Sound, Music, and Signal Processing: Concepts

NeXT Developer's Library

NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.

Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.


Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.

Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.

Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.

NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.

NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.

Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.

NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.

Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.

Sound, Music, and Signal Processing: Concepts

NeXT Computer, Inc., 900 Chesapeake Drive, Redwood City, CA 94063.

Manual written by Doug Fulton
Edited by Caroline Rose, Roy West, and Helen Casabona
Book design by Eddie Lee
Illustrations by Jeff Yaksick and Don Donoughe
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Introduction

Introduction

This manual describes the concepts that you should understand when creating applications that incorporate sound, music, signal processing, or that access the DSP. It's part of a collection of manuals called the *NeXT™ Developer's Library*; the illustration on the first page of this manual shows the complete set of manuals in this Library.

A version of this manual is stored on-line in the NeXT Digital Library (which is described in the user's manual *NeXT Applications*). The Digital Library also contains Release Notes, which provide last-minute information about the latest release of the software.

Conventions

Syntax Notation

Where this manual shows the syntax of a function, command, or other programming element, the use of bold, italic, square brackets, and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

print *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you type a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	Allows
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Notes and Warnings

Note: Paragraphs like this contain incidental information that may be of interest to curious readers but can safely be skipped.

Warning: Paragraphs like this are extremely important to read.

Chapter 1
System Overview

Chapter 1
System Overview

As illustrated in Figure 1-1, there are four levels of software between a NeXT application and the hardware that executes it:

- The NeXT Interface Builder™
- Objective-C language software "kits"
- The NeXT Window Server and specialized C libraries
- The Mach operating system

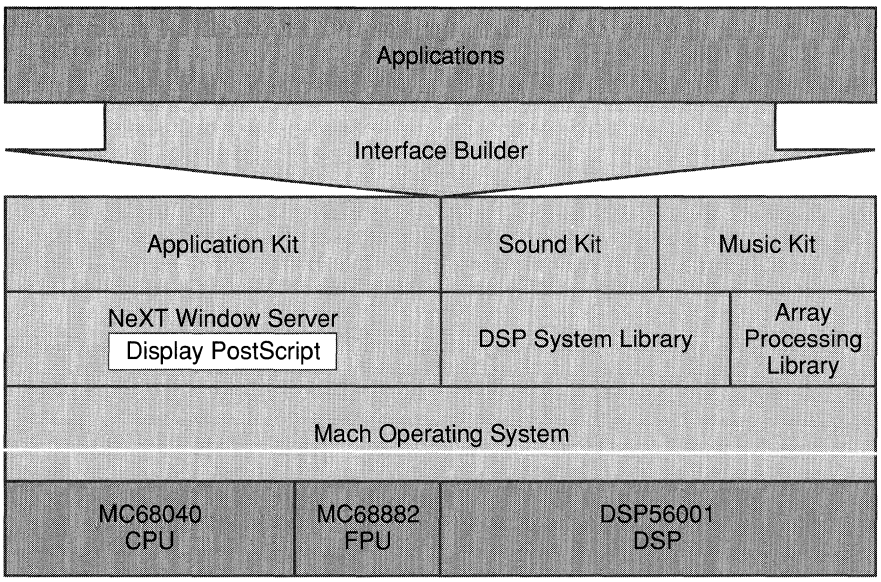

Figure 1-1. System Overview

Interface Builder is a powerful tool that lets you graphically design your application's user interface. It also makes it easy for you to establish connections between user-interface objects and your own code (for example, the code to execute when a button on the screen is clicked).

NeXT application programs are written in the Objective-C language, an extension to C that adds object-oriented concepts to the language. The software kits define a number of *classes*, or object templates, that you can use in your own applications. The software kits currently provided by NeXT are:

- An *Application Kit* that every application uses to implement the NeXT window-based user interface
- *Sound Kit*™ for adding sounds to your application
- *Music Kit*™ for music composition, synthesis, and performance

The NeXT *Window Server* is a low-level background *process* used by the Application Kit to manage windows and to send user *events*, such as mouse and keyboard actions, back to an application. Included in the Window Server is a Display PostScript® interpreter that's used for all drawing of text and graphics on the screen or printed page. The Display PostScript system was jointly developed by NeXT and Adobe Systems Incorporated as an enhancement of Adobe's PostScript page description language.

The Sound and Music Kits use the DSP56001 digital signal processor (the *DSP*) as a sound synthesizer. Objects in these kits communicate with the DSP by calling functions in the DSP system library. In addition to establishing and managing a channel of communication between your application and the DSP, the functions in the DSP system library also provide diagnostic capabilities and data conversion routines.

The functions in the array processing library use the DSP as an array processor, allowing your application to process multidimensional data with great speed and efficiency. Any application can include and use the array processing library.

Mach is a multitasking operating system developed at Carnegie Mellon University. It acts as an interface between the upper levels of software and the three Motorola microprocessors provided with NeXT computers: the MC68040 central processor, the MC68882 floating-point coprocessor, and the DSP56001 digital signal processor.

The rest of this chapter further describes the Sound and Music Kits, and the array processing library. Interface Builder, the Application Kit, the Window Server, and Mach are described in the companion *NeXTstep® Reference* and *NeXT Operating System Software* manuals.

Sound and Music Overview

NeXT computers provide a powerful system for creating and manipulating sound and music. The software for this system is divided into two kits: the Sound Kit and the Music Kit. The kit that you need depends on the demands of your application:

- The Sound Kit lets you incorporate prerecorded sound effects into your application and provides easy access to the microphone input so you can record your own sounds. The objects in the Sound Kit let you examine and manipulate sound data with microscopic precision.

- The Music Kit provides tools for composing, storing, and performing music. It lets you communicate with external synthesizers as well as create your own software instruments. Like the Sound Kit, the Music Kit provides objects that create and manipulate sounds with exquisite detail; but more important, the Music Kit helps you organize and arrange groups of sounds so you can design a performance.

The Sound Kit

A small number of system beep–type sound recordings, stored in files on the disk (called *soundfiles*), are provided by NeXT. Through the Sound Kit, you can easily access these files and incorporate the sounds into your application. It's also extremely easy to record new sounds into a NeXT computer. With a single message to the Sound Kit's Sound object, you can simply record your own sound effect through the microphone on the front of the MegaPixel Display (you can also plug a microphone into the jack on the back of the display). Sound playback is just as simple: Another message and the sound is played on the internal speaker and sent to the stereo output jacks at the back of the display.

When you record a sound using the Sound object, a series of audio "snapshots" or *samples* is created. By storing sound as samples, you can analyze and manipulate your sound data with an almost unlimited degree of precision. The SoundView class lets you see your sounds by displaying the samples in a window.

While the Sound Kit is designed primarily for use on sampled data, you can also use it to send instructions to the DSP. The speed of the DSP makes it an ideal sound synthesizer and, in general, DSP instructions take up much less space than sampled data. The Sound object manages the details of playing sounds for you, so you needn't be aware of whether a particular Sound contains samples or DSP instructions.

The Music Kit

The Music Kit provides a number of ways to compose and perform music. By attaching an external synthesizer keyboard to a serial port, you can play a NeXT computer as a musical instrument. Alternatively, you can compose music to be played by the computer by creating music data in a text editor or by creating an algorithm that generates it automatically. These approaches can be combined in performance. For instance, a musician can use an external keyboard to trigger precomposed events, allowing the computer to create sounds and gestures that are impossible on a traditional instrument, but at moments specified by the performer.

The Music Kit helps you construct applications that create, organize, process, and render music data. The Objective-C language classes provided by the Kit fall into three categories:

- Data representation
- Synthesis
- Performance

Data Representation

The data representation classes, illustrated in Figure 1-2, are used to encapsulate and organize music data.

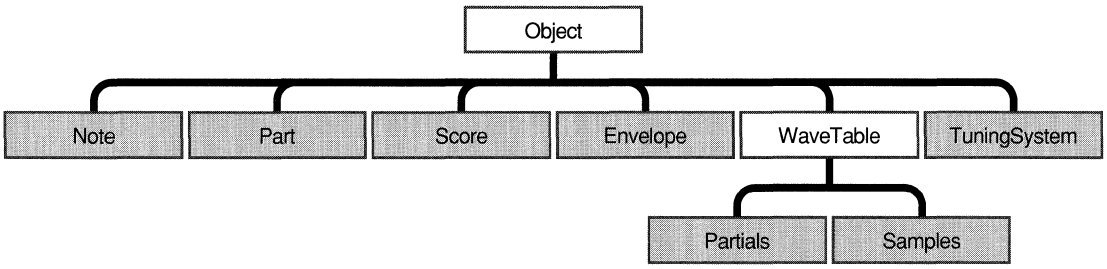

Figure 1-2. Music Data Representation Classes

Notes, Parts, and Scores form the core of music representation. Of paramount importance is the Note class: A Note object represents a musical note as a list of attributes, such as frequency, amplitude, and duration. Music applications use Note objects as a common currency: They're the basic package of musical information upon which the other objects act. Part and Score objects, as their names suggest, provide a means for organizing Note objects. The other data classes, Envelope, WaveTable (and its progeny), and TuningSystem, are designed to help define Note object attributes:

- Envelopes represent time-varying functions that can be used to continuously control the values of a Note's attributes (such as its amplitude and frequency).

- A WaveTable contains timbral information that's used during music synthesis on the DSP.

- A TuningSystem is a mapping of pitch names to specific frequencies, allowing an easy representation of alternate tunings.

The Music Kit defines an ASCII file format called *scorefile* that represents the music data objects as editable text in files on a disk. A few C-like programming constructs, such as variables and arithmetic operators, can be used in a scorefile to help create and fine-tune music data. You can also store music data as a Standard MIDI File.

Synthesis

Synthesizing music is potentially the most technically involved of the three Music Kit areas. At the easiest level, you can use and manipulate the software instruments, called SynthPatches, that are provided by the Music Kit. A SynthPatch subclass corresponds, roughly, to a voice preset on a MIDI synthesizer. However, the Music Kit SynthPatches are generally less confined than most MIDI presets: An enormously wide variety of sounds can be produced by the SynthPatches supplied by the Music Kit simply by varying the attributes of the Notes that they receive.

At a lower lever, you can design your own SynthPatch subclasses by interconnecting DSP synthesis modules that the Music Kit provides as objects called UnitGenerators. Finally, at the lowest level, you can design UnitGenerators yourself by writing DSP56000 assembly language macros and using the **dspwrap** tool to turn the macros into subclasses of UnitGenerator. This last level falls below the boundary of the Music Kit and is described in Chapter 7, "Programming the DSP." The principal Music Kit synthesis classes are shown in Figure 1-3.

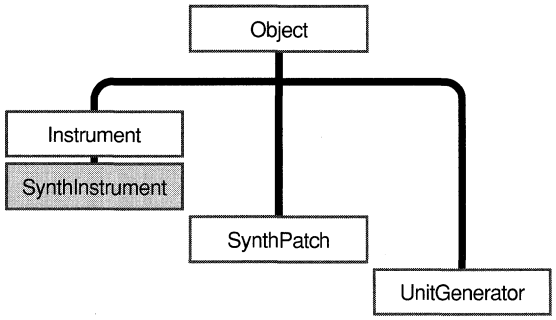

Figure 1-3. Synthesis Classes

The SynthInstrument class isn't strictly part of the synthesis machinery. However, it provides an easy way to allocate and control SynthPatch objects.

An additional class, not shown in the illustration above, is Orchestra. An Orchestra represents an entire DSP; the standard configuration includes a single DSP, thus most applications will create but a single Orchestra object. It's through an Orchestra that all synthesis resources, such as UnitGenerators and SynthPatches, are allocated.

Performance

During a Music Kit performance, Note objects are acquired, scheduled, and rendered (or *realized*). These functions are embodied by objects of the Performer, Conductor, and Instrument classes:

- Performer objects acquire Notes.
- Through messages scheduled with a Conductor object, a Performer forwards each Note it acquires to one or more Instruments. The Conductor thus controls the tempo of the performance.
- An Instrument receives Notes that are sent to it by a Performer and realizes them in some manner, typically by synthesizing them on the DSP or by sending them to an external MIDI instrument. Other types of realization include writing Notes to a scorefile or adding them to a Part.

Performer and Instrument are abstract classes; each subclass specifies a particular means of Note acquisition or realization. The Music Kit provides a number of Performer and Instrument subclasses.

Figure 1-4 shows the primary classes that are used to design a Music Kit performance.

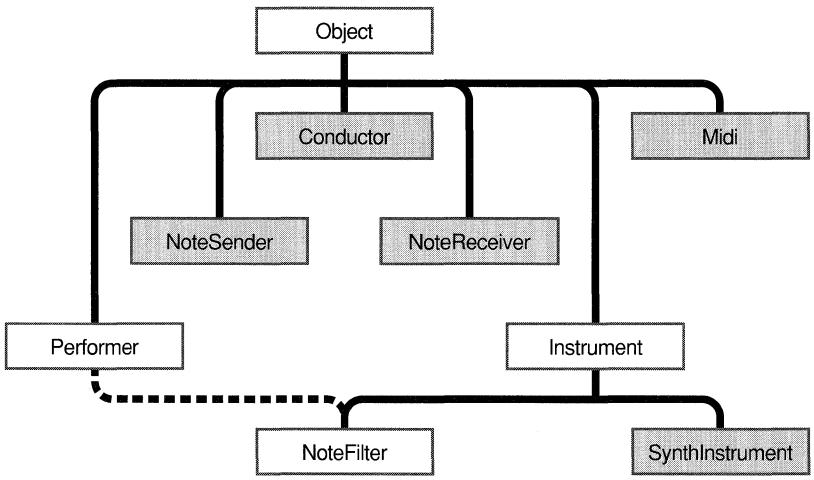

Figure 1-4. Performance Classes

In addition to the Performer, Conductor, and Instrument classes described above, five other classes are included in Figure 1-4: NoteSender, NoteReceiver, NoteFilter, SynthInstrument, and Midi.

- NoteSender and NoteReceiver objects are part of the implementation of Performer and Instrument: They're the ports through which Notes are sent by Performers and received by Instruments.

- A NoteFilter is a Performer/Instrument hybrid; while it inherits from Instrument, it also implements Performer protocol. Thus, it can receive Notes like an Instrument and then send them on to other Instruments, like a Performer. NoteFilters are interposed between Performers and Instruments and act as Note-processing modules.

- SynthInstrument is a subclass of Instrument that causes Notes to be realized on the DSP.

- A Midi object represents an external MIDI synthesizer that's attached to a NeXT computer through one of the serial ports. It can receive as well as send MIDI signals from and to the synthesizer it represents. While it inherits neither from Performer nor Instrument, it implements their protocols and contains NoteSenders and NoteReceivers.

A number of other Performer and Instrument subclasses are provided by the Music Kit. During a Music Kit performance, performance objects can be dynamically connected and reconnected. This allows you to mix and match Note sources with any means of realization. For example, the MIDI signals sent from an external MIDI synthesizer are automatically converted to Note objects by a Midi object. The Notes can then be sent to a SynthInstrument for realization on the DSP, or written to a scorefile by a ScorefileWriter Instrument.

Components of Sound and Music

Figure 1-5 shows the components for creating, playing, and storing music and sound with the hardware and software of a NeXT computer.

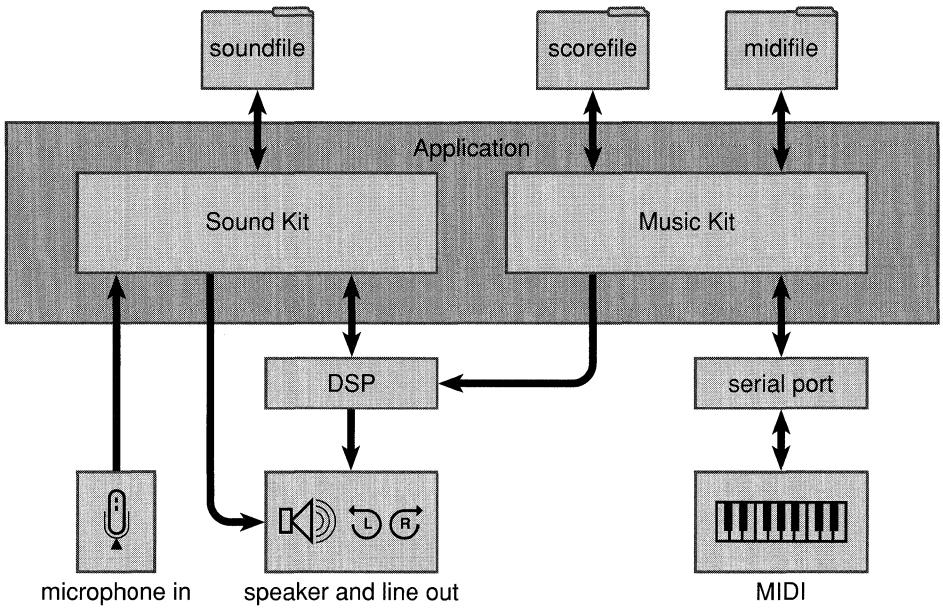

Figure 1-5. Music and Sound Components

Chapter 2
Sound

Chapter 2
Sound

This chapter describes the hardware and software provided by NeXT computers for recording, manipulating, playing back, and displaying sounds. The chapter is divided into three parts:

- The NeXT sound hardware
- A brief tutorial on sound and how it's represented on a NeXT computer
- The Sound Kit

Design Philosophy

NeXT computers provide a sound recording and playback system with powerful tools to aid in analyzing and manipulating acoustical data. Designed to satisfy the needs of the research scientist, this system is nevertheless extremely easy to use.

At the heart of the NeXT sound facilities are the Objective-C language classes provided by the Sound Kit. The Sound Kit manages the details of operating system communication, data access, and data buffering that are necessary for recording and playing sounds.

A number of system beep-type sounds are provided in files on the disk. You can easily incorporate these sounds into your application; the playback of an effect can be made to correspond to user actions or application events, such as the completion of a background process.

The sound software gives you full access to the data that makes up a sound. With some simple programming you can manipulate this data. For instance, you can alter the pitch of a sound or affect its playback speed. A sound can be played backwards, looped end to end, or chopped into pieces and reassembled in a different order. You can digitally splice and mix together any number of different sounds: A dog bark can be spliced into the middle of a doorbell; a clarinet tone can turn into a snore.

The digital hardware for sound recording and playback is ideal for research fields such as speech recognition, speech synthesis, and data compression. To ensure high-fidelity sound playback, NeXT computers use the same digital playback hardware found in commercial compact disc (CD) players.

Sound Hardware

Before you can process a sound, you must first get it into your NeXT computer. A microphone is provided on the front of the display, as well as a microphone jack on the back of the display that accepts a high-impedance microphone signal. The Sound Kit recording methods, described later in this chapter, automatically record and store sounds introduced through the microphone or the microphone jack.

For sound playback, the computer contains a speaker built into the display as well as stereo headphone and stereo line-out jacks. The keyboard volume and mute keys affect the built-in speaker and the headphone jack; the line-out jacks are provided to allow you to connect your NeXT computer to your own stereo for greater playback fidelity.

NeXT computers provide equipment to convert analog signals to digital and digital signals to analog. The following sections describe the NeXT digital sound hardware.

Voice-Quality Input

The microphone and microphone jack are connected to an *analog-to-digital converter* (ADC), known as the *CODEC* ("COder-DECoder"). The CODEC converter 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 samples from this converter can be stored on the disk or they can be forwarded to the DAC, described below, to reproduce the sound.

The CODEC's mu-law encoding allows a 12-bit dynamic range to be stored in eight bits. In other words, an 8-bit sound with mu-law encoding will yield the same amplitude resolution as an unencoded 12-bit sound. With this compression algorithm, the CODEC saves storage space. For example, one second of 8-bit mu-law audio takes 8012 bytes of storage. By comparison, one second of CD-quality sound occupies 88200 bytes, or about 11 times more storage space.

While 8-bit mu-law encoding provides only moderate fidelity, the CODEC is sufficient and useful in a number of sound application areas. For instance, all the elements necessary to implement voice mail—sending spoken mail messages through the network—are present. In such an application, compact data storage is more desirable than high fidelity.

The CODEC is available as a standard UNIX® device. It does have a special constraint in that once conversion starts, a new byte will come from the device every 124.8 microseconds. The program reading the CODEC must be prompt in absorbing this data or it will be lost. The operating system does some buffering of CODEC input data, but it's by no means unlimited. In most applications, the Sound Kit management of input data is sufficient.

High-Quality Sound Output

The high-quality stereo *digital-to-analog converter* (DAC) operates at 44100 samples per second (in each channel) with a 16-bit quantization, the same as in 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 has full de-glitching and anti-aliasing filters built in, so no external hardware is necessary for basic operation.

Like the CODEC, the DAC is available as a standard UNIX device. It's somewhat different from most devices in that it requires a great deal of data (176400 bytes per second at the high sampling rate). Any interruption in sending this data causes an interruption in the sound that will result in a pop in the audio output. Utilities are provided that ensure continuous data flow when sending sound data directly from the disk to the DAC.

Basic Sound Concepts

You don't need to know anything about sound or acoustics to use the NeXT sound facilities for simple recording and playback. However, to access and manipulate sound data intelligently, you should be familiar with a few basic terms and concepts. This section presents a brief tutorial on the basic concepts of sound and its digital representation, followed by an in-depth examination of SNDSoundStruct, the structure that's used by the NeXT sound software to represent sound.

What is Sound?

Sound is a physical phenomenon produced by the vibration of matter. The matter can be almost anything: a violin string or a block of wood, for example. As the matter vibrates, pressure variations are created in the air surrounding it. This alternation of high and low pressure is propagated through the air in a wave-like motion. When the wave reaches our ears, we hear a sound.

Figure 2-1 graphs the oscillation of a pressure wave over time.

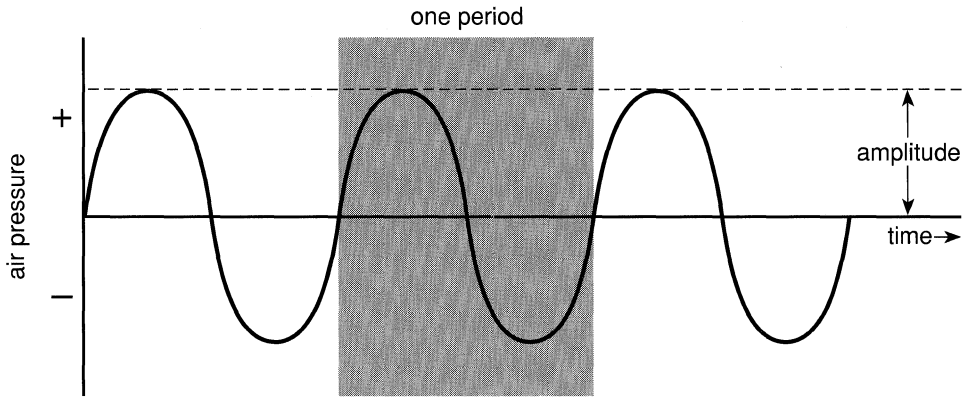

Figure 2-1. Air Pressure Wave

The pattern of the pressure oscillation is called a *waveform*. Notice that the waveform in Figure 2-1 repeats the same shape at regular intervals; the gray area shows one complete shape. This portion of the waveform is called a *period*. A waveform with a clearly defined period occurring at regular intervals is called a *periodic waveform*.

Since they occur naturally, sound waveforms are never as perfectly smooth nor as uniformly periodic as the waveform shown in Figure 2-1. However, sounds that display a recognizable periodicity tend to be more musical than those that are nonperiodic. Here are some sources of periodic and nonperiodic sounds:

Periodic

Musical instruments other than unpitched percussion
Vowel sounds
Bird songs
Whistling wind

Nonperiodic

Unpitched percussion instruments
Consonants, such as "t," "f," and "s"
Coughs and sneezes
Rushing water

Frequency

The *frequency* of a sound—the number of times the pressure rises and falls, or oscillates, in a second—is measured in *hertz* (Hz). A frequency of 100 Hz means 100 oscillations per second. A convenient abbreviation, kHz for *kilohertz*, is used to indicate thousands of oscillations per second: 1 kHz equals 1000 Hz.

The frequency range of normal human hearing extends from around 20 Hz up to about 20 kHz.

The frequency axis is logarithmic, not linear: To traverse the audio range from low to high by equal-sounding steps, each successive frequency increment must be greater than the last. For example, the frequency difference between the lowest note on a piano and the note an octave above it is about 27 Hz. Compare this to the piano's top octave, where the frequency difference is over 2000 Hz. Yet, subjectively, the two intervals sound the same.

Amplitude

A sound also has an *amplitude*, a property subjectively heard as loudness. The amplitude of a sound is the measure of the displacement of air pressure from its mean, or quiescent state. The greater the amplitude, the louder the sound.

How the Computer Represents Sound

The smooth, continuous curve of a sound waveform isn't directly represented in a computer. A computer measures the amplitude of the waveform at regular time intervals to produce a series of numbers. Each of these measurements is called a *sample*. Figure 2-2 illustrates one period of a digitally sampled waveform.

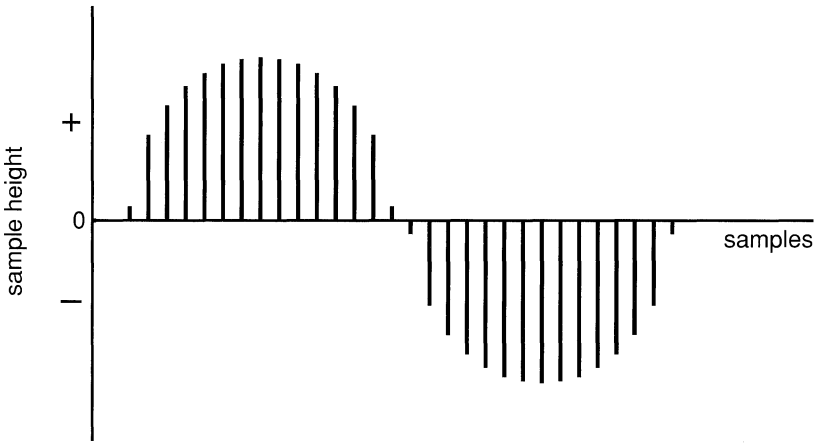

Figure 2-2. Sampled Waveform

Each vertical bar in Figure 2-2 represents a single sample. The height of a bar indicates the value of that sample.

The mechanism that converts an audio signal into digital samples is called an *analog-to-digital converter*, or *ADC*. To convert a digital signal back to analog, you need a *digital-to-analog converter*, or *DAC* (pronounced "dack").

Sampling Rate

The rate at which a waveform is sampled is called the *sampling rate*. Like frequencies, sampling rates are measured in hertz. The CD standard sampling rate of 44100 Hz means that the waveform is sampled 44100 times per second. This may seem a bit excessive, considering that we can't hear frequencies above 20 kHz; however, the highest frequency that a digitally sampled signal can represent is equal to half the sampling rate. So a sampling rate of 44100 Hz can only represent frequencies up to 22050 Hz, a boundary much closer to that of human hearing.

Quantization

Just as a waveform is sampled at discrete times, the value of the sample is also discrete. The *quantization* of a sample value depends on the number of bits used in measuring the height of the waveform. An 8-bit quantization yields 256 possible values; 16-bit CD-quality quantization results in over 65000 values. As an extreme example, Figure 2-3 shows the waveform used in the previous example sampled with a 3-bit quantization. This results in only eight possible values: .75, .5, .25, 0, –.25, –.5, –.75, and –1.

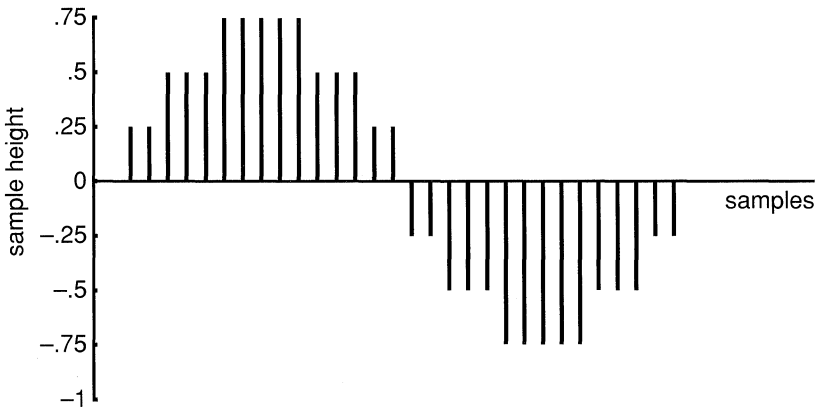

Figure 2-3. Three-Bit Quantization

As you can see, the shape of the waveform becomes less discernible with a coarser quantization. The coarser the quantization, the "buzzier" the sound.

Storing Sampled Data

An increased sampling rate and refined quantization improves the fidelity of a digitally sampled waveform; however, the sound will also take up more storage space. Five seconds of sound sampled at 44.1 kHz with a 16-bit quantization uses more than 400,000 bytes of storage—a minute will consume more than five megabytes. A number of data compression schemes have been devised to decrease storage while sacrificing some fidelity.

SNDSoundStruct: How a NeXT Computer Represents Sound

The NeXT sound software defines the SNDSoundStruct structure to represent sound. This structure defines the soundfile and Mach-O sound segment formats and the sound pasteboard type. It's also used to describe sounds in Interface Builder. In addition, each instance of the Sound Kit's Sound class encapsulates a SNDSoundStruct and provides methods to access and modify its attributes.

Basic sound operations, such as playing, recording, and cut-and-paste editing, are most easily performed by a Sound object. In many cases, the Sound Kit obviates the need for in-depth understanding of the SNDSoundStruct architecture. For example, if you simply want to incorporate sound effects into an application, or to provide a simple graphic sound editor (such as the one in the Mail application), you needn't be aware of the details of the SNDSoundStruct. However, if you want to closely examine or manipulate sound data you should be familiar with this structure.

The SNDSoundStruct contains a header, information that describes the attributes of a sound, followed by the data (usually samples) that represents the sound. The structure is defined (in **sound/soundstruct.h**) as:

```
typedef struct {
    int magic               /* magic number SND_MAGIC */
    int dataLocation;       /* offset or pointer to the data */
    int dataSize;           /* number of bytes of data */
    int dataFormat;         /* the data format code */
    int samplingRate;       /* the sampling rate */
    int channelCount;       /* the number of channels */
    char info[4];           /* optional text information */
} SNDSoundStruct;
```

SNDSoundStruct Fields

magic

magic is a magic number that's used to identify the structure as a SNDSoundStruct. Keep in mind that the structure also defines the soundfile and Mach-O sound segment formats, so the magic number is also used to identify these entities as containing a sound.

dataLocation

It was mentioned above that the SNDSoundStruct contains a header followed by sound data. In reality, the structure *only* contains the header; the data itself is external to, although usually contiguous with, the structure. (Nonetheless, it's often useful to speak of the SNDSoundStruct as the header and the data.) **dataLocation** is used to point to the data. Usually, this value is an offset (in bytes) from the beginning of the SNDSoundStruct to the

first byte of sound data. The data, in this case, immediately follows the structure, so **dataLocation** can also be thought of as the size of the structure's header. The other use of **dataLocation**, as an address that locates data that isn't contiguous with the structure, is described in "Format Codes," below.

dataSize, dataFormat, samplingRate, and channelCount

These fields describe the sound data.

dataSize is its size in bytes (not including the size of the SNDSoundStruct).

dataFormat is a code that identifies the type of sound. For sampled sounds, this is the quantization format. However, the data can also be instructions for synthesizing a sound on the DSP. The codes are listed and explained in "Format Codes," below.

samplingRate is the sampling rate (if the data is samples). Three sampling rates, represented as integer constants, are supported by the hardware:

Constant	Sampling Rate (Hz)	
SND_RATE_CODEC	8012.821	(CODEC input)
SND_RATE_LOW	22050.0	(low sampling rate output)
SND_RATE_HIGH	44100.0	(high sampling rate output)

channelCount is the number of channels of sampled sound.

info

info is a NULL-terminated string that you can supply to provide a textual description of the sound. The size of the **info** field is set when the structure is created and thereafter can't be enlarged. It's at least four bytes long (even if it's unused).

