  randomizeTimesUsing: (Random newForKymaWithSeed: 561)  
  pickingEventsUsing: (OneOverF newForKymaWithSeed: 17661)  
  totalBeats: 16  
  quantizeTo: 0.333  
  maxSpacing: 1.
```

Manipulating Event Collections

All event collections can be manipulated in various ways:

```
anEventCollection dim: aScale.  
anEventCollection trsp: aTransposition.  
anEventCollection inv: aPitch.  
anEventCollection retrograde.
```

`dim:` applies a scale to the duration of each event in the collection, `trsp:` transposes each event in the collection the given number of half-steps, `inv:` inverts each event in the collection about the given pitch, and `retrograde` reverses the time order of the event in the collection.

Finally, to actually output the events for the *MIDI Voice* to play, use:

```
anEventCollection playOnVoice: self bpm: 160.
```

Scripts and FileInterpreters

Scripts and the *FileInterpreters* are Kyma Sound classes that enable you to construct Sounds algorithmically. Both *Scripts* and the *FileInterpreters* take inputs that can include both Sounds and lifted Sounds. In a *Script*, you can combine and manipulate the inputs using Smalltalk. *TextFileInterpreters* and *MIDIFileInterpreters* let you use Smalltalk code to interpret data from text files and MIDI files.

Script

A *Script* Sound lets you refer to its inputs by name, assigning them start times and parameter values. In this respect, you could think of the inputs as “instruments” and the start time and parameter value assignments as the “score” (similar in some respects to the instruments and score of a Music-N language).

Unlike Music-N, however, a *Script* lets you use Smalltalk as part of the score. The script is not just an event list; it is a Smalltalk program. So you can think of the *Script* as being more than a score; it is a tool for constructing complex Sounds algorithmically.

In Kyma’s version of a “score,” not only can you supply start times and other parameters to an instrument, you can also change the instrument itself from the script.

TextFileInterpreter

Using a *TextFileInterpreter* Sound you can read and interpret text files. One application of the *TextFileInterpreter* is to use a score file from a Music-N language (like Csound) to control Kyma Sounds as its instruments. A *TextFileInterpreter*, however, is not limited to reading Music-N scores. In fact, you can use it to map any data to the parameters of a Sound. This capability makes a *TextFileInterpreter* useful for data sonification and generating signals for psychoacoustic experiments.

MIDIFileInterpreter

With a *MIDIFileInterpreter*, you can read information from a MIDI file and process or reinterpret it using Smalltalk before mapping it to the parameters of Kyma Sounds.

Events

Assigning Start Times

Each input of a *Script* forms the basis of an instrument. An instrument is not strictly a Sound; it is a template that can be used to create any number of instances of its basis Sound.

Each event in the *Script* corresponds to an instantiation (*i.e.*, the scheduling of a specific instance) of one of the instruments. The simplest event is the name of an instrument followed by a start time. For example, if you had an instrument named `anInstr`, and you wanted an instance of it to start at time 0, you would type

```
anInstr start: 0 s.
```

Suppose `anInstr` is based on a Sound whose duration is 5 seconds. Then the following *Script*

```
anInstr start: 0 s.  
anInstr start: 6 s.
```

specifies that you will hear `anInstr` followed by 1 second of silence followed by `anInstr`. You can also make the instances of `anInstr` overlap, for example,

```
anInstr start: 0 s.  
anInstr start: 1 s.  
anInstr start: 2 s.  
anInstr start: 3 s.
```

Beats and Metronome Indications

To specify times in beats instead of seconds, indicate the number of beats per minute at the beginning of the *Script*. For example,

```
MM := 90.
```

specifies that one beat is equivalent to 1/90 of a minute in subsequent events. You can redefine `MM` any number of times within a script. If you don't specify a value for `MM` in a top level *Script*, it will default to a value of 60, or one beat per second.

If you are using multiple *Scripts* or *FileInterpreter* Sounds within a shared top Sound, you should explicitly specify the desired value for `MM` in each script.

Setting Parameters

Parameter:Value Pairs

You can do more than simply assign start times to instruments; if the inputs have variables in them, each event in the script can also supply parameter values to the variables in the instrument. Any variable parameter can be set from the script.

To set parameters from the script, type the name of the input, the start time, and then a list of *parameter-Name:value* pairs. Each pair consists of the name of a variable in the input and a value for that variable. (To see a list of the variables in the selected Sound, choose **Get info** from the **Info** menu.)

For example, suppose a Sound named `anInstr` has some variable parameters in it named `?centerFreq`, `?dur`, and `?bandwidth`. Here is an example *Script* for setting the parameters of `anInstr`.

```
anInstr  
  start: 0 s  
  dur: 2 s  
  centerFreq: 1000 hz
```

```

        bandwidth: 10 hz.
anInstr start: 2 s.
anInstr start: 3 s freq: 4 a.
anInstr start: 3.2 s centerFreq: 4 a dur: 0.5 s.

```

Each event consists of the name of a Sound followed by a start time, any number of *parameter:value* pairs, and a terminating period. Any number of spaces, tabs, or carriage returns can separate the *parameter:value* pairs; carriage returns and tabs are encouraged when they will improve readability.

The unassigned parameter values for an event default to the values assigned to those parameters in the previous event. If a parameter has never been set in the script, Kyma will prompt you to supply a value for its variable.

The *parameter:value* pairs can occur in any order, and any parameter may be omitted. However, if the start time is omitted, the event is no longer a well-defined event, and it will not be scheduled at all.

Non-Numeric Parameters

An instrument's parameter values are not limited to numbers. A parameter's value can be any Smalltalk object, including a Sound; thus, strings, symbols, arrays, *etc.* can all appear as parameter values in a script. For example, the following is a legal **Script**:

```

anInstr start: 0 s wavetable: 'sine'.
anotherInstr start: 1 s weights: #(1 1 1 4).
aScriptingL start: 1 s script: 'intern start: 0 s'.

```

where the number sign and parentheses indicate an **Array** and single quotes indicate a **String**. (See *The Smalltalk-80 Language* on page 513 for more information.)

Instrument parameters can be set to Sounds as well. Suppose you have an instrument named *aNoise* based on a Sound having a variable ?dur, and an instrument named *aFilter* based on a Sound having a variable *input* (a *Variable* Sound).

You can set the value of *input* to *aNoise* as follows:

```

aFilter
  start: 1 s
  input: (aNoise dur: 1 s).

```

Since *aNoise* is given no start time, it is a Sound, but it will not be scheduled on the Cappybara. It *can* be the value for the *Variable* Sound. Then when you gave *aFilter* a start time, *aNoise* would be assigned the same start time. If you were to use

```

aFilter start: 1 s input: aNoise.

```

Because it has not been assigned a value in the script, Kyma would ask you to supply a value for ?dur.

Referring to the Orchestra

You can refer to a **Script** instrument in two ways: by its name or by its position in the “orchestra.” In the following event,

```

(orchestra at: 3) start: 3 s.

```

(orchestra at: 3) refers to the instrument based on the third input (counting from top to bottom, left to right). This can be useful when **Inputs** has been set to a *SoundCollectionVariable* or when you want to make an algorithmic choice of instrument.

Algorithmically-generated Events

A Script can include any Smalltalk-80 statements. For example, the script

```

anInstr start: 0 s freq: 100 hz dur: 1 s.
anInstr start: 1 s freq: 200 hz dur: 1 s.
anInstr start: 2 s freq: 300 hz dur: 1 s.

```

could be rewritten as a loop:

```
0 to: 2 do: [ :i |
  anInstr
    start: i s
    freq: ((i + 1) * 100) hz
    dur: 1 s].
```

In the loop above, the expression within brackets

```
anInstr start: i s
  freq: ((i + 1) * 100) hz
  dur: 1 s
```

is iteratively evaluated with *i* set to the values 0, 1, and 2.

To learn more about how you can use Smalltalk in a script, refer to the *Script* examples in the file **Scripts**.

Variables

Kyma Variables

A Kyma variable represents the value of a Sound parameter; it can be a *Variable* Sound, a *SoundCollectionVariable*, or a name preceded by a question mark.

You can use Kyma variables in a script. The following example plays a harmonic series (of ?total harmonics of 100 hz, spaced 1 second apart):

