Sound and Music on the NeXT Computer™

by David Jaffe, Julius Smith, and Lee Boynton

This document is somewhat old and out of date. It is useful in that it provides an overview of the NeXT music software, but some details may have changed since it was written. For accurate details, see elsewhere in the ***LocalLibrary/Documentation/MusicKit+DSP*** *documentation.*

The NeXT Computer™ provides a powerful system for creating and manipulating sound and music. The top-level software for this system is divided into two object-oriented libraries known as *kits*:

- The Sound Kit™ provides object-oriented access to the basic sound capabilities of the NeXT Computer, allowing sound recording, playback, display, and editing.

- The Music Kit™ provides classes for composing, storing, performing, and synthesizing music. It lets you communicate with external synthesizers as well as create your own software instruments.

Both kits are implemented in Objective-C. While they are largely independent of each other, they can also be used to a unified end. For instance, with the Sound Kit you can record sound data that can be used in a Music Kit performance.

Below the Sound and Music Kits lies the DSP56001 digital signal processor (the *DSP*). The Music Kit uses the DSP as a general music synthesizer; the Sound Kit uses it to perform signal processing (such as format conversion).

This paper gives an overview of the NeXT Sound and Music kits and examines their use of the DSP, particularly with regard to music synthesis. A brief description of the NeXT sound hardware is also included.

Hardware

Voice-Quality Input

At the back of the monitor is a high-impedence microphone jack that's connected to an analog-to-digital converter known as the *CODEC*. The CODEC converter, complete with anti-aliasing filter on input, uses an 8-bit mu-law encoded quantization and a sampling rate of 8012.8 Hz. This is generally considered to be fast and accurate enough for telephone-quality speech input. The CODEC output interfaces to the rest of the system through a DMA interface implemented in a gate array.

The CODEC's mu-law encoding allows a 12-bit dynamic range to be stored in 8-bits. In other words, an 8-bit sound with mu-law encoding will yield the same amplitude range as an unencoded 12-bit sound.

High-Quality Sound Output

The stereo digital-to-analog converter operates at 44100 samples per second (in each channel) with a 16-bit linear quantization, the same as in commercial CD players. A 1 kHz maximum-amplitude sinusoid played through the DAC will generate a 2 volt RMS signal at the audio output. The converter includes full de-glitching and anti-aliasing filters so no external hardware is necessary for basic operation.

The NeXT computer contains a speaker built into the monitor as well as stereo headphone and line-out jacks at the back of the monitor allowing you to connect the computer to your stereo for greater playback fidelity. The volume of the internal speaker and headphones can be controlled from the keyboard.

The DSP

Digital Signal Processing (DSP) hardware includes:

- The Motorola DSP56001 clocked at 25MHz
- Memory-mapped and DMA access (5MBytes/sec) to the DSP host interface
- 8K 24-bit words of zero-wait-state RAM, private to the DSP
- D–15 connector bringing out the DSP SSI and SCI ports

The DSP hardware is described in greater detail later in this paper.

Conventions

The descriptions of the Sound and Music Kits use the following conventions regarding Objective-C syntax:

- All Objective-C class names begin with a capital letter.
- Class names are used to denote instances when the context is clear. For example, "Performers send Notes to Instruments" implies "instances of the

Performer class send instances of the Note class to instances of the Instrument class".

- Instance methods appear in boldface, as in **perform**.

The Sound Kit

The Sound Kit™ provides object-oriented access to the basic sound capabilities of the NeXT Computer, allowing sound recording, playback, display, and editing. It's designed to accommodate both casual use of sound effects as well as detailed examination and manipulation of sound data.

To store a sound, the Sound Kit can write its data as a *soundfile*, a file format provided by NeXT. The extensible, language-independent soundfile format is also used by NeXT's *pasteboard*, a data structure that lets applications share data.

The Sound Kit makes extensive use of the virtual memory and inter-process messaging provided by Mach, the operating system used by the NeXT Computer. This allows efficient manipulation of large sounds by minimizing data relocation.

The Sound Class

The most important class in the Sound Kit is Sound. It defines an Objective-C wrapper around the data structure that contains raw sound data and provides the basic recording, playback, and editing operations. The class also manages the binding of names to Sound objects, allowing a simple, symbolic way to locate and share sounds within an application.

Sound objects can be instantiated from a soundfile, from the pasteboard, or created empty (ready for recording). Reading from a soundfile is instantaneous—the file's pages are simply mapped. The data isn't actually brought in from disk until it's referenced.

Basic Sound Operations

To record into or play a Sound object, you simply send it a **record** or a **play** message. Recording takes data from the CODEC microphone input; playback sends the data to the internal speaker and to the headphone and line-out jacks. Digital sound can also be directed to (and taken from) the DSP port to allow external conversion.

Playback and recording operations are performed asynchronously by background *threads* (a thread is a lightweight Mach subprocess that shares address space with its parent process). When you invoke the **play** or **record** method, a playback or recording thread is instantiated and the method returns immediately. When the thread finishes, the Sound object sends a notification message to a user-settable object called a *delegate*. For example, the **soundDidPlay:** message is sent to the delegate when a Sound has finished playing. The delegate, which can inherent from any class, introduces a limited emulation of multiple inheritance.

At any time, Sounds can be written to a soundfile or copied onto the pasteboard. With the basic editing methods you can delete and copy a section of a Sound, and you can insert one Sound into another.

Sound Data Management

The Sound Kit rarely moves sound data; instead, it's remapped in virtual memory. Copying operations employ "copy on write" protocol: A copied section of sound is physically shared in memory until one copy is written, requiring the physical memory for that section of the copy to be allocated. This memory management is transparent to the programmer and user. For example, even though the sampled data in an edited Sound can become fragmented in memory, the Sound can be immediately played back at any time.

Sound Formats

A number of formats for representing sampled sounds are supported, from compressed, low-bandwidth sounds to high-fidelity, CD-quality data. To accommodate the DAC, which only accepts 16-bit data at either 22.05 or 44.1 kHz, the Sound object uses the DSP for run-time format conversion. Instantaneous playback of most of the available formats is thus made possible.

In addition to sampled sounds, the Sound class also supports DSP sound synthesis instructions: Instead of samples, the Sound object contains loadable DSP code and data streams. In either case, playback of a Sound is transparent. You never need to know whether a Sound contains samples or DSP code in order to play it. However, the basic editing operations only apply to sampled sounds; they can't be used to manipulate DSP code.

While Sound objects are multi-channel, the system throughput imposes a limit on the number of channels at a given sampling rate that can be played in real time. The standard sound output (44.1 kHz stereo, 16-bit linear data) is a good match for the throughput abilities of both the optical and the hard disk drives.

The SoundView Class

The SoundView class provides a mechanism for displaying the sampled data in a single Sound object. A SoundView object can draw itself on the screen, and can scale, translate, and rotate its coordinates. Currently, a sound can be displayed as an oscilloscopic waveform or as an outline of its amplitude envelope.

A SoundView maintains a *selection* that's generally defined by the user with the mouse, and can perform basic edit operations—such as cut and paste—on the selection. Like a Sound object, a SoundView maintains a user-settable delegate to which it sends notification messages when the selection changes.