Format Codes

A sound's format is represented as a positive 32-bit integer. NeXT reserves the integers 0 through 255; you can define your own format and represent it with an integer greater than 255. Most of the formats defined by NeXT describe the amplitude quantization of sampled sound data:

Code	Format
SND_FORMAT_MULAW_8	8-bit mu-law samples
SND_FORMAT_LINEAR_8	8-bit linear samples
SND_FORMAT_LINEAR_16	16-bit linear samples
SND_FORMAT_EMPHASIZED	16-bit linear with emphasis
SND_FORMAT_COMPRESSED	16-bit linear with compression
SND_FORMAT_COMPRESSED_EMPHASIZED	A combination of the two above
SND_FORMAT_LINEAR_24	24-bit linear samples
SND_FORMAT_LINEAR_32	32-bit linear samples
SND_FORMAT_FLOAT	floating-point samples
SND_FORMAT_DOUBLE	double-precision float samples
SND_FORMAT_DSP_DATA_8	8-bit fixed-point samples
SND_FORMAT_DSP_DATA_16	16-bit fixed-point samples
SND_FORMAT_DSP_DATA_24	24-bit fixed-point samples
SND_FORMAT_DSP_DATA_32	32-bit fixed-point samples
SND_FORMAT_DSP_CORE	DSP program
SND_FORMAT_DSP_COMMANDS	Music Kit DSP commands
SND_FORMAT_DISPLAY	non-audio display data
SND_FORMAT_INDIRECT	fragmented sampled data
SND_FORMAT_UNSPECIFIED	unspecified format

All but the last five formats identify different sizes and types of sampled data. The others deserve special note:

- SND_FORMAT_DSP_CORE format contains data that represents a loadable DSP core program. Sounds in this format are required by the **SNDBootDSP()** and **SNDRunDSP()** functions. You create a SND_FORMAT_DSP_CORE sound by reading a DSP load file (extension ".lod") with the **SNDReadDSPfile()** function.

- SND_FORMAT_DSP_COMMANDS is used to distinguish sounds that contain DSP commands created by the Music Kit. Sounds in this format can only be created through the Music Kit's Orchestra class, but can be played back through the **SNDStartPlaying()** function.

- SND_FORMAT_DISPLAY format is used by the Sound Kit's SoundView class. Such sounds can't be played.

- SND_FORMAT_INDIRECT indicates data that has become *fragmented*, as described in a separate section, below.

- SND_FORMAT_UNSPECIFIED is used for unrecognized formats.

Fragmented Sound Data

Sound data is usually stored in a contiguous block of memory. However, when sampled sound data is edited (such that a portion of the sound is deleted or a portion inserted), the data may become discontiguous, or *fragmented.* Each fragment of data is given its own SNDSoundStruct header; thus, each fragment becomes a separate SNDSoundStruct structure. The addresses of these new structures are collected into a contiguous, NULL-terminated block; the **dataLocation** field of the original SNDSoundStruct is set to the address of this block, while the original format, sampling rate, and channel count are copied into the new SNDSoundStructs.

Fragmentation serves one purpose: It avoids the high cost of moving data when the sound is edited. Playback of a fragmented sound is transparent—you never need to know whether the sound is fragmented before playing it. However, playback of a heavily fragmented sound is less efficient than that of a contiguous sound. The **SNDCompactSamples()** C function can be used to compact fragmented sound data.

Sampled sound data is naturally unfragmented. A sound that's freshly recorded or retrieved from a soundfile, the Mach-O segment, or the pasteboard won't be fragmented. Keep in mind that only sampled data can become fragmented.

Sound C Functions

A number of C functions are provided that let you record, manipulate, and play sounds. These C functions operate on SNDSoundStructs and demand a familiarity with the structure. It's expected that most sound operations will be performed through the Sound Kit, where knowledge of the SNDSoundStruct isn't necessary. Nonetheless, the C functions are provided for generality and to allow sound manipulation without the Sound Kit. The functions are fully described in *Reference*.

The Sound Kit

The NeXT Sound Kit lets you access the sound hardware with a minimum of effort. Recording and playback of sound are particularly easy; the software manages data buffering, communication with the UNIX devices, synchronization with the operating system, and other such necessities. It's designed to accommodate both casual use of sound effects as well as detailed examination and manipulation of sound data.

The Sound Kit consists of three classes: Sound, SoundView, and SoundMeter.

The Sound Class

The Sound class provides a number of methods that let you access, modify, and perform sound data. The methods fall into four categories:

- Locating and storing sounds
- Recording and playback
- Editing
- Sound data manipulation

While a Sound object uses the SNDSoundStruct structure to represent its sound, you only need to be familiar with this structure if you're directly manipulating sound data.

Locating and Storing Sounds

Each Sound object represents a single sound. The Sound class provides four ways to install a sound in a Sound object. You can:

- Record a sound using the CODEC microphone input.
- Read sound data from a soundfile or Mach-O sound segment.
- Retrieve a sound from the pasteboard.

Sound recording (and playback) is described in the next section. Described here are the methods that let you read sounds from a soundfile or Mach-O segment and retrieve them from the pasteboard. As a shortcut to finding sounds, the Sound class provides a global naming mechanism that lets you identify and locate sounds by name.

Also described here are methods that let you store your sound by writing it to a soundfile or placing it on the pasteboard.

Soundfiles

Soundfiles are files on a disk that contain sound data. By convention, soundfile names are given a ".snd" extension. To read a soundfile into a Sound object, simply create the object and send it the **readSoundfile:** message:

```
#import <sound/sound.h>;                /* you must import this file */
. . .
id  aSound = [[Sound alloc] init];      /* create a Sound object */
int = [aSound readSoundfile:"KneeSqueak.snd"];   /* read a file */
```

The data in the named soundfile is read into the Sound object. The given file name is a complete UNIX pathname and must include the extension; in the example, the soundfile is searched for in the current working directory. Like many of the Sound methods, **readSoundfile:** returns an error code; the complete list of errors codes is given in the description of the **SNDSoundError()** C function in *Reference*. Success is indicated by the code SND_ERR_NONE.

These two operations, initializing a Sound object and reading a soundfile, are combined in the **initFromSoundfile:** method:

```
id  aSound = [[Sound alloc] initFromSoundfile:"KneeSqueak.snd"];
```

The method returns **nil** if the soundfile isn't found or if it can't be read. You can read a new soundfile into an existing Sound object at any time; the object's old sound is discarded.

NeXT provides a number of short sound effects (useful as system beeps) that are stored in the directory **/NextLibrary/Sounds**. These are:

Basso.snd
Bonk.snd
Frog.snd
Funk.snd
Pop.snd
SystemBeep.snd
Tink.snd

You can audition a soundfile by running the **sndplay** program from a Terminal or Shell window. For example:

```
sndplay /NextLibrary/Sounds/Frog.snd
```

Writing a soundfile from the data in a sound object is done by invoking the **writeSoundfile:** method:

```
[mySound writeSoundfile:"FleaSigh.snd"];
```

Even if the Sound object contains fragmented data, the data in the soundfile will be compact. However, the Sound object's data will remain fragmented.

The Mach-O Sound Segment

Reading a sound from the Mach-O sound segment is much like reading a soundfile: Like soundfiles, Mach-O sounds have a ".snd" extension. To read a Mach-0 sound, you invoke the **initFromMach0:** method:

```
id  mySound = [[Sound alloc] initFromMachO:"SonicBoom.snd"];
```

The Mach-O sound segment of your application is searched for the named sound. If it isn't found, the method returns **nil**.

You can install a sound (from a soundfile) into the Mach-O segment by supplying the **-segcreate** option when loading your application. For example:

```
cc ... -segcreate __SND SonicBoom.snd SonicBoom.snd
```

__SND is the name of the Mach-O sound segment. The first instance of **SonicBoom.snd** names the section of the Mach-O segment into which the soundfile is loaded. This is followed by the name of the soundfile (which must already exist). If you add a soundfile to your application through the Projects window in Interface Builder, the sound will automatically be included in the **make** script. Compiling a soundfile into your application lets you transport the application without regard for the original location of the file in the file system.

The Pasteboard

Placing a Sound object on the pasteboard lets you copy its data between running applications. To place a Sound on the pasteboard, invoke the **writeToPasteboard** method:

```
[mySound writeToPasteboard];
```

The object's data is compacted (if it's fragmented) and copied. The copy is then placed on the pasteboard.

To read data from the pasteboard into a Sound, invoke the **initFromPasteboard:** method:

```
id  mySound = [[Sound alloc] initFromPasteboard];
```

The sound data currently on the pasteboard is copied into the receiver of the message. Since the pasteboard can contain only one sound at a time, the method doesn't require an argument to further identify the sound. If there isn't a sound on the pasteboard, **initFromPasteboard** returns **nil**.

The Named Sound List

The Sound class maintains an application-wide list of named Sound objects called the *named Sound list*. The **addName:Sound:** class method lets you name a Sound object and add it to the named Sound list:

```
/* Add a Sound to the named Sound list. */
id namedSound = [Sound addName:"PopTop" sound:mySound];

/* Check for failure. */
if (namedSound == nil)
    . . .
```

The names in the named Sound list are unique; if you try to add a Sound by a name that's already in use, the effort is denied and **nil** is returned.

You can also name a Sound and place it on the named Sound list by sending **setName:** to the object:

```
id  namedSound = [mySound setName:"RedRover"];
```

setName: can be used to change the name of a Sound that's already on the named Sound list.

The **name** method retrieves a Sound object's name, whether given in a **setName:** message or through the **addName:sound:** method.

Named Sounds are visible to your entire application. To retrieve a named Sound and load a copy of its data into a new Sound object, invoke the **findSoundFor:** method:

```
id  newRedButton = [Sound findSoundFor:"RedButton"];
```

If **findSoundFor:** fails to find the Sound in the named Sound list, it gives its argument (the Sound name) a ".snd" suffix and looks for a named section in the Mach-O sound segment. If it's not found in the Mach-O segment, a soundfile (again, with the ".snd" extension) is searched for in these directories (in order):

1. ~/Library/Sounds/
2. /LocalLibrary/Sounds/
3. /NextLibrary/Sounds/

(~ represents the user's home directory.)

A Sound found through **findSoundFor:** is automatically added to the named Sound list.

To remove a named Sound from the named Sound list, invoke **removeSoundForName:**, passing the name of the object that you want to remove. Removing a named Sound neither frees the Sound nor changes the object's notion of its name (which it stores as an instance variable).

Identifying and locating Sounds through the named Sound list is generally the most efficient way to access sound data. The data in a named Sound is shared by all the objects that retrieve it.

Recording and Playing

To record a sound into a Sound object, simply create the object and send it the **record** message:

```
id  mySound = [[Sound alloc] init];
int errorCode = [mySound record];
```

Currently, the **record** method always records from the CODEC microphone input. The method returns immediately while the recording is performed by a background thread.

The value returned by **record** indicates the success or failure of the attempt to begin recording; SND_ERR_NONE indicates success.

The recording continues until the Sound object receives the **stop** message or until the Sound object can accommodate no more data. By default, the receiver of the **record** message is always set to accommodate ten minutes of 8 kHz mu-law sound (the type of sound data sent from the CODEC). You can set the size of the Sound object, prior to recording, to specify a different recording length. This is done through the **setDataSize:dataFormat:samplingRate:channelCount:infoSize:** method.

To play a sound, send the **play** message to the Sound object:

```
int errorCode = [mySound play];
```

Like recording, playback is performed by a background thread and the **play** method returns an error code. Playback continues until the entire sound is played or until the Sound object that initiated the playback receives the **stop** message.

A single Sound object can only perform one recording or playback operation at a time, thus the function of the **stop** method is never ambiguous: If the Sound is playing, **stop** stops the playback; if it's recording, it stops the recording.

You can temporarily suspend a playback or recording by sending the **pause** message to a Sound object. Like **stop**, the **pause** message halts whatever activity the Sound is currently engaged in; however, unlike **stop**, the Sound doesn't forget where it was. This allows the **resume** message to cause the Sound to continue its activity from the place at which it was paused.

The **record**, **play**, **pause**, **resume**, and **stop** methods (and the analogous action methods described in the next section) should only be used if you have a running Application object. To create a command-line program (similar to **sndrecord** or **sndplay**), you can use methods to create Sound objects and read sound data, but you should use the C functions **SNDStartRecording()**, **SNDStartPlaying()**, and **SNDStop()** to perform the Sound.

Action Methods

The Sound class methods **record:**, **play:**, **pause:**, **resume:**, and **stop:** are designed to be used as part of the target/action mechanism described in the *NeXTstep Concepts* manual. Briefly, this mechanism lets you assign a selected message (the action) and an object **id** (the target) to a Control object such that when the user acts on the Control, the action message is sent to the target object. In the following example, the three methods are assigned as action messages to three different Control objects—in this case, Buttons. The same Sound object is assigned as the Buttons' target:

```
/* Create a Sound object ... */
id  mySound = [[Sound alloc] init];

/* ...  and three Buttons.  */
id  recordButton = [[Button alloc] init],
    playButton = [[Button alloc] init],
    stopButton = [[Button alloc] init];

/* Set the action messages.  */
[recordButton setAction:@selector(record:)];
[playButton setAction:@selector(play:)];
[stopButton setAction:@selector(stop:)];

/* Set the targets.  */
[recordButton setTarget:mySound];
[playButton setTarget:mySound];
[stopButton setTarget:mySound];
```

In response to the user's clicking the different Buttons, the Sound object starts recording, starts playing, or stops one of these operations.

The Delegate

A Sound can have a delegate object. A Sound's delegate receives, asynchronously, the following messages as the Sound records or plays:

- **willPlay:** is sent just before the Sound begins playing.
- **didPlay:** is sent when the Sound finishes playing.
- **willRecord:** is sent just before recording.
- **didRecord:** is sent after recording.
- **hadError:** is sent if playback or recording generates an error.

To set a Sound's delegate object, invoke the **setDelegate:** method:

```
[mySound setDelegate:SoundDelegate];
```

A message is sent to the delegate only if the delegate implements the method that the message invokes.

Editing

The Sound class defines methods that support cut, copy, and paste operations for sampled sound data:

- **copySamples:at:count:** replaces the Sound's data with a copy of a portion of the data in its first argument, which must also be a Sound object.
- **insertSamples:at:** inserts a copy of the first argument's sound data into the receiving Sound object.
- **deleteSamplesAt:count:** deletes a portion of the Sound's data.

These methods all return **SNDSoundError()** type error codes (recall that SND_ERROR_NONE indicates success).

Note: The operations described here are also implemented in a more convenient form in the SoundView class; for example, replacing a portion of a Sound object with a portion of another Sound object requires all three methods listed above. By operating on a user-defined selection and using the pasteboard, the SoundView implements this operation in a single **paste:** method. The SoundView methods are less general than those in Sound, but if you want to include a simple graphic sound editor in your application, you should use the SoundView methods rather than these.

Delete

Deleting a portion of a Sound's data is direct; you simply invoke **deleteSamplesAt:count:**. For example:

```
/* Delete the beginning of mySound. */
int eCode = [mySound deleteSamplesAt:0 count:1000];
```

The first 1000 samples are deleted from the receiver of the message. The first argument specifies the beginning of the deletion in samples from the beginning of the data (counting from sample 0); the second argument is the number of samples to delete.

Copy and Paste

Copying a portion of one Sound and pasting it into another—or into itself, for that matter—requires the use of both **copySamples:at:count** and **insertSamples:at:**. In the following example, the beginning of **mySound** is repeated:

```
/* Create a stutter at the beginning of mySound. */
id tmpSound = [[Sound alloc] init];

int errorCode = [tmpSound copySamples:mySound at:0 count:1000];
if (errorCode == SND_ERROR_NONE)
    errorCode = [mySound insertSamples:tmpSound at:0];

[tmpSound free];
```

First, the data in **tmpSound** is completely replaced by a copy of the first 1000 samples in **mySound**. Note that the **copySamples:at:count** method doesn't remove any data from its first argument, it simply copies the specified range of samples from the first argument into the receiver. Next, **tmpSound** is prepended to **mySound**, creating a repetition of the first 1000 samples in **mySound**. The **insertSamples:** method inserts a copy of the argument into the receiver. Thus, the argument can be freed after inserting.

The two Sound objects involved in the **insertSamples:at:** method (the receiver and the first argument) must be compatible: They must have the same format, sampling rate, and channel count. If possible, the data that's inserted into the receiver of **insertSamples:at:** is automatically converted to be compatible with the data already in the receiver (see the description of the **SNDConvertSound()** C function in *Reference* for a list of the conversions that are supported). An error code indicating that the insertion failed is returned if the two Sounds aren't compatible or if the inserted data can't be converted.

Replace

Replacing is like copying and pasting, except that a region of the pasted-into Sound is destroyed to accommodate the new data. In the following example, the beginning of **oneSound** is replaced with a copy of the beginning of **twoSound**:

```
/* Replace the beginning of oneSound with that of twoSound.  */
int tmpCode = [tmpSound copySamples:twoSound at:0 count:1000];
int inCode;

if (tmpCode == SND_ERROR_NONE) {
    int oneCode = [oneSound deleteSamplesAt:0 count:1000];
    if (oneCode == SND_ERROR_NONE)
        inCode = [oneSound insertSamples:tmpSound at:0]; }

[tmpSound free];

/* Check inCode before performing further manipulations. */
. . .
```

Utility Methods

The editing methods described above only work on Sounds that contain sampled data. The **isEditable** method is provided to quickly determine whether a Sound object can be edited. The method returns YES if the object can be edited, NO if it can't.

The **compatibleWith:** method takes a Sound object as its argument and returns YES if the argument and the receiver are compatible. (The method also returns YES if one of the objects is empty; in other words, it's OK to insert samples into an empty object.) This method is useful prior to invoking the **insertSound:at:** method.

Another handy method is **sampleCount**, which returns the number of *sample frames* contained in the receiver. A sample frame is a channel-independent count of the samples in a Sound. For example, sending **sampleCount** to a two-channel Sound that contains three seconds worth of data returns the same value as sending it to a one-channel Sound that also contains three seconds of data (given that the two Sounds have the same sampling rate), even though the two-channel Sound actually contains twice as much data.

Other Editing Methods

The Sound class defines three more editing methods:

- **copy** returns a new Sound object that's a copy of the receiver.
- **copySound:** takes a Sound object as an argument and replaces the data in the receiver with the data in its argument. Since the entire range of data in the receiver is replaced, it needn't be editable, nor must the two Sounds be compatible.
- **deleteSamples** can only be sent to an editable Sound. It deletes the receiver's sound data.

Fragmentation

A Sound's data is normally contiguous in memory. However, when you edit a Sound object, its data can become fragmented, or discontiguous. Fragmentation is explained in the description of the SNDSoundStruct, earlier in this chapter. Briefly, fragmentation lets you edit Sounds without incurring the cost of moving large sections of data in memory. However, fragmented Sounds can be less efficient to play. The **needsCompacting** and **compactSamples** methods are provided to determine if a Sound is fragmented and to compact it. Note that compacting a large Sound that has been mercilessly fragmented can take a noticeably long time.

The SoundView Class

The SoundView class provides a mechanism for displaying the sound data contained in a Sound object. While SoundView inherits from the Application Kit's View class, it implements a number of methods that are also defined in Sound, such as **play:**, **record:**, and **stop:**. In addition, it implements editing methods such as **cut:**, **copy:**, and **paste:**.

SoundViews are designed to be used within a ScrollView. While you can create a SoundView without placing it in a ScrollView, its utility—particularly as it's used to display a large Sound—is limited.

Creating and Displaying a SoundView

To display a sound, you create a new SoundView with a particular frame, give it a Sound object to display (through **setSound:**), and then send the **display** message to the SoundView:

```
/* Create a new SoundView object. */
id  mySoundView = [[SoundView alloc] initFrame:&svRect];

/* Set its Sound object.  */
[mySoundView setSound:mySound];

/* Display the Sound object's sound data. */
[mySoundView display];
```

In the example, **svRect** is a previously defined NXRect. If autodisplaying is turned on (as set through View's **setAutodisplay:** method), you needn't send the **display** message; simply setting the Sound will cause the SoundView to be displayed.

For most complete sounds, the length of the Sound's data in samples is greater than the horizontal length of the SoundView in display units. The SoundView employs a reduction factor to determine the ratio of samples to display units and plots the minimum and maximum amplitude values of the samples within that ratio. For example, a reduction factor of 10.0 means that the minimum and maximum values among the first ten samples are plotted in the first display unit, the minimum and maximum values of the next ten samples are displayed in the second display unit and so on. You can set the reduction factor through the **setReductionFactor:** method.

Changing the reduction factor changes the time scale of the object. As you increase the reduction factor, more "sound-per-inch" is displayed. Of course, since more samples are used in computing the average amplitude, the resolution in a SoundView with a heightened reduction factor is degraded. Conversely, reducing the reduction factor displays fewer samples per display unit but with an improved resolution. You should be aware that changing the reduction factor on a large sound can take a noticeably long time.

SoundView Dimensions

In a SoundView, time runs from left to right; amplitude is represented on the y-axis, with 0.0 amplitude in the (vertical) center. When you set a SoundView's Sound, the amplitude data that's displayed is automatically scaled to fit within the given height of the SoundView.

The manner in which a SoundView's horizontal dimension is computed depends on the object's **autoscale** flag. If autoscaling is turned off, the length of a SoundView's frame is resized to fit the length of the Sound object's data while maintaining a constant reduction factor. In other words, a SoundView that's displaying a Sound that contains 10000 samples will be twice as long as one with a Sound that contains 5000 samples, given the same reduction factor in either SoundView.

Whenever the displayed data changes, due to editing or recording, the SoundView is resized to fit the length of the new data. This is particularly useful in a SoundView that's inside a ScrollView: The ScrollView determines the portion of data that's actually displayed, while the SoundView maintains a constant time scale. Changing the reduction factor with autoscaling turned off causes the SoundView to zoom in or out on the displayed data.

You can enable autoscaling by sending the message

```
/* Enable autoscale. */
[mySoundView setAutoscale:YES];
```

With **autoscale** enabled, the SoundView's frame size is maintained regardless of the length of the SoundView's Sound data. Instead, the reduction factor is recomputed so the length of the data will fit within the frame. When autoscaling is on, invoking **setReductionFactor:** has no effect.

Display Modes

A SoundView can display a sound as a continuous waveform, such as you would see on an oscilloscope, or as an outline of its maximum and minimum amplitudes. You set a SoundView's display mode by sending it the **setDisplayMode:** message with one of the following Sound Kit constants as an argument:

Constant	Meaning
SK_DISPLAY_WAVE	Waveform display
SK_DISPLAY_MINMAX	Amplitude outline display

Waveform display is the default.

The SoundView Selection

The SoundView class provides a selection mechanism. You can selectively enable the selection mechanism for each SoundView object by sending the **setEnabled:YES** message. When you drag in an enabled SoundView display, the selected region is highlighted. The method **getSelection:size:** returns, by reference, the number of the first sample and the number of samples in the selection.

Chapter 3
Representing Music Data

Chapter 3
Representing Music Data

This chapter describes the classes, methods, and C Functions that the Music Kit defines to represent music data.

The Note Class

Whether you are composing music, designing a performance scheme, or building software instruments, you need a thorough understanding of the Note class. The Note class provides a means for describing musical sounds; a Note object is a repository of musical information that's passed to and acted on by other Music Kit objects.

A Note contains three categories of information:

- A collection of *parameters*. Parameters describe the attributes of a musical sound, such as its frequency (pitch) and amplitude (loudness). A Note can contain any number of parameters, including none.

- A single *note type* that expresses the basic character of the Note object, whether it defines an entire musical note, or just its beginning, middle, or end.

- An integer identifier called a *note tag*. Note tags are used to associate a series of Notes. For example, two separate Note objects that define the beginning and the end of a musical note must have the same note tag value.

The three categories of Note information are examined in detail in the following sections.

Parameters

Parameters are the pith of a Note object. They're used to enumerate and describe the quantifiable aspects of a musical sound. The most important rule of parameters is that they don't *do* anything; in order for a parameter to have an effect, another object (or your application) must retrieve and apply it in some way. For example, the subclasses of SynthPatch provided by the Music Kit are designed to look for particular sets of parameters when synthesizing a Note. Some common parameters, such as those for frequency and amplitude, are looked for by all these classes.

A parameter consists of a unique integer tag, a unique print name (a string), and a value. The tag and name are used to identify the parameter:

- The parameter's tag identifies it within an application.
- The print name identifies the parameter in a scorefile.

Thus, the tag and the name are simply two ways of identifying the same parameter. To create a new parameter, you pass a print name (that you make up yourself) to the **parName:** class method. The method returns a unique tag for the parameter:

```
/* Create a new parameter tag (an int). */
int  myPar = [Note parName:"myPar"];
```

The name of the variable that represents the tag needn't be the same as the string name, although in the interest of clarity this is regarded as good form. The **parName:** method can also be used to retrieve the tag of a parameter that's already been created: **parName:** creates a new tag for each unique argument that's passed to it; subsequent invocations with the same argument will return the already-created tag.

Since the Music Kit SynthPatches look for particular parameters during synthesis, it follows that the Music Kit must also supply some number of parameter tags. These are listed and described in Appendix B, "Music Tables," in the *Sound, Music, and Signal Processing: Reference* manual.

Music Kit parameter tags are represented by integer constants such as **MK_freq** (for frequency) and **MK_amp** (for amplitude). The print names are formed by dropping the "MK_" prefix. Thus, **MK_freq** is represented in a scorefile as "freq" and **MK_amp** is "amp".

By definition, the parameter tags supplied by the Music Kit are sufficient for all uses of its SynthPatches and Midi. If you create your own SynthPatch subclass, you can create additional parameter tags to fully describe its functionality, but you should use as many of the Music Kit parameter tags as are applicable. For example, it's assumed that all SynthPatch subclasses will have a settable frequency; rather than create your own frequency parameter tag, you should use **MK_freq**. This promotes portability between SynthPatches.

Lest the emphasis on synthesis be misconstrued, keep in mind that a parameter's purpose is not restricted to that arena. Parameters can be used in any way that your application sees fit; for example, a graphic notation program could use parameters to describe how a Note object is displayed on the screen. However, you should also keep in mind that a parameter is significant only if some other object or your application looks for and uses it.

Parameter Values

The method you use to set a parameter's value depends on the data type of the value. The Note class provides six value-setting methods. The first three of these are:

- **setPar:toDouble:** sets the parameter value as a **double**.
- **setPar:toInt:** sets the value as an **int**.
- **setPar:toString:** sets the value as a pointer to a string.

The other three methods will be examined later.

The argument to the **setPar:** keyword is a parameter tag; the second argument is the value that you're setting. For example, to set the value of the bearing parameter (stereophonic location of a DSP synthesized sound) to 45.0 degrees (hard right), you could send any of the following messages:

```
/* Of course, you have to create the Note first. */
id  aNote = [[Note alloc] init];

/* Set the bearing. */
[aNote setPar:MK_bearing toDouble:45.0];

/* or */
[aNote setPar:MK_bearing toInt:45];

/* or */
[aNote setPar:MK_bearing toString:"45"];
```

You generally set bearing as a **double**—all the Music Kit SynthPatches apply bearing, as well as most other number-valued parameters, as a value of that type. However, retrieval methods are provided that perform type conversion for you. For example, the message

```
/* Retrieve the bearing parameter value as a double. */
double  theBearing = [aNote ParAsDouble:MK_bearing];
```

returns the **double** 45.0 regardless of which of the three methods you used to set the value. The retrieval methods include:

- **parAsDouble:** returns the value as a **double**.
- **parAsInt:** returns the value as an **int**.
- **parAsString:** returns a pointer (a **char** *) to a copy of the value.
- **parAsStringNoCopy:** returns a pointer to the value itself.

Note: You shouldn't alter the string returned by **parAsStringNoCopy:**. It's owned by the Note object.

If the parameter hasn't been set, the retrieval methods return values as follows:

- **parAsDouble:** returns MK_NODVAL.
- **parAsInt:** returns MAXINT.
- The string retrieval methods return an empty string.

Unfortunately, you can't use MK_NODVAL in a simple comparison predicate. To check for this return value, you must call the in-line function **MKIsNoDVal()**; the function returns 0 if its argument is MK_NODVAL and nonzero if not:

```
/* Retrieve the value of the amplitude parameter. */
double amp = [aNote parAsDouble:MK_amp];

/* Test for the parameter's existence. */
if (!MKIsNoDVal(amp))
... /* do something with the parameter */
```

For most uses of parameters—in particular, if you're designing a SynthPatch subclass—it's important to know whether the parameter was actually set before applying its value. You can compare the retrieved values with the values shown above to check whether the parameter had actually been set, or you can test the **BOOL** value returned by the **isParPresent:** method:

```
/* Only retrieve bearing if its value was set. */
if ([aNote parIsPresent:MK_bearing])
    double  theBearing = [aNote parAsDouble:MK_bearing];
```

Choosing a Value

To properly set a parameter's value, you need to know the range of values that are meaningful to the object that applies it. The Music Kit parameter lists given in Appendix B supply this information. If you're creating an application (or writing a scorefile) in order to synthesize Notes on the DSP or on an external MIDI synthesizer, you should consult these lists to make sure you're setting the Notes' parameters to reasonable and musically useful values.

Three of the most commonly used parameters, those for pitch, begin time, and duration, are special. See the section "Basic Parameters," later in this chapter, for a discussion of alternative ways to set and retrieve the values of these parameters.

Object-Valued Parameters

Some parameters take objects as their values. The methods for setting an object-valued parameter are:

- **setPar:toEnvelope:** sets the value as an Envelope object.
- **setPar:toWaveTable:** sets the value as a WaveTable object.
- **setPar:toObject:** sets the value as a non–Music Kit object.

Envelopes and WaveTables are described later in this chapter. The **setPar:toObject:** method is provided so you can set a parameter to an instance of one of your own classes. The class that you define should implement the following methods so its instances can be written to and read from a scorefile:

- **writeASCIIStream:** provides instructions for writing the object as ASCII text. In a scorefile, the text that represents an object—this includes Envelopes and WaveTables—is enclosed in square brackets ([]). The ASCII representation of an object must not include a close bracket. The method's argument is the NXStream to which the text is written.

- **readASCIIStream:** provides instructions for creating an object from its ASCII representation. When the method is called, the argument (an NXStream) is pointing to the first character after the open bracket. You should leave the argument pointing to the close bracket. In other words, you should read in whatever you wrote out.

Both of these methods are called automatically when you read a scorefile into your application (scorefile-reading methods are described later in this chapter).

You can retrieve an object-valued parameter through the following methods:

- **parAsEnvelope:** returns an Envelope object.
- **parAsWaveTable:** returns a WaveTable object.
- **parAsObject:** returns a non-Music Kit object.

Unlike the value retrieval methods shown in the previous section, these methods return **nil** if the parameter's value isn't the correct type.

Basic Parameters

A handful of attributes are common to all musical notes: pitch, loudness, begin time, and duration. Special methods and values are used to set the parameters that represent these attributes, as explained in the following sections.

Frequency and Pitch

Frequency and pitch are two terms that refer to the most fundamental aspect of a musical sound: its register or tonal height. Frequency is the exact measurement of the periodicity of an acoustical waveform expressed in hertz. Pitch, on the other hand, is an inexact representation expressed in musical names such as F sharp, A flat, or G natural.

When the DSP synthesizes a musical note, it produces a tone at a specified frequency. However, musicians think in terms of pitch. To bridge the gap between frequency and pitch, the Music Kit defines sets of *pitch variables* and *key numbers* that represent particular frequencies.

Pitch Variables

A pitch variable takes the following form:

pitchLetter[*sharpOrFlat*]*octave*

- *pitchLetter* is a lowercase letter from **a** to **g**. As in standard music notation, the Music Kit's pitch variables are organized within an octave such that **c** is the lowest pitch and **b** is the highest.

- The optional *sharpOrFlat* is **s** for sharp and **f** for flat. They raise or lower by a semitone the pitch indicated by *pitchLetter*.

- *octave* is 00 or an integer from 0 to 9. The octave component of the pitch name variable places the pitch class within a particular octave, where 00 is the lowest octave and 9 is the highest. Octaves are numbered such that **c4** is middle C.

Some examples of pitch variables are:

Pitch Variable	Pitch
ef4	E flat above middle C
gs3	G sharp below middle C
f00	F natural in the fifth octave below middle C
bs8	B sharp five octaves above middle C (the same as c9)

Notice that the natural sign isn't represented in the pitch variables. If neither the sharp nor the flat sign is present, the pitch is automatically natural. In addition, key signatures aren't represented; the accidentals that define a key must be present in each pitch that they affect.