```
1 to: ?total do: [ :i |
  pluck start: (i - 1) s freq: (100 * i) hz].
```

The variable ?total controls the number of times the code within the square brackets is executed; thus, it determines the number of events in this *Script*. When you load this *Script* Sound, Kyma will prompt you for a value for ?total. As the use of Kyma variables in this sample script suggests, scripts can be used as inputs of other scripts.

Variables in Scripts and FileInterpreters

Each event in a *Script* or *FileInterpreter* script corresponds to a mapping of variable names to values for those variables. When you load a *Script*, it binds the variables of its input(s) to the values supplied in the script.

You can see a list of the selected Sound's variables by choosing **Get info** from the **Info** menu.

If the script fails to map one of the variables to a value, then you must supply a value for that variable in some other way, either by entering values when prompted, by using the Sound as an example for a new class, or by making the Sound an input of another *Script* or *FileInterpreter*.

Kyma variables in the script of a *Script* or a *FileInterpreter* are treated just like variable parameters. These variables must be set from "outside" (that is, from some Sound to the right of them in the signal flow diagram).

Smalltalk Variables

You can also use Smalltalk variables in a script. A Smalltalk variable has no preceding question mark; instead it has to be declared at the beginning of the script in which it is used. Smalltalk variables are used as temporary storage locations within a script. They do not directly represent Sound parameters.

A Smalltalk variable is defined only within the script in which it appears. A Kyma variable is defined throughout the entire Sound structure in which it appears.

Smalltalk variables must be declared at the beginning of the Script by typing them between vertical lines,

```
| curTime curFreq |
```


You can assign values to Smalltalk variables using the `:=` assignment operator. For example, in the following *Script*,

```
| curTime curFreq |

curTime := 0.
curFreq := 100.
```

`curTime` is assigned the value 0 and `curFreq` is set to the value 100.

Additional kinds of Smalltalk variables (called block arguments and block temporaries) are described in *The Smalltalk-80 Language* on page 513.

TextFileInterpreter Messages

A *TextFileInterpreter* (a version of the *Script* Sound) reads data from text files. You can use a *TextFileInterpreter* to read and interpret scores written for a Music-N language. More generally, you can use a *TextFileInterpreter* to map any data to sound parameters.

In addition to allowing you to use general Smalltalk messages, a *TextFileInterpreter* also lets you use the following specialized messages for accessing text files:

Smalltalk Message	Result
<code>atEnd</code>	Returns <code>true</code> if at the end of the file, or <code>false</code> if not at the end.
<code>linesInFile</code>	Returns the number of lines in the text file.
<code>maxValue</code>	Returns the largest number in the file. If there are no numbers in the file, <code>maxValue</code> returns <code>-</code> .
<code>minValue</code>	Returns the smallest number in the file. If there are no numbers in the file, <code>minValue</code> returns <code>+</code> .
<code>nextNumber</code>	Returns the next number. Returns 0 at the end of the file or at a letter.
<code>nextParameters</code>	Returns the entire next line as an Array. Each item on the line that is separated by a space is put into a separate entry in the Array.
<code>reset</code>	Moves to the beginning of the file.

In the script of a *TextFileInterpreter*, you refer to the text file by the name `file`. You can address the file in its entirety, asking `file` for the number of lines it has and for its maximum and minimum values. For example, the following statements assign the largest value in the file to the Smalltalk variable `max`, the smallest value to `min`, and the number of lines to `lines`:

```
| max min lines |

max := file maxValue.
min := file minValue.
lines := file linesInFile.
```

You can also access the data in the file one line at a time. In the following statements, the Smalltalk variable `dataArray` is assigned an array containing each value found on the next line of the disk file:

```
| dataArray |

dataArray := file nextParameters.
```

The *TextFileInterpreter* keeps track of the last line it read from the file, so you can continue to read the “next” line until you reach the end of the file. To start over again at the beginning of a file, reset the file by using

```
file reset.
```

in the *Script*.

As a final illustration, let's take a slightly more complicated problem. Suppose that you have created a text file that had four items on each line, and that you wanted to interpret these four items as representing a start time, a duration, a frequency, and a wavetable, for example:

```
0      1      440.0      sine
0.5    1      453        sine
```

To interpret such a data file, you might use the following **Script**:

```
| event |

[file atEnd] whileFalse: [
  event := file nextParameters.
  anInstr
    start: (event at: 1) s
    duration: (event at: 2) s
    frequency: (event at: 3) hz
    wavetable: (event at: 4)].
```

The *TextFileInterpreter* continues to read lines out of the file until reaching the end of the file. Each line of the file is stored in turn as an array in the variable called *event*. The first position of the array is accessed as (*event at: 1*), the second by (*event at: 2*), etc.

MIDIFileInterpreter Messages

Much like the manner in which a *TextFileInterpreter* reads data from text files, a *MIDIFileInterpreter* reads data from MIDI files. A *MIDIFileInterpreter* interprets files using specialized Smalltalk messages that are sent to the MIDI file object.

Most of the specialized messages that you can use in a *MIDIFileInterpreter* address a single MIDI event. To navigate through the MIDI file, a *MIDIFileInterpreter* can send the following messages to the MIDI file:

Smalltalk Message	Result
<i>nextEvent</i>	Returns the next event in the MIDI file.
<i>reset</i>	Moves to the beginning of the file.
<i>atEnd</i>	Returns <i>true</i> if at the end of the file, or <i>false</i> if not at the end.

After using *nextEvent* to read the next event of a MIDI file, the *MIDIFileInterpreter* can send a number of messages to interpret the data in that MIDI event. (A MIDI "Note On" command coupled with the corresponding "Note Off" command constitutes an event.)

The following is a list of Smalltalk messages that can be sent to the *event*:

Smalltalk Message	Result
<i>channel</i>	Returns the MIDI channel (1-16) that the event is on.
<i>isChannelPressure</i>	Returns <i>true</i> or <i>false</i> depending on whether the event is a channel pressure.
<i>channelPressureValue</i>	If the event is a channel pressure, returns the value (0-127).
<i>isControl</i>	Returns <i>true</i> or <i>false</i> .
<i>controlNumber</i>	If the event is a control, returns the control number (0-127).
<i>controlValue</i>	If event is a control, returns the control value (0-127).
<i>lastControlValueForChannel:</i> <i>controlNumber:</i> <i>ifNone:</i>	Returns the last control value (0-127) seen for the given channel. The last argument is a block (i.e. code enclosed in brackets) that can return a default value if no control change for the given controller on the given channel has been found.
<i>isNoteOff</i>	Returns <i>true</i> or <i>false</i> .

<code>isNoteOn</code>	Returns <code>true</code> or <code>false</code> .
<code>noteNumber</code>	If the event is a note on or note off, returns the note number (0-127). MIDI note numbers correspond exactly to units of pitch in Kyma.
<code>duration</code>	If the event is a note on, returns the duration to the corresponding note off (in microseconds).
<code>isPitchWheel</code>	Returns <code>true</code> or <code>false</code> .
<code>pitchWheelChange</code>	If the event is a pitch wheel, returns the value (to ± 2048). You can scale this value in the Script.
<code>lastPitchWheelValueForChannel:</code> <code>sensitivityInHalfSteps:</code> <code>ifNone:</code>	Returns the last pitch wheel value seen for the given channel. If there has been no pitch wheel value received, the last argument is evaluated.
<code>pitchWheelPointsForChannel:</code> <code>sensitivityInHalfSteps:</code> <code>ifNone:</code>	Returns a collection of time points for the given channel's pitch wheel data. The last argument is evaluated if there has been none.
<code>isPolyphonicKeyPressure</code>	Returns <code>true</code> or <code>false</code> .
<code>polyphonicKeyPressureValue</code>	If event is a key pressure, returns the value (0-127).
<code>isProgramChange</code>	Returns <code>true</code> or <code>false</code> .
<code>programNumber</code>	If the event is a program change, returns the program number (0-127).
<code>isSwitch</code>	Returns <code>true</code> or <code>false</code> .
<code>isSwitchOn</code>	Returns <code>true</code> or <code>false</code> .
<code>isSwitchOff</code>	Returns <code>true</code> or <code>false</code> .
<code>status</code>	Returns the status byte as one of the following Symbols: <code>#noteOff</code> , <code>#noteOn</code> , <code>#polyphonicKeyPressure</code> , <code>#control</code> , <code>#programChange</code> , <code>#channelPressure</code> , <code>#pitchWheel</code> , or <code>#unknown</code> .
<code>time</code>	Returns the start time of the event (in microseconds).
<code>velocity</code>	If the event is a note on or note off, returns the key velocity (0-127).

Debugging

Inspecting Values

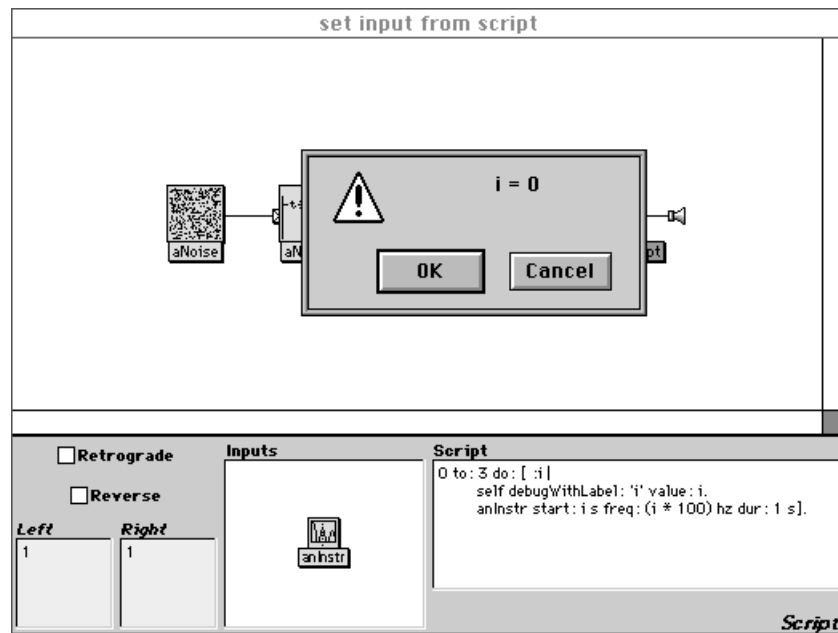
When debugging scripts, it can be useful to inspect the value of a Smalltalk variable as the script is being evaluated. To inspect the values of the variables, you can include the Smalltalk messages `debugWithLabel:value:` or `debug:` in the script. To illustrate, by including the statement