An example SoundView is shown in figure 1.

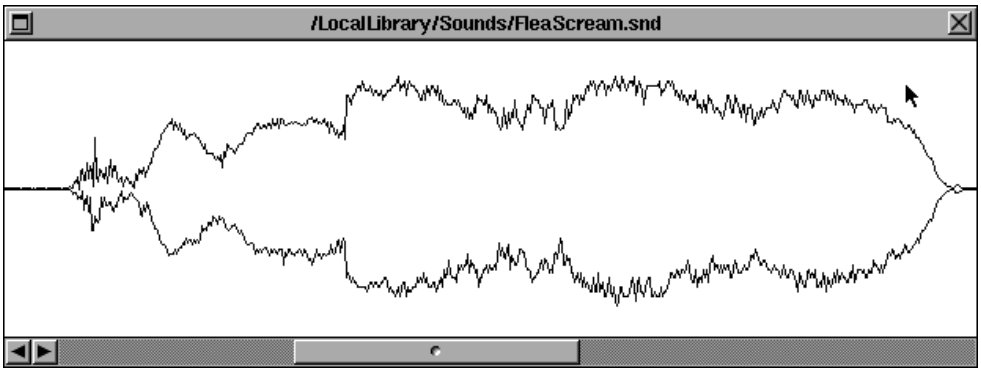

Figure 1; A SoundView

The Music Kit

The Music Kit™ provides tools for designing music applications. These tools address three topics: music representation, performance, and synthesis (digital sound generation and processing). The Objective-C classes defined in the Music Kit fall neatly into these three areas and are presented as such below.

The design goal of the Music Kit is to combine the interactive gestural control of MIDI with the precise timbral control of MUSIC 5-type systems in an extensible, object-oriented environment. To this end, the Music Kit is capable of fully representing MIDI. The Music Kit accepts MIDI in and can send MIDI out through the two serial ports at the back of the computer. Nonetheless, the Music Kit isn't limited by the MIDI specification; for example, its resolution of frequency and amplitude is much finer than MIDI's highly quantized values.

The Music Kit generates sounds by sending synthesis instructions to the DSP. The generality of the synthesis software far surpasses that of commercial synthesizers. While most synthesizers employ only one type of synthesis—the Yamaha DX–7 uses only frequency modulation, for example—the Music Kit can implement virtually any sound synthesis strategy. And since the synthesis engine (the DSP) and the control stream are brought together in a single high-performance computer, the Music Kit makes possible an unprecedented level of expressive control.

Music Representation

Music is represented in a three-level hierarchy of Score, Part, and Note objects. Scores and Parts are analogous to orchestral scores and the instrumental parts that they contain: a Score represents a musical composition while each Part corresponds to a particular means of realization. Parts consists of a time-sorted collection of Notes, each of which contains data that describes a musical event. When you play MIDI into a Music Kit application, the various MIDI channels become separate Part objects and the MIDI commands are turned into Notes.

A Score can contain any number of Parts and a Part any number of Notes. Methods are provided for rapid insertion, deletion, and lookup of Notes within a Part and Parts within a Score.

The Note Class

The Note is the basic package of musical information. The information in a Note object falls into four categories:

- A list of attribute-value pairs called *parameters* that describe the characteristics of a musical event
- A *noteType* that determines the general character of the Note
- An identifying integer called a *noteTag*, used to associate different Notes with each other
- A *timeTag*, or the onset time of the Note

Parameters

A parameter supplies a value for a particular attribute of a musical sound, its *frequency* or *amplitude*, for example. A parameter's value can be simple—an integer, floating point number, or character string—or it can be another object. The Note object provides special methods for setting the value of a parameter as an Envelope object or a WaveTable object. With the Envelope object you can create a value that varies over time. The WaveTable object contains sound or spectrum data that's used in wavetable synthesis.

The manner in which a parameter is interpreted depends on the Instrument that realizes the Note (the Instrument class defines the protocol for all objects that realize Notes). For example, one Instrument could interpret a heightened *brightness* parameter by increasing the amplitude of the sound, while another Instrument, given the same Note, might increase the sound's spectral content. In this way, parameters are similar to Objective-C messages: The precise meaning of either depends on how they are implemented by the object that receives them.

NoteTypes and NoteTags

A Note's noteType and noteTag are used together to help interpret a Note's parameters. There are five noteTypes:

- NoteDur represents an entire musical note (a note with a duration)
- NoteOn establishes the beginning of a note
- NoteOff establishes the end of a note
- NoteUpdate represents the middle of a note
- Mute is general-purpose; its use is defined by the application

NoteDurs and noteOns both establish the beginning of a musical note. The difference between them is that the noteDur also has information that tells when the note should end. A note created by a noteOn simply keeps sounding until a noteOff comes along to stop it. In either case, a noteUpdate can change the attributes of a musical note while it's sounding. The mute noteType is used to represent any additional information. For example, you can represent barlines and rehearsal numbers in Notes of type mute.

A noteTag is an arbitrary integer that's used to identify different Notes as part of the same musical note or phrase. For example, a noteOff is paired with a noteOn by matching noteTag values. You can create a legato passage with a series of noteOns, all with the same noteTag, concluded by a single noteOff.

The Music Kit's noteTag system solves many of the problems inherent in MIDI, which uses a combination of key number and channel to identify events that are part of the same musical phrase. For example, the Music Kit can create and manage an unlimited number of simultaneous legato phrases while MIDI can only manage 16 (in MIDI mono mode). Also, with MIDI's tagging system, mixing streams of notes is difficult—notes can easily get clobbered or linger on beyond their appointed end. The Music Kit avoids this problem by reassigning unique noteTag values when streams of Notes are mixed together.

TimeTags

A Note's timeTag specifies when the Note is to be performed. TimeTag values are measured in *beats* from the beginning of the performance, where the value of a beat can be set by the user. If the Note is a noteDur, its duration is also computed in beats.

Storing Music

An entire Score can be stored in a *scorefile*. The scorefile format is designed to represent any information that can be put in a Note object, including the Part to which the Note belongs. Scorefiles are in ASCII format and can easily be created and modified with a text editor. In addition, the Music Kit provides a language called *ScoreFile* that lets you add simple programming constructs such as variables, assignments, and arithmethic expressions to your scorefile.

Music Kit Performance

During a Music Kit performance, Note objects are dispatched, in time order, to objects that realize them in some manner—usually by making a sound on the DSP or on an external MIDI synthesizer. This process involves, primarily, instances of Performer, Instrument, and Conductor:

- A Performer acquires Notes, either by opening a file, looking in a Part or Score, or generating them itself, and sends them to one or more Instruments.
- An Instrument receives Notes sent to it by one or more Performers and realizes them in some distinct manner.
- The Conductor (there's usually only one Conductor object per performance) acts as a scheduler, ensuring that Notes are transmitted from Performers to Instruments in order and at the right time.

Before a Performer can send a Note to an Instrument, the two objects must be connected. A single Performer has a collection of outputs, each of which can be connected to any number of Instrument inputs. Performance connections are dynamic: You can connect and disconnect Instruments and Performers during a performance.