Correspondence Between Pitch Variables and Frequencies

Each pitch variable represents a predefined frequency. By default, the frequencies that correspond to the pitch variables define a *twelve-tone equal-tempered* tuning system, with **a4** equal to 440.0 Hz:

- Twelve-tone means that there are twelve discrete tones within an octave.

- Equal-tempered means that the frequency ratio between any pair of successive tones is always the same.

This is the tuning system used to tune modern fixed-pitch instruments, most notably the piano. The complete table of pitch variables and the corresponding default frequencies is given in Appendix B of *Reference*.

Key Numbers

Another way to specify the pitch is to use a key number. Key numbers are integers that correspond to the keys of a MIDI keyboard. As a MIDI standard, 60 is the key number for the middle C of the keyboard. The Music Kit provides constants to represent key numbers. The form of these constants is like that of the pitch variables, but with the letter **k** appended; for example:

Pitch Variable	Key Number
ef4	ef4k
gs3	gs3k
f00	f00k
bs8	bs8k

Key numbers are provided primarily to accommodate MIDI instruments. If you record a MIDI performance (using a Midi object), the pitch specifications will all be represented as key numbers. When you realize Notes on a MIDI synthesizer, the actual frequency represented by a particular key number is controlled by the synthesizer itself. The standard of "60 equals middle C" simply means that key number 60 creates a tone at whatever frequency the synthesizer's middle C key is tuned to produce.

Specifying Pitch in a Note

You can specify the pitch of Note objects as a frequency or pitch variable (a **double**), or as a key number (an **int**). These are represented by the parameter tags **MK_freq** and **MK_keyNum**. Regardless of how it's synthesized (on the DSP or on a MIDI instrument), the appropriate value is converted from whichever parameter is present.

To set a Note's pitch, you use the methods described earlier:

```
/* You must import this file when using pitch variables. */
#import <musickit/pitches.h>

/* Set the Note's pitch to middle C as a frequency. */
[aNote setPar:MK_freq toDouble:261.625];

/* The same using a pitch variable. */
[aNote setPar:MK_freq toDouble:c4];

/* And as a key number. */
[aNote setPar:MK_keyNum toDouble:c4k];
```

The conversion between frequencies or pitch variables and key numbers allows you to create Note objects that can be played on both the DSP and on a MIDI instrument using the same pitch parameter.

Retrieving Pitch from a Note

Special methods are provided to retrieve pitch:

- **freq:** returns a **double** value as a frequency.
- **keyNum:** returns an **int** as a key number.

If the **MK_freq** parameter isn't present but **MK_keyNum** is, the **freq:** method returns a frequency value converted from the **MK_keyNum** parameter. Similarly, **keyNum:** returns a key number value converted from **MK_freq** in the absence of **MK_keyNum**.

The Music Kit SynthPatches use **freq:** to retrieve pitch information; Midi uses **keyNum:**.

Keep in mind that either retrieval method converts a value from the opposite parameter only if its own parameter isn't set. In addition, you can set **MK_freq** and **MK_keyNum** independently of each other: Setting one doesn't reset the other.

Since frequencies are continuous and key numbers are discrete, the correspondence between them isn't exact; conversion from frequency to key number sometimes requires approximation. The pitch table in Appendix B of *Reference* gives the frequency range that corresponds to particular key numbers (in the default tuning system).

Loudness

The perceived loudness of a musical note depends on a number of factors, the most important being the amplitude of the waveform and its spectral energy, or brightness. All the Music Kit SynthPatches use the amplitude parameter, **MK_amp**; most also use **MK_bright**, the brightness parameter.

Amplitude

Amplitude is fairly straightforward: The value of the amplitude parameter determines the strength of the signal produced by the DSP. The value of **MK_amp** is retrieved as a **double**. Its value must be between 0.0 and 1.0, where 0.0 is inaudibly soft and 1.0 is a fully saturated signal. Perceived amplitude increases logarithmically: Successive Notes with incrementally increasing amplitude values are perceived to get louder by successively smaller amounts. For instance, the difference in loudness between amplitudes of 0.1 and 0.2 sounds much greater than the difference between 0.8 and 0.9.

Amplitude is set and retrieved through the normal methods; for example:

```
/* Set the amplitude of a Note. */
[aNote setPar:MK_amp toDouble:0.2];

/* Retrieve amplitude. */
double myAmp = [aNote parAsDouble:MK_amp];

/* Set the amplitude of a Note. */
[aNote setPar:MK_amp toDouble:MKdB(-15.0)];
```

The range of the decibel scale extends from negative infinity (inaudible) to 0.0 (maximally loud). Decibel scaling creates a linear correspondence between increasing value and perceived loudness: The perceived increase in loudness from −20.0 to −15.0 is the same as that from −15.0 to −10.0 (as well as from −10.0 to −5.0 and from −5.0 to 0.0).

Brightness

Brightness can be thought of as a tone control. The greater the value of **MK_bright**, the brighter the synthesized sound. As you decrease brightness, the sound becomes darker. **MK_bright** is used differently by the various SynthPatch subclasses; usually it's used to modify the values of other timbre-related parameters. Some SynthPatches, such as those that perform WaveTable synthesis, don't use **MK_bright** at all.

Brightness values are usually set and retrieved through **setPar:toDouble:** and **parAsDouble:** (the Music Kit SynthPatches always retrieve the value of **MK_bright** as a **double**). The range of valid brightness values is, in general, 0.0 to 1.0; you can actually set **MK_bright** to a value in excess of 1.0, although this may cause distortion in some SynthPatches. Specifying brightness in decibels is possible, but the scale tends to have less meaning here than it does for amplitude.

Begin Time and Duration

The begin time, or *time tag*, and duration parameters of a Note are set through the methods **setTimeTag:** and **setDur:**. Both methods take a **double** argument that's measured in *beats*. By default, a beat is one second long; however, you can change the size of a beat through methods defined in the Conductor class.

The retrieval methods **timeTag** and **dur** return a Note's time tag and duration. Because of the specialized methods for setting and retrieving these parameters, they don't have parameter tags to represent them, nor do they have print names. Their representation in a scorefile is explained in *Reference*.

For some applications, setting a Note's time tag isn't necessary; for instance, you can design a Performer object that creates Notes and performs them immediately. However, in many musical applications—in particular, for any application that adds Notes to a Part—time tags

are indispensable. For convenience, the **newSetTimeTag:** method lets you create a new Note and set the time tag value at the same time:

```
/* Create a new Note and set the time tag value to 3.5 beats. */
id  myNote = [Note newSetTimeTag:3.5];
```

A newly created Note otherwise has a time tag value of 1.0. Time tags are typically measured from the beginning of a performance (the Performer class provides methods that let you add a begin time offset). The Note in the example would thus be played after three and a half beats of a performance.

Duration is also in beats and indicates, ostensibly, the longevity of a Note during synthesis: When the duration has expired, the Note doesn't stop short; instead, its Envelope objects (if any) start to wind down. The actual length of the Note is its duration value plus the amount of time it takes for its amplitude Envelope to finish. This is described in detail in the section "The Envelope Class," later in this chapter.

Many Notes don't have a duration value. For example, some Note objects initiate a synthesized tone that plays until a subsequent Note object, also lacking a duration, specifically turns it off. The necessity or superfluity of the duration value is described in the following sections.

Note Type and Note Tag

A Note's note type describes its musical function with regard to the life of a synthesized sound. There are five note types. Briefly they are:

- NoteDur represents an entire musical note.
- NoteOn establishes the beginning of a note.
- NoteOff establishes the end of a note.
- NoteUpdate modifies a sounding note.
- Mute makes no sound.

Each of the five types is represented by an MKToken constant:

- MK_noteDur
- MK_noteOn
- MK_noteOff
- MK_noteUpdate
- MK_mute

Every Note has exactly one note type; the default is **MK_mute**. You set the note type with **setNoteType:** and retrieve it with **noteType**.

There are two styles for creating a complete musical note, either with a single noteDur or with a noteOn/noteOff pair.

Note tags are integers that are used to identify Note objects that are part of the same phrase; in particular, matching note tags are used to create noteOn/noteOff pairs and to associate noteUpdates with other Notes. The actual integer value of a note tag has no significance. The range of note tag values extends from 0 to $2^{31}-1$.

You set a Note's note tag through **setNoteTag:** and retrieve it with **noteTag**. The C function **MKNoteTag()** is provided to create note tag values that are guaranteed to be unique across your entire application—you should never create note tag values except through this function.

The following example, in which a noteOff is paired with a noteOn, demonstrates how to create and administer note tags:

```
/* Create a noteOn and a noteOff and set their time tags. */
id  aNoteOn = [[Note alloc] initWithTimeTag:1.0];
id  aNoteOff = [[Note alloc] initWithTimeTag:3.5];
[aNoteOn setNoteType:MK_noteOn];
[aNoteOff setNoteType:MK_noteOff];

/* Create a new note tag for the noteOn. */
[aNoteOn setNoteTag:MKNoteTag()];

/* Set the noteOff note tag to that of the noteOn. */
[aNoteOff setNoteTag:[myNoteOn noteTag]];
```

The following sections further examine each note type and discuss note tags as they apply to each type.

NoteDur

The information in a noteDur defines an entire musical note. A noteDur is distinguished by having a duration ("Dur" stands for duration). Of the five note types, only noteDur can have a duration value—invoking **setDur:** automatically sets a Note's duration to **MK_noteDur**.

You can associate any number of noteUpdates with a noteDur, thereby changing the attributes of the musical note while it's sounding. In order to associate a noteUpdate to a noteDur, they must be given the same note tag, as described above. NoteUpdates are described in a subsequent section.

NoteOn and NoteOff

The other way to define a complete musical note is to use a noteOn/noteOff pair. A noteOn starts a musical note and a subsequent noteOff terminates it. Each noteOn/noteOff pair must share a unique note tag.

If the same note tag is given to successive noteOns that aren't articulated by intervening noteOffs, the second and subsequent noteOns retrigger the Note's Envelopes when it's synthesized on the DSP.

A noteOff triggers the release portion of a Note's Envelope. Any parameters that it contains are applied to that portion of the Note, however brief. See the "The Envelope Class" section, later in this chapter.

NoteUpdate

NoteUpdates are used to alter the parameters of a musical note that's already underway. A noteUpdate is associated with another Note by virtue of matching note tags. In the following example, a noteUpdate is used to change the pitch of a musical note represented by a noteDur:

```
id   myNoteDur, myNoteUpdate;

/* Create a Note with a time tag and set its pitch, and duration. */
myNoteDur = [[Note alloc] initWithTimeTag:1.0];
[myNoteDur setPar:MK_freq toDouble:c4];
[myNoteDur setDur:3.0];

/* Create a noteUpdate with a time tag and set its pitch. */
myNoteUpdate = [[Note alloc] initWithTimeTag:2.5];
[myNoteUpdate setNoteType:MK_noteUpdate];
[myNoteUpdate setPar:MK_freq toDouble:d4];

/* Set the note tags to the same value. */
[myNoteDur setNoteTag:MKNoteTag()];
[myNoteUpdate setNoteTag:[myNoteDur noteTag]];
```

The effect of the two Notes is a single, two-beat-long musical note that changes pitch after one-and-a-half beats.

Only the parameters that are explicitly present in the noteUpdate are applied to the sounding note: If a particular parameter is present in the original Note but is absent in an associated noteUpdate, the value of the original parameter is retained.

A noteUpdate with no note tag affects all the currently sounding Notes that are being realized through the same SynthInstrument object.

Mute

A mute is normally ignored by SynthPatch and Midi objects, so it can't be used to represent a sound-making event. Mutes are useful for representing structural breakpoints such as bar lines. If you send the **setNoteTag:** message to a mute, its note type is changed to **MK_noteUpdate**.

The Envelope Class

An envelope is a function that varies over time. Envelopes are extremely important to synthesized music because they allow continuous control of the attributes of a sound. For example, with an envelope you can specify how quickly a musical note speaks and how long it takes to die away. Without envelopes, a synthesized tone would snap on, maintain a steady amplitude for its entire duration, and then snap off. ("Snap" can be taken literally: Both the arrival and the departure of the sound would be accompanied by an audible click.)

An envelope is depicted as a continuous line on an xy coordinate system, where time moves forward from left to right on the x-axis, and the envelope's value at a particular time is given as y. Figure 3-2 shows some typical envelope shapes. The top two envelopes, with their characteristic initial rise and ultimate fall, are typical of those used to control amplitude. The bottom one, applied to frequency, would introduce some warble at the beginning and end of a note.

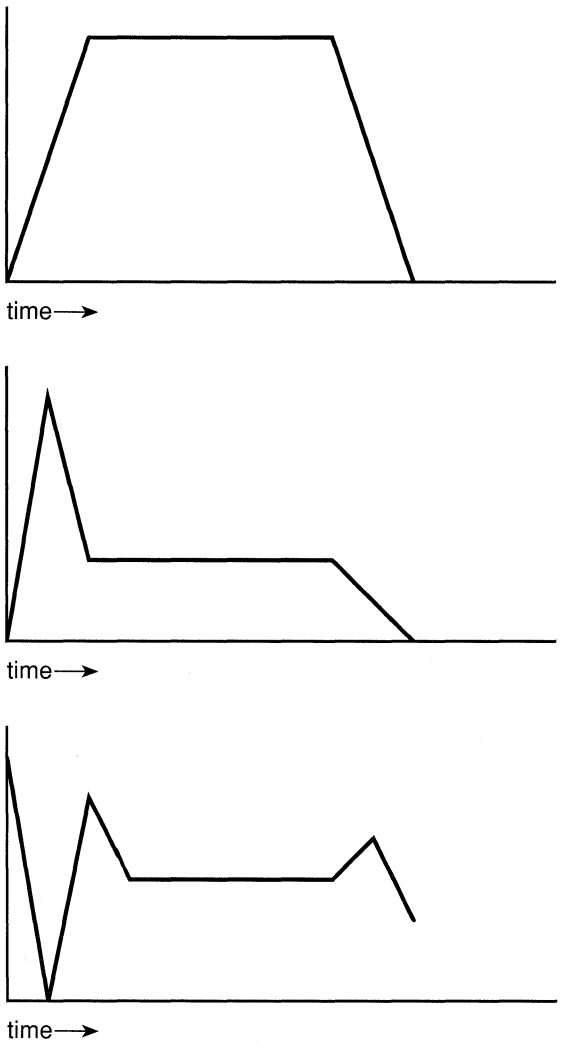

Figure 3-2. Typical Envelopes

Instances of the Music Kit's Envelope class are used to represent envelope functions. An Envelope object contains a series of x,y coordinates, or *breakpoints*, that mark a change in an envelope's direction or trajectory. Figure 3-3 superimposes breakpoints on the previously illustrated envelope shapes (an open circle denotes the location of a breakpoint).

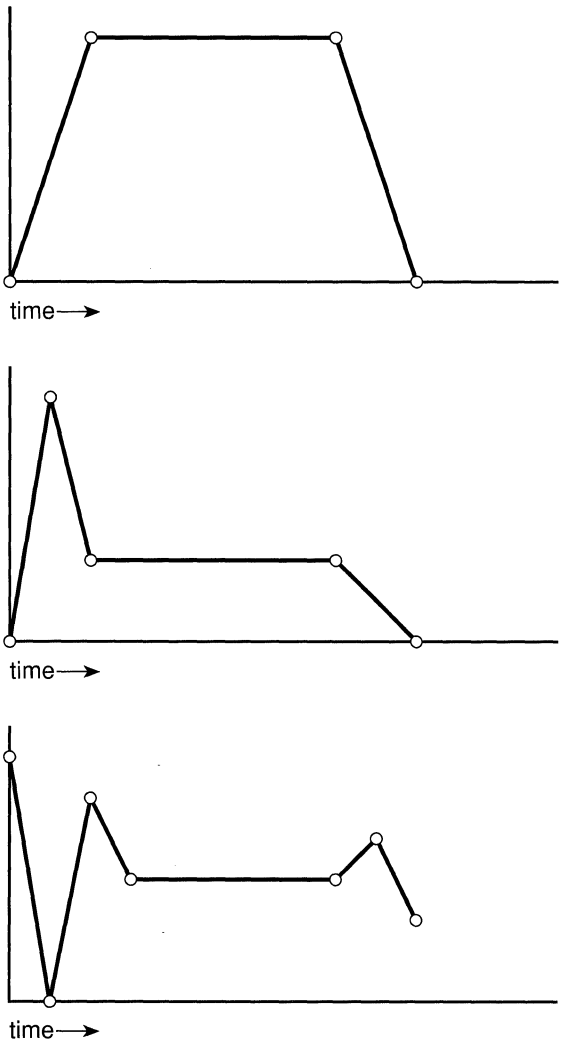

Figure 3-3. Breakpoint Envelopes

An Envelope object can have any number of breakpoints, allowing you to create arbitrarily complex functions.

You can use an Envelope object to control virtually any attribute of a sound synthesized on the DSP. While Envelope control is indispensable for amplitude, it can also be used to good effect for frequency and timbre-related attributes associated with particular synthesis techniques.

Besides providing continuous control of a sound's attributes, an Envelope can also be used to retrieve discrete values of y for given values of x. The retrieved values can then be used, for example, to set the same parameter in a series of Notes, allowing you to control the parameter's evolution over an entire musical phrase.

The following sections examine the methods that define Envelope objects and demonstrate how to use them in DSP synthesis and for discrete-value retrieval.

Defining an Envelope

The (x,y) value pairs that define an Envelope's shape are set through the **setPointCount:xArray:yArray:** method. The first argument is the number of breakpoints in the Envelope; the other two arguments are arrays of x values and y values:

```
/* Create an Envelope object. */
id      anEnvelope = [[Envelope alloc] init];

/* Create and instantiate arrays for the x and y values. */
double  xVals[] = {0.0, 1.0, 4.0, 5.0};
double  yVals[] = {0.0, 1.0, 1.0, 0.0};

/* Define the Envelope with data. */
[anEnvelope setPointCount:4 xArray:xVals yArray:yVals];
```

The elements in the x and y arrays are paired in the order given. Thus, the first breakpoint in an Envelope is created from the first element in the x array and the first element in the y array, the second breakpoint is created from the second elements of either array, and so on. Figure 3-4 illustrates the Envelope object defined in the example. The x and y values for each breakpoint are shown in parentheses.

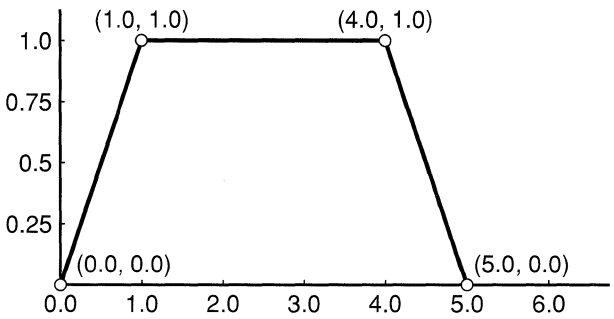

Figure 3-4. Simple Envelope

The way the x and y values are interpreted depends on the way the Envelope is used. In general, an Envelope is scaled by other values, allowing the same Envelope object to be stretched and squeezed to fit a number of different contexts.

Envelopes and the DSP

The most important use of an Envelope is to provide continuous control over a musical attribute of a Note that's synthesized on the DSP. To do this, you supply the Envelope object as a parameter to the Note. For example, an Envelope used to control amplitude is set as a Note's **MK_ampEnv** parameter:

```
/* Create a Note and an Envelope. */
id  aNote = [[Note alloc] init];
id  anEnvelope = [[Envelope alloc] init];

/* Create x and y value arrays and define the Envelope. */
double  xVals = {0.0, 1.0, 4.0, 5.0};
double  yVals = {0.0, 1.0, 1.0, 0.0};
[anEnvelope pointCount:4 xArray:xVals yArray:yVals];

/* Set the Envelope to control aNote's amplitude. */
[aNote setPar:MK_ampEnv toEnvelope:anEnvelope];
```

The Envelope defined here is the same as the one illustrated in Figure 3-4, above. When **aNote** is synthesized its amplitude follows the curve shown in the illustration. It rises from zero, maintains a steady state, and then falls back to zero.

As with any parameter, an Envelope-valued parameter is only meaningful if it's looked for and used by the SynthPatch object that synthesizes the Note. Appendix B lists and describes the Envelope parameters used by the Music Kit SynthPatch subclasses.

In addition, the Music Kit SynthPatches are designed such that Envelopes are only significant in a noteOn or a noteDur. Setting an Envelope parameter in a noteOff or a noteUpdate has no immediate effect, although it's used if the phrase is rearticulated and the rearticulating Note (by definition, a noteOn or noteDur) doesn't specify the Envelope parameter itself.

Scale and Offset

Associated with each Envelope parameter provided by the Music Kit are two related parameters that interpret the Envelope's y values. The names of these parameters are formed as **MK_***attribute***0** and **MK_***attribute***1**:

- **MK_***attribute***0** is the value of the Envelope when y is 0.0.
- **MK_***attribute***1** is the value of the Envelope when y is 1.0. As a convenience, the parameter **MK_***attribute* is defined as a synonym for **MK_***attribute***1**.

The parameters that interpret the amplitude Envelope, for example, are **MK_amp0** and **MK_amp1** (which is synonymous with **MK_amp**). Since amplitude should always rise from and fall back to 0.0 (to avoid clicks), you'll probably never need to set the value of **MK_amp0**—if the parameter isn't set, its value defaults to 0.0. The amplitude Envelope is normally interpreted by setting the value of **MK_amp** (only):

```
/* Set the amplitude Envelope (as previously defined). */
[aNote setPar:MK_ampEnv toEnvelope:anEnvelope];

/* The value of MK_amp sets the value when y is 1.0. */
[aNote setPar:MK_amp toDouble:0.15];
```

During synthesis, **aNote**'s amplitude is scaled according to the value of **MK_amp**, as depicted in Figure 3-5 (notice that the breakpoint values themselves don't change, only their interpretations are affected).

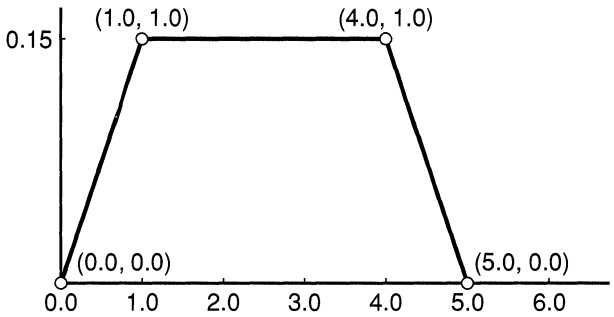

Figure 3-5. Scaled Amplitude Envelope

Technically, the interpretation of a particular value of y is calculated according to the following formula:

interpretedValue = (*scale** y) + *offset*

where *scale* is calculated as **MK_***attribute***1** – **MK_***attribute***0** and *offset* is simply the value of **MK_***attribute***0**.

The Stickpoint

When a SynthPatch receives a noteOn or noteDur, it starts processing the Note's Envelopes, reading their breakpoints one by one. The y values are scaled and offset as described above; the x values are taken as seconds (with modifications described in the next section). If the Note's duration (in seconds) is greater than the duration of the Envelope—in other words, if the Envelope runs out of breakpoints before the DSP is done synthesizing the Note—then the final y value is maintained for the balance of the Note.

To accommodate Notes of different lengths, the Envelope object lets you define one of its breakpoints as a *stickpoint*. When the SynthPatch reads an Envelope's stickpoint, it "sticks" until a noteOff arrives (or the declared duration of a noteDur elapses). The Envelope shown in the previous example, with its flat middle section, can easily be redefined using a stickpoint, as follows:

```
/* Instantiate arrays for x and y. */
double  xVals = {0.0, 1.0, 2.0};
double  yVals = {0.0, 1.0, 0.0};

/* Define the Envelope and set the MK_amp constant. */
[anEnvelope pointCount:3 xArray:xVals yArray:yVals];
[aNote setPar:MK_amp toDouble:0.15];

/* Set the Envelope's stickpoint. */
[anEnvelope setStickpoint:1];
```

The argument to **setStickpoint:** is a zero-based index into the Envelope's breakpoints. In the example, **anEnvelope**'s second breakpoint is declared to be the stickpoint. Figure 3-6 shows how the stickpoint allows the same Envelope to be applied to Notes (or Note phrases) with different durations. The stickpoint is shown as a solid circle; the sustained portion of the Envelope is indicated as a dashed line. The tempo in the illustration is assumed to be 60.0.

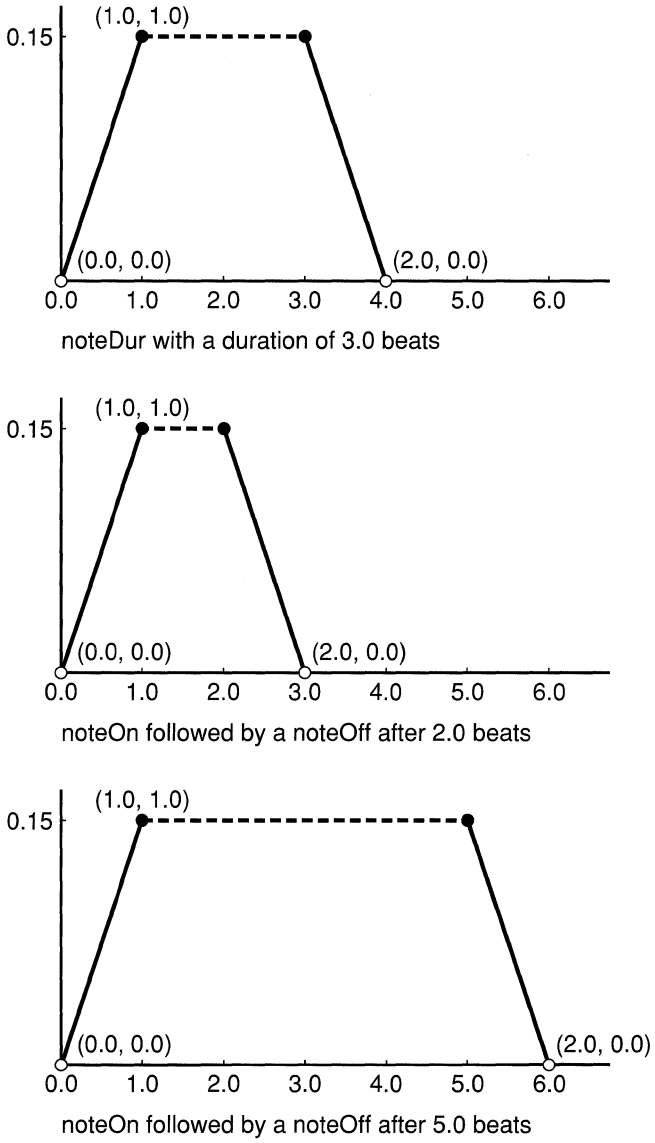

Figure 3-6. Envelope with Stickpoint

Notice that the duration between the end of the stickpoint segment and the following breakpoint is always the same (one second, as defined by the Envelope itself), regardless of the length of the Note.

Attack and Release

An Envelope object is divided into three parts: attack, sustain, and release. The stickpoint defines the sustain; the attack is the portion that comes before the stickpoint and the release is the portion that comes after it. An Envelope can have any number of breakpoints in its attack and release segments.

You can specify the absolute duration of the attack portion of an Envelope by setting the value of the **MK_***attribute***Att** parameter; the release is set through **MK_***attribute***Rel**. For example, the amplitude attack and release parameters are **MK_ampAtt** and **MK_ampRel**, respectively. The values of these parameters are taken as the number of seconds (given as **double**s) to spend in either segment, as illustrated in Figure 3-7. The x values of the breakpoints in the two segments are scaled within the given durations to maintain their defined proportions.

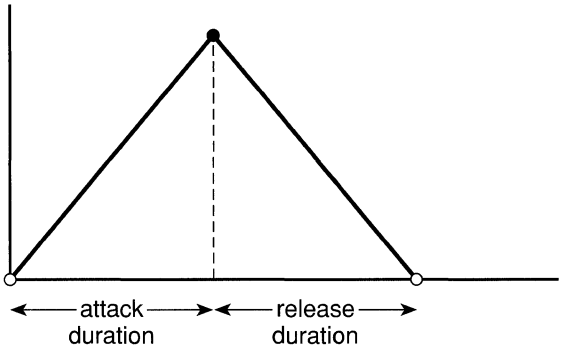

Figure 3-7. Attack and Release

Note: Since they're set as seconds (not beats), the attack and release times aren't affected by tempo.

Figure 3-8 shows the same (amplitude) Envelope used in the previous examples with various attack and release values.

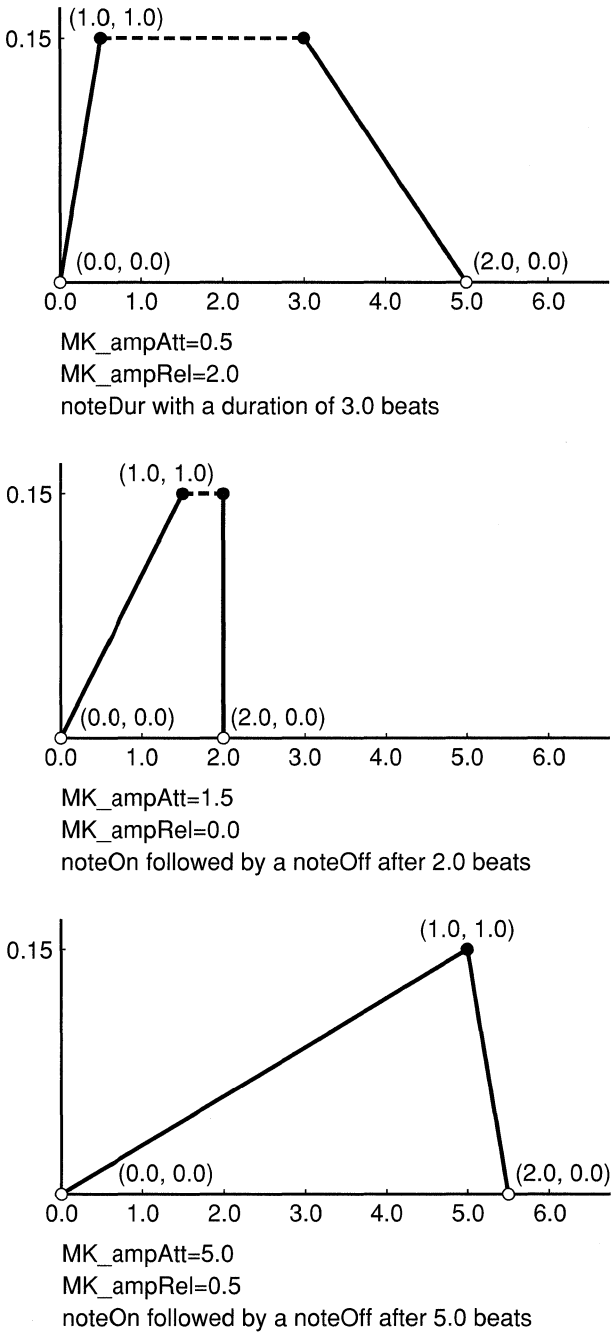

Figure 3-8. Envelope with Attack and Release

Figure 3-9 shows what happens when a noteOff arrives (or a noteDur expires) during the attack portion of the Envelope—in other words, before the stickpoint is reached. For this illustration, both **MK_ampAtt** and **MK_ampRel** are assumed to have values of 1.0.