```
self debugWithLabel: 'i' value: i.
```

in a script, the phrase “`i =`” and the value of `i` will be displayed in a dialog. (`debug:` returns just the value, without the label.) For example, in this segment,

```
0 to: 3 do: [ :i |
    self debugWithLabel: 'i' value: i.
    anInstr start: i s freq: (i * 100) hz dur: 1 s].
```

Kyma will display the following on the first iteration of the loop:



If you click **OK**, the program will proceed, and the next value of *i* displayed will be 1. To stop evaluating the script, click **Cancel**.

Expand

If a *Script* or one of the *FileInterpreters* isn't behaving the way you think it should, try expanding it so that you can see what kind of structure your script is producing. Choose **Expand** from the **Action** menu to expand a Sound once. You can then select specific inputs for further expansion.

Like many Sounds in Kyma, a *Script* Sound is actually a shorthand representation for the Sound as it is actually realized on the Capybara. A Sound may have to expand several times before it consists of only primitive Sounds, *i.e.* Sounds that have assembly language representations on the Capybara.

Commented Examples

The following examples can be found in the file called **Scripts** in the **Examples** folder.

The first example is called *Kurt's Rhythmic Pulsation*. It uses a Sound called *thump* that has variables ?frequency and ?cycles.

```
"Set the metronome to 300 beats per minute."
MM := 300.

"Set i to the values 0, 1, ... 200 on each iteration."
0 to: 200 do: [ :i |

    "If i mod 4 is 0 (i.e. if i is a multiple of 4), then start the instru-
    ment called thump at i beats with a frequency of 100 hertz and 25
    cycles within each envelope."
    i \\ 4 = 0 ifTrue: [
        thump
            start: i beats
            frequency: 100 hz
            cycles: 25].

    "etc..."
    i \\ 2 = 0 ifTrue: [
        thump
            start: i beats
            frequency: 250 hz
            cycles: 25].

    i \\ 3 = 0 ifTrue: [
        thump
            start: i beats
            frequency: 150 hz
            cycles: 25].

    "If i is less than 50 AND i is a multiple of 5, then start the instru-
    ment called thump at i beats with a frequency of 400 hertz and 25
    cycles within each envelope."
    (i < 50 and: [i \\ 5 = 0]) ifTrue: [
        thump
            start: i beats
            frequency: 400 hz
            cycles: 25].

    i \\ 13 = 0 ifTrue: [
        thump
            start: i beats
            frequency: 700 hz
            cycles: 25].

    i \\ 7 = 0 ifTrue: [
        thump
            start: i beats
            frequency: 50 hz
            cycles: 50].

    "If i is greater than 50 AND i is not a multiple of 7, 5 or 3, then
    start the instrument called thump at i beats with a frequency of 1000
    hertz and 100 cycles within each envelope."
    (i > 50 and: [i \\ 7 ~= 0
        and: [i \\ 5 ~= 0 and: [i \\ 3 ~= 0]]])
        ifTrue: [
            thump
                start: i beats
                frequency: 1000 hz
                cycles: 100].

    "End of 0 to: 200 do: loop."
    ].
```

The next example is called *Serial Example*. In this example, *inst* takes two arguments: *offset* and *sound*. The argument *sound* must be another Sound; in this case it is called *aPluck*. The Sound *aPluck* also takes two arguments: *left* and *right*. Notice that whenever *aPluck* is provided as the argument to *inst*, *aPluck* is not assigned a start time.

inst looks at the pitch of its Sound argument and adds the value of *offset* to that pitch. For instance, if the frequency of *aPluck* were 4 a and the *offset* were 3, the result would be a copy of *aPluck* whose frequency was 5 c.

In this example, *p0* is an array of intervals that determines the sequence of pitches and the start times of *aPluck*.

```
| p0 start |
```

```
MM := 480.
```

```
"Set p0 to an array of 12 intervals, expressed as the number of half
steps to be added to the pitch of aPluck."
```

```
p0 := #(0 1 11 5 2 4 3 9 10 6 8 7).
```

```
start := 0 beats.
```

```
"Step forward through p0 by setting i to 1, 2,..., 12 successively and
using it as the index into the array. Each copy of inst starts (p0 at:
i) beats after the previous one. The pitch of each copy of inst is the
pitch of aPluck plus the offset (p0 at: i) halfsteps. For example, on
the third time through the loop, (p0 at: i) = 5. Thus, a copy of inst
will start 5 beats after the previous copy and its pitch will be the
pitch of aPluck plus 5 halfsteps."
```

```
1 to: 12 do: [ :i |
    start := start + (p0 at: i).
    inst
        start: start beats
        offset: (p0 at: i)
        sound: (aPluck left: 1 right: 1)].
```

```
"r5: This time, step backwards through the array, and start 5 halfsteps
above the base pitch. Reset start so that these events overlap some of
the previously scheduled events."
```

```
start := 6.
12 to: 1 by: -1 do: [ :i |
    start := start + (p0 at: i).
    inst start:
        start beats
        offset: (p0 at: i) + 5
        sound: (aPluck left: 0 right: 1)].
```

```
"ri0: This time, go through the array backwards, negate each of the
offsets, and subtract 24 (i.e. transpose them two octaves down). Reset
start so that these events occur simultaneously with the events of r5."
```

```
start := 6.
12 to: 1 by: -1 do: [ :i |
    start := start + (p0 at: i).
    inst
        start: start beats
        offset: (p0 at: i) negated - 24
        sound: (aPluck left: 0.6 right: 0)].
```

```
"Play a low frequency sound starting at 9 seconds."
```

```
low
    start: 9 s
    carFreq: 100 hz
```

```

        cmRatio: 2.0
        dur: 0.5 s.