The Conductor provides control over the timing of a performance by letting you set the tempo dynamically as well as pause and resume an entire performance. You can also pause and resume individual Performers. A method is provided for updating the Conductor's notion of the current time when an asynchronous event, such as a mouse click, is received. This makes it easy to incorporate the actions of the user in an interactive application.

This system is useful for designing a wide variety of applications that process Notes sequentially. For example, a Music Kit performance can be configured to perform MIDI or DSP sequencing, graphic animation, MIDI real-time processing (such as echo, channel mapping, or doubling), sequential editing on a file, mixing and filtering of Note streams under interactive control, and so on.

Performer and Instrument Subclasses

Both Performer and Instrument are abstract classes. This means that you never create and use instances of these classes directly in an application. Rather, they define common protocol (for sending and receiving Notes) that's used by their subclasses. The subclasses build on this protocol to generate or realize Notes in some application-specific manner.

The Music Kit provides a number of Performer and Instrument subclasses. The principle Performer subclasses are:

- ScorePerformer and PartPerformer. These read Notes from a designated Score and Part, respectively. ScorePerformer is actually a collection of PartPerformers, one for each Part in the Score.
- ScorefilePerformer reads a scorefiles, forming Note objects from the contents of the file.
- MidiIn creates Note objects from the byte stream generated by an external MIDI synthesizer attached to a serial port.

The Instrument subclasses provided by the Music Kit are:

- SynthInstrument objects realize Notes by synthesizing them on the DSP.
- MidiOut turns Note objects into MIDI commands and sends the resulting byte stream back out to an external MIDI synthesizer connected to a serial port.
- ScoreRecorder and PartRecorder receive Notes and add them to a Score and Part, respectively.
- ScorefileWriter writes scorefiles.
- NoteFilter is a subclass of Instrument that also implements Performer's Note-sending protocol, thus it can both receive and send Notes. Any number of NoteFilter objects can be interposed between a Performer and an Instrument. NoteFilter is, itself, abstract. The action a NoteFilter object takes in response to receiving a Note is defined by the subclass. For example, you can create a NoteFilter subclass that creates and activates a new Performer for every Note it receives.

Music Synthesis

By using the DSP for music synthesis, the Music Kit can generate sound with an attention to detail that equals MUSIC 5-type systems—and it can do so in real time, without first writing the results as sampled data.

The principal synthesis classes are UnitGenerator, SynthData, SynthPatch, and SynthInstrument:

- UnitGenerators and SynthData are the basic building blocks of DSP synthesis; they correspond directly to code or data on the DSP.
- A SynthPatch object is a configuration of SynthElements that defines a particular synthesis strategy (e.g. frequency modulation).
- SynthInstrument is a subclass of Instrument that realizes Notes by assigning them to particular SynthPatch instances. It performs what in MIDI synthesizers is called "voice allocation".

Another class, Orchestra, is provided to manage the DSP for you. For instance, allocation of all UnitGenerators, SynthData and SynthPatch objects is handled by the Orchestra. Each DSP is represented by a single Orchestra object. The basic NeXT configuration has one DSP; thus there is ordinarily only one instance of Orchestra. The state of an Orchestra can be saved as DSP code in a soundfile.

The UnitGenerator and SynthData Classes

Each subclass of UnitGenerator, an abstract class, implements a particular synthesis function, defined by the corresponding DSP unit generator 56000 assembly code macro. The Music Kit supplies classes that implement oscillators, envelope handlers, filters, mixers, and so on. In addition, you can create your own UnitGenerator subclasses from DSP macro source using the program **dspwrap**.

SynthData objects correspond to DSP data memory. An important use of SynthData is to provide *patchpoints* or locations that can be written to by one UnitGenerator and read by another. For example, simple frequency modulation can be implemented by setting the output of an oscillator to write to a patchpoint that's read by the frequency input of another oscillator. SynthData objects can also be used to hold wavetables, delay memory, constants, and so on.

The SynthPatch Class

SynthPatch is also an abstract class; each subclass represents a configuration of UnitGenerator and SynthData objects. The SynthPatch class defines a standard set of methods that are implemented by each subclass to define the manner in which a Note's parameters are applied. In general, the manner in which parameters are interpreted depends on the Note's noteType:

- A noteOn heralds a new Note stream (a phrase consisting of Notes with the same noteTag) or rearticulation of an existing stream.
- When a noteUpdate is received, the SynthPatch makes a transition to accommodate the new parameter values.
- A noteOff causes the SynthPatch to wind down. For example, if envelope handlers are used, they are told to begin the release portion of the envelopes. The SynthPatch will continue to respond to noteUpdates after it receives a noteOff and before it reaches the actual end of the Note.
- A noteDur is treated like a noteOn. A noteOff is automatically created to correpond to the end of the noteDur.

The SynthInstrument Class

Each SynthInstrument object knows how to create and manage instances of a particular SynthPatch class. When it receives a Note, the SynthInstrument decides whether to create a new SynthPatch instance or to apply the Note to a SynthPatch that's already running. It makes this decision by looking at the Note's noteTag.

Each SynthPatch instance that a SynthInstrument manages corresponds to a particular noteTag. When a Note is received, the SynthInstrument checks to see if a SynthPatch with that noteTag is running, passing the Note along to the SynthPatch if it exists, creating a new SynthPatch if it doesn't. The total number of SynthPatches that a SynthInstrument can create is limited by the memory restrictions of the DSP. If a SynthInstrument can't find the resources to play a new Note, it normally preempts its oldest running SynthPatch. However, you can subclass SynthInstrument to provide a different preemption strategy, such as one based on amplitude.

Allocation can be done automatically or manually. In automatic allocation, new SynthPatches are obtained from the Orchestra and returned to the Orchestra. Thus, SynthPatches are shared among all SynthInstruments. In manual allocation, each SynthInstrument is assigned a fixed pool of

SynthPatches from which it allocates particular instances. Manual allocation requires more forethought, but it can also result in more efficient use of DSP resources.

Music Processing

Music processing is a trivial variant of music synthesis. You merely use a different UnitGenerator in your SynthPatch—the In1aUG reads the left channel of the sound input stream, while the In1bUG reads the right channel of the sound input stream.

Real-Time Issues

The DSP can only do so much computing and remain in real time. In cases where the DSP can't keep up with real time, music can be precomputed, possibly in multiple passes, and stored in a soundfile for later playback.

Any interactive music application requires that the latency between a user's action and its musical result be kept to a minimum. MIDI synthesizers, for example, operate in this realm—the lag between pushing down a key and the instantiation of the note must be imperceptible. We call this *critical real time*. The Music Kit design philosophy is to provide critical real-time response without sacrificing generality, extensibility and clarity of design.

Sound and Music Kit Summary

The Sound and Music Kits both provide Objective-C classes supporting music and sound. While they are largely independent of each other, they also interact. For instance, with the Sound Kit you can record sound data that can be played in a Music Kit performance. In the other direction, we plan to provide a soundfile type that contains an arbitrary musical performance using the Music Kit. Already, Music Kit UnitGenerators are being used to build system beep-type sound effects. Short, DSP-generated sounds take up much less disk space than sampled data, and usually make no use of the disk during playback.