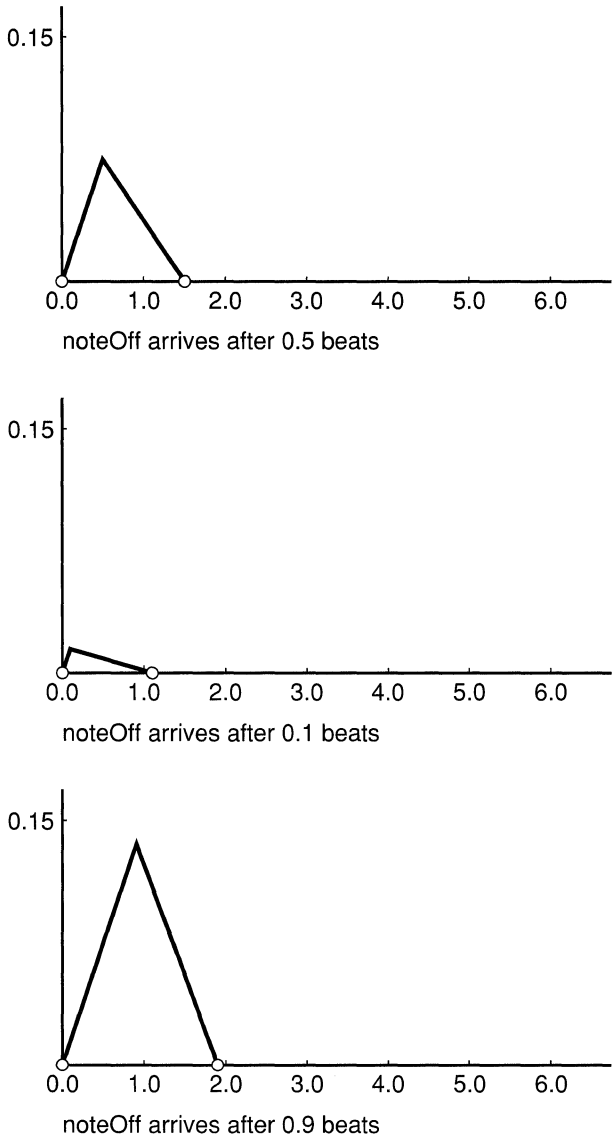

Figure 3-9. Early noteOff

When the noteOff arrives, the Envelope heads for the first breakpoint in the release (the first breakpoint after the stickpoint) from wherever it happens to be at the time. The release takes its full duration (as defined in the Envelope itself, or by **MK_ampRel**, if present) regardless of whether the noteOff arrives before or after the stickpoint is reached.

Modeling a Note without Sustain

Not every instrument can create a sustained tone; the amplitude envelope of a piano tone, for example, is characterized by a sharp rise and fall followed by a gradual but steady decay to quiescence. This is depicted in Figure 3-10.

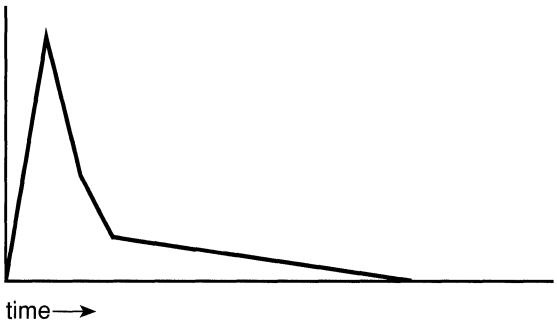

Figure 3-10. Piano Envelope

To simulate this sort of envelope shape, yet still accommodate Notes of any length, the Envelope object definition would look something like this:

```
double  xVals = {0.0, 0.05, 0.2, 0.5, 8.0, 8.15};
double  yVals = {0,0, 1.0,  0.5, 0.3, 0.0, 0.0};

[anEnvelope setPointCount:6 xArray:xVals yArray:yVals];

/* Set the stickpoint to breakpoint 4, xy:(8.0, 0.0). */
[anEnvelope setStickpoint:4];
```

The Envelope is depicted in Figure 3-11.

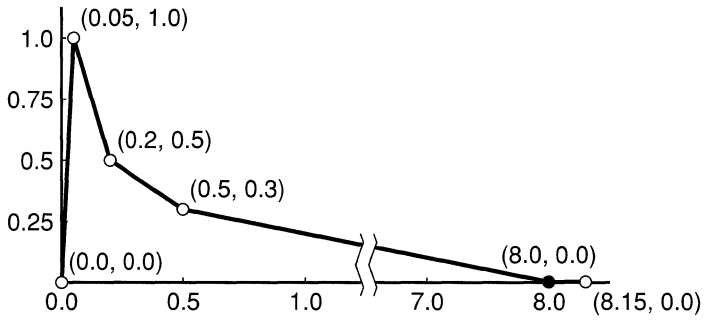

Figure 3-11. Simulated Piano Envelope

Notice that the Envelope's stickpoint is, curiously enough, set to a breakpoint that has a y value of 0.0. Equally curious is the release portion of the Envelope: a flat piece of seemingly useless real estate. However, consider the result of the two possible scenarios:

- The noteOff arrives after the stickpoint is reached. In this case, the synthesized sound has already decayed to an amplitude of 0.0. When the noteOff arrives, the release portion is indeed executed, but since the amplitude is already at 0.0, the release portion doesn't produce an audible effect.

- The noteOff arrives before the stickpoint is reached. The release portion is triggered, causing the amplitude to decay to 0.0 in 0.15 seconds.

Attack and release durations on a nonsustaining instrument are generally invariant, so you would rarely set the **MK_ampAtt** and **MK_ampRel** parameter.

Portamento

The Music Kit provides an additional parameter, **MK_portamento**, with which you can further manipulate your Envelopes' attack times. Like the **MK_***attribute***Att** parameters, **MK_portamento** takes a **double** value that's measured in seconds, but rather than affect the entire attack portion, it sets the duration between the first two breakpoints only. Also, as used by the SynthPatches provided by the Music kit, **MK_portamento** affects all the Envelopes in a Note—there aren't individual portamento parameters for amplitude, frequency, and so on. In a Note that contains a portamento value and one or more attack scalers, the attacks of the individual Envelopes are scaled before the value of **MK_portamento** is applied.

MK_portamento is provided so you can easily and quickly control, to some degree, the rearticulation of a Note's Envelopes. As such, it's only significant in a Note that rearticulates a phrase—it's ignored in a noteDur with no note tag, and has no immediate effect in a noteOn or a noteDur with a previously inactive note tag (although, in the latter case, the value of **MK_portamento** is stored in anticipation of subsequent rearticulations).

You should keep in mind that portamento is optional. It can be quite useful if you're modelling an instrument that has different attack characteristics depending on whether a Note is the beginning of a new phrase or part of a legato passage. For example, in some instruments, such as a horn, the attack of an initial musical note—in amplitude, frequency, and timbre—is more drawn out than in the subsequent notes of a phrase. To simulate such an instrument, it's convenient to use **MK_portamento** to affect all the Envelopes at once.

Smoothing

The previous examples have shown the lines that connect an Envelope's breakpoints as straight segments. In reality, as synthesized by the DSP, these segments follow an asymptotic curve. In an asymptotic curve, the target is never fully reached—the curve rises (or falls) in successively smaller steps as it approaches the target. However, there's a point in the curve where the target is perceived to have been attained. This point is controlled by the smoothing value.

By default, smoothing is 1.0, a value that's used to mean that the point at which the target is perceived to have been reached is equal to the difference between the x values in successive breakpoints; in other words, it takes the entire time between breakpoints to reach the target y value. Other values are, similarly, the ratio of curve duration to overall duration between a pair of breakpoints. For example, a smoothing of 0.5 means it takes half the time between a pair of breakpoints to (perceptually) complete the curve between the breakpoints' y values. A smoothing value in excess of 1.0 falls short of the target altogether (it takes longer than the allotted time to reach the target).

You can set the smoothing value for each breakpoint by defining the Envelope through an alternative method:

setPointCount:xArray:orSamplingPeriod:yArray:
smoothingArray:orDefaultSmoothing:

Smoothing is set as either an array of values (one for each breakpoint) passed to as the argument to the **smoothingArray:** keyword, or as a default value passed as the argument to the **orDefaultSmoothing:** keyword. The default smoothing is used only if the argument to **smoothingArray:** is NULL.

Smoothing is, admittedly, a somewhat elusive concept, best explained by illustration. Figure 3-12 shows the shape of an Envelope that uses various smoothing values between successive breakpoints. The values in parentheses are the x, y, and smoothing values for each breakpoint.

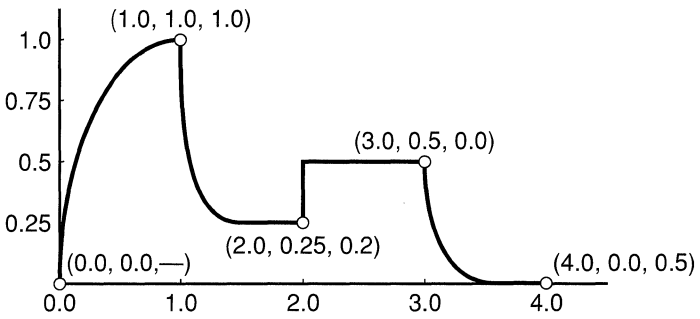

Figure 3-12. Envelope with Smoothing

Notice that the smoothing value is omitted from the first breakpoint. While a smoothing value must be supplied as the first element in the smoothing array (if you use the **smoothingArray:** keyword), this value is actually ignored when the Envelope is synthesized. This is because a breakpoint's smoothing value applies to the curve leading into it—the curve from the previous breakpoint to the current one. Since there isn't a previous breakpoint before the first one, the smoothing value for breakpoint 0 is thrown away.

Returning to the illustration, the smoothing value for the second breakpoint is 1.0; thus the curve leading from the first breakpoint into the second breakpoint takes up the entire duration between the two points. The smoothing value for the third break point is 0.2; the

curve leading into the third breakpoint reaches the target y value with time to spare. The fourth breakpoint has a smoothing of 0.0. This means that it takes no time to reach the target; the Envelope immediately leaps to the target y value. (Note that a smoothing of 0.0 is the only way to ensure that the asymptotic curve will, in truth, reach its target.) The final breakpoint smoothing value is 0.5. Accordingly, the curve reaches the target halfway between breakpoints.

While smoothing control is provided for completeness, most musical applications will be satisfied with the default smoothing provided by the **setPointCount:xArray:yArray:** method.

Sampling Period

You may have noticed that the Envelope definition method that brought you smoothing also introduced an alternate way to set an Envelope's x values. Rather than define x values in an array, you can also set them as a default increment by passing a (**double**) value to the method's **orSamplingPeriod:** keyword. Again, the default argument is used only if the array argument (in this case, the argument to **xArray:**) is NULL.

If you use a sampling period, the first x value is always 0.0. Successive x values are integer multiples of the sampling period value.

Discrete Value Lookup

The other way to use an Envelope is to retrieve a discrete value of y for a given x. This is performed in a single method, **lookupYForX:**, which takes a **double** argument that specifies an x value and returns the y value that corresponds to it. If the x value doesn't lie directly on a breakpoint, a linear interpolation between the y values of the surrounding breakpoints is performed to determine the appropriate value. For example, consider a simple, two-breakpoint Envelope defined as follows:

```
double  xVals = {0.0, 1.0};
double  yVals = {0.0, 2.0};
id anEnvelope = [[Envelope alloc] init];

[anEnvelope setPointCount:2 xArray:xVals yArray:yVals];
```

The message

```
double  interpY = [anEnvelope lookupYForX:0.5];
```

returns 1.0. The computation is illustrated in Figure 3-13.

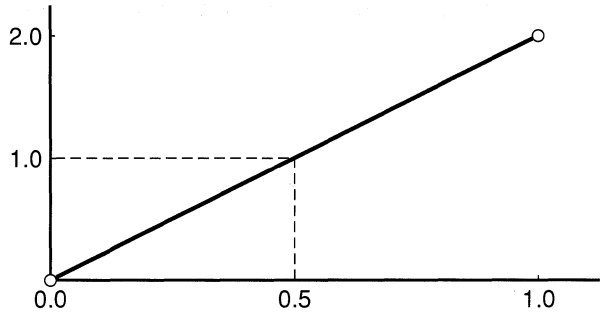

Figure 3-13. Linear Interpolation

With discrete-value lookup, the Envelope's stickpoint and smoothing values are ignored. Also, using an Envelope in this way doesn't require its presence in a Note object; thus, the parameters that help shape an Envelope used for DSP synthesis, such as **MK_amp**, **MK_ampAtt**, and **MK_ampRel**, aren't applied to discrete-value lookup.

If you request a discrete y value for an x that's out of bounds, the **lookupYForX:** method returns the y value of the breakpoint at the exceeded boundary. For example (using the same Envelope), the message

```
/* Specify an x for which there is no y. */
double  interpY = [anEnvelope lookupYForX:1.5];
```

returns 2.0, as it also would for any argument greater than 1.0. Similarly, any argument that's less than 0.0 would return (from this Envelope) 0.0.

Envelopes in Scorefile Format

When you write a Score to a scorefile, either through a message to the Score object or by using a ScorefileWriter in a performance, the Envelope objects that appear in the Notes in the Score are written out as a series of breakpoints in parentheses. The Envelope's stickpoint, if any, is indicated by the presence of a vertical bar following the so-designated breakpoint. The entire Envelope representation is enclosed by square brackets. For example:

```
[(0.0, 0.0, 1.0) (0.3, 1.0, 1.0) | (0.5, 0.0, 1.0)]
```

The three values inside the parentheses are, in order, the breakpoint's x, y, and smoothing values. The smoothing value is always written out—keep in mind that smoothing defaults to 1.0. In this example, the second point is the stickpoint.

If you give the Envelope a name before you write the scorefile, the Envelope is only written in this long form once; subsequent references (in the scorefile) are made to the Envelope object by its name. To name an Envelope, call the **MKNameObject()** C function:

```
MKNameObject("env1", anEnvelope);
```

It's a good idea to name your Envelope objects. This saves space in the scorefile and also makes processing the file during a performance more efficient.

A named Envelope appears in a scorefile statement as:

```
BEGIN;
. . .
noteOn ... ampEnv:envelope env1=[(0.0, 0.0, 1.0) ... ] ... ;
. . .
```

(The noteOn type is used here only as an example.) **envelope** is a keyword that declares the following name (**env1** in the example) to represent an Envelope.

If you write your own scorefile, you should be aware of the following:

- The x, y, and smoothing values can be expressions. Because of this, the three values must be separated by commas.
- The smoothing value is sticky; it applies to the breakpoint in which it appears and to all subsequent breakpoints in that Envelope declaration (until another smoothing value is encountered).
- If you don't specify a smoothing value, it defaults to 1.0.
- You should declare and set all your Envelope objects as **envelope** variables in the header of the scorefile. This makes reading the file more efficient.

For more on the scorefile format and ScoreFile language, see *Reference*.

The WaveTable Class

WaveTable objects are used exclusively in DSP synthesis to describe and create musical timbres. While WaveTable synthesis has limitations, it's a particularly easy and direct way to create a library of sounds. However, to intelligently define a WaveTable, you need to be familiar with a few basic concepts of musical acoustics. The section "What is Sound?" in Chapter 2 introduces some of these fundamentals; the cogent points from Chapter 2 are summarized and new concepts that pertain to WaveTables are introduced in the next section.

Summary of Musical Acoustics

When matter vibrates, a pressure disturbance is created in the surrounding air. The pressure disturbance travels as a wave to your ears and you hear a sound. A sound, particularly if it's a musical sound, can be characterized by its *waveform*, the shape of the air pressure's rise and fall. Waveforms created by musical instruments are generally periodic; this means that the pressure rises and falls in a cyclical pattern.

A periodic waveform has two basic characteristics, frequency and amplitude:

- The number of pattern repetitions, or *periods*, within a given amount of time determines a sound's frequency (pitch). Frequency is measured in *hertz* (abbreviated Hz), or cycles per second. For example, a musical sound with a period that repeats itself 440 times a second produces a tone at 440 Hz (A above middle C).

- The amplitude of a sound wave is the amount of energy in the air pressure disturbance. Amplitude is heard as loudness—the greater the energy, the louder the sound. (There are other factors that contribute to the loudness of a sound, but amplitude is generally the most important.) A number of different methods are used to measure loudness; of greatest use for musical purposes is to describe the loudness of a sound (or, as we shall see, a component of a sound) in comparison with another sound (or sound component).

A special waveform is the *sine wave*: Sine waves are important to musical acoustics in that they define the basic component used to describe musical sounds: Any periodic waveform can be broken down into one or more sine waves. The sine waves that make up a musical sound have frequencies that are (usually) integer multiples of a basic frequency called the *fundamental*. For example, if you pluck the B string on a guitar, it produces a fundamental frequency of approximately 494 Hz. However, the sound that's produced contains sine waves with frequencies that are integer multiples of 494:

494 * 1 = 494
494 * 2 = 988
494 * 3 = 1482
494 * 4 = 1976
and so on

Sine wave components, or *partials*, that are related to each other as integer multiples of a fundamental frequency are said to make up a *harmonic series*. A musical sound can also have partials that are inharmonically related to the fundamental; for example, the shimmer and pungency of a bell's tone is created by the abundance of inharmonic partials. However, as explained later, a WaveTable object is best suited to represent timbres that are created from a harmonic series.

The partials in a sound have amplitudes that are measured in relation to each other. For the guitar, the amplitude of each successive sine wave is generally less than that of the previous partial.

The fundamental (the partial at the fundamental frequency) needn't have the greatest amplitude of all the partials, nor must successive partials decrease in amplitude. Some instruments, such as the bassoon, have very little energy at the fundamental. Nonetheless, your ears decode the information in a harmonic series such that there is rarely confusion about the frequency of the fundamental; in other words, we almost always hear the fundamental as the frequency that's the least common denominator of the partials that make up the sound.

Constructing a WaveTable

A WaveTable object represents one complete period of a musical waveform. There are two ways to create a WaveTable, as embodied by WaveTable's subclasses, Partials and Samples:

- With a Partials object, you can define a WaveTable by specifying the individual partials that make up the waveform.

- A Samples object represents a waveform as a series of sound samples. It uses a Sound object (defined by the Sound Kit) as its data.

The Partials Class

You define a Partials object by supplying the frequency, amplitude, and phase information for a series of partials. This is done through the method **setPartialCount:freqRatios:ampRatios:phases:orDefaultPhase:**. The first argument is the number of partials; the next three arguments are arrays of **double** data that specify the frequency ratios, amplitude ratios, and initial phases of the partials, respectively. You can also set the phase as a constant by passing a **double** as the argument to the **orDefaultPhase:** keyword. In this case, you must pass NULL as the argument to **phases:**.

In the following example, a waveform is created from a series of partials that are integer multiples of a fundamental frequency; the partials decrease in amplitude as they increase in frequency.

```
/* Create the Partials object. */
id      aPartials = [Partials new];
double  freqs[] = {1.0,2.0,3.0,4.0,5.0,6.0};
double  amps[] =  {1.0,0.5,0.25,0.12,0.06,0.03};

/* Fill the object with data. */
[aPartials setPartialCount:6
           freqRatios:freqs
           ampRatios:amps
           phases:NULL
           orDefaultPhase:0.0];
```

Note: Phase is generally unimportant in creating musical timbres, although it can drastically affect a waveform that's used as a low-frequency control signal, such as vibrato.

The frequencies in a Partials object are specified as ratios, or multiples, of a fundamental frequency (the fundamental is represented by a frequency ratio of 1.0). The actual (fundamental) frequency of the waveform created from a Partials depends on the how the object is used by the SynthPatch that synthesizes it. In general, the waveform is "transposed" to produce the frequency specified in a Note's frequency parameter, **MK_freq**. Similarly, the amplitude of each partial is relative to the value of another parameter, usually **MK_amp**.

The Samples Class

The Samples class lets you create a WaveTable through association with a Sound object (an instance of the Sound Kit's Sound class). This is done by invoking the Samples' **setSound:** method:

```
/* You must import the Sound Kit's header file. */
#import <soundkit/soundkit.h>

. . .
    /* Create a Samples object and a Sound object. */
    id  aSamples = [Samples new];
    id  aSound = [Sound new];

    /* Fill the Sound with data. */
    . . .

    /* Associate the Sound with the Samples. */
    [aSamples setSound:aSound];
```

A copy of the Sound object is created and stored in the Samples object when **setSound:** is invoked, so it's important that you fill the Sound with data before invoking the method. Chapter 2 describes ways to create Sound data.

You can also associate a Samples with a Sound by reading a soundfile, through the **readSoundfile:** method. The Samples object creates a Sound object and then reads the soundfile by sending **newFromSoundfile:** to the Sound. The argument is a UNIX pathname that must include the soundfile-identifying ".snd" extension.

A Samples' Sound object must contain one channel of 16-bit linear, sampled data. The length of the data (the number of samples) must be a power of two.

Setting a WaveTable in a Note

To hear the timbre represented by a WaveTable object, you set the WaveTable as a parameter of a Note and then play the Note using a SynthPatch that recognizes the parameter. Most of the Music Kit SynthPatches recognize the **MK_waveform** parameter:

```
/* Create a Note. */
id  aNote = [Note new];

/* Set its MK_waveform value. */
[aNote setPar:MK_waveform toWaveTable:aPartials];
```

In this example, the value of **MK_waveform** is set to the previously defined Partials object, **aPartials**. The manner in which the Partials object is used during synthesis depends on the SynthPatch to which the Note is sent.

Grouping Notes

The Part and Score classes are designed to group Note objects. As their names imply, a Part represents a series of Notes that are realized on the same Instrument; a Score represents all or part of a composition and consists of some number of Parts. For storage, a Score can be written to the disk as a scorefile or a midifile.

The first part of this section explains the methods used to add Notes to Parts, Parts to Scores, and to write a Score as a file. The second part, "Retrieving Scores, Parts, and Notes," presents methods and techniques for retrieving, querying, and further manipulating Parts, Scores, and the Notes they contain. Finally, the special Notes called Part info and Score info are described.

Constructing a Score

Adding a Note to a Part

A Part is a time tag sorted collection of Note objects. A Part can contain any number of Notes. While the Notes within a Part are sorted by time tag value, every Note in the Part needn't have a unique time tag; a Part can represent simultaneous (and otherwise overlapping) Notes. However, a single Note can only belong to one Part at a time.

There are three methods for adding a Note to a Part:

- **addToPart:**, a Note method
- **addNote:**, a Part method
- **addNoteCopy:**, also a Part method

The first two are functionally equivalent, allowing you to add a Note to a Part by messaging either object:

```
[myNote addToPart:myPart];
/* is the same as */
[myPart addNote:myNote];
```

Since a Note object can only belong to one Part at a time, when you add a Note to a Part, it's first removed from the Part that it's currently a member of, if any. Both **addNote:** and **addToPart:** return the **id** of the Note's old Part (the Part from which the Note was removed).

addNoteCopy: creates (and returns) a new Note object as a copy of its argument and adds the copy to the receiver. The argument Note itself isn't removed from its Part. The method creates the Note copy by invoking Note's copy method.

You can continue to modify a Note (setting its parameter values, note type, time tag, and so on) after it has been added to a Part object. A Part sorts its Notes automatically, so you can

add them in any order—you don't have to add them by order of their time tag values. If you change the time tag of a Note after you add it to a Part, the Note is automatically repositioned in the Part.

Methods for adding a collection of Notes to a Part—for instance, adding the contents of one Part to another Part—are described below, in the section "Adding and Removing Groups of Notes."

Naming a Part

A Part object has a print name that's used when writing it to a scorefile (or, more accurately, when the Score to which the Part belongs is written to a scorefile). The **MKNameObject()** function is used to name a Part; like all Music Kit names, a Part's name is case-sensitive and consists of a letter followed by a string of alphanumeric characters:

```
MKNameObject("Solo", aPart);
```

To retrieve a Part's name, call the C function **MKGetObjectName()**:

```
char *aName = MKGetObjectName(aPart);
```

Adding a Part to a Score

To add a Part to a Score, invoke one of these methods:

- **addToScore:**, a Part method
- **addPart:**, a Score method

A Score can contain any number of Parts; a Part can belong to only one Score at a time. Both of the Part-adding methods first remove the Part from its present Score, if any.

Just as you can modify a Note after it has been added to a Part, you can continue to modify a Part (by adding and removing Notes, for example) after it has been added to a Score.

Writing a Score to a File

You write a scorefile by sending one of the following messages to a Score object:

- **writeScorefile:** takes a file name (**char ***) argument.
- **writeScorefileStream:** takes a stream pointer (NXStream *) argument.

The first of these opens and closes the file for you. Every time you send the **writeScorefile:** message, the named file is overwritten. By convention, scorefiles are given a ".score" extension.

writeScorefileStream: expects a stream pointer that's already open for writing. The method leaves the stream open after it returns, allowing you to write additional, application-specific information to the end of the file. It's left to your application to close the stream.

Rather than write an entire Score to a file, you can specify a particular section by sending the **writeScorefile:firstTimeTag:lastTimeTag:timeShift:** message (a similar method, with an initial keyword of **writeScorefileStream:**, is provided for writing a section of a Score to a stream). For example:

```
[myScore writeScorefile:"Adagio.score" firstTimeTag:3.5
        lastTimeTag:10.2 timeShift:0.0]
```

Here, only those Notes with time tag values between 3.5 and 10.2, inclusive, are written to the file. As a convenience, you can use the constant MK_ENDOFTIME as the **lastTimeTag:** argument to write from some position in the Score to its end:

```
[myScore writeScorefile:mySfile firstTimeTag:3.5
        lastTimeTag:MK_ENDOFTIME timeShift:0.0]
```

To write from the beginning of the Score, you specify 0.0 as the **firstTimeTag:** argument. The **timeShift:** takes a value that specifies the number of beats by which the Notes are time-shifted as they're represented in the file. Only the file representation is affected by this value; in other words, the Notes themselves aren't time-shifted (their time tags aren't affected).

For each of the scorefile-writing methods, there is an analogous method that writes a Standard MIDI file.

Retrieving Scores, Parts, and Notes

Reading a File

You can fill a Score with information simply by reading a scorefile (or MIDI file; for brevity, scorefiles are used exclusively in the examples). The methods provided for reading a scorefile are analogous to those for writing:

- **readScorefile:** takes a file name argument.
- **readScorefileStream:** takes an NXStream pointer argument.

When you read a scorefile into a Score object, Part and Note objects are automatically created to accommodate the information in the file.

You can specify a section of the scorefile and a time shift on that section by adding the **firstTimeTag:lastTimeTag:timeShift:** keywords. For example:

```
[myScore readScorefile:"Adagio.score"
    firstTimeTag:6.5 lastTimeTag:MK_ENDOFTIME timeShift:-6.5]
```

Notes represented in the file that have time tag values greater than or equal to 6.5 are read into the Score. Within the Score, each Note's time tag is shifted ahead in time (toward the chronological beginning of a Part) by 6.5 beats.

Finding a Part in a Score

The complete set of a Score's Parts, regardless of how the Score was created, can be retrieved through Score's **part** method. The method returns the Parts in a List. The Score method **partCount** gives the number of Parts that it contains.

Retrieving a Note from a Part

There are a number of ways to retrieve a Note from a Part. One way is to access the Note by its time tag value through one of the following Part methods:

- The **atTime:** method returns the first Note object with the specified time tag value.
- **atOrAfterTime:** returns the first Note with a time tag greater than or equal to the argument.

Both methods take a **double** argument and they both return **nil** if they can't find an appropriate Note. The following example illustrates the difference between the two methods:

```
/* Create a Part, two Notes, and an id for return values. */
id  aPart = [Part new],
    aNote = [Note newSetTimeTag:2.0],
    bNote = [Note newSetTimeTag:3.0],
    returnNote;

/* Add the Notes to the Part. */
[aPart addNote:bNote];
[aPart addNote:aNote];

/* Retrieve the Note at time 1.5.  (Returns nil; no such Note.) */
returnNote = [aPart atTime:1.5];

/* Retrieve the Note at or after 1.5.  (Returns aNote.) */
returnNote = [aPart atOrAfterTime:1.5];
```

You can also retrieve a Note by its ordinal position within its Part by sending the message **nth:** to the Part object. It takes an integer argument *n* and returns the *n*th Note, zero-based, in the Part. Using the same Part object from the previous example, the message

```
[aPart nth:1]
```

returns **bNote**, the second Note in the Part. Recall that within a Part, Notes are ordered by their time tag values; the order of Notes with equivalent time tags reflects the order in which they were added to the Part.

The time tag and the ordinal methods of retrieving Notes are combined in the methods **atTime:nth:** and **atOrAfterTime:nth:**. The first of these methods returns the *n*th Note with the specified time tag; through this method you can retrieve a particular Note in a chord. The **atOrAfterTime:nth:** method returns the *n*th Note with a time tag equal to or greater than the first argument.

The **next:** method also retrieves Notes based on ordinal position: It returns the Note that immediately follows the Note given as the argument. **next:** can be used to access each Note in a Part in turn:

```
/* Initially set aNote to the first Note in the Part. */
id  aNote = [aPart nth:0];

/* Access each Note in the Part and change it to MK_mute. */
while (aNote = [aPart next:aNote])
    [aNote setNoteType:MK_mute];
```

next: returns **nil** if there is no next Note, or if the argument isn't a member of the Part. A more efficient way of accessing each Note in a Part is to create a List of the Notes in a Part and step down the List. The previous example can be rewritten as follows:

```
/* Create a Sequence over the Part's Notes. */
id  notes = [aPart notes];
int noteCount = [aPart noteCount];
int i;
/* Access each Note in the Sequence and change its note type. */
for (i=0; i<noteCount; i++)
    [[notes objectAt:i] setNoteType:MK_mute];
```

Removing a Note from a Part

The Part class defines two methods for removing individual Notes from a Part object:

- **removeNote:** removes the Note object specified in the argument.
- **removeNotes:** removes all Notes common to the receiver and the List specified in the argument.

You can also remove a Note from its Part by sending the **removeFromPart:** message to the Note object. The Part object from which the Note was removed is returned.

Adding and Removing Groups of Notes

The Part class defines methods that allow you to add and remove Notes as a collection. There are two methods for adding a collection of Notes:

- **addNotes:timeShift:**
- **addNoteCopies:timeShift:**

The first argument is a List of Notes. The second argument is an optional value in beats (a **double**) that's used to offset each Note's time tag. The **addNotes:timeShift:** method removes each Note in the collection from the Part that it's currently a member of before adding it to the receiver. The **addNoteCopies:timeShift:** method adds copies of each Note in the List.

To remove a List of Notes, you can invoke the method **removeNotes:**, which takes a List of Notes and removes each one from the receiver.

Finally, you can remove all the Notes from a Part through the **empty** method.

Chapter 4
Music Synthesis

Chapter 4
Music Synthesis

There are four levels of Music Kit classes involved in synthesizing music on the DSP:

- At the top level is the SynthInstrument class; an inheritor from Instrument, it provides an interface between the Music Kit performance mechanism and DSP synthesis. A SynthInstrument distributes Notes that it receives during a performance to individual SynthPatch objects that control the actual synthesis.

- SynthPatch, an abstract class, embodies the principles of music synthesis. Each subclass of SynthPatch represents a particular synthesis strategy; each SynthPatch object synthesizes one Note at a time, using the strategy defined by its class.

- The UnitGenerator class is also abstract; each of its subclasses represents a signal processing function that runs on the DSP. These are the basic building blocks of music synthesis that a SynthPatch subclass configures to define its synthesis strategy. At the same level as UnitGenerator is the SynthData class. Instances of SynthData represent portions of DSP memory that hold synthesis data such as wavetables and delay memory. In addition, SynthData objects are used to transmit data between UnitGenerator objects.

- The Orchestra represents the DSP itself. It manages the allocation of DSP resources and controls communication between the DSP and the host. Requests for use of the DSP are always made through messages to the Orchestra; this includes requests made by a SynthInstrument for more SynthPatches, and those of a SynthPatch for UnitGenerators and SynthData objects.

The SynthInstrument class is described in the next chapter, "Music Performance." The rest of this chapter is devoted to the other classes listed above.

The Orchestra Class

The Orchestra class manages DSP resources used in music synthesis. Each instance of Orchestra represents a single DSP; in the basic NeXT configuration, there's only one DSP so you create only one Orchestra object.

The methods defined by the Orchestra class let you manage a DSP by allocating portions of its memory for specific synthesis modules and by setting its processing characteristics.

You can allocate entire SynthPatches or individual UnitGenerator and SynthData objects through Orchestra methods. Primary among these are:

- **allocSynthPatch:** allocates an instance of the given SynthPatch subclass.
- **allocUnitGenerator:** does the same for a UnitGenerator subclass.
- **allocSynthData:length:** allocates a portion of DSP memory of a given length.

Keep in mind, however, that similar methods defined in other classes—specifically, the SynthPatch allocation methods defined in SynthInstrument, and the UnitGenerator and SynthData allocation methods defined in SynthPatch—are built upon and designed to usurp the allocation methods defined by Orchestra. You only to need to allocate synthesis objects directly if you want to assemble sound-making modules at a low level.