"r0: This time, step backwards through the array. Reset start so that
this group of events begins at 10 seconds."
start := 10 s.
12 to: 1 by: -1 do: [ :i |
    start := start + (p0 at: i).
    inst
        start: start beats
        offset: (p0 at: i)
        sound: (aPluck left: 1 right: 1)].

"p6: Step forward through the array, and start 5 halfsteps above the
base pitch. Reset start so that this group of events begins 6 beats af-
ter 10 seconds."
start := 10 s + 6 beats.
1 to: 12 do: [ :i |
    start := start + (p0 at: i).
    inst
        start: start beats
        offset: (p0 at: i) + 5
        sound: (aPluck left: 0 right: 1)].

"Step backwards through the array, and add the inverted interval to the
base pitch two octaves lower. Reset start so that this group of events
begins simultaneously with the previous group."
start := 10 seconds inBeats + 6.
12 to: 1 by: -1 do: [ :i |
    start := start + (p0 at: i).
    inst
        start: start beats
        offset: (p0 at: i) negated - 24
        sound: (aPluck)].

"Play a final low frequency sound at the end."
low
    start: start beats
    carFreq: 100 hz
    cmRatio: 2.0
    dur: 0.5 s.

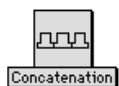
```

The Class Editor

In the course of designing new Sounds, you may find that you have constructed a complex Sound with only a few relevant control parameters. Suppose, for example, that you have constructed a 16-harmonic additive synthesis Sound. To change the fundamental, you could edit all sixteen frequency fields. However, since you know the relationship between the harmonics, you could instead specify them symbolically as (!Frequency, !Frequency*2, ..., !Frequency*16). Then when you changed !Frequency, all the *Oscillators* would maintain their harmonic relationship to one another. You can specify the same kind of relationship with variables (i.e., ?freq, ?freq*2, ..., ?freq*16).

Since ?freq is the only parameter needed to control the whole Sound, you can hide the low-level details of the structure by *encapsulating* the structure into a single, new kind of Sound object. In Kyma, this process is called defining a new *class* of Sounds. This new class can have its own parameters, its own icon, and its own editor.

Sound Classes and Sounds



Each Sound is a specific instance of a Sound class. There can be any number of instances of each class. The Sound class contains information that all of its instances have in common. For example, all instances of class *Concatenation* share the same icon (shown in the left margin).

They also share the same parameter names: **Name**, **Inputs**, **Retrograde**, **Reverse**. In Kyma, you never work directly with the system classes; instead, you modify the *instances* of the system classes that appear in the system prototypes window.

Creating and Editing Sound Classes

The **Action** menu gives you three options for creating and editing classes. The **Edit class** operation allows you to edit previously defined classes. **New class from example** (active only in the Sound file window) opens an editor on a new class based upon the selected Sound. **Retrieve example from class** (active only in the Sound class editor) extracts the Sound that is being used as the template for a user-defined class and saves it to the Sound file window.

You can create a custom Sound class by introducing variable into an example Sound parameters (see *Variables* on page 510). The variables in the example Sound become the parameters of the new Sound class.

Setting Up the Example

Start with the Sound that you would like to use as an example for a new class of Sounds. The first step is to decide which of the Sound's parameters should be constant, which parameters should be variable, and whether there should be any relations maintained between parameters.

Edit the Sound, replacing the parameters that are allowed to vary from instance to instance with variables, *Variables*, or *SoundCollectionVariables*.

To specify relations between parameters, use expressions involving variables and constant values. For example, an additive synthesis Sound might consist of a *Sum* with 16 *Oscillators* as inputs. By setting their frequency values to the expressions

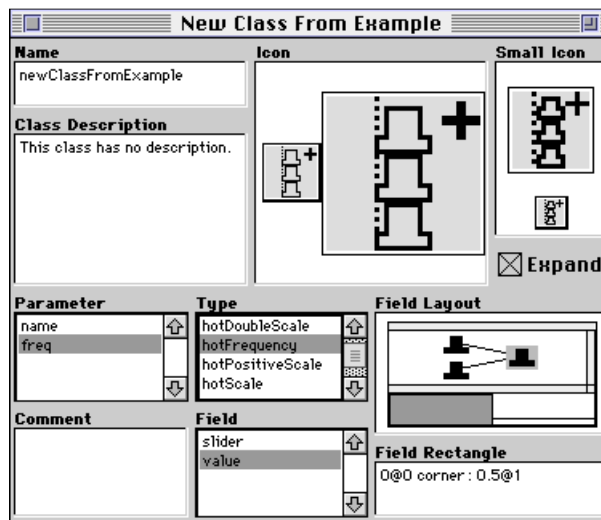
?freq, ?freq*2, ... , ?freq*16

you stipulate that fifteen *Oscillators* have frequencies that are harmonics of the first, no matter what the frequency of the first *Oscillator*.

Class Editor

After you have saved your newly edited Sound (be sure it is still selected), select **New Class from example...** from the **Action** menu.

This opens a class editor that will let you define a new Sound class based upon your example.



A class editor has several parts:

- An icon editor for designing the class icon
- An icon editor for designing the class small icon
- Field Layout: a miniature version of the Sound editor
- A text editor for the new Sound class name
- A text editor for the class description
- Parameter: a list of the class parameters
- Type: the parameter type of the selected parameter
- Field: a list of the legal field types for the selected parameter
- Field Rectangle: the relative position of the selected parameter's field within the Sound editor
- Comment: a text editor for editing the selected parameter's description
- A check box for specifying that a Sound should expand before transformations are applied to it

Using the Icon Editors

Click in a white area to draw a black dot, and click in a black area to erase. If you hold the mouse button down as you move the mouse, you can draw or erase continuously. While you are designing the icon, you can also see an actual size version of it below or to the left of the large version.

There are two icons for each class. The small icon is used whenever you select **Small icon** in the dialog under **View** in the **Edit** menu. The other icon is the one you see by default.

To set the large and small icons to be the same as those of another Sound, drag that Sound into the icon editor.

Parameter Types and Fields

When you select a name from the **Parameter** list, you see its type selected in the **Type** list and the kind of parameter field it uses selected in the **Field** list. The position of the parameter field is shown highlighted in the miniature Sound editor labeled **Field Layout**.

Parameter Types

Each variable name in the example Sound becomes a parameter name in this new class. You can set parameter types to be the same as those in another Sound by dragging that Sound icon onto the **Parameter** list in the Class editor.

To manually set the type of a parameter, first select the parameter name in the list labeled **Parameter**. Then choose from among the given list of types under the **Type** heading. The type of a parameter defines

its range of legal values, its default value, its list of units, and whether it requires units (e.g., hz or s). The type is used by the Sound editor to check whether the values entered in the parameter fields make sense for those particular parameters. Later, when you compile a Sound of this new class, any parameter fields with invalid values will flash and the host computer will beep twice.

The best way to learn about types is to use the class editor to inspect the parameter types of some of the system Sound classes. See the table at the end of this chapter for an exhaustive listing of the types and their ranges.

Parameter Fields

Field describes the kind of field the parameter will have in the Sound editor. The kind of field determines how you will be able to enter values for that parameter in the Sound editor. For example, a `hotValue` field lets you type Event Values as well as constants, and a `soundCollection` field allows you to drag multiple Sounds into the field.

In the Sound class editor, the contents of the **Field** list depends upon the currently selected **Type**; only those fields that are legal for the selected type are shown.

The different kinds of fields available are as follows:

Field	Description
<code>array</code>	Allows you to type in values separated by spaces; this field accepts them in order as an array.
<code>boolean</code>	A check box that allows you to choose true or false.
<code>codeString</code>	Used in <i>Script</i> , the <i>FileInterpreters</i> , and <i>ParameterTransformer</i> for the Script or Transformation field. It parses what you have entered and checks its syntax whenever you try to load or edit another Sound. A <code>codeString</code> provides you with more detailed error messages than a value field does.
<code>fileName</code>	A browse button that opens a file list dialog.
<code>grammar</code>	The field used for the production rules of ContextFreeGrammar. See the Prototypes Reference on page 218.
<code>legalNameSound-Collection</code>	Accepts any number of Sounds with legal variable names. This is needed in <i>Script</i> and the <i>FileInterpreters</i> so that you can refer to the Sounds by name in the Script .