The Sound and Music Kits make it easy to design and use sound and music applications on the NeXT Computer. Nonetheless, the Kits are simply tool boxes, they aren't applications themselves. Their vitality will be determined by the imagination of the software developers who build on their foundation. We feel that the concepts on which the Sound and Music Kits are based provide a framework capable of meeting the needs of a wide variety of application developers.

Figure 2 shows the components for creating, playing, and storing music and sound with the hardware and software of the NeXT Computer.

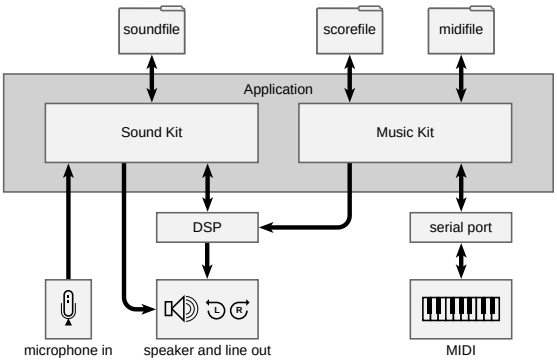

Figure 2; Sound and Music on the NeXT Computer

The DSP

The Motorola DSP56001 microprocessor is a state-of-the-art digital signal processor that can execute 12.5 million instructions per second and, in a single instruction, can perform a 24 by 24-bit multiply, a 48 plus 56-bit addition, two parallel data moves, an instruction fetch, and two general index updates. The 24-bit data paths and architecture optimized for digital signal processing make the DSP56001 exceptionally well suited for digital audio.

The DSP host interface is connected to the host processor through a high-speed DMA interface (one of twelve independent DMA channels) implemented in a large gate array known as the *DMA chip*. Moreover, the DMA chip provides memory-mapped access to the eight byte registers of the DSP host interface.

DSP on-chip memory is divided into three banks of 512 24-bit words. The *p* memory bank provides 512 words of program RAM on chip. The *x* and *y* memory banks each provide 256 words of on-chip data ROM (read-only memory) and 256 words of on-chip data RAM. The internal x ROM contains mu-law and A-law expansion tables, while the internal y ROM contains one complete cycle of a sine wave. The concatenation of x and y memories into a single 48-bit word memory is referred to as the *l* memory space.

Off-chip DSP memory on the NeXT machine exists in two address ranges, each of which spans all of external memory. In the first address range (8K to 16K), x, y, and p memories are overlaid, that is, an external memory reference points to the same off-chip location regardless of the memory space specified. Note that in this address range, there is no l memory space support. (The high and low word are mapped to the same word.) In the second address range, 40K to 48K, x and y are split into separate 4K partitions, and p is overlays the entire 8K. This address region allows external l memory use, and supports algorithms (such as the Motorola benchmarks involving complex data) which expect x and y memories to be physically separate.

The three on-chip DSP memory banks can be accessed in parallel, with as many as three moves in one instruction cycle (80ns). Off-chip DSP memory can't be accessed in parallel, requiring three instruction cycles to move a word in all three memory spaces.

DSP Software

This section describes the mapping of a general purpose computer music system to the DSP56001 architecture.

The Music Kit DSP Monitor provides timed-message support, *unit generators* (DSP subprograms) for generating computer music, audio buffering and DMA support, and other features needed by the Music Kit.

The **dspwrap** program creates an Objective-C class from a unit generator macro (written in DSP assembly language). Each of these classes is a subclass of the Music Kit's UnitGenerator. A number of UnitGenerator subclasses are provided by the Music Kit. These include signal generators, digital filters, mixers, and envelope handlers.

The Music Kit DSP Monitor

This section outlines aspects of the DSP Monitor that support the downloading and control of DSP musical instruments in real time.

The Orchestra Loop

The orchestra loop is a DSP program that executes a large block of code repetitively, once for each *tick*. A tick is some number (currently 8) of digital sound samples. If not for efficiency considerations, an orchestra program would compute a single sample of digital sound on each iteration. The architecture of the DSP makes it much more efficient to compute digital sound in little blocks called ticks.

The following is an example orchestra program. The user prepares such a program only when debugging newly developed unit generators. Unit generators managed by the Music Kit are dynamically loaded into the DSP, and the orchestra program is built up inside the DSP on the fly at run-time. We show it here to illustrate its structure:

```
;; test.asm
;; To assemble: asm56000 -a -b -l -I/usr/include/dsp/smsrc/ test
;;
    include 'music_macros'
    ; utility macros
    beg_orch 'test'          ; begin orchestra main program
        new_yib yvec,8,0
    ; Allocate y output vector
        beg_orcl             ; begin orchestra loop
            unoise orch,1,y,yvec,0
        ; make noise, seed 0
            out2sum orch,1,y,yvec,0.5,0.5
    ; center it (.5,.5)
        end_orcl             ; end of orchestra loop
    end_orch 'test'          ; end of orchestra main program
```

The example allocates a length 8 signal vector called *yvec* in internal y DSP memory with the invocation of the **new_yib** macro. Inside the orchestra real-time loop, delimited by the **beg_orcl** and **end_orcl** macros, there is an instance of the **unoise** unit generator which simulates white noise, and an instance of the **out2sum** unit generator which sums its argument into the sound-out buffer. (This buffer is cleared by code emitted in the **beg_orcl** macro.) The **end_orcl** macro simply jumps back to a label at the beginning of the orchestra loop defined in **beg_orcl**. This loop executes until the chip is reset or Host Flag 0 is set in the DSP host interface. This is all there is to an orchestra program running in the DSP.

More elaborate examples are built up by inserting more unit generators like **unoise** into the orchestra loop. To do this while the loop is running requires precisely timed software downloads. To be able to drastically rebuild the orchestra loop between tick computations, there needs to exist an adequate supply of buffering of the output sound data in the DSP. We currently use 1024 words total for the dual stereo sound-out DMA buffers.

The macro **beg_orcl** emits code to perform the following once-per-tick services in the orchestra loop:

- Request DMA transfer of the completed sound-out buffer if necessary.
- Compare the time stamp of the next timed message (queued in DSP memory) to the current DSP time (in samples), and execute all messages timed for "now".
- Add 8 to the 48-bit current time variable to update it for one iteration of the orchestra loop.
- Reset the three unit-generator memory argument pointers R_X, R_Y, and R_L to their beginning values for the orchestra loop.

The Importance of Vectorized Computations

In the current implementation, the tick size is 8 samples. Since the overhead of setting up DSP index and ALU registers is often comparable to the amount of actual work done in the inner loop of a tick computation, a tick size of 8 brings the set-up overhead for each unit generator close to ten percent in most cases.

It's important not to make the tick size any larger than necessary because it also determines the size of a *patchpoint*, memory that's used for communication between unit generators. It's highly desirable that all patchpoints fit on-chip in the DSP. This is because the three-way parallel data move capability of the DSP requires that at least two of the data moves be to or from on-chip memory. Only one parallel external memory read or write is possible because only one set of data and address pins is brought out of the chip.