Sharing Allocations

To avoid creating duplicate synthesis modules on the DSP, each instance of Orchestra maintains a shared object table. Objects on the table are SynthPatches, SynthDatas, and UnitGenerators; each is indexed by some other object that represents the shared object. For example, the OscgafUG UnitGenerator (a family of oscillators) lets you specify its waveform-generating wave table as a Partials object (you can also set it as a Samples object; for the purposes of this example we only consider the Partials case). When its wave table is set through the **setTable:length:** method, the oscillator allocates a SynthData object from the Orchestra to represent the DSP memory that will hold the waveform data computed from the Partials. It also places the SynthData on the shared object table using the Partials as an index by sending the message

```
[Orchestra installSharedSynthData:theSynthData for:thePartials];
```

If another oscillator's wave table is set as the same Partials object, the already allocated SynthData can be returned by sending the message

```
id aSynthData = [Orchestra sharedObjectFor:thePartials];
```

The method **installSharedObject:for:** is provided for installing SynthPatches and UnitGenerators.

Orchestra's Device Status

Before you can do anything with an Orchestra—particularly, before you can allocate synthesis objects—you must create and open it. As usual, creation is done through the **alloc** and **init** methods; to open an Orchestra, you send it the **open** message. This provides a channel of communication with the DSP that the Orchestra represents. The DSP can be opened by only one application at a time, so you should always check the value returned by open; the method returns **nil** if the DSP couldn't be opened.

Once you've allocated the objects that you want, either through the methods described above or through those defined by SynthInstrument and SynthPatch, you can start the synthesis by sending the **run** message to the Orchestra. The **stop** method halts synthesis and **close** breaks communication with the DSP. These methods change the Orchestra's status, which is always one of the following MKDeviceStatus values:

Status	Meaning
MK_devOpen	The Orchestra is open but not running.
MK_devRunning	The Orchestra is open and running.
MK_devStopped	The Orchestra has been running but is now stopped.
MK_devClosed	The Orchestra is closed.

You can query an Orchestra's status through the **deviceStatus** method.

Orchestra Output

When the Orchestra is running it produces a stream of samples that, by default, are sent to the stereo digital to analog converter (DAC), which converts the samples into an audio signal. Instead, you can tell the Orchestra to write the samples to a soundfile by invoking the method **setOutputSoundfile:** (you must set the soundfile before sending **run** to the Orchestra).

The Music Kit SynthPatch Subclasses

The Music Kit provides a number of SynthPatch subclasses, instances of which you can use in your application. SynthPatch objects are "data-driven"; during a performance, they're sent Note objects (through methods described in the next section) from which they pluck the parameters of interest. Thus, in order to use a SynthPatch, you must know not only what sort of synthesis it embodies, but which parameters it expects to see in the Notes it receives. A table of the Music Kit parameters organized by SynthPatch synthesis technique is given in Appendix B, "Music Tables." Below, the Music Kit SynthPatch subclasses are listed and briefly described.

Pluck

Pluck employs physical modelling to synthesize the sound of a plucked string. The real-world mechanics of a plectrum plucking a string are replaced on the DSP by a burst of noise filling a length of delay memory. The delay memory is looped and filtered, causing the initial noise burst to gradually become more harmonic as the spectral energy subsides towards the fundamental, emulating the strike-and-fade characteristics of a real plucked string.

Wavetable Synthesis

Wavetable synthesis is a technique in which a length of memory is filled with one or more periods of a waveform; the memory is then looped during playback to produce a continuous signal. While wavetable synthesis is extremely easy to use—no messy formulas are needed to create a musical timbre—it's somewhat limited to the extent that you don't have control over the individual elements that create the timbre (which you do with techniques such as frequency modulation).

The Music Kit's wavetable synthesis SynthPatches use single-period wavetables; they are:

SynthPatch	Description
Wave1	One wavetable
Wave1v	One wavetable with vibrato
Wave1i	One wavetable with frequency interpolation
Wave1vi	One wavetable with vibrato and interpolation
DBWave1v	One database wavetable with vibrato
DBWave1vi	One database wavetable with vibrato and interpolation
DBWave2vi	Two database wavetables with vibrato and interpolation

The database wavetable SynthPatches access the Music Kit's WaveTable Database, a library of predefined timbres. These are listed in the "Music Tables" appendix.

Frequency Modulation

In frequency modulation (fm) synthesis, the output of one oscillator controls the frequency of another oscillator. If the frequency of the controlling oscillator (or "modulator") is subaudio, the tone produced by the controlled oscillator (or "carrier") will exhibit vibrato. However, as the modulator's frequency is increased, the carrier's vibrato also increases until the individual undulation become indistinguishable and sidebands, or reflections of the modulator's frequency around the carrier's frequency, appear. If the oscillators are producing sine waves with identical or harmonically-related frequencies, the sidebands produce a harmonic series. As the oscillators' waveforms become more complex, the sidebands become more numerous, but they may still produce a harmonic series. However, if the oscillators' frequencies aren't harmonically related, the result can be a clangorous mess (which is good if you're trying to make bell sounds, one of the more popular and occasionally unavoidable results of fm synthesis).

SynthPatch	Description
Fm1	Simple (one-modulator) fm
Fm1i	Simple fm with frequency interpolation
Fm1v	Simple fm with vibrato
Fm1vi	Simple fm with interpolation and vibrato
Fm2cvi	Cascade fm
Fm2cnvi	Cascade fm with random modulation (noise) on the modulators
Fm2pvi	Parallel fm
Fm2pnvi	Parallel fm with noise

Building a SynthPatch

The SynthPatch class is abstract; each subclass of SynthPatch describes a unique strategy for creating a musical sound. It does this by implementing methods that provide two things:

- A *patch* specification. A patch is a configuration of DSP synthesis elements.
- A scheme for playing the patch. This consists of defining the conditions in which the patch is turned on and off and how Note parameters are used to control it while it's running.

Designing a patch is actually quite simple: The Music Kit provides an object-oriented interface to the DSP, thus protecting the SynthPatch designer from the rigors of programming directly in DSP56000 assembly code. While concern for efficiency makes some knowledge of DSP memory organization necessary, SynthPatch design makes greater demands of your imagination in creating new sound-making schemes than of your ability to examine and grasp the small print of signal processing.

The Music Kit defines a number of conventions for controlling a patch. Most of these conventions are manifested as methods that are declared as subclass responsibilities by the SynthPatch class. Other conventions are given as general guidelines that should be followed to maintain consistency with the SynthPatch subclasses provided by the Music Kit.

A Simple SynthPatch

This section describes, by example, the basic steps for creating a SynthPatch subclass. The example SynthPatch produces a single sine wave (with a settable frequency, amplitude, and bearing) for each Note it receives. The design is broken into two parts: designing the patch specification, and playing the patch. While the methodology shown for playing the patch introduces a number of SynthPatch design conventions, it lacks some important features that enhance musical flexibility. These features are shown in the more complex SynthPatch design demonstrated in the section "A Better SynthPatch," later in this chapter.

Designing the Patch Specification

Every SynthPatch contains a recipe for creating a patch. The ingredients of the patch are UnitGenerator and SynthData objects (collectively referred to as *synthElements*):

- Each UnitGenerator subclass represents a specific signal processing function. The Music Kit supplies a number of UnitGenerator subclasses that perform functions such as creating and combining signals, filtering, and adding the finished product to the output sample stream.

- SynthData objects represent data. These objects can be used for downloading information to the DSP; for example, a WaveTable is represented on the DSP as a SynthData. Another important use of a SynthData is to provide a location through which one UnitGenerator can send data to another UnitGenerator. This type of SynthData object is called a *patchpoint*.

The list of synthElement specifications and instructions for connecting these elements to each other are encapsulated in a PatchTemplate object. Every SynthPatch subclass creates at least one PatchTemplate—most create only one. A PatchTemplate is created and returned by the SynthPatch class method **patchTemplateFor:**, a subclass responsibility. In the following example, a single sine wave SynthPatch is declared and its **patchTemplateFor:** method is implemented:

```
/* The following files must be imported. */
#import <musickit/musickit.h>
#import <musickit/unitgenerators.h>
#import <objc/List.h>

/* We call our simple SynthPatch 'Simplicity'. */
@implementation Simplicity

/* A static integer is created for each synthElement. */
static int  osc,              /* sine wave UnitGenerator */
            stereoOut,        /* sound output UnitGenerator */
            outPatchpoint;    /* SynthData */

+ patchTemplateFor:aNote
/* The argument is ignored in this implementation. */
{
    /*
     * Step 1:  Create an instance of the PatchTemplate class.  This
     * method is automatically invoked each time the SynthPatch
     * receives a Note.  However, the PatchTemplate should only be
     * created the first time this method is invoked.  If the object
     * has already been created, it's immediately returned.
     */
    static id theTemplate = nil;
    if (theTemplate)
        return theTemplate;
    theTemplate = [PatchTemplate new];

    /*
     * Step 2:  Add synthElement specifications to the PatchTemplate.
     * The first two are UnitGenerators; the last is a SynthData
     * that's used as a patchpoint.
     */
    osc = [theTemplate addUnitGenerator:[OscgUGxy class]];
    stereoOut = [theTemplate addUnitGenerator:[Out2sumUGx class]];
    outPatchpoint = [theTemplate addPatchpoint:MK_xPatch];
```

```
    /* Step 3:  Specify the connections between synthElements. */
    [theTemplate to:osc sel:@selector(setOutput:) arg:outPatchpoint];

    /* Always return the PatchTemplate. */
    return theTemplate;
}
```

After creating the PatchTemplate instance (step 1 in the example), synthElement specifications are added to it (step 2) using methods defined by the PatchTemplate class. There are three basic methods to do this (a fourth method will be discussed later):

- addUnitGenerator:
- addSynthData:length:
- addPatchpoint:

Each of these methods returns an integer value that's used as an index to the added synthElement. Subsequent references to the synthElements are always made through these indices. Since all instances of a particular SynthPatch subclass use the same set of indices, the variables that store the values returned by these methods must be declared statically and be made global to the entire class.

Finally, instructions for connecting the synthElements are specified by invoking PatchTemplate's **to:sel:arg:** method (step 3). The arguments to this method are the receiver, selector, and argument, respectively, of a message that will be sent when a SynthPatch instance is created. Simplicity specifies a single connection:

```
[theTemplate to:osc sel:@selector(setOutput:)
    arg:outPatchpoint]
```

When an instance of Simplicity is created and played, the output of the UnitGenerator indexed by **osc** will be set to the SynthData indexed by **outPatchpoint**.

Simplicity's SynthElements Examined

To understand the synthElements used in the example, you need to be familiar with a simple detail of DSP memory organization. DSP memory is divided into three sections, x, y, and p. x and y memory are used for data; p memory is used for program code. Thus, SynthData objects represent data in either x or y memory, while UnitGenerators represent DSP functions that always reside in p memory. This is illustrated in Figure 4-1.

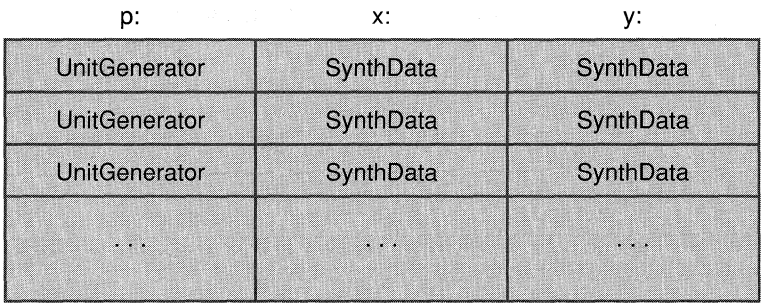

Figure 4-1. DSP Memory Division

The Music Kit further divides DSP memory into logical segments, represented as integer constants. In designing a SynthPatch, you only need to be concerned with four of these segments:

- **MK_xPatch** is used for patchpoints in x memory.
- **MK_yPatch** is for patchpoints in y memory.
- **MK_xData** is for non-patchpoint SynthData objects in x memory.
- **MK_yData** is for non-patchpoint SynthData objects in y memory.

A SynthData object is specified by its segment. For example, Simplicity's patch contains a single SynthData (a patchpoint) that resides in x memory, as set in the message

```
/* Add an x segment patchpoint. */
[theTemplate addPatchpoint:MK_xPatch]
```

UnitGenerators also refer to x and y memory in order to properly read and write data. Recall the first UnitGenerator added to Simplicity's patch:

```
[theTemplate addUnitGenerator:[OscgUGxy class]]
```

The "x" and "y" in the class name OscgUGxy refer to x and y memory spaces, respectively. OscgUGxy is a simple UnitGenerator that has a single input for reading data and a single output for writing data. The order of these data spigots, or *memory arguments*, is given in the UnitGenerator name as output followed by input. Thus, OscgUGxy's writes data to x memory (output) and reads it from y memory (input). The Music Kit provides a class for each memory permutation: OscgUGyy, OscgUGyx, OscgUGxy, and OscgUGxx. These are called *leaf classes* of the *master class* OscgUG. Aside from the differing memory references, the leaf classes are exactly the same. Every UnitGenerator function provided by the Music Kit is similarly organized into a master class and a complete set of leaf classes.

Note: When describing a subclass of UnitGenerator, it's convenient to refer to the master class rather than a specific leaf class. Furthermore, the "UG" (which stands for "UnitGenerator") is often dropped from the master class name.

The Oscg family of UnitGenerators provides a general oscillator function (the "g" in Oscg stands for "general"). An oscillator is a module that creates a signal by cycling over a table of values, called a *lookup table*, that represents a single period of a waveform. In a general oscillator, the lookup table isn't part of the UnitGenerator. You can supply the oscillator with a lookup table by using a WaveTable object (this will be demonstrated in a subsequent example). Alternatively, the oscillator can use the built-in sineROM, a read-only section of y memory that contains a single period of a sine wave. Simplicity's OscgUGxy does the latter: It reads the sineRom, therefore its input *must* read from y memory. In the example, the connection between the sineROM and OscgUGxy is made by default—it needn't be specified through the **to:sel:arg:** method.

One of the conventions of designing a patch is to balance, as much as possible, the use of x and y memory. Since Simplicity's oscillator must read from y memory in order to read the sineROM, its output is set to x memory. So of the four Oscg leaf classes, OscgUGxy is chosen.

The other UnitGenerator in Simplicity's patch, Out2sumUGx, is a special UnitGenerator that adds a stream of (two-channel) sample data to the stereo output sample stream. The single memory argument (the "x" in Out2sumUGx) is the UnitGenerator's input. Simplicity uses the Out2sumUGx leaf class so the UnitGenerator can read the MK_xPatch patchpoint that's written to by OscgUGxy. Figure 4-2 shows a diagram of the complete patch, superimposed on the DSP memory layout.

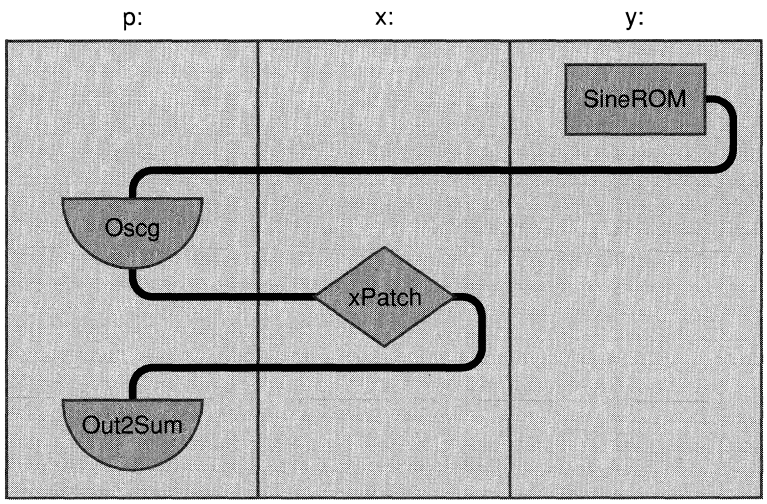

Figure 4-2. Simplicity's Patch

Notice that Figure 4-2 shows a connection between the patchpoint and the input of Out2Sum, a connection that isn't specified in Simplicity's PatchTemplate. By convention, the connection to the output UnitGenerator is implemented in a method that's invoked when the SynthPatch receives a noteOn. This method, called **noteOnSelf:**, is examined in the next section.

Note: The illustration in Figure 4-2 introduces schematic conventions that will be used throughout this section:

- UnitGenerators are drawn as half-circles (oscillators) or as inverted triangles (everything else).
- A UnitGenerator's inputs are at the top of the icon, its outputs are on the bottom.
- Patchpoints are drawn as diamonds. Other SynthData objects, including the predefined SynthData that represents the DSP sineROM, are rectangles.
- Data written to a SynthData arrives at the left side of the icon. Data is read from the right.

It's often convenient to represent a patch without including the patchpoints and without superimposing the schematic on the DSP memory diagram. Figure 4-3 shows Simplicity's patch in an abbreviated form. A SynthData's memory space is indicated by an x or y inside the icon. The spaces from which and to which a UnitGenerator reads and writes data is similarly indicated just to the right of each input and output.

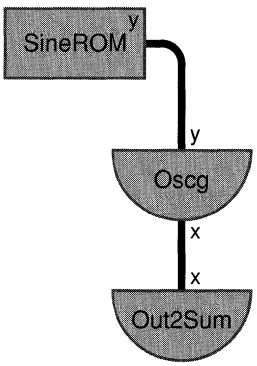

Figure 4-3. Conventional Patch Diagram

Playing the Patch

Keep in mind that a SynthPatch object is ordinarily created and controlled by an instance of SynthInstrument. During a Music Kit performance, the SynthInstrument distributes the Notes it receives to the various SynthPatch objects that it controls through the methods **noteOn:**, **noteUpdate:**, and **noteOff:** (the SynthInstrument treats a noteDur as a noteOn and manufactures a noteOff to balance it; also, the SynthInstrument normally suppresses mutes). The design of a SynthPatch subclass must include a methodology to control the patch in response to these messages. This is done by implementing the following methods:

- noteOnSelf:
- noteUpdateSelf:
- noteOffSelf:
- noteEndSelf

As their names imply, the first three of these methods are invoked automatically when the SynthPatch receives a **noteOn:**, **noteUpdate:**, or **noteOff:** message, respectively. **noteEndSelf** is automatically invoked when the Note is completely finished and is provided to accommodate the release portion of the SynthPatch's Envelopes.

Note: While these four methods aren't subclass responsibilities, the default implementations provided by the SynthPatch class do nothing. Thus, if you don't provide an implementation of, for example, **noteUpdateSelf:**, your SynthPatch won't respond to noteUpdates.

The noteOnSelf: Method

Simplicity, our example SynthPatch, implements **noteOnSelf:** as follows:

```
- noteOnSelf:aNote
{
    /* Step 1:  Read the parameters in the Note and apply them to the
       patch. */

    [self applyParameters:aNote];

    /*
     * Step 2:  Turn on the patch by connecting the Out2sumUGx object
     * to the patchpoint and sending the run message to all the
     * synthElements.
     */
    [[self synthElementAt:stereoOut]
            setInput:[self synthElementAt:outPatchpoint]];
    [synthElements makeObjectsPerform:@selector(run)];

    return self;
}
```

The first of the two steps, applying the Note parameters to the patch, is performed in the **applyParameters:** method. The implementation of this method is described in a later section.

The second step, turning on the patch, distinguishes the **noteOnSelf:** method from the others. The first message in step 2 sets the input of **stereoOut** (the Out2Sum UnitGenerator) to the patchpoint **outPatchpoint** (recall that this connection was purposely left unspecified in the PatchTemplate). The final message sends **run** to each of the SynthPatch's synthElements. This causes the UnitGenerators to begin operating.

Note: While it isn't necessary to send **run** to the patch's patchpoint, it's convenient to send it to all synthElements as shown in the example. SynthData implements **run** to do nothing, so there's no harm in sending this message to a patchpoint.

The extremely important step of actually creating and connecting the objects that make up Simplicity's patch is performed automatically. As described in the next chapter, part of a SynthInstrument's duties when it receives a Note is to allocate an appropriate SynthPatch object to synthesize the Note. It does this by sending **patchTemplateFor:** to its SynthPatch subclass with the received Note as the argument. As we have seen, this method returns a PatchTemplate object. It then allocates a SynthPatch according to the specifications in the PatchTemplate and forwards the Note to the SynthPatch through the **noteOn:** method. Thus, by the time the SynthPatch receives the **noteOnSelf:** message (which is sent by **noteOn:**) the patch has already been created.

A SynthPatch contains a List of the objects that make up its patch in its **synthElements** instance variable. In the example above, a use of this instance variable is given as the receiver of the message that causes the UnitGenerators to start running:

```
[synthElements makeObjectsPerform:@selector(run)];
```

You can retrieve a particular object from the **synthElements** List by invoking the **synthElementAt:** method, passing the index of the synthElement as the argument. This is demonstrated in the example above in the line

```
[[self synthElementAt:stereoOut]
        setInput:[self synthElementAt:outPatchpoint]];
```

synthElementAt:stereoOut returns an instance of the object indexed by **stereoOut**. In other words, it returns an instance of the Out2sumUGx class, as specified in Simplicity's **patchTemplateFor:** method. Similarly, **synthElementAt:outPatchpoint** returns Simplicity's patchpoint.

Finally, the return value of **noteOnSelf:** is significant: If the method returns **nil**, the argument Note isn't synthesized. Simplicity's implementation always returns **self** so all noteOns that it receives are synthesized.

The noteUpdateSelf: Method

Simplicity's implementation of **noteUpdateSelf:** is straightforward; it simply applies its argument's parameters to the patch:

```
- noteUpdateSelf:aNote
{
    [self applyParameters:aNote];
    return self;
}
```

The value returned by **noteUpdateSelf:** is ignored.

The noteOffSelf: and noteEndSelf Methods

noteOffSelf: and **noteEndSelf** work together to wind down and deactivate a SynthPatch. As mentioned earlier, **noteOffSelf:** is automatically sent when a noteOff is forwarded to the SynthPatch through the **noteOff:** method. When a noteOff arrives, the SynthPatch doesn't stop; rather, the noteOff is taken as a signal to begin the release portions of any Envelopes that are part of the patch. The value returned by **noteOffSelf:** is taken as the amount of time, in seconds, to wait before invoking **noteEnd**; this value is usually the release time of the SynthPatch's amplitude envelope. Since Simplicity doesn't have any Envelopes, its implementation of **noteOffSelf:** always returns 0.0 (an example of a SynthPatch that uses Envelopes is given later):

```
- (double)noteOffSelf:aNote
{
    [self applyParameters:aNote];

    /* No Envelopes, so no release time is needed. */
    return 0.0;
}
```

Even though a noteOff is the beginning of the end of a SynthPatch's activity, the Note may contain some parameters; these parameters are applied just as in the other methods, by invoking **applyParameters:**.

After waiting the prescribed amount of time, the **noteEnd** message is sent. **noteEnd** invokes **noteEndSelf**, a method that deactivates the SynthPatch:

```
- noteEndSelf
{
    /* Deactivate the SynthPatch by idling the output. */
    [[self synthElementAt:stereoOut] idle];
    return self;
}
```

The **idle** method is implemented by all subclasses of UnitGenerator. In its implementation of **idle**, Out2sum connects its input to a predefined patchpoint that always contains zero data (the data in the patchpoint consists wholly of zeroes). This effectively turns off the patch. Notice that the SynthPatch isn't freed, nor is its patch (as specified in the PatchTemplate) dismantled. An important convention of SynthPatch design is to perform the minimum amount of work necessary when deactivating the object. This makes both the deactivation itself and a subsequent reactivation (when another noteOn arrives) as efficient as possible.

It should be noted that **noteEnd** (and, thus, **noteEndSelf**) is also invoked when the SynthPatch is created, thereby ensuring that the patch is silent until the first Note is received. This also explains why the final connection to Out2sum isn't specified in the PatchTemplate—if it was so specified, the connection would be made only to be immediately severed upon reception of the **noteEnd** message (the patch is created before **noteEnd** is sent).

Applying Parameters

The final step in our SynthPatch design is to supply it with parameter values. As mentioned earlier, Simplicity has three settable attributes: frequency, amplitude, and bearing. Simplicity implements the method **applyParameters:** to read the appropriate parameters from its argument Note and apply their values to the patch:

```
- applyParameters:aNote
{
    /* Retrieve and store the parameters. */
    double  myFreq = [aNote freq];
    double  myAmp = [aNote parAsDouble:MK_amp];
    double  myBearing = [aNote parAsDouble:MK_bearing];

    /* Apply frequency if present. */
    if ( !MKIsNoDVal(myFreq) )
        [[self synthElementAt:osc] setFreq:myFreq];

    /* Apply amplitude if present. */
    if ( !MKIsNoDVal(myAmp) )
        [[self synthElementAt:osc] setAmp:myAmp];

    /* Apply bearing if present. */
    if ( !MKIsNoDVal(myBearing) )
        [[self synthElementAt:stereoOut] setBearing:myBearing];
}
```

First, the parameters are retrieved from the argument Note. Notice that the **freq** method is used to retrieve frequency; recall from the description of the Note class that this method returns the value of **MK_freq** or, in **MK_freq**'s absence, a value converted from **MK_keyNum**.

To apply a parameter value to the patch, you send a message to the UnitGenerator that controls that aspect of the patch. The Oscg UnitGenerator controls frequency and amplitude, so **setFreq:** and **setAmp:** are sent to the patch's OscgUGxy object. Out2sum controls bearing, so it receives **setBearing:**. These methods are defined in the UnitGenerators' master classes.

The complete source code for Simplicity is provided as an example SynthPatch in the files **Simplicity.m** and **Simplicity.h** in the directory

/NextDeveloper/Examples/MusicKit/exampsynthpatch

A Better SynthPatch

Build a better SynthPatch and the world will beat a path to your door. This section improves the SynthPatch design shown in the previous sections. Of greatest significance is the envelope control that's added to both frequency and amplitude. The patch specification is accordingly more complex than in Simplicity. In addition, a number of conventions are introduced in the methods that play the patch, not only to accommodate envelope control but, more important, to make the SynthPatch more efficient and more adaptable to the caprice of musical performance.

The SynthPatch designed here is called Envy. Like Simplicity, it produces a single sine wave with a settable frequency, amplitude, and bearing.

Designing the Patch

The following example shows the implementation of Envy's **patchTemplateFor:** method:

```
/* Statically declare the synthElement indices. */
static int  ampAsymp,       /* amplitude envelope UG */
            freqAsymp,      /* frequency envelope UG */
            osc,            /* oscillator UG */
            stereoOut,      /* output UG */
            ampPp,          /* amplitude patchpoint */
            freqPp,         /* frequency patchpoint */
            outPp;          /* output patchpoint */

+ patchTemplateFor:aNote
{
    /* Step 1: Create (or return) the PatchTemplate. */
    static id theTemplate = nil;
    if (theTemplate)
        return theTemplate;
    theTemplate = [PatchTemplate new];

    /* Step 2:  Add the SynthElement specifications. */
    ampAsymp = [theTemplate addUnitGenerator:[AsympUGx class]];
    freqAsymp = [theTemplate addUnitGenerator:[AsympUGy class]];
    osc = [theTemplate addUnitGenerator:[OscgafiUGxxyy class]];
    stereoOut = [theTemplate addUnitGenerator:[Out2sumUGx class]];
    ampPp = [theTemplate addPatchpoint:MK_xPatch];
    freqPp = [theTemplate addPatchpoint:MK_yPatch];
    outPp = ampPp;
```

```
    /* Step 3:  Specify the connections. */
    [theTemplate to:ampAsymp sel:@selector(setOutput:) arg:ampPp];
    [theTemplate to:freqAsymp sel:@selector(setOutput:) arg:freqPp];
    [theTemplate to:osc sel:@selector(setAmpInput:) arg:ampPp];
    [theTemplate to:osc sel:@selector(setIncInput:) arg:freqPp];
    [theTemplate to:osc sel:@selector(setOutput:) arg:outPp];

    /* Return the PatchTemplate. */
    return theTemplate;
}
```

The three-step design outline is the same as in Simplicity: The PatchTemplate is created, the synthElement specifications are added to the PatchTemplate, and the connections between SynthElements are specified. However, two new UnitGenerator families, Oscgafi and Asymp, are introduced. These are examined in the next sections; briefly, Oscgafi is a general oscillator that allows another UnitGenerator to its amplitude and frequency. Asymp is an envelope handler; it's used to apply Envelope objects to the patch. The patch is illustrated in Figure 4-4.

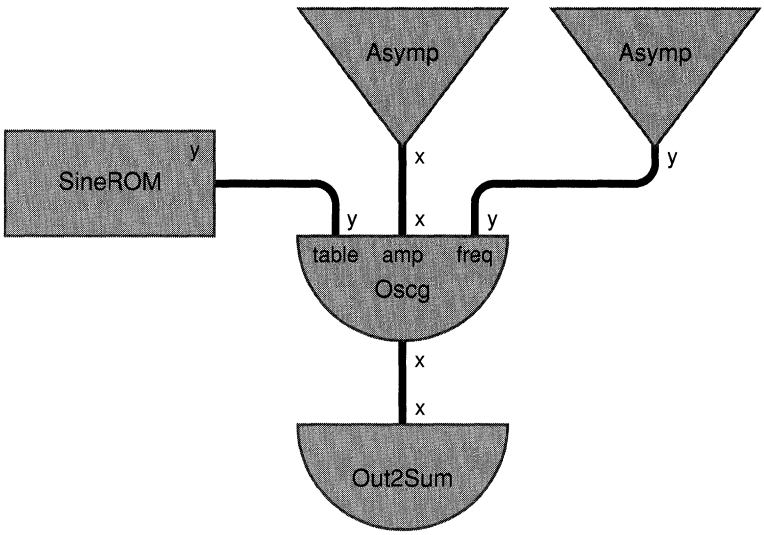

Figure 4-4. Envy's Patch

Returning to the code example, notice that **ampPp** and **outPp** are given the same value:

```
ampPp = [theTemplate addPatchpoint:MK_xPatch];
. . .
outPp = ampPp;
```

When the patch is created, these two synthElement indices will refer to the same object. In other words, the patchpoint that's used in the connection between **ampAsymp** and **osc** is reused in the connection between **osc** and **stereoOut**. Reusing patchpoints makes the patch smaller and more efficient. However, you can only reuse patchpoints if the patch's UnitGenerators are executed in a predictable order. Consider how Envy's UnitGenerators use the shared patchpoint:

1. **ampAsymp** writes to **ampPp**.
2. **osc** reads from **ampPp** and writes to **outPp**.
3. **stereoOut** reads from **outPp**.

For the shared patchpoint scheme to work, the UnitGenerators must be executed in the order given—chaos would reign should **stereoOut** read from **outPp** before **osc** writes to it. The order in which a SynthPatch's UnitGenerators are executed is the order in which their specifications are added to the PatchTemplate. Thus, Envy's UnitGenerators are executed in the following order:

1. ampAsymp
2. freqAsymp
3. osc
4. stereoOut

Since **ampAsymp** is executed before **osc**, and **osc** before **stereoOut**, the patchpoint between **ampAsymp** and **osc** can be reused as the patchpoint between **osc** and **stereoOut**.

Note: For some patches, the order in which the UnitGenerators are executed doesn't matter. You can add UnitGenerators and declare their execution to be unordered by using the **addUnitGenerator:ordered:** method, passing NO as the second argument (the **addUnitGenerator:** is actually a shorthand; it invokes **addUnitGenerator:ordered:** with YES as the second argument). While you can't share patchpoints in a patch that uses unordered UnitGenerators, allocating the patch is somewhat more efficient.

Oscgafi

The Oscgafi UnitGenerator is the most flexible oscillator provided by the Music Kit. In addition to allowing envelope control of amplitude and frequency, it also performs an interpolation, minimizing the noise that's sometimes introduced when reading the lookup table. The components of the UnitGenerator's name summarize these features:

Component	Meaning
Osc	Oscillator
g	General
a	Amplitude control
f	Frequency control
i	Interpolation

Oscgafi has four memory arguments, in this order:

1. output
2. amplitude control input
3. frequency control input
4. lookup table input

The permutations of a UnitGenerator with four memory arguments results in 16 leaf classes. Envy uses the xxyy version (OscgafiUGxxyy), so the memory arguments correspond to memory space as follows:

Argument	Space
output	x
amplitude control	x
frequency control	y
lookup table	y

A notable difference between Oscg and Oscgafi is that in the latter, frequency and amplitude aren't set directly through messages to the oscillator. To control these attributes, you affect the UnitGenerators that are connected to Oscgafi's inputs. The Music Kit provides a C function, called **MKUpdateAsymp()** that does this for you. This function is described later as it's used to apply parameter values to Envy's patch.