<code>radioButtons</code>	Presents built-in choices (depending on the type of field) as a column of radio buttons. For example, if you choose radio button in the class editor for the type of the WordSize field in a <i>DiskRecorder</i> , the WordSize field will display radio buttons corresponding to the three word size choices.
<code>radioButtonsTwoColumns</code>	Same as <code>radioButtons</code> , but displays in two columns.
<code>samplesFileList</code>	The field used in the Segments parameter of the <i>SamplesFileSplicer</i> .
<code>sound</code>	Accepts only one Sound at a time.
<code>soundCollection</code>	Accepts any number of Sounds.
<code>string</code>	A text editor that will accept any sequence of characters. It cannot be set to a variable.
<code>symbolList</code>	A pop-up list of choices that presents a list of values.
<code>value</code>	Allows you to type any value, variable, or expression. The expression is evaluated when you load or edit another Sound.
<code>wavetable</code>	Allows you to select a sample from a file dialog or to type in the name of a sample file or a variable.

Positioning the Fields

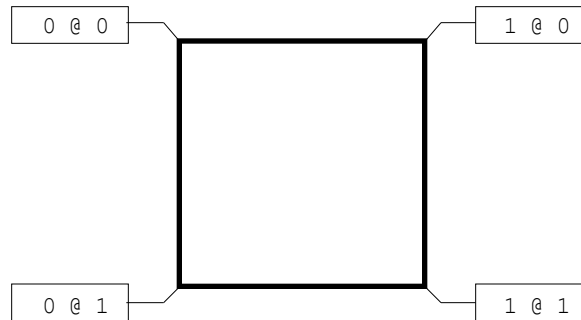
Field Layout shows you what the Sound editor for Sounds of this class will look like. When you select a parameter, its field is highlighted in gray.

To change the position of a field, type the coordinates of a new rectangle in the **Field Rectangle** text editor and then press **Enter**.

The format for specifying rectangle coordinates is as follows:

upperLeftX @ upperLeftY corner: lowerRightX @ lowerRightY

where the parameter fields are laid out in a square whose corner points are as shown below:



The *x* and *y* values are not in absolute units like centimeters or pixels; they just indicate a position relative to a maximum *x* and *y* of 1 @ 1.

Whenever you type in a new rectangle, you should adjust each of the other parameter field positions as necessary to avoid any overlapping fields.

Class Description and Comments

Each class has a **Class Description**, a paragraph or two describing how Sounds of that class behave.

In addition to an overall class description, each parameter has its own **Comment**. A parameter comment provides later users of Sounds of this class with instructions for how to enter a value for the parameter, any restrictions on its range of legal values, and an indication of how it relates to the other parameters and to the class as a whole.

The **Edit** menu is active in the class editor for editing the class description and the parameter comments.

Expand

When the **Expand** box is checked, the new class will expand before any transformations are applied to it (see *Controlling Expansion* on page 545 for more information).

Closing the Class Editor

The first time you close the editor on a new Sound class, you will be asked whether you want to save an instance of the new class. Click **Yes** unless you would like to start over from scratch. An instance of the new class is created using default values for each of the parameters. To change those values, double-click on the Sound and edit it.

When you close the class editor window after subsequent editing of that Sound class, you will be asked if you want to save the changes made to the class. Choose **Yes** to save the changes, **No** to discard the changes, or **Cancel** to return to the class editor without closing.

Debugging

When a Sound that you have constructed has some unexpected behavior, it can be helpful to expand it, so that you can see how Kyma is really interpreting your Sound. The **Expand** operation from the **Action** menu expands a Sound one level only. You can then select specific inputs for further expansion.

Organizing the New Classes

To keep track of your custom-designed Sound classes, you can keep them all in one Sound file that you can open as a custom prototype strip. You can organize the Sounds into categories by placing related Sounds together in collections.

Parameter Types and Ranges

The parameter types and their ranges are listed in the table beginning on the next page. In most cases, the type of the parameter is the same as its name. If you're not sure of a parameter's type, you can find it by editing the class of the Sound in which it appears.

Type	Range
almostOneScale	0.9 to 1.0
array	any list of one or more objects
boolean	true or false
channelNbr	Mono or Stereo
className	<i>obsolete</i>
codeString	Smalltalk code
complex	any complex number
delayedSoundCollection	<i>obsolete</i>
delayLine	wavetable name (see <i>What is a Wavetable?</i> on page 493)
diskFormat	SDI, SDII, AIFF, IRCAM or WAVE
doubleScale	-2.0 to 2.0
duration	1 to 140737488355327 samp
dynamicsType	compressor or expander
envelope	a graphical envelope
envelopeSegmentType	linear or exponential
feedbackDelayType	comb or allpass
fileName	any legal file name
filterType	<i>obsolete</i>
frequency	-25000.0 hz to 25000.0 hz
function	<i>obsolete</i>
genericSource	Disk, Live, or RAM
grammar	see <i>Prototypes Reference</i> for <i>ContextFreeGrammar</i>
halfScale	0.0 to 0.5
hotArray	any list of one or more objects; each element in the array can be an Event Value
hotDoubleScale	-2.0 to 2.0
hotFrequency	-25000.0 hz to 25000.0 hz
hotPositiveScale	0 to 1.0
hotScale	-1.0 to 1.0
hotTime	0 samp to 140737488355327 samp
hotValue	number or an Event Value
hotValueList	<i>obsolete</i>
integer	-2147483647 to 2147483647
interpolation	none or linear
legalName	a letter followed by zero or more letters and numbers
legalNameSoundCollection	one or more Sounds with legal names
limeFileName	any legal Lime file name
lowestAnalyzedFreq	Above 1F (44 hz), Above 4F (345 hz), Above 2F (87 hz), Above 5F (690 hz), or Above 3F (173 hz)
lowRateRecording	<i>obsolete</i>
mapping	OnePerHalfStep, EquallySpaced, ByBasePitch, or ByPitchRange
midiChannel	1 to 16
midiControllerNumber	0 to 127

midiFileName	any legal MIDI file name
midiSource	MIDIInput, MIDIFile, or Script
modulation	frequency or none
name	one or more characters
object	any Smalltalk object
optionalMidiChannel	0 to 16, 0 indicates the default MIDI channel
oscillatorBankInputType	SpectralShape or SOSAnalysis (<i>obsolete</i>)
panFunction	linear or power
polesAndZeroes	any list of complex poles and zeroes
positiveFrequency	0.001 hz to 25000.0 hz
positiveInteger	0 to 214748364
positiveScale	0 to 1.0
processor	1 to 8
recording	<i>obsolete</i>
response	BestFreq, BetterTime, BetterFreq, or BestTime
samplesFileList	any list of sample file segments (used in <i>DiskSplicer</i>)
samplesFileName	any legal sample file name
scale	-1.0 to 1.0
shapeSource	wavetable or polynomial
singleWavetableDelay	0 to 4096 samp
sound	any Sound
soundCollection	any collection of one or more Sounds
spectrumFileName	any legal spectrum file name
spectrumWavetable	spectrum wavetable name (see <i>What is a Wavetable?</i> on page 493)
string	one or more characters
textFileName	any legal text file name
time	0 samp to 140737488355327 samp
upScale	-16.0 to 16.0
value	any Smalltalk value
wavetable	wavetable name (see <i>What is a Wavetable?</i> on page 493)
wavetableAccess	none, readWrite, readOnly, or writeOnly
wavetableDelay	0 samp to 4096 samp
wavetableIndex	0 to 1044480 (<i>obsolete</i>)
wavetableLength	0 samp to 1044480 samp (<i>obsolete</i>)
wavetableOfStandardLength	name of any 4096-sample wavetable (see <i>What is a Wavetable?</i> on page 493)
wordSize	8, 16, or 24

The type called `object` is the least restrictive of all parameter types and can include any Smalltalk object. The parameter type `value` includes any number or expression, but it does not include compound structures such as Sets or Arrays.

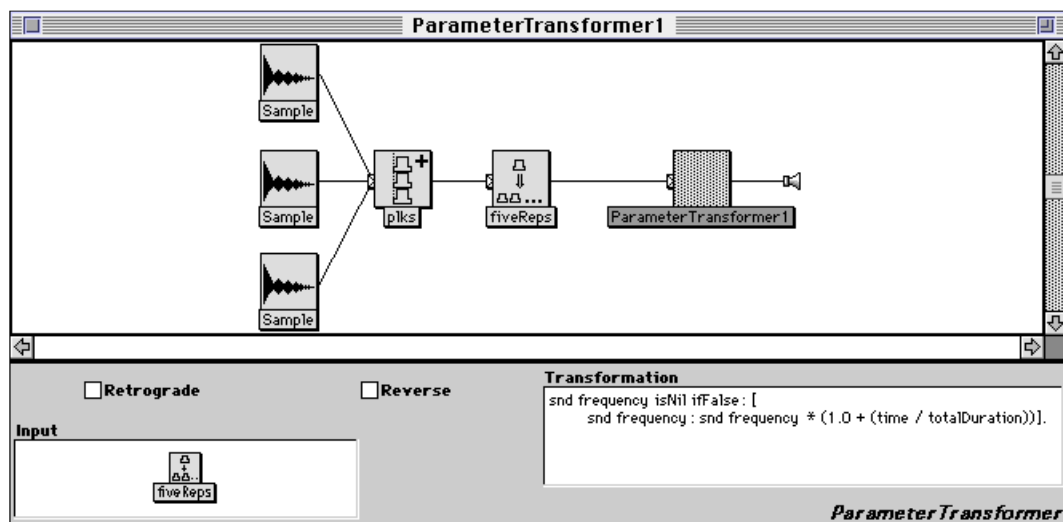
Parameter Transformers

Unlike a *Script*, which can generate any number of events from any number of inputs, a *ParameterTransformer* takes a single input and creates a single transformed version of it. When you play a *ParameterTransformer*, it resets or changes the specified parameter values of its input before beginning to generate sound. Only the initial parameter values are modified; no transformations take place while the Sound plays.

Because a *ParameterTransformer* can alter parameter values throughout the entire sound structure of its input, its use requires careful attention to the details of how each parameter functions within the Sound.

Transformations

The **Transformation** field of a *ParameterTransformer* takes any number of transforming statements. The parameters to be transformed are identified by name, and all parameters of that name throughout the input's structure are transformed.



Setting Parameters

To set a parameter in the input, ignoring the current value of the parameter, type the following into the **Transformation** field: `snd` followed by the name of the parameter, a colon, the value for that parameter, and then a period. (Make sure to use the parameter name exactly as it appears above the field in the Sound editor, but without the first letter capitalized.) For example,