The hardware DO instruction in the Motorola DSP56001 further enhances the benefits of vectorized computations by performing the loop test and branch in parallel with the block iteration; while there is a three instruction-cyle (six clock cycle) overhead incurred to set up the loop, the individual loop iterations suffer no test and branch overhead.

Finally, the dual parallel indexing ALUs provide zero-overhead memory address updates for two parallel data transfers, with skip factors, modulo (wrap-around) addressing, and even bit-reverse indexing for FFT data shuffling provided as indexing modes.

Thus, vectorized computations are far more efficient than computing a single audio sample per iteration of the orchestra loop. The price for this efficiency is a loss of control bandwidth since parameter updates (envelope break-points, filter coefficients, etc.) are only installed once at the beginning of each tick.

Unit Generators

The *unit generator* is a fundamental building block of sound synthesis. It can also be regarded as one computational "black box" in a real-time signal processing diagram. Almost every unit generator has an output signal that it writes into a patchpoint every time it runs (once per tick). At the heart of every unit generator is a DO loop that executes 8 times (once per sample) to produce an output tick. Most unit generators also have one or more signal inputs that are also 8-sample long vectors.

By carefully arranging the order of execution of unit generators within the orchestra loop, it's often possible to significantly reduce the number of patchpoints required. For example, if unit generators A, B, and C are arranged in a linear chain, i.e., A—> B—> C, and if no other unit generators depend on their outputs, then only one patch-point is needed if A runs before B and B runs before C. The patchpoint between A and B is simply reused as the patchpoint between B and C.

Note also that the order in which unit generators are executed in the orchestra loop determines whether or not there is an 8-sample delay in the connection between them. For example, if in the above example, the order of execution is C,B,A, then then C's output will be delayed two ticks relative to A's output plus whatever delay is built into B and C. For this reason, the Music Kit provides a way to control the order of execution of unit generators.

The following is a partial list of the NeXT unit generators:

add2 - add two signals to produce a third
allpass1 - one-pole digital allpass filter section
asymp - one segment of an exponential (ADSR type) envelope
biquad - direct-form, two-pole, two-zero, filter section
constant - generate a constant signal
delay - sample-based delay line using non-modulo indexing
delayticks - tick-based delay line using non-modulo indexing
dswitch - switch from input 1 to input 2 after delay
impulses - periodic impulse-train generator
mul2 - multiply two signals to produce a third
onepole - one-pole digital filter section
onezero - one-zero digital filter section
orchloopbegin - begin DSP orchestra loop (invokes **beg_orcl** macro only)
orchloopend - end DSP orchestra loop (invokes **end_orcl** macro only)
oscg - simplest oscillator with general address mask
oscgaf - oscillator with amplitude and frequency envelopes
oscgf - oscillator with multiplicative frequency input
osci - interpolating oscillator
oscs - simplest oscillator
oscw - oscillator based on 2D vector rotation
out2sum - sum signal vector into sound output buffer.
patch - patch one signal vector to another
sawtooth - sawtooth oscillator
scale - scale a signal vector by a scalar using mpyr
scl1add2 - add scaler times first signal to the second
slpdur - linear envelope generation using slopes/durations
twopole - two-pole digital filter section
unoise - uniform pseudo-random number generator
unoisehp - highpassed uniform pseudo-random number generator

Appendix A provides more information about the unit generator implementation.

DSP Messages

A DSP message is an unsolicited message from the DSP to the host processor. When the DSP writes a word to its Transmit Data registers, an interrupt is generated in the host. In response, the host reads as many words as possible and places them into a Mach message which the user can elect to receive on a particular Mach port.

Each DSP message is 3 bytes. This is to allow a DSP message to be atomically written into the host-interface Transmit Byte Registers. The first byte is an opcode, and it is followed by two data bytes. Opcodes between 0 and 127 are normal messages, and opcodes between 128 and 255 are error messages (the MSB of the first byte is set if and only if the DSP message is an error message of some kind). Error message are separated from other DSP messages by the Mach driver so that they can be received on separate ports. DSP error messages are normally followed immediately by a PC back-trace, the time the error occurred (in samples), and other transient state information.

Inside the DSP, outgoing messages are enqueued, and the queue is emptied by the Host Transmit Data interrupt which strikes every time the host reads a word from the DSP. If the DSP message queue fills up, the DSP blocks, unless a flag has been set which allows overwriting of unsent DSP messages in the circular buffer.

If a DSP-to-host DMA transfer is in progress, DSP messages can be enqueued, but they will not be sent until after the DMA terminates. If a host-to-DSP DMA transfer is in progress, DSP messages can be sent, but there is no interrupt of the host. Instead, the user task must read the host interface register ISR containing the RXDF flag. When this flag is set, it means there is data in the receive byte registers RXH, RXM, and RXL.

Host Messages

A host message is a message from the host processor to the DSP. There are three types:

1. Untimed
2. Timed (absolute or relative)
3. Timed with a time stamp of 0

In more detail,

(1) Untimed messages are executed immediately at interrupt level on the DSP.

(2) Timed messages are copied to a large Timed Message Queue (TMQ) within the DSP (currently close to 1000 words). They are executed when the sample counter reaches or exceeds the time stamp of the message. Relative times (not used by the Music Kit) are measured from the DSP sample counter at the time the DSP is copying the message to the TMQ. Only “real time” applications can use timed messages, for unless there is an active orchestra loop, no sample counter is maintained. (“Real time” doesn’t necessarily mean true real time, e.g., when directing output to disk.)

(3) A zero time stamp means “now” but defferred to the end of the current tick. This is useful for real-time events when an orchestra loop is running. The messages will bypass the timed message queue, but they will not update unit generator parameters at an unpredictable point in the orchestra loop execution.

Untimed and timed-zero messages are processed at a higher priority than timed messages. This is so that they will bypass messages stored up in the timed message queue.

The easiest way to execute a host message from C is to use the function DSPCall from libdsp. The format of this call is

```
ierr = DSPCall(Opcode,Nargs,Arg1,...,ArgNargs);
```

where all arguments are of type int, and *Opcode* is the actual address of the DSP subroutine, or the address of its permanently allocated dispatch address. The timed version is

```
ierr = DSPCallTimed(timeStamp,Opcode,Nargs,Arg1,...,ArgNargs);
```

A host message consists of a number of writes to the TX registers followed by a host command. Each TX write by the host interrupts the DSP. A fast, two-word interrupt handler pushes the word onto the Host Message Stack (HMS). Thus, the orchestra loop continues to execute as the arguments of the host message trickle in from the host. The last argument written is the dispatch address. Finally, a host command kicks off processing of the message.

Host Commands

DSP host commands are interrupts generated in the DSP when the host writes the Command Vector Register (CVR) of the Host Interface (cf. the DSP56000 User’s Manual, p. 7–9). There are a total of 32 types of interrupts possible within the DSP. The first 19 are predefined by Motorola (cf. p. 8–6). The remaining 13 interrupt cases are called *host commands* and are either defined by NeXT or available for definition and support by the user.