In addition, Oscgafi's frequency input is actually an increment input—the oscillator's frequency is defined by the increment, or step size, that it uses when reading its lookup table. This explains why **osc** is connected to the **freqPp** patchpoint through the **setIncInput:** method:

```
[theTemplate to:osc sel:@selector(setIncInput:) arg:freqPp];
```

Oscgafi's **incAtFreq:** method is provided to translate frequencies into increments. This, too, will be used when applying parameter values.

Asymp

The Asymp UnitGenerator is an envelope handler; it translates the data in an Envelope object and loads it onto the DSP. An Envelope object is associated with an Asymp through the **MKUpdateAsymp()** function.

Envy uses two Asymps, one to control the frequency of Oscgafi and the other to control its amplitude. The leaf classes are chosen to match the memory arguments in Oscgafi: The Asymp leaf class that controls amplitude is AsympUGx; for frequency, it's AsympUGy.

Playing the Patch

A number of new conventions for playing and controlling a SynthPatch are introduced in the following sections. In particular, the conventions regarding preemption, rearticulation, and "sticky" parameters are demonstrated.

Declaring the Parameters

A convention of SynthPatch design (one that wasn't followed in the implementation of Simplicity) is to create an instance variable for each parameter the SynthPatch responds to. The variables are used to maintain the state of the object's patch.

Because of the introduction of envelope control into the patch, Envy responds to several more parameters than did Simplicity. These are shown as they are declared as instance variables in the SynthPatch's interface file (**Envy.h**):

```
@interface Envy:SynthPatch
{
    /* Amplitude parameters. */
    id      ampEnv;         /* the Envelope object for amplitude */
    double  amp1,           /* amplitude at y=1 */
            amp0,           /* amplitude at y=0 */
            ampAtt,         /* ampEnv attack duration in seconds */
            ampRel;         /* ampEnv release duration in seconds*/

    /* Frequency parameters. */

    id      freqEnv;        /* the Envelope object for frequency */
    double  freq1,          /* frequency at y=1 */
            freq0,          /* frequency at y=0 */
            freqAtt,        /* freqEnv attack duration in seconds*/
            freqRel;        /* freqEnv release duration in seconds */

    /* Other parameters. */
    double  portamento;     /* transition time in seconds */
    double  bearing;        /* stereo location */
}
```

A set of defaults for the parameter instance variables should also be included in a SynthPatch design. Envy implements a method called **setDefaults** to provide this:

```
- setDefaults
{
    ampEnv      = nil;
    amp0        = 0.0;
    amp1        = MK_DEFAULTAMP;          /* 0.1 */
    ampAtt      = MK_NODVAL;              /* parameter not present */
    ampRel      = MK_NODVAL;              /* parameter not present */

    freqEnv     = nil;
    freq0       = 0.0;
    freq1       = MK_DEFAULTFREQ;         /* 440.0 */
    freqAtt     = MK_NODVAL;              /* parameter not present */
    freqRel     = MK_NODVAL;              /* parameter not present */

    portamento = MK_DEFAULTPORTAMENTO;    /* 0.1 */
    bearing     = MK_DEFAULTBEARING;      /* 0.0 (center) */

    return self;
}
```

By convention, a SynthPatch's parameter instance variables should be set to their default values before the SynthPatch begins a new phrase. This is done by invoking **setDefaults** from the **noteEndSelf** method. Keep in mind that **noteEnd**, which invokes **noteEndSelf**, is invoked when a new SynthPatch is created, so **setDefaults:** will be invoked before the first Note arrives as well as after the end of each phrase.

However, one other condition must be considered—that of the preempted SynthPatch—which, by definition, isn't sent the **noteEnd** message. The SynthPatch class defines a method **preemptFor:** that you can redefine in your subclass to reset the parameter instance variables to their default values and to provide any other special behavior for a preempted SynthPatch. The method is invoked just before the SynthPatch receives, in a **noteOn:** message, the Note for which it was preempted (the argument to **preemptFor:** is this same Note). Envy implements **preemptFor:** to preempt the amplitude Envelope and invoke **setDefaults:**. It ignores the argument:

```
- preemptFor:aNote
{
    [[self synthElementAt:ampAsymp] preemptEnvelope];
    [self setDefaults];
    return self;
}
```

The noteOnSelf: and noteUpdateSelf: Methods

These two methods follow the same form as those in Simplicity:

```
- noteOnSelf:aNote
{
    /* Apply the parameters to the patch. */
    [self applyParameters:aNote];

    /* Make the final connection to the output sample stream.*/
    [[self synthElementAt:stereoOut] setInput:outPp];

    /* Tell the UnitGenerators to begin running. */
    [synthElements makeObjectsPerform:@selector(run)];

    return self;
}

- noteUpdateSelf:aNote
{
    /* Apply the parameters to the patch. */
    [self applyParameters:aNote];

    return self;
}
```

Once again, both methods invoke **applyParameters:** to apply the Note's parameters to the patch. In addition, **noteOnSelf:** completes the connection between Oscgafi and Out2sum, and it tells the UnitGenerators to run by sending **run** to each of the synthElements.

The noteOffSelf: and noteEndSelf Methods

Recall that the value returned by **noteOffSelf:** is taken as the amount of time to wait (in seconds) before **noteEnd** is invoked. Typically, this value is the release time of the SynthPatch's amplitude Envelope:

```
- (double)noteOffSelf:aNote
{
    /* Apply the parameters. */
    [self applyParameters: aNote];

    /* Signal the release portion of the frequency Envelope. */
    [[self synthElementAt:freqAsymp] finish];

    /* Same for amplitude, but also return the release duration. */
    return [[self synthElementAt:ampAsymp] finish];
}
```

An Asymp responds to the **finish** message by signaling the release portion of its Envelope; the method returns the duration of the release.

As in Simplicity, **noteEndSelf** sends **idle** to the Out2sum object to remove it from the output sample stream. It also aborts the frequency Envelope (we're assured that the amplitude Envelope has finished since its demise is what causes this method to be invoked) and then invokes the **setDefaults** method, as dictated in "Declaring the Parameters," above.

```
- noteEndSelf
{
     /* Remove the patch's Out2sum from the output sample stream. */
    [[self synthElementAt:stereoOut] idle];

    /* Abort the frequency Envelope. */
    [[self synthElementAt:freqAsymp] abortEnvelope];

    /* Set the instance variables to their default values. */
    [self setDefaults];

    return self;
}
```

Phrase Status

The manner in which a Note's parameters are applied to a patch can depend on the performance context in which the SynthPatch receives the Note. This context, called *phrase status*, is represented as an MKPhraseStatus constant and is automatically set when the SynthPatch receives a phrase event message (such as **noteOn:** and **noteUpdate:**). There are seven phrase states:

- **MK_phraseOn** means that the received Note is a noteOn and the SynthPatch has been freshly allocated to synthesize the Note. This status indicates the beginning of a new phrase.
- **MK_phraseRearticulate** indicates a noteOn that rearticulates an existing phrase.
- **MK_phraseOnPreempt** is also used for noteOns, but it indicates that the SynthPatch has been preempted to synthesize the Note. Like **MK_phraseOn**, this status means that a new phrase is beginning.
- **MK_phraseUpdate** means that the Note is a noteUpdate and the SynthPatch is in the attack or stickpoint portions of its Envelopes (the SynthPatch's synthStatus is **MK_running**).
- **MK_phraseOff** indicates a noteOff.
- **MK_phraseOffUpdate** is for a noteUpdate that arrives during the release portion of the Envelopes (synthStatus is **MK_finishing**).
- **MK_phraseEnd** is used to indicate the end of a phrase.

A SynthPatch's phrase status, which is retrieved by sending **phraseStatus** to the SynthPatch, is provided solely as a convenience to SynthPatch designers and is only valid within the implementations of the **noteOnSelf:**, **noteUpdateSelf:**, **noteOffSelf:**, and **noteEndSelf** methods. Sent to a SynthPatch from outside these methods, **phraseStatus** returns **MK_noPhraseActivity**.

You can use phrase status in the design of your SynthPatch in a test that leads to specialized behavior. For example, you may want to apply certain noteUpdate parameters differently, depending on whether the phrase status is **MK_phraseUpdate** or **MK_phraseOffUpdate**. Two conventional uses of phrase status—as an argument to the **MKUpdateAsymp()** function and to determine if a Note is the beginning of a new phrase—are demonstrated in the next section.

Applying Parameters

The way that Envy applies a Note's parameters is more sophisticated than the manner employed by Simplicity. A convention ignored in the design of Simplicity holds that a noteOn that rearticulates a phrase should inherit, if necessary, the values of the parameters in the phrase so far. For example, if a rearticulating noteOn doesn't contain an amplitude Envelope, it uses the one set in the previous noteOn. The implementation of Envy accommodates this by storing its parameter values as instance variables: It can supply a "missing" parameter by using the value stored in the appropriate variable.

Determining the correct value for amplitude and frequency is more complicated in the implementation of Envy than in that of Simplicity. Because of its use of Envelope objects, Envy's amplitude and frequency depend on the values of a number of related parameters. The Music Kit provides a C function called **MKUpdateAsymp()** that helps to untangle this web. The function takes, as arguments, all the objects and parameter values associated with a particular Envelope and applies them in a predictable manner to the attribute that the Envelope controls.

MKUpdateAsymp() takes eight arguments:

1. An Asymp object
2. An Envelope object
3. The Envelope's value when y = 0.0
4. The Envelope's value when y = 1.0
5. The Envelope's attack time
6. The Envelope's release time
7. The portamento value
8. The current phrase status

The function's behavior is described in the Volume 2, Chapter 3. Briefly, it applies an Envelope (argument 2) to an Asymp (argument 1) after properly scaling the Envelope's value range (arguments 3 and 4) and setting its attack and release times (arguments 5 and 6). Portamento (argument 7) is used only if the phrase status (argument 8) is **MK_phraseRearticulate**.

Envy's implementation of the **applyParameters:** method demonstrates the conventional way to apply parameters to a patch that includes Envelopes:

```
- applyParameters:aNote
{
    /* Retrieve and store the parameters. */
    id     myAmpEnv     = [aNote parAsEnvelope:MK_ampEnv];
    double myAmp0       = [aNote parAsDouble:MK_amp0];
    double myAmp1       = [aNote parAsDouble:MK_amp1];
    double myAmpAtt     = [aNote parAsDouble:MK_ampAtt];
    double myAmpRel     = [aNote parAsDouble:MK_ampAtt];

    id     myFreqEnv    = [aNote parAsEnvelope:MK_freqEnv];
    double myFreq0      = [aNote parAsDouble:MK_freq0];
    double myFreq1      = [aNote freq];
    double myFreqAtt    = [aNote parAsDouble:MK_freqAtt];
    double myFreqRel    = [aNote parAsDouble:MK_freqRel];

    double myPortamento = [aNote parAsDouble:MK_portamento];
    double myBearing    = [aNote parAsDouble:MK_bearing];

    /* Store the phrase status. */
    MKPhraseStatus phraseStatus = [self phraseStatus];

    /* Is aNote a noteOn? */
    BOOL isNoteOn = [aNote noteType] == MK_noteOn;

    /* Is aNote the beginning of a new phrase? */
    BOOL isNewPhrase = (phraseStatus == MK_phraseOn) ||
                       (phraseStatus == MK_phraseOnPreempt);

    /* Used in the parameter checks. */
    BOOL shouldApplyAmp = NO;
    BOOL shouldApplyFreq = NO;
    BOOL shouldApplyBearing = NO;

    /* The same portamento is used in both frequency and amplitude. */
    if ( !MKIsNoDVal(myPortamento) ) {
        portamento = myPortamento;
        shouldApplyAmp = YES;
        shouldApplyFreq = YES; }

    /* Check the amplitude parameters and set the instance
       variables. */
    if (myAmpEnv != nil) {
        ampEnv = myAmpEnv;
        shouldApplyAmp = YES; }

    if (!MKIsNoDVal(myAmp0)) {
        amp0 = myAmp0;
        shouldApplyAmp = YES; }

    if (!MKIsNoDVal(myAmp1)) {
        amp1 = myAmp1;
        shouldApplyAmp = YES; }
```

```
    if (!MKIsNoDVal(myAmpAtt)) {
        ampAtt = myAmpAtt;
        shouldApplyAmp = YES; }

    if (!MKIsNoDVal(myAmpRel)) {
        ampRel = myAmpRel;
        shouldApplyAmp = YES; }

    /* Apply the amplitude parameters. */
    if (shouldApplyAmp || isNoteOn)
        MKUpdateAsymp([self synthElementAt:ampAsymp],
                      ampEnv, amp0, amp1, ampAtt, ampRel,
                      portamento, phraseStatus);

    /* Check the frequency parameters and set the instance
       variables. */
    if (myFreqEnv != nil) {
        freqEnv = myFreqEnv;
        shouldApplyFreq = YES; }

    if (!MKIsNoDVal(myFreq0)) {
        freq0 = myFreq0;
        shouldApplyFreq = YES; }

    if (!MKIsNoDVal(myFreq1)) {
        freq1 = myFreq1;
        shouldApplyFreq = YES; }

    if (!MKIsNoDVal(myFreqAtt)) {
        freqAtt = myFreqAtt;
        shouldApplyFreq = YES; }

    if (!MKIsNoDVal(myFreqRel)) {
        freqRel = myFreqRel;
        shouldApplyFreq = YES; }

    /* Apply the frequency parameters. */
    if (shouldApplyFreq || isNoteOn)
        MKUpdateAsymp([self synthElementAt:freqAsymp], freqEnv,
                      [[self synthElementAt:osc] incAtFreq:freq0],
                      [[self synthElementAt:osc] incAtFreq:freq1],
                      freqAtt, freqRel, portamento, phraseStatus);

    /* Check and set the bearing. */
    if (!MKIsNoDVal(myBearing)) {
        bearing = myBearing;
        shouldApplyBearing = YES; }

    if (shouldApplyBearing || isNewPhrase)
        [[self synthElementAT:stereoOut] setBearing:bearing];

    return self;
}
```

As in Simplicity's implementation of **applyParameters:**, the value of each parameter is stored and then checked to determine whether the parameter is actually present in the argument Note. In addition, this implementation updates the values of the instance variables to those of the parameters that are present.

Each parameter that affects amplitude is checked in its own conditional statement. If the parameter is present, the **shouldApplyAmp** variable is set to YES, indicating that the amplitude Envelope needs to be updated. Finally, the value of **shouldApplyAmp** is logically or'd with the value of **isNoteOn**, which is YES if **aNote** is a noteOn. Thus, the **MKUpdateAsymp()** call for amplitude is made if any of the tested parameters are present, and it's always made if **aNote** is a noteOn.

The conditionals for applying the frequency parameters are the same as those for amplitude. Notice, however, that the **freq0** and **freq1** values aren't passed directly to **MKUpdateAsymp()**. Instead, they're used to retrieve increment values from **osc** through its **incAtFreq:** method. As mentioned earlier, Oscgafi's frequency value isn't set as a frequency in hertz, but rather as an increment into its lookup table.

Finally, the bearing parameter is tested, its instance variable is set, and the parameter is applied to the patch. Notice that bearing is automatically applied if **aNote** is the beginning of a new phrase. Unlike amplitude and frequency, bearing isn't controlled by an Envelope, so it doesn't need to be automatically applied if the Note is simply a rearticulation of an existing phrase.

Adding a WaveTable

Envy, although otherwise entertaining, is of limited timbral interest—it can only produce a sine wave. Replacing Envy's sine wave with a WaveTable is quite simple; its patch isn't affected, nor are the implementations of the **noteOnSelf:** type methods. The only real change is in the implementation of **applyParameters:**.

First, however, you must provide an instance variable and default value for **MK_waveform**—the parameter that identifies SynthPatch's WaveTable object:

```
@interface Envy:SynthPatch
{
    id waveform;
    . . .
}

- setDefaults
{
    waveform = nil;
    . . .
}
```

The **applyParameters:** method is rewritten to attend to the new parameter:

```
- applyParameters:aNote
{
    . . .
    /* Create local variables for the new parameter. */
    id  myWaveform = [aNote parAsWaveTable:MK_waveform];
    BOOL shouldApplyWave = NO;

    /* Test the parameters. */
    if (myWaveform != nil) {
        waveform = myWaveform;
        shouldApplyWave = YES; }

    if (shouldApplyWave || isNewPhrase)
    [[self synthElementAt:osc] setTable:waveform length:0
                                 defaultToSineROM:isNewPhrase];
    . . .
}
```

The **setTable:length:defaultToSineROM:** method sets the oscillator's lookup table to the specified WaveTable object. Passing 0 as the length of the table causes the Music Kit to compute a default value. The argument to **defaultToSineROM:** is a BOOL value that determines whether the sineROM should be used if memory for the WaveTable can't be allocated. In this implementation, the argument is YES only if **aNote** is the beginning of a new phrase. For all other phrase states, the previously set WaveTable is used if a new one can't be allocated.

WaveTables are shared among SynthPatches—if two SynthPatches declare the same WaveTable object (with the same length), the WaveTable is allocated once and the SynthPatches read from the same memory. This feature is provided automatically by all the Oscg-type oscillators.

Creating a UnitGenerator Subclass

Each UnitGenerator subclass is an object-oriented interface to a DSP macro, called a *unit generator*, that's written in DSP56001 assembly language. The Objective-C code in the subclass is generated automatically from a DSP macro by the **dspwrap** program. To build your own UnitGenerator subclass that you can use in SynthPatch design, you run **dspwrap** on a DSP56001 assembly language macro that you've created. In addition, you can modify and wrap the DSP macros that NeXT provides as source code in the directory **/usr/lib/dsp/ugsrc**.

The design of DSP assembly language macros is outside the scope of the present discussion. The following sections show how to use **dspwrap** and how to further modify the UnitGenerators that it creates.

Using dspwrap

The **dspwrap** program is used to create array processing C functions as well as UnitGenerator subclasses. To indicate that you want to create the latter, you call the program with the **-ug** switch followed by the name of the file that contains the assembly code macro (you must include the ".asm" extension when specifying the file). For example, the invocation

```
dspwrap -ug unoisehp.asm
```

creates a master UnitGenerator class called UnoisehpUG as well as the appropriate leaf classes. These are embodied in the following files, which are automatically generated by **dspwrap**:

- **UnoisehpUG.m** is the implementation file of the master class.
- **UnoisehpUG.h** is the master class interface file.
- **UnoisehpUGx.m** and **UnoisehpUGy.m** are leaf class implementations.
- **UnoisehpUGx.h** and **UnoisehpUGy.h** are the leaf interface files.
- **unoisehpUGInclude.m** is imported by the master class.

The number of leaf classes that are created depends on the number of address-valued memory arguments, described below, in the macro: A different leaf class is created for each combination of x and y DSP memory spaces. The **unoisehp** macro, which implements a high-pass random number generator, has only one such argument—its output—so two leaf classes are generated, one for either memory space.

Some other files, such as documentation and DSP assembler and linker files, are also created. These can be moved, deleted, or disregarded as you see fit. For the present purposes, only the files listed above are important.

Modifying the Class

There are two basic reasons to modify a UnitGenerator class that's generated by **dspwrap**:

- To set the values of the DSP memory arguments that are passed to the macro
- To define the class's response to some common messages

Of the files generated by **dspwrap**, you should modify only those that define the master class. In other words, continuing with the Unoisehp example, only **UnoisehpUG.m** and **UnoisehpUG.h** should be edited. The entire implementation file of the UnoisehpUG master class as generated by **dspwrap** is shown below:

```
#import <musickit/musickit.h>
#import "UnoisehpUG.h"
@implementation UnoisehpUG:UnitGenerator
{}

/* DSP memory arguments. */
enum args { aout, seed};

#import "unoisehpUGInclude.m"
```

Setting the Arguments

The two **enum** variables shown above, **aout** and **seed**, are DSP memory arguments. A memory argument represents a location on the DSP from which the unit generator that's executing can read information sent to it by the Music Kit. There are two types of memory arguments:

- Address-valued arguments administer the location in DSP memory from which, or to which, the executing unit generator reads or writes data.
- Datum-valued arguments take a value that's used as part of the unit generator's computation.

In the example, **aout** variable is an address-valued argument that represents the UnitGenerator's output patchpoint. We create a Unoisehp method named **setOutput:** that sets this argument:

```
-setOutput:outputPatchPoint
{
    return [self setAddressArg:aout to:outputPatchPoint];
}
```

The **setAddressArg:to:** method is defined by UnitGenerator to set an address-valued DSP memory argument.

The other **enum** variable, **seed**, is a datum-valued memory argument. It's set through UnitGenerator's **setDatumArg:to:** method, as demonstrated in our implementation of Unoisehp's **setSeed:** method:

```
-setSeed:(int)aSeed
{
    return [self setDatumArg:seed to:aSeed];
}
```

Defining the Class's Response

During a performance, a UnitGenerator can expect to receive the following messages:

- **run** tells the receiver to begin doing whatever it does.
- **finish** winds down the receiver before coming to a halt.
- **idle** provides instructions for halting the receiver's activity.

Invocations of these methods were shown in the implementations of the Simplicity and Envy SynthPatches. Remember that **finish** returns the amount of time the UnitGenerator needs to complete its mission—the amount of time to wait before **idle** should be sent. The default implementations of these methods can be sufficient. For further tuning, you should implement, in your master class, the methods **runSelf**, **finishSelf**, and **idleSelf**—methods that are automatically invoked when the corresponding performance messages are received.

You almost always provide an implementation of **idleSelf** to ensure that your UnitGenerator is brought to a halt in a manner befitting its activity. The Unoisehp implementation of this method sets its output to the DSP's *sink*, a location that, by convention, is never read:

```
-idleSelf
{
    /* Set the output (aout) to sink. */
    [self setAddressArgToSink:aout];
    return self;
}
```

Chapter 5
Music Performance

Chapter 5
Music Performance

This chapter describes the classes and functions that you use to create a musical performance with the Music Kit. The material presented here falls roughly into three sections: realizing music data, Music Kit scheduling, and guidelines for building an efficient application. To fully appreciate this chapter, you should be familiar with the material in the preceding two chapters, Chapter 3, "Representing Music Data," and Chapter 4, "Music Synthesis."

Design Philosophy

The primary task in a musical performance is to take a representation of music and render it in an appropriate fashion. Traditionally, this task is met by a performer wielding an instrument. Similarly, the Music Kit has its Performer and Instrument objects: A Performer obtains a Note object, either by reading it from a database such as a Part or scorefile or by improvising the Note itself. The Performer then relays the Note to an Instrument that provides the machinery for rendering, or *realizing*, the Note.

Of Instrument and Performer, the former is the more crucial to a Music Kit performance. Realization is the whole point of a performance, thus Instruments are ubiquitous. Performers provide a convenient means for acquiring or generating Notes but they're not essential to a performance application; in fact, some applications can't use Performers but, instead, must manufacture Notes themselves and send them directly to Instruments. In any case, the means by which a Note is acquired is separated from its means of realization and the mechanism for connecting an agent of acquisition to that of realization is extremely general. Specifically, any Performer (or your application) can be connected to any—and any number of—Instruments. These connections can be created and severed dynamically, allowing you to create a kaleidoscopic network of Note sources and destinations.

With Performer and Instrument, the Conductor class completes the triumvirate of preeminent performance classes. A Conductor object acts as a time keeper as it determines the tempo of the performance and starts and stops groups of Performers. Essentially, a Conductor is a timed-message sending object that dispatches Objective-C messages at distinct times (in this regard, it's similar to the timed entry mechanism of Display PostScript, but with a more sophisticated interface). For example, it's through a Conductor that a Performer schedules the messages that relay Notes to an Instrument.

Asking a computer to perform music often pushes it to its computational limits. Because of the demands of music performance, the Music Kit provides hooks into the operating system that allow you to create applications that can run with heightened priority. While

this can sometimes be a dangerous game—it isn't easy stopping a highly charged fugue that has jumped the rails—well-debugged and fine-tuned applications will find great benefit in these enhancements.

Performance Outline

A Music Kit performance can be divided into three phases:

- Preparation: You must determine the sources and destinations of the Notes that you wish to perform and establish other characteristics of the performance.
- Commencement: This is always accomplished by sending **startPerformance** to the Conductor class.
- Termination: This is achieved by sending **finishPerformance** to the Conductor class.

This sequence of activities can be repeated any number of times while an application is running. While a single application can have but a single performance in progress at any particular time, that single performance can do any number of things. For example, if you want to capture MIDI input as a scorefile while playing along to a Part that's synthesized on the DSP, you perform both tasks in a single performance—you don't have to set up separate environments. Obviously, the two endeavors involve different classes of objects, but when the Conductor class receives the **startPerformance** message, everybody starts wheezing.

Much of the work that goes into a performance is involved in setting it up; in fact, many of the methods that are defined by the classes that are involved in a performance can only be invoked before the performance starts (or between performances). As you design your own application, you should consult the class descriptions in Chapter 2 of *Reference* to check for the conditions under which a method may be invoked.

The following sections examine the three primary classes involved in a performance: Instrument, Performer, and Conductor.

The Instrument Class

Instruments are the agents through which Note objects are realized. Since realization is the ultimate destiny of a Note object, the Instrument class is the ultimate focus of a Music Kit performance: Every performance involves at least one Instrument.

At the heart of an Instrument is its **realizeNote:fromNoteReceiver:** method. The Instrument class itself is abstract and doesn't implement this method; it's the responsibility of each Instrument subclass to provide an implementation of **realizeNote:fromNoteReceiver:** to establish the manner in which instances of the subclass

realize Notes. The Music Kit includes a number of Instrument subclasses (and pseudo-Instruments) that realize Notes by synthesizing them on the DSP or an external MIDI synthesizer, and that store Notes by adding them to a Part or Score or by writing them to a file.

One of the principals of Instrument design is that an Instrument doesn't create the Notes that it realizes. Instead, it realizes Notes that are sent to it by another object or by your application. The machinery by which an Instrument receives Notes is well defined by the Music Kit; unlike that for Note-realization, you don't reinvent the Note-reception mechanism for each Instrument subclass.

The following sections examine the Instrument class according to its two tasks: receiving and realizing Notes. The latter topic centers upon descriptions of the Instrument subclasses included in the Music Kit, followed by an example of Instrument subclass design.

Receiving Notes

An Instrument receives Notes through the same method that defines their realization, **realizeNote:fromNoteReceiver:**. However, you don't send Notes to an Instrument by invoking this method directly; instead, you send the Notes to one of the Instrument's NoteReceiver objects which, in turn, invokes **realizeNote:fromNoteReceiver:** for you.

NoteReceiver is an auxiliary class that acts as a "Note port" for an Instrument. This is manifested as the three features that a NoteReceiver brings to an Instrument:

- It provides methods through which an Instrument receives Notes (in other words, methods that invoke Instrument's **realizeNote:fromNoteReceiver:**).

- It lets you control the stream of Notes into an Instrument by allowing you to toggle its ability to forward Notes to an Instrument.

- It acts as a jack into which you can plug a Performer object's NoteSender. This connects the Performer that owns the NoteSender to the Instrument that owns the NoteReceiver, such that the Notes generated by the Performer are automatically delivered to and realized on the connected Instrument. The methods by which you marry a NoteReceiver to a NoteSender are described later in the section on Performers and NoteSenders.

Creating and adding NoteReceivers is usually part of the design of the Instrument subclass, although some subclasses require you to create and add NoteReceivers yourself. In either case, the method by which a NoteReceiver is added to an Instrument is the same: Instrument's **addNoteReceiver:**. While an Instrument can own more than one NoteReceiver, a single NoteReceiver can belong to only one Instrument: An invocation of **addNoteReceiver:** removes the argument (the NoteReceiver) from its current owner.

The receiveNote: Method

The fundamental method through which a NoteReceiver itself receives Notes—and thereby forwards them to its owning Instrument—is called **receiveNote:**. However, before you can send **receiveNote:** to a NoteReceiver, you must be able to locate the object. If you created and added NoteReceivers to an Instrument yourself, finding these objects shouldn't pose a problem—you made them, you should know where they are. But keep in mind that many Instruments create and add NoteReceivers for you. Any Instrument's NoteReceivers can be retrieved as a List object by sending the **noteReceivers** message to the Instrument; members of this List can then be sent the **receiveNote:** message. As a convenience, the first NoteReceiver in this List can be retrieved through the **noteReceiver** method. This is particularly handy for Instruments that define only one NoteReceiver, or in situations where you don't care which NoteReceiver you send the Note to. In the following example, an instance of SynthInstrument is created and a Note is sent to its NoteReceiver. SynthInstrument is a subclass of Instrument that realizes Notes by synthesizing them on the DSP; it creates a single NoteReceiver as part of its **init** method:

```
/* Create a SynthInstrument and send it a Note (assumed to exist). */
aSynthIns = [[SynthInstrument alloc] init];

[Conductor lockPerformance];
[[aSynthIns noteReceiver] receiveNote:aNote];
[Conductor unlockPerformance];
```

You'll note the presence of the Conductor class methods **lockPerformance** and **unlockPerformance**; invocations of these two methods should bracket all invocations of **receiveNote:** (with some exceptions, as noted in the sections on Conductors and Performers, later in this chapter). Briefly, these methods ensure that your cast of characters are in synch and primed for timely Note realization. A fuller explanation of **lockPerformance** and **unlockPerformance** is given in "Locking the Performance," later in this chapter.

You can create and add any number of NoteReceivers to the Instruments that you allocate, even if these Instruments create NoteReceivers themselves. However, with some exceptions—notably that described in the next section—there isn't much reason to do so: One NoteReceiver is as good as the next. Any number of Note-producing agents—whether Performers or different mechanisms in your application—can all send **receiveNote:** to the same NoteReceiver. In general, adding a gaggle of NoteReceivers to an Instrument doesn't make the Instrument more interesting, it simply makes it bigger.

Squelching a NoteReceiver

You can throttle a NoteReceiver's ability to invoke **realizeNote:fromNoteReceiver:** by sending it the **squelch** message. To release this clench, you invoke **unsquelch**. Through this simple feature, you can easily and quickly mute an Instrument.

While a NoteReceiver is squelched, the **receiveNote:** method ceases to function: It immediately returns **nil** without forwarding the argument Note to the NoteReceiver's owner (unsquelched, **receiveNote:** returns **self**). Realization of the Note isn't merely deferred, it's abandoned for good. You can determine whether a NoteReceiver is squelched by sending it the **isSquelched** message: If the message returns YES, the object is squelched.

The squelch feature, in certain applications, argues for the use of multiple NoteReceivers for a single Instrument. For example, you can create an application in which Notes are generated from MIDI input and read from a Part object at the same time. If you feed these two Note sources to the same Instrument, you may want to create and add two distinct NoteReceivers, one for either source, so you can independently squelch the two streams of Notes.

Performance Status

An Instrument is considered to be in performance from the time that you send **receiveNote:** to any of its unsquelched NoteReceivers until the performance is over—in other words, until the Conductor class receives the **finishPerformance** message. You can query an Instrument's performance status by sending it the **inPerformance** message, where a return of YES signifies that the object is currently in performance.

The performance status of an Instrument is significant because some Instrument methods aren't effective while the Instrument is in performance. This can be a particularly devilish source of confusion since the state of an Instrument's performance doesn't require a performance, in the larger sense, to be in progress. Specifically, if you send **receiveNote:** to an Instrument's NoteReceiver *before* the Conductor class receives **startPerformance**, the Instrument will, nonetheless, be considered to be performing (and thus the aforementioned performance-status dependent methods will have no effect).

Realizing Notes

There are very few rules when it comes to realizing Notes: You can implement **realizeNote:fromNoteReceiver:** to realize a Note in almost any arbitrary manner. As mentioned earlier, the Music Kit includes Instruments that synthesize and store Notes as their forms of realization. For many performance applications, the Instruments provided by the Music Kit are sufficient. The following sections describe these subclasses.

SynthInstrument

A SynthInstrument is by far the most complicated Instrument; it realizes Notes by causing them to be synthesized on the DSP. It operates on three basic principles:

- Every instance of SynthInstrument is associated with a single subclass of SynthPatch.
- Before or during a performance, a SynthInstrument object allocates (through requests to the Orchestra) and manages instances of its SynthPatch subclass. These SynthPatch objects are used to synthesize the Notes that the SynthInstrument receives through its NoteReceiver.