```
snd left: 0.25.
```

will set all **Left** parameters to the value 0.25. Any Sound that has a **Left** parameter, wherever it is in the input's structure, will be set to 0.25. Any Sounds that do not have a left parameter will be unchanged. (Note that **Left** is actually the name of the parameter and not the name of a variable or Event Value within the parameter field.)

Transforming Parameters

You can use the current value of a parameter in computing a new parameter value. In order to obtain the current value of any parameter, use `snd` followed by the parameter name. Transforming statements can then use the current value of a parameter in calculating a new value for the same parameter. For example, the statement

```
snd frequency: snd frequency * 2.
```

would transform the input's frequency up an octave. One parameter can also be a function of another. For instance, the following transforming statement makes the input's duration a function of its frequency:

```
snd duration: (1.0 / snd frequency hz).
```

In this case, the duration of the input will be exactly one cycle long.

If a transforming statement tries to perform arithmetic on a parameter name that is not present in the input or one of its inputs, Kyma will return an error message, informing you that a message was sent to `nil`. For example, suppose the current Sound lacks the parameter **CmRatio**. When Kyma encounters the transforming statement

```
snd frequency: snd cmRatio * 0.5.
```

you will get an error for trying to multiply `nil` by `0.5`. To protect against using `nil` in a calculation, test the value and only perform the calculation if the value is not `nil`. For example, the following transforming statement is protected against calculating with `nil`:

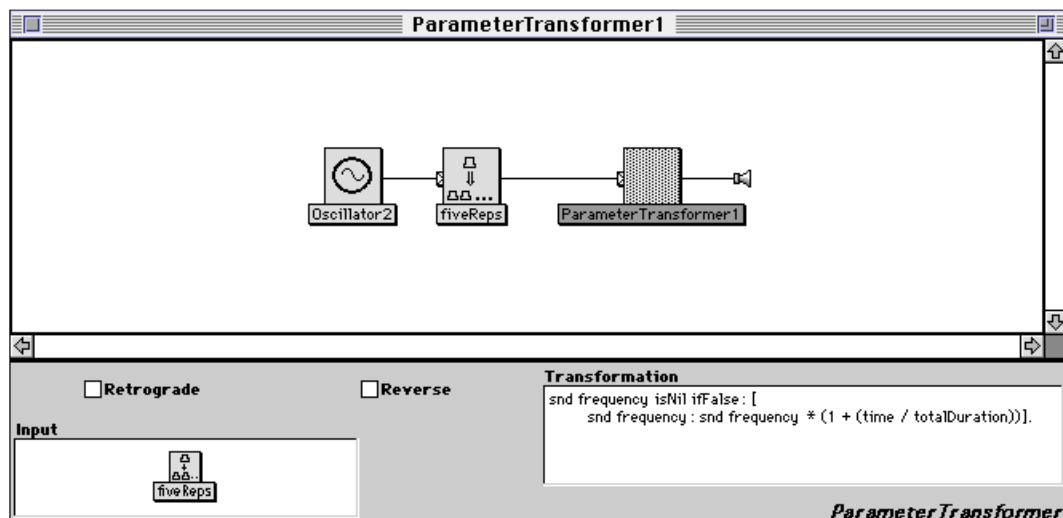
```
snd cmRatio isNil iffFalse: [snd frequency: snd cmRatio * 0.5].
```

The effect is to set the frequency to `cmRatio * 0.5` in Sounds where **CmRatio** is present and to skip over Sounds where that parameter is absent.

Time-varying Transformations

Two values, `time` and `totalDuration`, are available in *ParameterTransformers*. The value of `time` is the start time in microseconds of the Sound being transformed. The value of `totalDuration` is the duration in microseconds of the *ParameterTransformer's* input. You can use `time` and `totalDuration` to construct transformations that depend on the start time of an input within the structure.

Suppose the input of a *ParameterTransformer* is a *Repetition* of an *Oscillator*.



The result of the transformation

```
snd frequency: snd frequency * (1 + (time / totalDuration)).
```

is a sequence of Sounds with higher and higher frequencies.

Transformations are made to the *initial* values of a Sound's parameters; thus, any variation in the transformation over time is manifested only at the start time of each Sound, not over the course of a Sound and not on a sample-by-sample basis.

Skipping Levels

In the default case, a transformation is applied to each level of the input's structure. If you want to stop the transformation from continuing beyond a certain level of the sound structure, use

```
snd doNotTransformSubSounds.
```

To continue applying transformations to inputs, use