The DSP Mach driver reserves two DSP host-commands:

- DSP_HC_DMAWT - DMA write termination (host-to-DSP)
- DSP_HC_KERNEL_ACK - Interrupt kernel when DSP can accept data

The DSP_HC_DMAWT host command terminates DMA writes to the DSP; it’s necessary because host-receive interrupts are usurped by the DMA transfer when the direction is into the DSP, thereby stifling host messages.

The DSP_HC_KERNEL_ACK host command provides a way for the kernel to sleep when the DSP busy, and obtain an interrupt when the DSP is no longer busy. This works because the DSP can accept exactly one pending host command while it has host interrupts disabled. When the DSP enables host interrupts, the **ack** host message is executed, and the result is the DSP message DSP_DM_KERNEL_ACK which causes an interrupt of the host. The kernel intercepts the DSP message and goes on with whatever it was doing when the DSP originally blocked (typically sending a series of messages to the DSP).

The Music Kit DSP Monitor additionally defines one more DSP host-command:

- DSP_HC_XHM - Execute host message

The DSP_HC_XHM host command triggers all host messages to the DSP. It causes the DSP to jump to the address sitting on top of the HMS. This will initiate a subroutine call which implements the host message and consumes the arguments on the HMS.

When processing a host message, the DSP sets the “DSP Busy” flag (HF2). The host looks for this flag to clear as a signal that the message has been digested. (More precisely, to avoid a race, the host must watch for HC to clear, wait at least 400ns, and then wait for HF2 to clear.)

Host Message Format

A host message typically contains a variable number of arguments followed by an opcode (dispatch address) in the lower two bytes or’d with a time stamp type (absolute, relative, or untimed) in the upper byte. Timed host messages include also a time stamp after the opcode:

<Arg1>...<ArgN> {<TimeStamp>} <TimeStampType OpCode> [<hostCommand>]

Timed Message Queue (TMQ)

Timed host messages are enqueued on the TMQ. The TMQ is a circular buffer similar to the DMA buffers and the HMS. Relative time is converted to absolute time before enqueuing. Timed messages can be inserted only at the top of the queue which means timed messages must be sent in time order. Real-time preemption of the timed message queue is accomplished using untimed or timed-zero host messages. If the time stamp is earlier than that of the current DSP time, whether because a message arrived out of order or because the host process sent the message too late, it will be executed immediately, and an underrun condition will be logged.

When a host message of maximum size will not fit in the TMQ, host flag HF3 is set in the DSP host interface to tell the driver not to send any more. As soon as a maximum length message can fit, HF3 clears, and the driver will resume sending timed messages. The maximum host message size is currently 30 words for arguments, opcode, and time stamp (if any). Low-level routines in the DSP C library break up long timed messages into digestible chunks. This same software layer also optimizes by combining short messages with the same time stamp into a single Mach message.

The maximum absolute time representable in the TMQ is $2^{47}-1$ samples (approximately one century). The maximum relative time is $2^{24}-1$ samples, which is about 6 minutes at 44.1 kHz.

Untimed Message Queue (UTMQ)

The UTMQ contains all timed messages with a time stamp less than or equal to the current sample number at the time the message was received. The contents of the UTMQ are executed at the beginning of each tick before checking the TMQ.

Implementation of Queues

All Queues (FIFOs) are implemented using the *modulo storage* feature of the DSP56001. The benefit of doing this is memory protection. For example, DMA transfers are double-buffered; a two-word interrupt handler, driven by data transmission interrupts, is set-up which looks, in the host-message handling case, as follows:

```
movep x:$FFEB,y:(R_HMS)+
nop
```

Without modulo storage (M_HMS set to the double-buffer length minus 1), it would be possible for an unchecked transfer to wipe out DSP y memory. Using modulo storage, the transfer can only spin around inside the storage ring.

Message queues are additionally protected by special markers at the beginning and end. They are checked to make sure they have not been overwritten.

User Memory Segments in the DSP Orchestra

Each user memory space (x,y,p) has an upper, middle, and lower segment, corresponding to the three location counters in each memory space provided by the Motorola DSP assembler. The following table describes their use by the Music Kit DSP Monitor:

Space	Lower Segment	Middle Segment	Upper Segment
p	On-chip subroutines	Orchestra tick-loop	Off-chip subroutines and orchestra loop
x	On-chip patch points	UG arguments	Off-chip patch points and data
y	On-chip patch points	UG arguments	Off-chip patch points and data
l	On-chip long data	UG arguments	(No off-chip long data)

The assignment of location counters to each of these memory segments is given below:

Space	Lower	Middle	Upper
p	pl:	p:	ph:
x	xl:	x:	xh:
y	yl:	y:	yh:
l	ll:	l:	lh:

A hard partition is defined by the Orchestra class (in the Music Kit) between lower-segment and middle-segment on-chip memory. Any on-chip requests which do not fit in the provided partition are relocated off-chip (i.e., the lower and upper segments are combined off-chip). The on-chip request is analogous to the register allocation request in C: it is advised by the programmer, but the system does not guarantee it will happen.

Note that only the upper segment is certainly in external memory. The lower segment can be in either as discussed above. The middle p segment will often straddle on-chip and off-chip memory. A *leaper* unit generator is inserted to connect the on-chip to the off-chip portion of the orchestra loop.

Within the Music Kit, the memory segments above are mapped onto logical memory segments as follows:

- xData - X data memory. Always off-chip. Used for delay and wave tables.
- yData - Same as x data memory since spaces are overlaid off-chip.
- pLoop - Orchestra loop. On-chip with overflow to off-chip.
- pSubr - Orchestra subroutines. Always off-chip.
- lArg - L memory arguments. Always on-chip.
- xArg - X memory arguments. On-chip with overflow to off-chip.
- yArg - Y memory arguments. On-chip with overflow to off-chip.

Use of Host Flags

The four host flags are used as follows:

- HF0 - Tell DSP to abort current program

- HF1 - Tell DSP that requested DMA transfer is pending
- HF2 - Tell host a host message is being executed
- HF3 - Tell host Timed Message Queue is full

While processing a host message, HF3 is always cleared. This is done so that the combination of both HF2 and HF3 may mean that the DSP has aborted. In this state, the DSP is at a breakpoint, and it must be either restarted, or taken over by a debugger.

Interrupt Priority Levels

The priority levels used within the Music Kit DSP Monitor are as follows:

- 0 - Default (user level).
- 1 - Host communications service (DMA, message interrupts, host commands)
- 2 - Serial port service.
- 3 - Critical sections and non-maskable interrupt service.

Acknowledgements

Douglas Fulton documented the NeXT sound and music system, and has made substantial design improvements in clarifying general protocol and the identity and mechanisms of the classes. He also helped us with this paper. Gregg Kellogg is the author of the DSP, Sound, and MIDI device drivers (with Doug Mitchell now taking over the MIDI driver). Michael McNabb brought wave table synthesis to the Music Kit and designed and built a number of UnitGenerator and SynthPatch subclasses. John Strawn wrote most of the matrix and array processing macros. Mike Minnick helped finish the DSP array processing tools and wrote most of the programming examples. Roger Dannenberg influenced both the Music Kit noteTag design and the design of the performance mechanism (using a data flow discrete simulation model). Andy Moorer helped shape the Envelope strategy, suggested the unit-generator memory-argument scheme, and provided expert consultation in many areas. Dana Massie contributed speech coding, sampling-rate conversion, and signal conditioning modules for the Sound Kit. Doug Keislar helped with testing, developer support, and demos. The software of William Schottstaedt and others at CCRMA (Stanford University) served as a model for some of the mechanisms in the Music Kit. John Chowning, Max Mathews and others in the computer music community have lent moral, technical, and visionary support. Bud Tribble initiated the decision by NeXT to place a DSP56001 in every machine, and Steve Jobs' unflagging sense of direction has been critical to the music project.