Following these principles, the primary decisions you need to make regarding a SynthInstrument are which SynthPatch subclass to assign it and which of two schemes it should use to allocate instances of the SynthPatch subclass.

Setting the SynthPatch Subclass

You set a SynthInstrument's SynthPatch subclass by invoking the **setSynthPatchClass:** method. As the method's argument you specify one of the SynthPatch classes included in the Music Kit, or one of your own. If you have a multiple-DSP system, the SynthInstrument will allocate its SynthPatch objects on the first DSP that has sufficient available resources. You can restrict allocation to a specific DSP by invoking **setSynthPatchClass:orchestra:**, passing an Orchestra object as the second argument.

Note: If you use a SynthPatch class included in the Music Kit, you must import the file **musickit/synthpatches/*ClassName*.h**, where ***ClassName*** is the name of the class you wish to use. Alternatively, the file **musickit/synthpatches/synthpatches.h** will import all the Music Kit SynthPatch interface files.

The **setSynthPatchClass:** method (and the **...orchestra:** version) checks its (first) argument to ensure that it's a class that inherits from SynthPatch, returning **nil** if it doesn't. It also returns **nil** (and doesn't set the class) if the SynthInstrument is already involved in a performance.

Allocation Mode

SynthInstrument defines two allocation modes:

- In *automatic allocation mode* the SynthInstrument allocates SynthPatch objects as it receives Notes, tagging each SynthPatch with the note tag of the Note for which it was allocated. As it receives subsequent Notes, the SynthInstrument compares each Note's note tag to those of the SynthPatches it has already allocated. If it finds a match, the Note is sent to that SynthPatch. If it doesn't match, the SynthInstrument allocates a new SynthPatch.

- In *manual allocation mode* the SynthPatch objects are allocated all at once, before Notes begin to arrive. The number of SynthPatches to allocate is set through the method **setSynthPatchCount:**. The number of SynthPatches that are actually allocated may be less than the number requested, depending on the availability of DSP resources at the time that the message is sent. The method returns the number of SynthPatches that were actually allocated.

By default, a SynthInstrument is in automatic allocation mode. Simply sending **setSynthPatchCount:** places it in manual mode. To set it back to automatic mode, you send it the **autoAlloc** message. A SynthInstrument can't switch modes while it's performing.

You can query a SynthInstrument's allocation mode by invoking **allocMode**, a method that returns one of the following integer constants:

Constant	Mode
MK_AUTOALLOC	Automatic allocation
MK_MANUALALLOC	Manual allocation

Changing the SynthPatch Count

You can change the number of manually allocated SynthPatches at any time—even during a performance—simply by resending the **setSynthPatchCount:** message. (If the object is performing, it must have been placed in manual mode before the performance began.) Notice, however, that the argument is always taken as the total number of SynthPatches that are allocated to the SynthInstrument—it doesn't represent the number of new objects to allocate. For example, in the following sequence of messages, a total of four SynthPatches are allocated.

```
/* Allocate three SynthPatches. */
[aSynthIns setSynthPatchCount:3];
. . .

/* Allocate one more. */
[aSynthIns setSynthPatchCount:4];
```

The **synthPatchCount** method returns the number of manually allocated SynthPatches. Thus, the previous example can be rewritten as

```
/* Allocate three SynthPatches. */
[aSynthIns setSynthPatchCount:3];
. . .

/* Allocate one more. */
[aSynthIns setSynthPatchCount:[aSynthIns synthPatchCount]+1];
```

If the SynthInstrument is in automatic mode, **synthPatchCount** returns 0.

Deallocating SynthPatch objects is also possible:

```
/* Allocate three SynthPatches. */
[aSynthIns setSynthPatchCount:3];
. . .

/* Deallocate two of them. */
[aSynthIns setSynthPatchCount:[aSynthIns synthPatchCount]-2];
```

If the argument signifies a deallocation, the SynthInstrument's idle SynthPatches, if any, are deallocated first; the balance are deallocated as they become inactive.

The PatchTemplate

Some SynthPatches come in a variety of configurations. For example, the Fm1vi SynthPatch (frequency modulation with optional vibrato) is configured differently depending on whether the Note it's synthesizing specifies vibrato. It does this to be as efficient as possible—excluding vibrato from Fm1vi's configuration means that it uses less of the Orchestra's resources.

A SynthPatch represents each of its configurations as a different PatchTemplate object. When you set a SynthInstrument to manual allocation mode, you can specify the number of SynthPatches with a particular PatchTemplate by invoking the **setSynthPatchCount:forPatchTemplate:** method. The second argument is the **id** of the PatchTemplate that you want. This is returned by sending the **patchTemplateFor:** message to the SynthInstrument's SynthPatch class, with a Note object as the argument. This is best explained by example:

```
/* Create a SynthInstrument. */
id  aSynthIns = [[SynthInstrument init] alloc];

/* Create a variable to store the PatchTemplate. */
id  noVibTemplate;

/* Create a dummy Note. */
id  aNote = [[Note init] alloc];
```

```
/* Set its vibrato amplitudes to 0.0. */
[aNote setPar:MK_svibAmp toDouble:0.0];
[aNote setPar:MK_rvibAmp toDouble:0.0];

/* Retrieve the PatchTemplate senza vibrato. */
noVibTemplate = [[aSynthIns synthPatchClass]
            patchTemplateFor:aNote];

/* Allocate three vibratoless SynthPatches. */
[aSynthIns setSynthPatchCount:3
            forPatchTemplate:noVibTemplate];
```

If the PatchTemplate isn't specified, the SynthPatches are created using the *default PatchTemplate*. Each subclass of SynthPatch designates one of its PatchTemplates as the default for that class; by convention, the most extravagant PatchTemplate is provided as the default. Thus, the default Fm1vi PatchTemplate includes vibrato. In the example, the Fm1vi PatchTemplate without vibrato is retrieved by passing (as the second argument to **patchTemplateFor:**) a Note that explicitly sets the vibrato parameters to 0.0.

Within the same SynthInstrument, you can manually allocate SynthPatches that use different PatchTemplates. The following extension of the previous example demonstrates this:

```
/* Allocate one vibratoless SynthPatch. */
[aSynthIns setSynthPatchCount:1
            forPatchTemplate:noVibTemplate];

/* And two with vibrato. */
[aSynthIns setSynthPatchCount:2];
```

setSynthPatchCount: always uses the default PatchTemplate (which, as mentioned earlier, includes vibrato for Fm1vi). When the SynthInstrument in the example receives a Note that initiates a new phrase, it automatically forwards the Note to the proper SynthPatch: If the Note contains vibrato parameters with zero values (similar to the dummy Note used in the previous example), it's forwarded to the SynthPatch that excludes vibrato; otherwise, it's sent to one of the other two SynthPatches.

The SynthInstrument keeps a count of the number of SynthPatches it has manually allocated for each PatchTemplate. The count for a particular PatchTemplate is returned by the method **synthPatchCountForPatchTemplate:**. The **synthPatchCount** method, used in an example in the previous section, returns the count for the default PatchTemplate.

If the SynthPatch is in automatic mode, you don't need to specify which PatchTemplate to use. The SynthInstrument automatically creates a SynthPatch with the correct PatchTemplate to accommodate the parameters in the Notes it receives.

Talking to the SynthPatches

A SynthInstrument's primary task is to forward Notes to its SynthPatches. The SynthPatch class defines three methods that are designed to be invoked by a SynthInstrument for this purpose:

- **noteOn:** is used to forward a noteOn type Note.
- **noteUpdate:** forwards noteUpdates.
- **noteOff:**, likely enough, forwards noteOffs.

The Note itself is passed as the argument to the method. Invocation of these methods is automatic; when a SynthInstrument receives a Note during a performance, it automatically invokes the appropriate method.

Notice that noteDurs and mutes aren't included in this scheme. A noteDur is split into a noteOn/noteOff pair. If the noteDur doesn't have a noteTag, a unique noteTag is created and given to the two new Notes. Mutes are simply ignored.

Besides forwarding Notes, the **noteOn:** and **noteOff:** methods also set a SynthPatch's *synthesis status*. This describes the object's current synthesis state as one of the following MKSynthStatus constants:

Constant	Meaning
MK_idle	The SynthPatch is currently inactive.
MK_running	The SynthPatch is synthesizing the body of a musical note.
MK_finishing	The SynthPatch is in the release portions of its Envelopes.

The **noteOn:** message sets the synthesis status to MK_running; **noteOff:** sets it to MK_finishing. In either of these states, the SynthPatch is considered to be active. The status is set to MK_idle when the release portion of the SynthPatch's amplitude Envelope (in other words, the object set as the value of the MK_ampEnv parameter of the Note that the SynthPatch is realizing) is complete. None of the other Envelopes are taken into consideration when determining if the SynthPatch is idle: It's assumed that the amplitude Envelope will ultimately fall to an amplitude of 0.0, thus whatever course the other Envelopes take after that point is for nought since the SynthPatch will no longer be making any noise.

The Update State

The SynthInstrument class gives special consideration to a noteUpdate that doesn't have a note tag:

- The noteUpdate is forwarded to all the active SynthPatches (within the SynthInstrument that received the Note).

- The Note's parameters are stored in the SynthInstrument's *update state*.

Every SynthInstrument has an update state. When a SynthInstrument begins a new phrase, the parameters in the update state are merged into the Note that initiated the phrase (always a noteOn, whether by nature or due to a noteDur split). The update state parameters never overwrite the value of a parameter already present in a noteOn—in the case of a parameter collision, the noteOn's parameter takes precedence. Conversely, a noteOn's parameters never affect the update state, it can only be changed by another noteUpdate with no note tag.

As a demonstration of these principles, consider the following scorefile excerpt:

```
/* Scorefile body excerpt. */
t 0.0;
part1 (noteUpdate) amp:.25;

t 1.0;
part1 (noteOn 1) freq:c4;     /* amplitude is .25 */
t 2.0;
part1 (noteOff 1);
t 3.0;
part1 (noteOn 2) freq:d4 amp:.75; /* amplitude is .75 */
t 4.0;
part1 (noteOff 2);

t 5.0;
part1 (noteOn 3) freq:e4;     /* amplitude is, once again, .25 */
t 6.0;
part1 (noteUpdate) amp:.5;
t 7.0;
part1 (noteUpdate 3) amp:.25;
t 8.0;
part1 (noteOff 3);

t 9.0;
part1 (noteOn 4) freq: f4;    /* amplitude is .5 */
t 10.0;
part1 (noteOff 4);
```

The initial note tag-less noteUpdate sets the amplitude parameter in the SynthInstrument's update state; notice that the update state is set even though the SynthInstrument doesn't have any active SynthPatches. Of the four subsequent noteOns, the first, third, and fourth don't have amplitude parameters so they inherit the one in the update state. The second noteOn has its own amplitude; it ignores the parameter in the update state.

While the third musical note is sounding, two more noteUpdates arrive. The first has no note tag, so it affects both the active SynthPatch and the update state. The second noteUpdate's note tag matches that of the active SynthPatch; it's forwarded to the SynthPatch but doesn't affect the update state.

Preempting a SynthPatch

While the DSP makes a great synthesizer, its resources are by no means unlimited. It's possible to ask it to synthesize, at the same time, more Notes than it can accommodate. The number of Notes that can be synthesized at one time depends on a number of factors, the most significant being the sampling rate and the requirements of the SynthPatches that are being used. There sometimes comes a time in the life of a SynthInstrument when it tries to allocate just one more SynthPatch and finds that the well is empty.

When a SynthInstrument can't get the resources to synthesize a new Note, it tries to preempt an active SynthPatch rather than lose the Note. The following steps are taken to determine which SynthPatch to preempt:

1. The preempted SynthPatch should have sufficient resources to synthesize the Note, thus the SynthInstrument first looks for a SynthPatch that uses the same PatchTemplate that's needed to synthesize the new Note.

2. If there's more than one such SynthPatch, the one that first received a **noteOff:** message, if any, is preempted. In other words, the preemption scheme first looks for a SynthPatch whose synthesis status is MK_finishing.

3. If there aren't any finishing SynthPatches, the oldest SynthPatch is chosen.

4. If a SynthPatch with the appropriate PatchTemplate isn't available, the SynthInstrument tries other SynthPatches until one is found that has sufficient resources to synthesize the new Note.

A SynthInstrument object can only preempt its own SynthPatches—it can't steal one from another SynthInstrument. The search for a preemptible SynthPatch is sometimes unsuccessful; for example, if there are no more resources to build a new SynthPatch, the new Note can't be synthesized.

Providing Your Own Preemption Scheme

The determination of which SynthPatch to preempt is performed in the **preemptSynthPatchFor:patches:** method: The method's return value is taken as the SynthPatch to preempt. If you want to provide your own system for preempting SynthPatches, you have to create your own subclass of SynthInstrument in which to reimplement this method.

The method is automatically invoked when the SynthInstrument has to preempt a SynthPatch. The two arguments are:

- The newly arrived Note
- The first in a list of candidate SynthPatches

Notice that the second argument is a single SynthPatch object. To get to the next object in the list, you send **next** to the SynthPatch at hand.

Midi

The Midi class isn't a true Instrument—it inherits from Object. However, it creates NoteReceivers and implements **realizeNote:fromNoteReceiver:** and so can be used as an Instrument.

A Midi object creates 17 NoteReceivers, one for the Basic Channel and one each for the 16 Voice Channels. You can retrieve the NoteReceiver that corresponds to a particular channel through the **channelNoteReceiver:** message, passing the channel number that you want: 0 retrieves the NoteReceiver for System and Channel Mode Messages; 1-16 retrieves the NoteReceiver for the corresponding Voice Channel. You create a Midi object to correspond to a serial port, specified as "midi0" or "midi1," in the **allocFromZone:onDevice:** method.

When it receives a Note, the Midi object translates it into a series of MIDI messages, based on the Note's note type and parameters that it contains. It then sends the messages out the serial port.

A Midi object has a device status that's much like the device status of the Orchestra object, as described in Chapter 4. Before you can send Notes to a Midi object (more accurately, before the object will do anything with these Notes), you must open and run it, through the **open** and **run** methods.

PartRecorder and ScoreRecorder

PartRecorder is a fairly straightforward Instrument: The Notes that it receives through its single NoteReceiver are copied and added to the Part with which it's associated. The NoteReceiver is created automatically; the Part must be set by your application, through the **setPart:** method.

A PartRecorder sets a Note's time tag to the current time as it receives the Note; the measure of the current time is either in beats or seconds, depending on the value of its "time unit." You can set the time unit through the **setTimeUnit:** method, passing either MK_second or MK_beat as the argument. Other than the manipulation of the time tag, the Note is unchanged by the PartRecorder.

ScoreRecorder isn't actually an Instrument; it's used to correspond to Score just as a PartRecorder corresponds to a Part. A ScoreRecorder actually creates and controls some number of PartRecorders, one for each Part in its Score.

ScorefileWriter

A ScorefileWriter object realizes Notes by writing them to a scorefile. It's the one Instrument defined by the Music Kit that requires you to create and add NoteReceivers from your application. Each NoteReceiver that you add corresponds to a Part that will be represented in the scorefile that's written. Typically, you name the NoteReceivers that you create; these names are used to identify the corresponding Part-representations in the scorefile:

```
/* Create a ScorefileWriter and add to it 3 NoteReceivers. */
id aSFWriter = [[ScorefileWriter alloc] init];
id fordReceiver = [[NoteReceiver alloc] init];
id pageReceiver = [[NoteReceiver alloc] init];
id quicklyReceiver = [[NoteReceiver alloc] init];

/* Name the NoteReceivers. */
MKNameObject("Ford", fordReceiver);
MKNameObject("Page", pageReceiver);
MKNameObject("Quickly", quicklyReceiver);

/* Add the NoteReceivers. */
[aSFWriter addNoteReceiver:fordReceiver];
[aSFWriter addNoteReceiver:pageReceiver];
[aSFWriter addNoteReceiver:quicklyReceiver];

/* Set the ScorefileWriter's file by name. */
[aSFWriter setFile: "Falstaff.score"];
```

Creating an Instrument Subclass

While there are no strict rules governing realization, intelligent Instrument design should follow these guidelines:

- An Instrument should realize Notes as it receives them. It's possible to design an Instrument that, for instance, reschedules its Notes to be realized later, but for the sake of generality, an Instrument should act immediately upon the Notes it receives.
- If an Instrument needs to alter or store a Note, it should create a copy of the Note and act upon the copy.
- An Instrument shouldn't be a source of Notes. The task of generating new Notes belongs to a Performer or to your application. The role of an Instrument is to respond, not to conceive. This doesn't mean that an Instrument can't create Notes, but it should only do so in response to receiving a Note.

Along with these guidelines, keep in mind that an Instrument should, if possible, create and add to itself some number of NoteReceivers, usually in its **init** method.

The Conductor Class

The Conductor class defines the mechanism that controls the timing of a Music Kit performance. This control is divided between the class object and instances of Conductor:

- The Conductor class itself represents an entire Music Kit performance. The class methods perform global operations such as setting characteristics that apply to all Conductor instances, and starting and stopping a performance.

- Each Conductor instance embodies a *message request queue*, a list of messages that are to be sent to particular objects at specific times. Most of the instance methods are designed to affect a Conductor's queue in some way. The most commonly invoked of these are the methods that enqueue message requests, and those that determine how quickly a Conductor processes the requests in its queue (in other words, the Conductor's tempo).

Conductors Created by the Music Kit

The Music Kit automatically creates two Conductor instances for you, the *clockConductor* and the *defaultConductor*:

- As its name implies, the clockConductor acts as a clock: It ticks away at a steady and immutable 60.0 beats per minute. Any timing information that's reckoned by the other Conductor instances is computed in reference to the clockConductor.

- Many applications need only a single Conductor instance (in addition to the clockConductor); the defaultConductor is created as a convenience to meet this need. The defaultConductor is more pliable than the clockConductor in that its tempo can be altered and its activities can be temporarily suspended during a performance.

The clockConductor is retrieved by sending the **clockConductor** message to the Conductor class; similarly, **defaultConductor** retrieves the defaultConductor.

The Message Request Queue

Every instance of Conductor (this includes the clockConductor and defaultConductor) maintains a message request queue. This queue consists of a list of structures, each of which encapsulates a request for a message to be sent to some object. Every request is given a timestamp that indicates when its message should be sent. The requests in a message request queue are sorted according to these timestamps. When a performance starts (through the **startPerformance** class method), the Conductor instances begin processing their message queues, sending the requested messages at the appropriate times.

Note: The structures in the message request queues are of type MKMsgStruct. All the fields of this structure are private: You can examine them, but you should never alter their values directly. Detailed knowledge of the MKMsgStruct isn't necessary. The structure is defined without further explanation in the file **/usr/include/musickit/Conductor.h**.

Creating and Scheduling a Message Request

To enqueue a message request with a Conductor, you invoke the **sel:to:atTime:argCount:** or **sel:to:withDelay:argCount:** method. The arguments to these methods are similar:

Keyword	Argument
sel:	Selector that identifies the method you wish to invoke
to:	The object that implements the desired method
atTime: *or* withDelay:	The time at which you wish the method to be invoked
argCount:	The number of method arguments, followed by the arguments themselves, separated by commas

The difference between the two methods is the manner in which the time argument is interpreted. A message request enqueued through the **...atTime:...** method is sent at the specified time measured from the beginning of the performance. If you use the **...withDelay:...** method, the requested message is sent after the specified amount of time has elapsed since the **sel:to:withDelay:argCount:** method itself was invoked (given that a performance is in progress). Invoked before a performance begins, the two methods are identical.

Once you've made a message request through one of these methods, you can't rescind the action; if you need more control over message requests—for example, if you need to be able to reschedule or to remove a request—you should use the following C functions:

- **MKNewMsgRequest(double** *time*, **SEL** *selector*, **id** *receiver*, **int** *argCount*,...) creates a new MKMsgStruct structure and returns a pointer to it. The arguments are similar, although in a different order, to those of the **sel:to:atTime:argCount:** method.

- **MKScheduleMsgRequest(MKMsgStruct** **aMsgStructPtr*, **id** *conductor*) places the structure pointed to by *aMsgStructPtr*, which was previously created through **MKNewMsgRequest()**, in *conductor*'s message request queue.

- **MKRepositionMsgRequest(MKMsgStruct** **aMsgStructPtr*, **double** *time*) repositions a message request within a Conductor's queue. The value of the *time* argument is absolute: It indicates the request's new position as the number of beats since the beginning of the performance.

- **MKCancelMsgRequest(MKMsgStruct** **aMsgStructPtr*) removes a message request.

The Conductor class provides two special message request queues, one that contains messages that are sent at the beginning of a performance and another for messages that are sent after a performance ends. The class methods **beforePerformanceSel:to:argCount:** and **afterPerformanceSel:to:argCount:** enqueue message requests in the before- and after-performance queues, respectively.

Controlling a Performance

As previously mentioned, a Music Kit performance starts when the Conductor class receives the **startPerformance** message. This starts the Conductor's clock ticking (as represented by the clockConductor). If you're synthesizing music on the DSP or sending messages to an external MIDI synthesizer, you should send the **run** message to the Orchestra class or to your Midi object at virtually the same time that you invoke **startPerformance**:

```
/* Start Midi, the DSP, and the performance at the same time. */
[aMidi run];  /* assuming aMidi was previously created */
[Orchestra run];
[Conductor startPerformance];
```

When it receives **startPerformance**, the Conductor class sends the messages in its before-performance queue and then the Conductor instances start processing their individual message request queues. As a message is sent, the request that prompted the message is removed from its queue. The performance ends when the Conductor class receives **finishPerformance**, at which time the after-performance messages are sent. Any message requests that remain in the individual Conductors' message request queues are removed. Note, however, that the before-performance queue isn't cleared. If you invoke **beforePerformanceSel:to:argCount:** during a performance, the message request will survive a subsequent **finishPerformance** and will affect the next performance.

By default, if all the Conductors' queues become empty at the same time (not including the before- and after-performance queues), **finishPerformance** is invoked automatically. This is convenient if you're performing a Part or a Score and you want the performance to end when all the Notes have been played. However, for many applications, such as those that create and perform Notes in response to a user's actions, universally empty queues aren't necessarily an indication that the performance is over. To allow a performance to continue even if all the queues are empty, send **setFinishWhenEmpty:NO** to the Conductor class.

While a performance is in progress, you can pause all Conductor's by sending **pausePerformance** to the Conductor class. A paused performance is resumed through the **resumePerformance** method. Individual Conductor objects can be paused and resumed through the **pause** and **resume** methods.

Setting the Tempo

A Conductor's tempo controls the rate with which it processes the requests in its message request queue. Two methods are provided for setting a Conductor object's tempo:

- **setTempo:**, which takes a **double** argument, sets the tempo in beats-per-minute.
- **setBeatSize:** also takes a **double**, but it sets the tempo by defining the duration, in seconds, of a single beat.

Regardless of which method you use to set the tempo, the values returned by the retrieval methods **tempo** and **beatSize** are computed appropriately, as shown in the following example:

```
double bSize;

/* Sets the defaultConductor's tempo. */
[[Conductor defaultConductor] setTempo:240.0];

/* Return the beat size; bSize will be 60.0/240.0, or 0.25. */
bSize = [[Conductor defaultConductor] beatSize];
```

You can change a Conductor's tempo at any time, even during a performance. If your application requires multiple simultaneous tempi, you need to create more than one Conductor, one for each tempo. A Conductor's tempo is initialized to 60.0 beats per minute.

Locking the Performance

Every Conductor instance has a notion of the current time, measured in beats. This notion is updated by the Conductor class only when a message from one of the request queues is sent; *all* Conductors are updated when *any* Conductor sends such a message. If your application sends a message (or calls a C function) that depends on a Conductor's notion of time being current, you must first send **lockPerformance** to the Conductor class. Every invocation of **lockPerformance** should be balanced by an invocation of **unlockPerformance**. For example, if you send **receiveNote:** to an Instrument's NoteReceiver, you must bracket the message with **lockPerformance** and **unlockPerformance**. (However, invocations of **receiveNote:** that are requested through a Conductor's messge request queue shouldn't be bracketed by these methods.)

Conductors and Notes

Every Note is associated with a Conductor object. By default, a Note is associated with the defaultConductor. You can't change a Note's Conductor directly; only a Performer object can do that (as described in the section on Performers).

The association between a Note and a Conductor is of particular importance if the Note is a noteDur that's sent to a SynthInstrument or Midi object. Both of these Instruments split a noteDur into a noteOn/noteOff pair. The noteOn is realized immediately and the noteOff is scheduled for realization at a later time, as indicated by the original Note's duration value. To do this, a request for the noteOff to be sent in a **receiveNote:** message is enqueued with the Note's Conductor. The exact time at which the Note arrives depends, therefore, on this Conductor's tempo.

A Note's Conductor is also important if you send the Note to an Instrument through NoteReceiver's **receiveNote:atTime:** or **receiveNote:withDelay:** methods. These methods cause the NoteReceiver to enqueue a **receiveNote:** request with the Note's Conductor at the specified time: The former method takes the **atTime:** argument as an absolute measure from the beginning of the performance, while the latter measures the **withDelay:** argument as some number of beats from the time that it's invoked.

Conductors and Envelopes

The relationship between an Envelope and a Conductor is as important as it is inflexible: The dispatching of an Envelope's breakpoints during DSP synthesis is always done through message requests with the clockConductor. You don't have to do anything to obtain this behavior, it happens automatically through **MKUpdateAsymp**(), the function that you use in the design of a SynthPatch subclass to apply an Envelope to a synthesis patch.

This association is particularly important not for the particular Conductor with which the breakpoints are scheduled, but that they are scheduled at all. Since the clockConductor handles breakpoint dispatching, this means that its queue may be filled with breakpoint messages without you knowing it. As a result, if you set the performance to finish when the queues are empty, the performance won't finish until all the breakpoint messages are sent from the clockConductor's queue. This is generally desirable behavior. Where things can become confusing is if you pause an entire performance (through Conductor's **pausePerformance** class method) while Envelopes are being handled. Not only will all Note handling stop, all Envelopes will freeze as well. This usually isn't pleasant.

One way to avoid the problem is to pause all your Conductor objects, through the **pause** method, rather than pause the entire performance.

The Performer Class

A Performer object does two things:

- It generates or otherwise obtains a series of Note objects during a performance.
- It acts as a cover for Conductor's scheduling mechanism by sequentially and repeatedly requesting invocations of its own **perform** method.

The Performer class itself is abstract; you create a subclass of Performer to correspond to a unique sources of Notes. The Music Kit includes subclasses that read Notes from a Part (PartPerformer) or a scorefile (ScorefilePerformer)—the latter actually inherits from the FilePerformer class, a subclass of Performer that defines methods for managing files. The Music Kit also includes pseudo-Performers that fashion Notes from MIDI input (Midi), and that read Notes from a Score (ScorePerformer, which creates a PartPerformer for each Part in the Score). Consult Chapter 2 of *Reference* for further descriptions of these classes.

Using a Performer object is quite simple; creating your own subclass is a bit more complicated and requires a firm understanding of how a Performer goes about its business. These two topics are presented below.

Using a Performer

To use a Performer, you need to do two things: connect it to an Instrument and tell it to go. Every Performer contains some number of NoteSenders, auxiliary objects that are created by the Performer to act as Note "spigots." NoteSenders are analogous to an Instrument's NoteReceivers.

To connect a Performer to an Instrument, you retrieve a NoteSender and NoteReceiver from either, respectively, and connect these objects through the **connect:** method, as defined by both NoteSender and NoteReceiver. For example, to connect a PartPerformer to a SynthInstrument, you would do the following:

```
/* aPartPerformer and aSynthIns are assumed to exist. */
[[aPartPerformer noteSender] connect:[aSynthIns noteReceiver]];
```

Since both classes define the **connect:** method, the following is equivalent:

```
/* aPartPerformer and aSynthIns are assumed to exist. */
[[aSynthIns noteReceiver] connect:[aPartPerformer noteSender]];
```

The **noteSender** method returns one of a Performer's NoteSenders, just as the **noteReceiver** method retrieves one of an Instrument's NoteReceivers. If you're using a Music Kit Performer subclass, you should refer to its description to determine if it creates more than one NoteSender. If it creates only one, then the **noteSender** method is sufficient. If it creates more than one, you can retrieve the entire set as a List through the **noteSenders** method and then choose the NoteSender that you want by plucking it from the List. A

ScorefilePerformer, for example, creates a NoteSender for each Part that's represented in its scorefile.

Activating a Performer

To make a Performer run, you send it the **activate** message. This prepares the object for a performance but it doesn't actually start performing Notes until you send **startPerformance** to the Conductor class. If you invoke **activate** while a performance is in progress (in other words, *after* you send **startPerformance**), the Performer will immediately start running. In addition, the Performer may require subclass-specific preparation; for example, you have to set a PartPerformer's Part before you send it the **activate** message.

While a Performer is running, you can pause and resume its activity through the **pause**, **pauseFor:**, and **resume** methods. To completely stop a Performer you invoke **deactivate**. In addition, all Performers are automatically deactivated when the Conductor class receives the **finishPerformance** message. A Performer can be given a delegate object that can be designed to respond to the messages **performerDidActivate:**, **performerDidPause:**, **performerDidResume:**, and **performerDidDeactivate:**. These messages are sent by the Performer at the obvious junctures in its performance.

Performers and Conductors

Every Performer object is associated with a Conductor. If you don't set a Performer's Conductor explicitly (through **setConductor:**), it will be associated with the defaultConductor. The rate at which a Performer performs its Notes is controlled by its Conductor's tempo. In general, all the Performers you create can be associated with the same Conductor. The only case in which a Performer demands its own Conductor is if you want the Performer to proceed at a different tempo from its fellow Performers.

Creating a Performer Subclass

The design of a Performer subclass must address three tasks: acquiring a Note, sending it into a performance, and scheduling the next Note.

Acquiring Notes

Each subclass of Performer defines a unique system for acquiring Notes. You can design your own Performers that, for example, read Notes from a specialized database or create Notes algorithmically. Regardless of how a Performer acquires its Notes, it does so as part of the implementation of its **perform** method.

The **perform** method can be designed to acquire any number of Notes with a single invocation.

Sending Notes

To send a Note into a performance, a Performer relies on its NoteSender objects. A Performer creates and adds some number of NoteSenders to itself, usually as part of its **init** method. NoteSenders are created through the usual sequence of **alloc** and **init** messages; they're added to a Performer through Performer's **addNoteSender:** method. A Performer can add any number of NoteSenders to itself, although it's anticipated that most Performers will need only one.

As part of its implementation of the **perform** method, a Performer passes the Note it has acquired as the argument in a **sendNote:** message, which it sends to its NoteSenders. Each NoteSender then relays the Note to the NoteReceivers to which it's connected; each NoteReceiver passes the Note to the Instrument that it (the NoteReceiver) belongs to. Thus, by sending **sendNote:** to a NoteSender, a Performer communicates Notes to one or more Instruments. If more than one Note is acquired in the **perform** method, each is sent in a separate **sendNote:** message.

Note: Methods that are invoked from within the **perform** method—and this includes the **sendNote:** method—*shouldn't* be bracketed by **lockPerformance** and **unlockPerformance**.

Scheduling Notes

As described above, every time a Performer receives the **perform** message it acquires a Note and then sends it to its NoteSenders. The final obligation of the **perform** method is to schedule its own next invocation. This is done by setting the value of the **nextPerform** instance variable. The value of **nextPerform** is measured in beats according to the tempo of the Conductor and, most important, it's taken as a time delay: If you set **nextPerform** to 3.0, for example, the **perform** method will be invoked after 3.0 beats.

To get things started, a Performer's first **perform** message is automatically scheduled to be sent just after the Performer is activated. You can delay this initial invocation by setting the **nextPerform** variable from within the **activateSelf** method. The default implementation of **activateSelf** does nothing; a subclass can implement it to provide pre-performance initialization just such as this.

An important implication of this scheduling mechanism is that a Performer must be able to determine when it wants to perform its next Note at the time that it acquires and performs its current Note.

Fine-Tuning a Performance Application

Responsiveness

The responsiveness of a performance to the user's actions depends on whether the Conductor class is clocked or unclocked, and upon the value of the performance's *delta time*. By default, the Conductor class is clocked. This means that the messages in the message request queues are sent at the times indicated by their timestamps. When the Conductor class is clocked, a running Application object must be present (unless the performance is being run in a separate thread, as described below).