```
snd transformSubSounds.
```


Similarly, if you would like to transform only the inputs (that is, you would like the transformation to skip over the current Sound), you would use

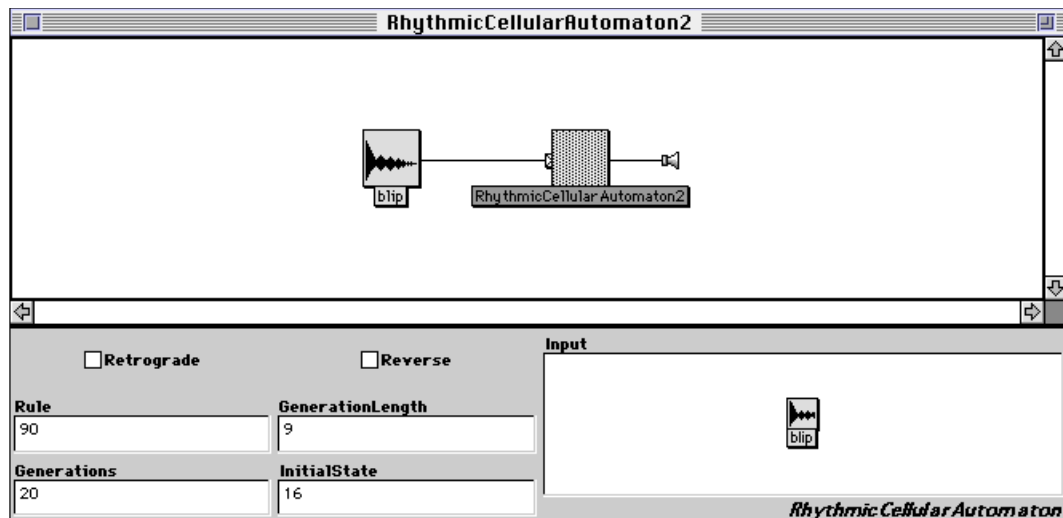
```
snd doNotTransformThisSound.
```

Controlling Expansion

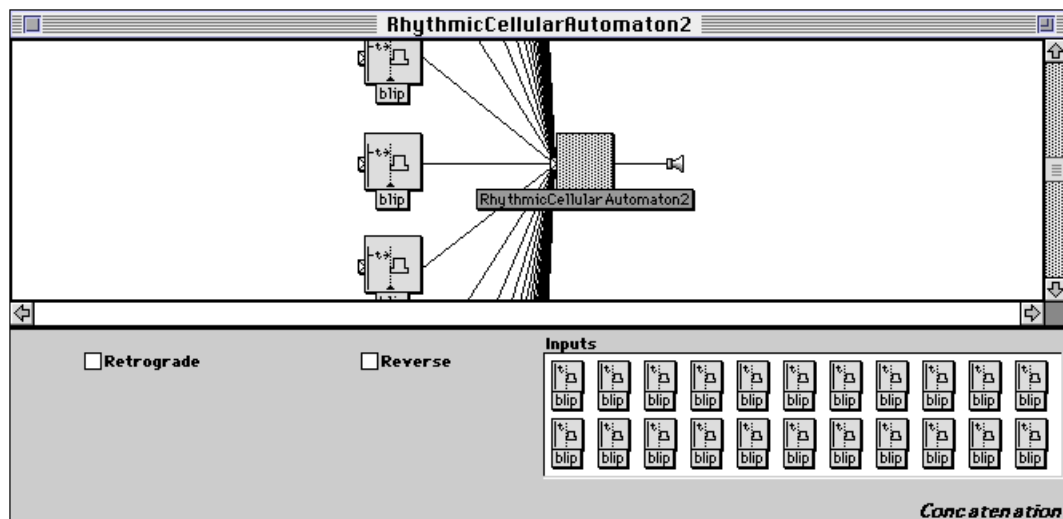
Most of the system prototypes provided in Kyma are actually combinations of several Sounds. When one of these prototype Sounds is played, it is first expanded. When you apply a transformation to the parameters of a Sound, it may make a difference whether you transform the parameters before or after the Sound has been fully expanded.

When you create a new Sound class from an example (see *The Class Editor* on page 536), you can choose whether the new kind of Sound should expand before transformations are applied to its parameters. If you check the **Expand** box, the Sound will expand before transformation; if the box is not checked, parameter transformations will be applied to the Sound parameters before the Sound expands.

For example, a *RhythmicCellularAutomaton* takes a single input and repeats it in a rhythmic pattern generated algorithmically according to the **Rule** that you supply. (See the *Prototypes Reference* on page 218 or the on-line help for a more detailed explanation of how the **Rule** is used to generate the pattern.)

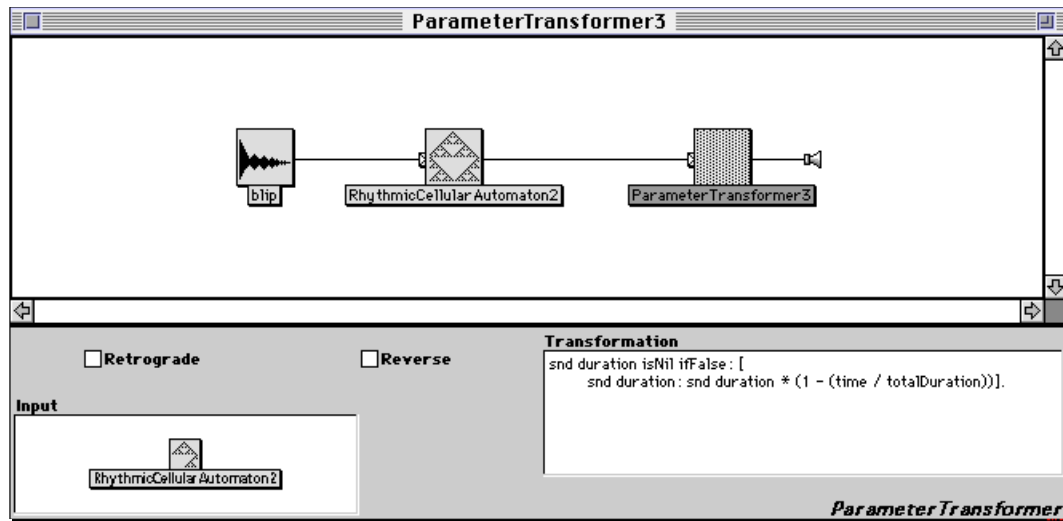


A *RhythmicCellularAutomaton* expands to a *Concatenation* of *TimeOffsets*. Each *TimeOffset* contains a copy of the *RhythmicCellularAutomaton*'s input.



Suppose you had a *ParameterTransformer* of a *RhythmicCellularAutomaton*, and that the **Transformation** field read

```
snd duration isNil iffFalse: [
  snd duration: snd duration * (1 - (time / totalDuration))].
```



You might expect the result to be the rhythmic pattern speeding up linearly over time. On the other hand, you know that a *ParameterTransformer* can only affect its input's initial state — the values of its parameters at its start time. Since the *RhythmicCellularAutomaton* has only a single input and since that input starts at time 0, you might expect the transformation to be

```
snd duration: snd duration * (1 - (0.0 / totalDuration)).
```

that is

```
snd duration: snd duration * 1.
```

In other words, you might expect the transformed *RhythmicCellularAutomaton* to be the same as the untransformed version.

What happens when you apply a time-dependent transformation to a *RhythmicCellularAutomaton* or, in fact, to any other Sound that expands when you play it? Since the *ParameterTransformer* first expands the *RhythmicCellularAutomaton* and then applies the transformation to the resulting *Concatenation*, the result is a rhythmic pattern that speeds up linearly with time, because each blip has a shorter duration than the one before it. On the other hand, the **SilentTime** parameter in the *TimeOffsets* remains untouched because, even though **SilentTime** is a duration, only the parameter fields named **Duration** are affected by the *ParameterTransformer*.

Suppose you wanted to be able to change the **Rule** parameter of the *RhythmicCellularAutomaton*. If you use the **Transformation**:

```
snd rule: 183.
```

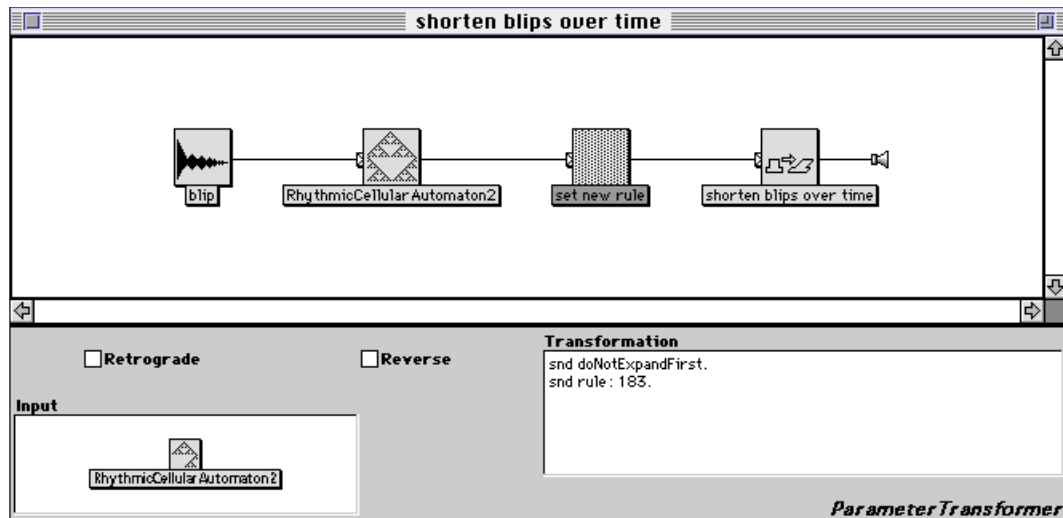
the **Rule** will be unchanged. Why? Because the *ParameterTransformer* first expands the *RhythmicCellularAutomaton* to a *Concatenation* and then tries to apply the transformation. Since there is no **Rule** parameter in a *Concatenation*, the transformation cannot be applied.

If you wanted to change the **Rule** parameter, you would have to override the default behavior of the *ParameterTransformer* by adding a line to the **Transformation**:

```
snd doNotExpandFirst.
snd rule: 183.
```

Then the *ParameterTransformer* would apply the transformation before expanding the *RhythmicCellularAutomaton*, and the rule would be modified.

What if you wanted to both transform the rule and cause the rhythmic pattern to speed up over time? In that case, you would use two nested *ParameterTransformers*:



The right one would have the **Transformation**:

```
snd doNotExpandFirst.  
snd rule: 183.
```

and its input would have the **Transformation**:

```
snd duration isNil ifFalse: [  
    snd duration: snd duration * (1 - (time / totalDuration))].
```

The first *ParameterTransformer* would change the **Rule** parameter of the *RhythmicCellularAutomaton*, and the next *ParameterTransformer* would expand the *RhythmicCellularAutomaton* using the new rule and would then apply the duration transformation.

Background Materials

Reference Books and Textbooks

Auditory Display

- Blattner, M. & R. B. Dannenberg, editors. 1992. *Multimedia Interface Design*. New York: ACM Press.
- Buxton, W., W. Gaver, and S. Bly. *Auditory interfaces: The Use of Non-Speech Audio at the Interface*. Cambridge: Press of the University of Cambridge, in final preparation.
- Kramer, G., editor. 1994. *Auditory Display: The Proceedings of the First International Conference on Auditory Display*. Addison Wesley. CD in a sleeve.

Acoustics

- Pierce, John. R. 1992. *The Science of Musical Sound, Revised Edition*. New York, New York: WH Freeman and Company.
- Rossing, T. D. 1992. *The Science of Sound, Second Edition*. Reading, MA: Addison-Wesley Publishing Company.
- Sundberg, J. 1992. *The Science of Musical Sounds*. Academic Press.

Sound Perception and Cognition

- Begault, D. 1994. *3-D Sound for Virtual Reality and Multimedia*. Cambridge: Academic Press.
- Blauert, J. 1983. *Spatial Hearing*. Cambridge: MIT Press.
- Bregman, A. 1990. *Auditory Scene Analysis: The Perceptual Organization of Sound*. Cambridge: MIT Press.
- Deutsch, D. 1982. *The Psychology of Music*. Academic Press.
- Gelfand, S. A. 1990. *Hearing: An Introduction to Psychological and Physiological Acoustics, Second Edition*. New York, NY: Marcel Dekker, Inc.
- Roederer, J. G. 1975. *Introduction to the Physics and Psychophysics of Music, Second Edition*. New York: Springer-Verlag, New York, Heidelberg, and Berlin.
- Zwicker, E. and H. Fastl. 1990. *Psychoacoustics: Facts and Models*. New York: Springer-Verlag.

Representation of Digital Audio Signals

- De Poli, G., A. Piccialli, and C. Roads. 1991. *Representations of Musical Signals*. Cambridge: MIT Press.
- Marsden, A. and A. Pople. 1991. *Computer Representations and Models in Music*. Academic Press.

Software Technology

- Pope, S. T., editor. 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press.
- Tarabella, L. 1992. *Informatica e Musica*. Milan: G. E. Jackson.

Sound Synthesis Algorithms

- Chamberlin, H. 1987. *Musical Applications of Microprocessors, Second Edition*. Hayden Books, Indianapolis, IN.
- Dodge, C. and T. A. Jerse. 1985. *Computer Music: Synthesis, Composition, and Performance*. Schirmer Books: New York, NY.
- Moore, F. R. 1990. *Elements of Computer Music*. Prentice Hall, Englewood Cliffs, NJ.
- Roads, C. 1993. *Computer Music Tutorial*. The M.I.T. Press, Cambridge, MA.

Wells, T. 1981. *The Technique of Electronic Music*. Schirmer Books, New York.

Digital Audio Signal Processing

Oppenheim, A. V. and R. W. Schaffer. 1975. *Digital Signal Processing*. Englewood Cliffs: Prentice-Hall, Inc.

Proakis, J. G. and D. G. Manolakis. 1988. *Introduction to Digital Signal Processing*. New York: Macmillan.

Strawn, J. ed. 1985. *Digital Audio Signal Processing*. Los Altos, CA: William Kaufman, Inc.

Digital Audio Engineering

Ballou, G., editor. 1987. *Handbook for Sound Engineers: The New Audio Cyclopedia*. Indianapolis, IN: Howard W. Sams & Company.

Benson, K. B., editor. 1988. *Audio Engineering Handbook*. New York, NY: McGraw-Hill Book Company.

Chamberlin, H. 1987. *Musical Applications of Microprocessors, Second Edition*. Hayden Books, Indianapolis, IN.

Olson, H. F. 1957. *Acoustical Engineering*. Professional Audio Journals.

Pohlmann, K. C. 1989. *Principles of Digital Audio*, Second Edition. Indianapolis, IN: Howard W. Sams & Company.

Strawn, J. 1985. *Digital Audio Engineering*. William Kaufman, Inc.

Watkinson, J. 1989. *The Art of Digital Audio*. Boston, MA: Focal Press.

Algorithmic Music Composition

Balaban, M. K. Ebcioglu, O. Laske. *Understanding Music with AI: Perspectives on Music Cognition*. Cambridge: The AAAI Press / The MIT Press.

Xenakis, I. 1992. *Formalized Music: Thought and Mathematics in Composition, Revised Edition*. Stuyvesant, NY: Pendragon Press.

MIDI and Real-time Event Processing

Braut, C. 1994. *The Musician's Guide to MIDI*. Sybex.

Rothstein, J. 1991. *MIDI: A Comprehensive Introduction*. A-R Editions.

Rowe, R. 1993. *Interactive Music Systems*. MIT Press. (includes CD-ROM with software and sound examples).

Yavelow, C. 1993. *The Macintosh Music and Sound Bible*. IDG Books.

History

Chadabe, J. 1997. *Electric Sound: The Past and Promise of Electronic Music*. Prentice-Hall.

Anthologies

Mathews, M. V., and J. R. Pierce, eds. 1989. *Current Directions in Computer Music Research*. MIT Press.

Roads, Curtis. 1989. *The Music Machine*. Cambridge, MA: MIT Press.

Roads, C. and J. Strawn. 1985. *Foundations of Computer Music*. MIT Press.

Proceedings of the 1977-1997 International Computer Music Conferences. International Computer Music Association.

Professional Societies

ACM SIGSound: the special interest group on sound computation

Audio Engineering Society

IEEE Acoustics Speech and Signal Processing

IEEE Task Force on Computer Music
International Computer Music Association (ICMA)
SEAMUS: Society for Electro-Acoustic Music in the United States

Conferences

Audio Engineering Society Conferences and Conventions
Acoustical Society of America Conferences
International Computer Music Conference
International Conference on Auditory Display
IEEE International Conference on Acoustics, Speech, and Signal Processing
SEAMUS Conference

Journals

Computer Music Journal
Journal of the Acoustical Society of America
Journal of the Audio Engineering Society
IEEE Signal Processing Magazine
IEEE Transactions on Signal Processing
IEEE Transactions on Audio
Journal of New Music Research
Leonardo Music Journal
Music Perception

Magazines

Audio Media
Electronic Musician
EQ
Future Music
Keyboard
Keyboards: Claviers–Informatique Musicale–Home Studio
Keys
Mix
Pro Sound News
Sound on Sound
Studio Sound