The Sound Kit was designed and implemented by Lee Boynton, the Music Kit was designed and implemented by David Jaffe, and the DSP computer-music and array-processing software was designed and implemented by Julius Smith.

Appendix A - DSP Unit Generator Implementation

This appendix discusses an example unit generator, explaining its features and discussing various issues which were addressed in the design.

An Example Unit Generator

Below is an example unit-generator source file. It contains a single unit generator macro, **add2** which is found by the Motorola DSP assembler when assembling a stand-alone orchestra program. For the Music Kit, the macro is automatically wrapped (by **dspwrap**) in a small main DSP program, assembled in relative mode, and packaged into an Objective C object class which is called upon by the Music Kit to dynamically load an instance of the unit generator into the DSP during a musical performance. In macro source files, the following mnemonics are used for the DSP index registers:

```
R_X     =   R0  = x memory argument pointer
R_Y     =   R4  = y memory argument pointer

R_I1    =   R1  = input register 1 (can be used for anything)
R_I2    =   R5  = input register 2 (can be used for anything)
R_O     =   R6  = output register  (can be used for anything)

R_HMS   =   R3  = pointer to top of host message stack
R_DMA   =   R7  = index register used for DMA transfers

Analogous names are used for the N and M registers, e.g.,
N_X = N0 and M_X = M0.
```

The following is the unit generator source code:

```
;;  Modification history
;; ————————————————
;;  10/19/87/jos - initial file created from scale.asm
;;
;;————————————————————————————DOCUMENTATION————————————————————--------
;;  NAME
;;    add2 (UG macro) - add two signals to produce a third
;;
;;  SYNOPSIS
;;    add2 pf,ic,sout,aout0,i1spc,i1adr0,i2spc,i2adr0
```

```
;;
;;  MACRO ARGUMENTS
;;    pf     = global label prefix (any text unique to invoking macro)
;;    ic     = instance count (such that pf\ _add2_\ ic\ _ is globally unique)
;;    sout       = output memory space ('x' or 'y')
;;    aout0      = initial output address in memory sout
;;    i1spc      = input 1 memory space ('x' or 'y')
;;    i1adr0     = initial input address in memory i1spc
;;    i2spc      = input 2 memory space ('x' or 'y')
;;    i2adr0     = initial input address in memory i2spc
;;
;;  DSP MEMORY ARGUMENTS
;;    Arg access  Argument use
                Initialization
;;    ——————      ——————————
                  ——————————
;;    x:(R_X)+    address of input 1 signal    i1adr0
;;    y:(R_Y)+    address of input 2 signal    i2adr0
;;    y:(R_Y)+    address of output signal     aout0
;;
;;  DESCRIPTION
;;    The add2 unit-generator sums two patch points, forming a
;;    third. The output can be the same patch-point as either input.
;;    The inner loop is two instructions if the memory spaces
;;    for in1, in2, and out are x,y,x. In all other cases the
;;    inner loop is three instructions.
;;
;;  DSPWRAP ARGUMENT INFO
;;    add2 (prefix)pf,(instance)ic,
;;        (dspace)sout,aout0,(dspace)i1spc,i1adr0,(dspace)i2spc,i2adr0
;;
;;  MAXIMUM EXECUTION TIME
;;    116 DSP clock cycles for one "tick" which equals eight audio samples.
;;
      beg_def 'add2'
      ; begin macro definition
      define add2_pfx "pf\ _add2_\ ic\ _"
          ; pf = <name>_pfx of invoker
      define add2_pfxm ""“add2_pfx”“"
          ; this form need in macro args
add2  macro pf,ic,sout,aout0,i1spc,i1adr0,i2spc,i2adr0
      beg_mac 'add2'
      ; begin macro body
      new_xarg add2_pfxm,i1adr,i1adr0
          ; allocate x memory argument
      new_yarg add2_pfxm,i2adr,i2adr0
          ; allocate y memory argument
      new_yarg add2_pfxm,aout,aout0
          ; allocate y memory argument
      move y:(R_Y)+,R_I2
              ; input 2 address to R_I2
      move y:(R_Y)+,R_O
              ; output address to R_O
      move x:(R_X)+,R_I1
              ; input 1 address to R_I1

      move i2spc:(R_I2)+,A
          ; load input 2 to A
      move i1spc:(R_I1)+,X0
          ; load input 1 to X0
      do #I_NTICK,add2_pfx\ tickloop
          ; enter do loop
          add X0,A  i1spc:(R_I1)+,X0
      ; do add and fetch input 1
          if "i1spc"=='x'&&"i2spc"=='y'
              if "ospc"=='x' ; xyx
                  move A,sout:(R_O)+ i2spc:(R_I2)+,A ; optimal
              else ; xyy
                  move A,sout:(R_O)+
                  move i2spc:(R_I2)+,A
              endif
          else
              move A,sout:(R_O)+
              move i2spc:(R_I2)+,A
          endif
add2_pfx\ tickloop
      end_mac 'add2'
      endm
      end_def 'add2'
```

There are several points to note about this example. The extensive documentation at the top of the file is turned into a UNIX™ -style man page by the **dspwrap** program. (Certain fields such as the DSPWRAP ARGUMENT INFO field, used internally by **dspwrap** in generating Objective C code, are suppressed).

The first two macro arguments, *prefix* (pf) and instance count (ic), are used in generating unique global symbols within the macro. They can be arbitrary text, although, by convention, the instance count increments through the integers from 1, and the prefix is a unique label prefix passed down from above.

The purpose of having both a prefix and an instance count is to support unique label generation at all levels in a *nested* macro expansion. An example illustrating this feature is given below:

```
; main DSP program illustrating nested unique name generation

     include 'music_macros'
          ; utility macros

     beg_def 'first'
          ; begin macro definition
     define first_pfx "pf\ _first_\ ic\ _"
     ; pf = <name>_pfx of invoker
     define first_pfxm ""first_pfx""
          ; use in macro args
firstmacro pf,ic
     beg_mac 'first'
     ; begin macro body
          second first_pfxm,1
          ; nested invocation
          second first_pfxm,2
     end_mac 'first'
     endm
     end_def 'first'

     beg_def 'second'
     define second_pfx "pf\ _second_\ ic\ _"
     define second_pfxm ""second_pfx""
second  macro pf,ic
     beg_mac 'second'
          msg 'innermost'
second_pfx\ label
          dc 0
     end_mac 'second'
     endm
     end_def 'second'

; orchestra program
;
     beg_orch 'test'          ; begin orchestra main program
          beg_orcl            ; begin orchestra loop
               first orch,1   ; instance 1 of macro first
               first orch,2   ; instance 2 of macro first
          end_orcl            ; end of orchestra loop
     end_orch 'test'          ; end of orchestra main program
```

When this program is assembled, it prints "innermost" four times, and the following four labels appear in the assembly output file:

```
_SYMBOL P
orch_first_1__second_1_label I 000056
orch_first_1__second_2_label I 000057
orch_first_2__second_1_label I 000058
orch_first_2__second_2_label I 000059
```

The benefit of this use of prefix and instance count is the availability of global handles on all interesting quantities at all levels of macro expansion.

After the prefix and instance count arguments of the add2 unit generator, there is the macro argument *sout* which can be either y or x. It specifies the memory space of the output signal vector. The next macro argument, *aout0*, specifies the address of the output signal. Similarly, i1spc, i1adr0, i2spc, and i2adr0 specify the memory space and memory address of the two length 8 input patchpoints.

An attempt is made to optimize each memory space combination. The **dspwrap** program emits an Objective C UnitGenerator subclass for each combination of memory spaces. For example, the **add2** example above would generate eight subclasses corresponding to each combination of memory space for the two inputs and single output.

The statement

```
     new_xarg add2_pfxm,i1adr,i1adr0
          ; allocate x memory argument
```

specifies memory argument allocation and expands to

```
     xdef orch_add2_1_i1adr
     org x:
orch_add2_1_i1adr
     bsc 1,yvec   ; allocate x memory argument
     org p:
```

where *orch_add2_1_* is the assumed expansion of the macro argument *add2_pfxm* , and *yvec* is the assumed expansion of the macro argument *i1adr0*. Thus, the argument's symbolic name is exported by the **xdef** statement, and the next available element of internal x memory is allocated as the corresponding memory argument of the unit generator.

Unit Generator Memory Arguments

Memory arguments contain the entire *run-time state* of the unit generator. In the execution of an orchestra loop, each unit generator loads its ALU and index registers from its memory arguments. Any state needed for the next tick's computation (such as delayed signals in a digital filter) is written by the unit generator into the memory arguments on exit. To minimize pointer initialization overhead, the memory arguments are accessed sequentially in internal x and y RAM in the order needed by the unit generator. This eliminates initialization of the three memory argument pointers, except once at the beginning of the orchestra loop. Loading of addresses from memory arguments was shown in the add2 example above and is repeated below:

```
move y:(R_Y)+,R_I2    ; input 2 address to R_I2
move y:(R_Y)+,R_O     ; output address to R_O
move x:(R_X)+,R_I1    ; input 1 address to R_I1
```

There were no data memory arguments in the add2 example, but they are handled in exactly the same way. Data arguments are things like amplitude, frequency, and current phase of a sinusoidal oscillator, or digital filter coefficients and delayed signal values. Often, two data arguments can be loaded in parallel, for example,

```
move x:(R_X)+,X0 y:(R_Y)+,Y0
; load gain1 to X0 and gain2 to Y0
```

Each unit generator is responsible for leaving R_X, R_Y, and R_L pointing one past its argument block on exit. If a unit generator needs to save run-time state (such as current envelope amplitude), it's best to place this state last in the memory arguments. Having all running state in the memory arguments facilitates memory compaction needed with dynamic loading.

In addition to x and y memory arguments, l memory arguments are supported. Since l memory is the concatenation of x and y memories (48 bits), l memory arguments are allocated *down* from the top of on-chip x and y RAMs. Long memory arguments are used for oscillator phase and table increment (which determines frequency of oscillation), exponential envelope state, and the slope and current value of the linear amplitude ramper.

All memory arguments are given symbolic names (as shown in the add2 example above). These symbols are collected into the assembly output file and are needed to write (or read) the arguments from a host task. The Music Kit takes care of this linkage for most users. However, these hooks can support more general real-time signal processing applications which need not involve the Music Kit.

Unit Generator Macro Arguments

Three cases summarize the kinds of unit generator macro arguments discussed above:

- prefix and instance count text (arguments *pf* and *ic*)
- space and initial address of input and output signals (e.g. *sout,aout0*)
- initialization of data memory arguments (e.g. filter coefficients)

Miscellaneous Considerations when Writing Unit Generators

Our experience in writing unit generators indicates that the natural limit on the size of a unit generator is that which fills up the ALU and/or index registers. If there is a need in the inner loop to swap data in and out of a register, it's a good idea to break up the computation between two unit generators.

We have found that, it is useful to constrain the input signal memory spaces (for optimum performance) before constraining its output memory space. For example, given a choice between reading two inputs in parallel (which requires them to be in opposite memory spaces) and writing the output signal as a single parallel move (in which case either x or y may be specified) versus reading one input in parallel with writing the output (which forces the input to be in an opposite space relative to the output), the former is preferred. This allows, when building the synthpatch, choosing whichever output space optimizes the subsequent reading unit generator. This scheme provides optimal structures easily whenever at most one unit generator reads each output signal.

Appendix B - Update to "An Overview of the Sound and Music Kits for the NeXT Computer"

The article "An Overview of the Sound and Music Kits for the NeXT Computer" describes the state of the 0.9 release of the music and sound software. This update describes the 1.0 release of the software. Additionally, an upcoming 2.0 release (not described here) has significant new features, improved performance and more complete documentation.

The 1.0 release of the Sound Kit is optimized significantly from the 0.9 version and provides additional functionality. In 0.9, the Sound class provided efficient insert and delete operations of sampled sounds. In 1.0, the SoundView makes use of this ability. The reading and writing the pasteboard is now "lazy", providing rapid cut and paste of sound samples. The SoundMeter class is new and provides a simple bar-graph meter of a sampled sound's activity while playing or recording. It displays a moving average of the sound amplitude with "peak hold".

The 1.0 release of the system software includes a Mach MIDI driver that provides time-stamped parsed input and scheduled output and communicates with the serial ports on the NeXT machine. With the addition of an inexpensive adapter, each of the two serial ports can connect to a MIDI cable. Programming examples to directly play and record Standard MIDI file are provided. Higher-level access to the Mach MIDI driver is provided by the 1.0 Music Kit via the Midi class. Programming examples are included that illustrate using the Music Kit to do MIDI echo, MIDI record to and playback from a scorefile, etc. In addition, the Score object can now read and write Standard MIDI

files.

The 1.0 Music Kit UnitGenerator Library includes processing modules such as filters and oscillators. The Music Kit SynthPatch library includes general-purpose instruments to support a wide range of synthesis techniques including simple frequency modulation with complex waveforms, cascade and parallel modulation FM, FM with noise modulation, wave table synthesis with interpolation between wave tables, and plucked string synthesis. The FM and wave table synthesis have the option of periodic vibrato, random vibrato, and amplitude and frequency envelopes with an arbitrary number of breakpoints. A library of wave tables for various timbres in various ranges is supplied.

Note that there was a change in the Music Kit class hierarchy since the article was published. The SynthElement class, which was the superclass of UnitGenerator and SynthData, was eliminated. UnitGenerator and SynthData now inherit directly from Object.

Finally, we'd like to acknowledge the contributions to the project made by Mike Minnick and John Strawn.

David Jaffe & Lee Boynton