If you don't need interactive control over a performance, you may find it beneficial to set it to unclocked by sending **setClocked:NO** to the Conductor class. In an unclocked performance, messages in the message request queues are sent one after another as quickly as possible, leaving it to some other device—the DSP or the MIDI device driver—to handle the timing of the actual realization.

Setting the delta time further refines the responsiveness of a performance. Delta time is set through the **MKSetDeltaT()** C function; the argument defines an imposed time lag, in seconds, between the Conductor's notion of time and that of the DSP and MIDI device drivers. It acts as a timing cushion that can help to maintain rhythmic integrity by granting your application a sort of computational head start: As you set the delta time to larger values, your application has more time to process Notes before they are realized. However, this computational advantage is obtained at the expense of degraded responsiveness. Choosing the proper delta time value depends on how responsive your application needs to be. For example, if you are driving DSP synthesis from MIDI input, a delta time of as much as 10 milliseconds (0.01 seconds) is generally acceptable. If you are adjusting Note parameters by moving a Slider with the mouse, a delta time of 100 milliseconds or more can be tolerated. Finding the right delta time for your application is largely a matter of experimentation.

Separate-Threaded Performance

To enhance the efficiency of a performance, you can run it in its own thread. This is done by sending **useSeparateThread:YES** to the Conductor class. Running a performance in its own thread separates it from the main event loop, thus allowing music to play with greater independence from your application's other computations. However, certain restrictions must be adhered to when running a performance in its own thread:

- You can't use ScorefileWriter or ScorefilePerformer objects in the performance.
- You can't invoke an Orchestra method that changes the Orchestra's status; these are **open**, **run**, **stop**, **close**, and **abort**.

- You can't send messages to instances of classes defined by the Application Kit, or to instances of the Sound Kit's SoundView and SoundMeter classes. In addition, you can't read or write soundfiles, or play or record sounds through an instance of the Sound Kit's Sound class (note, however, that you *can* use the sound library functions).

- You can't call DPS client functions.

- You can't call NXStreams functions.

- You can't call C functions that rely on standard input and output; these are functions such as **printf()** and **scanf()**. Because of this, DSP error logging and Music Kit tracing can't be used. In addition, if you need to handle Music Kit errors, you must provide your own error handler function through **MKSetErrorProc()**.

These restrictions apply only to that part of your application that's running in the performance thread; specifically, messages sent by a Conductor through its message request queue, and method invocations and C function class that are part of the design of a Performer or Instrument (or pseudo-Performers such as Midi) must follow these restrictions. For example, you can't use an Instrument that sends messages to an Application Kit Window object; however, you can send messages to the Window from your application's main thread.

The performance thread can cause a restricted method to be invoked or a restricted function to be called by sending a Mach message to a message port. To do this, you must first register the port through **DPSAddPort()** in the main thread. This is demonstrated in the Ensemble programming example (**/NextDeveloper/Examples/MusicKit/Ensemble**).

An important restriction in a multi-threaded performance is that *all* messages (or groups of messages) to Music Kit objects sent from the main thread should be bracketed with **lockPerformance** and **unlockPerformance**.

Performance Priority

Give your application an unfair advantage through the **setTheadPriority:** Conductor class method. This method sets the Mach-scheduling priority of the performance thread, whether or not it's separate. Performance priority values are between 0.0 and 1.0, where 0.0 is unheightened (the default) and 1.0 is the maximum priority for a user process. Normally, Mach priorities degrade over time; you can subvert this degradation by giving ownership of your application to **root** and setting the application's protection to include the **set user ID** bit. In a Terminal window, you would type the following:

```
su root
chown root yourAppHere
chmod u + s yourAppHere
```

Performing with the DSP

You can also shape your performance's capabilities by affecting the Orchestra class, thus influencing the manner in which DSP resources are used.

Sampling Rate

The DSP can output stereo samples at two rates, 22050 samples per second, or 44100 samples per second. By default, it runs at the low sampling rate. You can improve a performance's response time with regard to DSP synthesis by using the high sampling rate, as accomplished by sending the message

```
[Orchestra setSamplingRate:44100.0];
```

However, by asking the DSP to run at a higher sampling rate, you rob it of some of its power. In general, the DSP can be considered to be twice as "big" at the low sampling rate as at the high. In other words, if the DSP is able to synthesize twelve simultaneous voices at the low sampling rate using a particular SynthPatch, it may only be able to synthesize six such voices at the high sampling rate.

Sound Buffer Sizes

While the speed of the DSP makes real-time synthesis approachable, there's always an imposed time delay that's equal to the size of the buffer used to collect computed samples before they're shovelled to the DAC. To accommodate applications that require the best possible response time (the time between the initiation of a sound and its actual broadcast from the DAC), a smaller sample output buffer can be requested by sending the **setFastResponse:YES** message to an Orchestra. However, the more frequent attention demanded by the smaller buffer will detract from the DSP's synthesis computation and, again, fewer simultaneous voices may result.

Headroom

The Orchestra doesn't know, at the beginning of a Note, if the DSP can execute a given set of UnitGenerators quickly enough to produce a steady supply of output samples for the entire duration of the Note. However, it makes an educated estimate and will deny allocation requests that it thinks will overload the DSP and cause it to fall out of real time. Such a denial may result in a smaller number of simultaneously synthesized voices.

You can adjust the Orchestra's DSP processing estimate, or *headroom*, by invoking the **setHeadroom:** Orchestra method. This takes an argument between –1.0 and 1.0; a negative headroom allows a more liberal estimate of the DSP resources—resulting in more simultaneous voices—but it runs the risk of causing the DSP to fall out of real time.

Conversely, a positive headroom is more conservative: You have a greater assurance that the DSP won't fall out of real time but the number of simultaneous voices is decreased. The default is a somewhat conservative 0.1.

Chapter 6
Array Processing

Chapter 6
Array Processing

An *array processor* is a digital hardware device capable of performing mathematical computations on arrays of data with tremendous efficiency and speed. This chapter outlines the design of C programs that use the Motorola DSP56001, or *DSP*, as an array processor. In addition, the C functions provided by the NeXT DSP system and array processing libraries are summarized, and instructions for creating your own array processing functions (from DSP56001 assembly language macros) are given.

Digital signal processing is a type of array processing that performs computations on signal data. Signals are typically one-dimensional streams produced by measuring the changes in physical phenomena, such as sound, over a period of time. The DSP is used as a signal processor when, for example, it synthesizes music or converts sound data for playback. Real-time signal processing with the DSP is discussed at the end of this chapter.

Design Philosophy

Traditionally, array processors have been employed in fields such as weather forecasting and other physical modeling pursuits to perform "number crunching" on large amounts of data. To attain the speed necessary to process these mountains of data within a tolerable time, array processors use parallel hardware and a computational pipeline in which numerical operations, data input/output, program fetching, and address updating are all done in parallel. Furthermore, an array processor is generally provided as a peripheral device with its own private high-speed memory, allowing it to work efficiently and exclusively on the task at hand. After initiating a process on an array processor, the host processor is free to continue with other chores, unburdened by the array processing operations.

The fundamental difference between array processing and more conventional mathematical computing is the explicit use of arrays—rather than individual numbers—as the primitive objects upon which the computations are performed. The speed and parallelism provided by an array processor are made possible by the use of arrays: Because array elements are stored contiguously, they can be fetched in parallel with numerical computations, and the address of the next element can be automatically computed.

NeXT computers use the Motorola DSP56001 as a general-purpose, fixed-point array processor. This state-of-the-art microprocessor can execute up to 12.5 million instructions per second and, with a single instruction, can perform a 24- by 24-bit fixed-point multiply, a 48- plus 56-bit addition, two 16-bit address updates, and three parallel memory moves.

In addition, 8K 24-bit words of zero-wait-state static RAM are available to the DSP for private data storage. Thus, the DSP provides the architectural features of an array processor.

Fundamental array processing operations, such as vector add and matrix multiply, are provided as C functions in the NeXT array processing library. A key design element is the extensibility of this software. It's a simple matter to create your own array processing function from a DSP assembly language macro by using the **dspwrap** utility. Creating your own macro requires familiarity with DSP assembly language; to help acquaint you with this language, NeXT provides a number of macros in source code form—many of which were used to generate the array processing functions—as programming examples. You can write and wrap your own DSP macros, or you can combine the macros provided by NeXT to create more complex and efficient array processing functions.

Creating an Array Processing Program

Programs that access the DSP as an array processor follow a basic five-step design:

1. Initialize the DSP.
2. Transfer data from host memory to DSP memory.
3. Download and perform array processing functions on the data in the DSP.
4. Transfer the processed data back to host memory.
5. Free the DSP.

The DSP is assigned to a single program until it's done processing that program's data. Initializing the DSP will assign it (step 1); no other program will be able to use the DSP until it's freed (step 5). This means, for example, that you can't launch an application that synthesizes music or that depends on the DSP for sound playback conversion while you're running your array processing program.

A program running on the host processor not only sends data to the DSP (step 2), it also sends instructions for processing the data (step 3). After the DSP has completed its processing, the program must ask for the processed data to be sent back to the host (step 4).

The following program fragment demonstrates the five main steps involved in an array processing program:

```
/*
 * Perform array processing operations on two arrays (a[] and b[]),
 * returning results to a third array (c[]).  To compile:
 *     cc thisFile.c -larrayproc -ldsp_s -lsys_s
 */

#import <dsp/arrayproc.h>
#define N 200                    /* number of elements in each array */
#define AADR DSPAPGetLowestAddress()  /* address of a in DSP memory */
#define BADR (AADR+N)            /* address of b in DSP memory */
#define CADR (BADR+N)            /* address of c in DSP memory */
#define INC 1                    /* element increment for all arrays */

main() {
/* The arrays are declared to contain data of type float. */
    float  aArray[N], bArray[N], cArray[N];

    /* Place initial values in aArray[] and bArray[]. */

    . . .

    /* Step 1, initialize the DSP. */
    DSPAPInit();

    /* Step 2, transfer the data arrays to the DSP. */
    DSPAPWriteFloatArray(aArray, AADR, INC, N);
    DSPAPWriteFloatArray(bArray, BADR, INC, N);

    /*
     * Step 3, download and perform array processing functions on the
     * data.  As an example, the vector plus vector function is used
     * here.
     */
    DSPAPvpv(AADR, INC, BADR, INC, CADR, INC, N);

    /* Step 4, return the result to the host. */
    DSPAPReadFloatArray(cArray, CADR, INC, N);

    /* Step 5, release the DSP. */
    DSPAPFree();

    /* Do something interesting with cArray[]. */
    . . .
}
```

Programming Examples

The directory **/NextDeveloper/Examples/DSP/ArrayProcessing** contains example array processing programs in the following subdirectories:

apsound	Time-reverse a sound file.
matrix	Multiply two matrices.
fdfilter	Perform convolution in the frequency domain.

In addition, the subdirectory **libap** demonstrates how to create your own array processing library, and the **fuse** subdirectory illustrates the "fusing" of several array processing macros into a single C function.

DSP Memory Map

Figure 6-1 and Figure 6-2 show the layout of DSP memory for array processing. The actual addresses delimiting each memory region are in the header file **dsp/dsp_memory_map_ap.h**. Generally, the array processing monitor occupies the lowest and highest addresses in each space. On-chip p memory is nominally reserved for the array processing program itself, on-chip x memory is for array processing function arguments, and on-chip y memory is available for the user. Note, however, that the array processing functions provided by NeXT use only the x memory space for all data arrays.

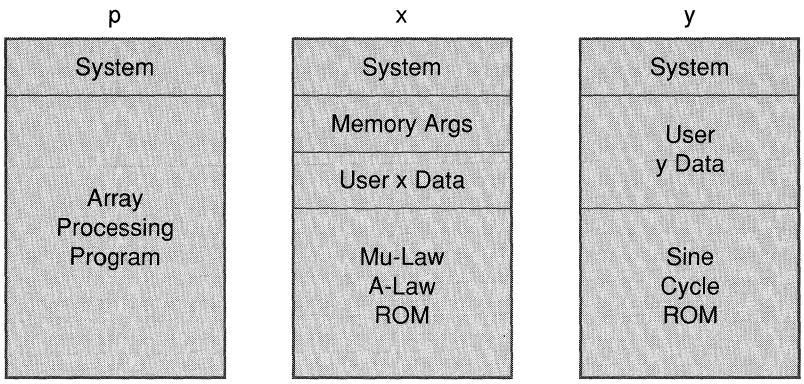

Figure 6-1. DSP Internal Memory Map

There are two images of external memory:

- In image 1 (shown below in Figure 6-2), external memory appears as one giant array that can be carved arbitrarily into x, y, and p segments.
- In image 2 (not shown), the external x and y memory banks are physically separate, as they are on the chip.

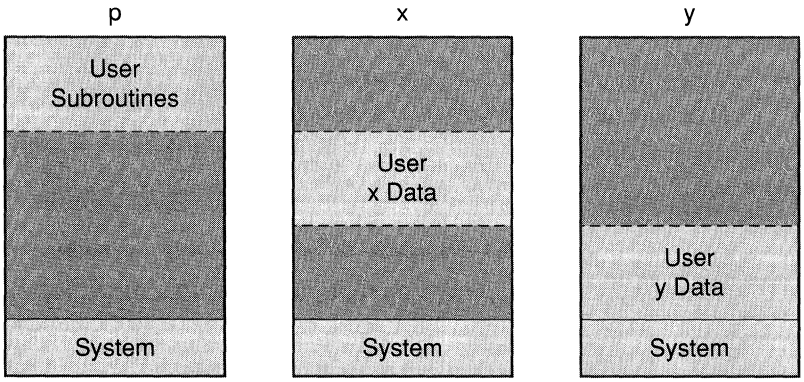

Figure 6-2. DSP External Memory Map (Image 1)

More information on the physical memory map is provided in Appendix F of *Reference*.

Data Formats

The DSP stores data in a 24-bit, fractional, two's complement, fixed-point number representation that's declared on the host as type DSPFix24. A DSPFix24 number with bits b_0, b_1, ..., b_{23} (each bit being 0 or 1) has the value

$$-b_0 + b_1 / 2 + b_2 / 4 + ... + b_{23} / 2^{23}$$

The minimum representable value is −1 (binary 10...0), and the maximum value is $1-2^{-23}$, which is often referred to as "1 minus epsilon" (binary 01...1). If n denotes a DSPFix24 number, then $-1.0 \le n < +1.0$. Inside the DSP, numbers are routinely clipped to the interval [−1,1) as they're transferred from one of the 56-bit accumulators to 24-bit memory. It's left to the programmer to ensure that computations in the DSP don't create values outside this range (other than temporarily within an accumulator).

Since a DSPFix24 number n is within the range $-1.0 \le n < +1.0$, a C **float** number f destined for the DSP must also be within $-1.0 \le f < +1.0$. A C **int** number i must be within the range $-2^{23} \le i < 2^{23}$, or between −8388608 and 8388607. On the host, a DSPFix24 number is stored as an **int;** 24 bits are right-justified in 32 bits and the sign extension isn't required.

Complex Array Format

Array processing functions that use complex data interleave the real and imaginary parts of the data. This means that arrays of complex numbers are stored in a single DSP memory space as [$x(1)$, $y(1)$, $x(2)$, $y(2)$, ..., $x(N)$, $y(N)$], where $x(i)$ is the real part and $y(i)$ is the imaginary part of the ith element of the length N complex array.

The one exception to this is the Fast Fourier Transform function, **DSPAPfftr2a()**, which requires that the real part be stored in x memory and the imaginary part in y memory.

DSP System Library

The DSP system C library, **/usr/lib/libdsp_s.a**, provides generic data transfer and conversion functions that support both array processing and the sound synthesis operations of the Music Kit. In addition, it lets you boot the DSP with an arbitrary monitor and provides functions for reading and writing the host interface, as described in Chapter 5, "Programming the DSP."

The header file **dsp/dsp.h** provides procedure prototypes (either directly or by including other such header files) for the functions in the DSP system library. The DSP system functions have the prefix "DSP".

Data Format Conversion

Data on the host must be converted to DSPFix24 format before it's transferred to the DSP. Similarly, data returned to the host by the DSP is in this format and should be converted to a data type, such as **float** or **int**, that's useful to your program.

System functions that transfer data to or from the DSP, such as **DSPAPWriteFloatArray()** and **DSPAPWriteIntArray()**, do the necessary data type conversions for you. However, for greater control you can convert and transfer data by calling two separate C functions: one that does the conversion and another that performs the transfer. The following example shows separate DSP system function calls for data conversion and data transferral:

```
/*
 * Program example demonstrating the separation of data conversion
 * from data transferral.
 */

#include <dsp/arrayproc.h>
#define N 200
#define AADR DSPAPGetLowestAddress()
#define BADR (AADR+N)
#define CADR (BADR+N)
#define INC 1
```

```
main()
{
    /* Arrays for float data. */
    float  aFloat[N], bFloat[N], cFloat[N];

    /* Analogous arrays for DSPFix24 data. */
    DSPFix24 aFix[N], bFix[N], cFix[N];

    /* Place float data in aFloat[] and bFloat[]. */
    . . .

    /* Initialize the DSP. */
    DSPAPInit();

    /* Convert data from float to DSPFix24 format. */
    DSPFloatToFix24Array(aFloat, aFix, N);
    DSPFloatToFix24Array(bFloat, bFix, N);

    /* Transfer DSPFix24 data to the DSP without conversion. */
    DSPAPWriteFix24Array(aFix, AADR, INC, N);
    DSPAPWriteFix24Array(bFix, BADR, INC, N);

    /* Perform array processing functions on the data. */
    . . .

    /* Return result data to the host and put it in c[]. */
    DSPAPReadFix24Array(cFix, CADR, INC, N);

    /* Convert DSPFix24 to float, if necessary. */
    DSPFix24ToFloatArray(cFix, cFloat, N);

    /* Free the DSP. */
    DSPAPFree();

    /* Do something interesting with cFloat[]. */
    . . .
}
```

DSPFloatToFix24Array() and **DSPFix24ToFloatArray**() are data conversion functions that take three arguments:

- Data is read from an array in host memory given as the first argument.
- The data is converted and then written to a host array given as the second argument.
- The third argument is a count of the number of elements in either array.

Analogous functions are provided to convert to and from type **int**, and to and from type **double**. Keep in mind that the data conversion functions don't scale the data for you; values that aren't within the bounds described in the section "Data Formats" above are clipped.

Note: int arrays can be cast to type DSPFix24 and written to the DSP without explicit conversion; the uppermost byte of each **int**—which, if properly scaled, should only contain the sign extension—is ignored during the transfer. Similarly, if data that's retrieved from

the DSP is to be interpreted as nonnegative integers, it isn't necessary to convert from DSPFix24 to **int**. On the other hand, negative integers from the DSP require sign extension in the uppermost byte, as provided by **DSPFix24ToIntArray**().

Data that's converted with a data conversion function can be transferred with the functions **DSPAPWriteFix24Array**() and **DSPAPReadFix24Array**(), as shown in the example. These are the fastest, lowest level DSP array transfer functions—no data format conversion is carried out before or after the transfer.

The procedure prototypes for the conversion functions are in **dsp/DSPConversion.h**.

Note: In addition to the DSPFix24 data type, the DSP also accepts 16-bit and 8-bit numbers, but these exist primarily for the Sound Kit. The Sound Kit uses the 16-bit format to process sound for CD-quality output and the 8-bit format for the mu-law encoded voice-quality sound input.

Array Processing Library

The array processing library, **/usr/lib/libarrayproc.a**, contains C functions that provide many of the elementary array operations needed for array processing (and signal processing) applications. In addition, the library provides functions that help communicate with and control the DSP in a manner that's best suited for array processing needs.

The array processing library functions—except those with prefix "DSPAPGet", as described below—return an error code that's 0 for success and nonzero for failure. The nature of the error is displayed if the **DSPEnableErrorFile**() function has been called, as explained in Chapter 5.

All the functions in the array processing library have the prefix "DSPAP".

System Support Functions

System support functions for array processing provide DSP control, data transfer between DSP memory and host memory, and error handling.

DSP Control Functions

DSP control functions manage the acquisition of the DSP, providing exclusive access to the DSP chip. The most important of these are **DSPAPInit**() and **DSPAPFree**():

- **DSPAPInit**() acquire and initializes the DSP for array processing.
- **DSPAPFree**() releases and resets the DSP.

The other DSP control functions, listed in the header file **dsp/DSPControl.h**, needn't be called directly if you're using **dspwrap**-generated functions. However, you can use them to implement advanced features, such as stacking of array processing functions (which are all relocatable modules) and overlapping of data transfers and program execution on the DSP.

Data Transfer Functions

These functions provide array copying between host memory and DSP memory. The functions operate on specific data types or configurations and all come in pairs, one for sending (writing) an array from the host to the DSP and another for retrieving (reading) data from the DSP to the host. They each take four arguments:

- A pointer to the array on the host
- The address of the array on the DSP
- A "skip factor," the DSP address increment used in the transfer
- The number of array elements to transfer

The data type of the first argument depends on the array that's being written or read. The other three arguments are **int**s. Keep in mind that the skip factor always applies to the array in DSP memory, whether you're reading or writing. For example, if you write an array with a skip factor of 3, the first element from the host array is written as the first word in the DSP array, the second element from the host is the fourth word on the DSP, the third host element is the seventh DSP word, and so on. Similarly, if you read an array with a skip factor of 3, the first host element is taken from the first DSP word, the second host element from the fourth DSP word, and so on. To write and read contiguous elements, use a skip factor of 1.

There are seven write/read pairs of transfer functions, representing seven different data types or configurations:

- **DSPAPWriteFix24Array()** and **DSPAPReadFix24Array()** transfer arrays of type DSPFix24. These functions are also used to transfer unpacked byte arrays. Array data on the host is right-justified in 24 bits on the DSP.

- **DSPAPWriteIntArray()** and **DSPAPReadIntArray()** transfer arrays of type **int**. The write function is the same as that for DSPFix24 arrays. The read function, on the other hand, is different from the DSPFix24 read function: The 24-bit values received from the DSP are sign-extended.

- **DSPAPWritePackedArray()** and **DSPAPReadPackedArray()** transfer arrays of data packed in type **unsigned char**. Each successive three bytes of host data is transferred to or from a single word on the DSP.

- **DSPAPWriteShortArray()** and **DSPAPReadShortArray()** transfer arrays of 16-bit data packed in type **int**. These are used primarily for processing sound data; each 32-bit word in the source array provides two successive 16-bit samples in the DSP. The DSP receives each 16-bit word right-justified in 24 bits, with no sign extension.

- **DSPAPWriteByteArray()** and **DSPAPReadByteArray()** transfer arrays of 8-bit data packed in type **unsigned char**. These are also used for processing sound data; each 32-bit word in the source array provides four successive 8-bit samples in the DSP, right-justified in 24 bits and with no sign extension.

- **DSPAPWriteFloatArray()** and **DSPAPReadFloatArray()** transfer arrays of **float**s. The data is converted to type DSPFix24 before being sent to the DSP, and converted back to **float** when read. Otherwise, these functions are the same as those that read and write **int** data. Host floating-point data must lie between −1.0 and 1.0 in order to be accurately represented in DSP fixed point.

- **DSPAPWriteDoubleArray()** and **DSPAPReadDoubleArray()** transfer arrays of **double**s. The data is converted to and from type DSPFix24. Note that double-precision offers no advantage relative to single-precision **float** data; the larger double-precision mantissa isn't representable in DSP fixed point.

Functions Returning Address Limits

The DSP memory map addresses that pertain to array processing are defined as constants in the header file **dsp/dsp_memory_map_ap.h**. The following functions return these values and are provided so you can avoid compiling in constants that may change in the future. None of these functions take arguments.

- **DSPAPGetLowestAddress()** returns the lowest address available for the user in external memory, using image 1 of the memory map. This value is represented by the constant DSPAP_XLE_USR and is always the start of external memory (0x2000 in the present hardware). Note that x, y, and p memory spaces are overlaid in image 1; in other words, DSP memory locations x:*n*, y:*n*, and p:*n* refer to the same physical memory cell for each *n* of the DSP external RAM.

- **DSPAPGetHighestAddress()** returns the highest address available for the user in image 1 of external memory (DSPAP_XHE_USR).

- **DSPAPGetLowestAddressXY()** returns the lowest address available for the user in external memory using image 2 (DSPAP_XLE_USG). This value points to the same physical location as **DSPAPGetLowestAddress()**, but in the address partition where x and y memory spaces are physically separated.

- **DSPAPGetHighestXAddressXY()** returns the highest address available for the user in the x partition of external memory, image 2 (DSPAP_XHE_USG).

- **DSPAPGetHighestYAddressXY()** returns the highest address available for the user in the y partition of external memory, image 2 (DSPAP_YHE_USG).

- **DSPAPGetHighestAddressXY()** returns the minimum of DSPAP_XHE_USG and DSPAP_YHE_USG.

- **DSPAPLoadAddress()** returns the start address of the array processing program in DSP on-chip program memory. The start address is equal to DSPAP_PLI_USR (the start of internal program memory for the user) plus the size of the preamble program that appears before the array processing program.

Array Processing Functions

The array processing functions perform manipulations on data residing in the DSP. They're generated through **dspwrap** from DSP56001 assembly language macros: **dspwrap** assembles a binary image from the macro source and wraps a C function around it. When the function is called, the binary image at its heart is downloaded to the DSP and executed there.

The source code files for the macros from which the array processing functions were generated are provided in the directory **/usr/lib/dsp/apsrc**. These files are made available as programming examples and to allow you to wrap combinations of existing macros into new functions.

Naming Conventions

The following abbreviation conventions apply to the DSP macro and array processing function names.

Abbreviation	Meaning
c	Complex
v	Vector
m	Matrix if operand, minus if operator
s	Scalar
t	Times
p	Plus
i	Immediate
b	Backwards
br	Bit-reversed
mag	Magnitude
max	Maximum
min	Minimum
lim	Limit
rand	Random
real	Real part
imag	Imaginary part

For example, the macro name **vtspv** means "vector times scalar plus vector," and **sumvmag** means "sum vector magnitudes." In addition to these abbreviations, there are two other sources of mnemonics: the DSP56001 instruction set itself (for example, **veor** means "vector exclusive or") and the DSP software published by Motorola. Any term not covered

by these conventions is spelled out in full; for example, **vsquare** stands for "vector square," which squares each element of a vector.

The following sections provide one-line summaries of the array processing functions. For brevity, each function is given by the name of its underlying DSP macro; in other words, without the "DSPAP" prefix. More detailed descriptions of the functions are provided in Volume 2, Chapter 3. A list of the calling sequences (only) for the functions appears in Volume 2, Appendix B.

Real Vector and Matrix Operations

mtm	matrix times matrix
vreal	vector real part extraction
vimag	vector imaginary part extraction
vclear	vector clear
vfill	vector fill with a constant value
vfilli	vector fill immediate (fill from argument)
vmove	vector move
vmoveb	vector move backwards
vmovebr	vector move bit-reversed
vabs	vector absolute value
vnegate	vector negate
vpv	vector plus vector
vpvnolim	vector plus vector, no limiting
vmv	vector minus vector
vtv	vector times vector (pointwise multiply)
vtvpv	vector multiply plus vector
vtvpvtv	vector multiply plus vector multiply
vtvmvtv	vector multiply minus vector multiply
vps	vector plus scalar
vpsi	vector plus scalar immediate
vts	vector times scalar
vtsi	vector times scalar immediate (scalar in argument)
vtsmv	vector times scalar minus vector
vtspv	vector times scalar plus vector
vtvms	vector times vector minus scalar
vtvps	vector times vector plus scalar

vramp	vector ramp
vrampi	vector ramp immediate (slope in argument)
vrand	vector random numbers
vreverse	vector reverse elements
vsquare	vector square
vswap	vector swap

Complex Vector Operations

The complex vector operations perform elementary operations on arrays of complex data. Complex numbers, each consisting of a real part and an imaginary part, arise naturally in the application of spectrum analysis.

cvcombine	complex vector combine (two real to complex)
cvconjugate	complex vector conjugate
cvfill	complex vector fill with a constant value
cvfilli	complex vector fill immediate (from argument)
cvmandelbrot	complex vector Mandelbrot set generator
cvmcv	complex vector minus complex vector
cvmove	complex vector move
cvnegate	complex vector negate
cvpcv	complex vector plus complex vector
cvreal	complex vector from real (zero imaginary part)
cvtcv	complex vector times complex vector
fftr2a	radix 2 FFT (requires xy memory partition)

Maximum and Minimum Operations

maxmagv	scalar maximum magnitude of vector elements
minmagv	scalar minimum magnitude of vector elements
maxv	scalar maximum of vector elements
minv	scalar minimum of vector elements
vmax	vector maximum of two vectors
vmin	vector minimum of two vectors

Vector Sum Operations

sumv	vector element sum
sumvmag	vector magnitude sum
sumvnolim	vector element sum, no limiting

Vectorized DSP Instruction Operations

This family of array processing functions lets you apply individual DSP instructions to each element (or pair of elements) in an entire vector (or pair of vectors). For example, **DSPAPvand()**, applies the DSP's **and** instruction to each successive pair of elements from the two input vectors. Note that the logical operations **and**, **or**, and **eor**, are bitwise operations on successive vector elements—as opposed to word-oriented operations—and provide a full word of output information.

vand	vector and
vor	vector or
veor	vector exclusive or
vlsl	vector logical shift left
vlsr	vector logical shift right
vasl	vector arithmetic shift left
vasr	vector arithmetic shift right

Creating New Array Processing Functions

You can create your own array processing function by writing DSP56001 assembly language and processing it with **dspwrap**. Programming examples that illustrate this process are provided in the directory **/NextDeveloper/Examples/DSP/ArrayProcessing**. The relevant examples are:

libap	Example of creating a custom library of array processing routines
fuse	Example of fusing supplied array processing macros into a single C function

The basic procedure is to copy an existing array processing macro that's closest to your needs (from the directory **/usr/lib/dsp/apsrc**), modify it to further suit your purposes, run **dspwrap** to generate the C function interface, and link the function into your own array processing library. Note that existing array processing macros can be invoked inside your new array processing macro.

The following demonstrates a typical invocation of **dspwrap**:

```
dspwrap -ap -nodoc mymacro.asm
```

The **-ap** command-line argument indicates that the program is being used to generate an array processing function. **-nodoc** suppresses automatic documentation generation. The file **mymacro.asm** contains a DSP macro for which the program creates a C function named **DSPAPmymacro()**. The C function is written to a file named **DSPAPmymacro.c**.

Further details about **dspwrap** are provided in a UNIX manual page.

Real-Time Digital Signal Processing

A *digital signal* is a sequence of numerical measurements of a physical variable, such as temperature or air pressure, over time. For example, the DSP is used as a signal processor when it synthesizes music or processes recorded sounds. Normally, a signal processing system is thought of as operating on continuous streams of signal data in real time. However, a signal can be broken into a succession of arrays, allowing it to be processed in terms of array processing operations. When implemented using array processing, the operations are performed on a succession of these arrays (or vectors), typically out of real time. It's possible to perform array-oriented operations on signals and keep up with real time, although the output is always delayed by an amount no less than the time it takes to process a single array.

The UnitGenerator class, part of the Music Kit, is designed to support the needs of real-time digital signal processing. In contrast to array processing functions, which cause DSP macros to execute inside the DSP without host interaction, the UnitGenerator class supports real-time communication between the host and the DSP. Parameters of a UnitGenerator can be updated after every "tick" of samples of the output signal have been computed. (The Music Kit presently uses 16 samples per tick.) Thus, at the expense of more DSP resources devoted to real-time communication and buffering, it's possible to configure a network of signal processing modules for real-time processing of signals coming in and going out of a DSP serial port or through the host interface.

Similar to array processing functions, UnitGenerator subclasses are built around DSP assembly language macros. By combining UnitGenerators into an Orchestra object, you can assemble a large amount of DSP code that can be sent to the DSP with a single instruction. One Orchestra can perform a series of computations that would otherwise require several array processing function calls.

For more information on the UnitGenerator class, see Chapter 3, "Music." The DSP source code for the Music Kit monitor is in **/usr/lib/dsp/smsrc/mkmon8k.asm**. It can be adapted to accommodate different tick sizes and buffer lengths.

Chapter 7
Programming the DSP

Chapter 7
Programming the DSP

This chapter explains how to access the DSP from software on a NeXT computer. A brief review of the hardware is presented, followed by a description of ways to access the DSP. Familiarity with the DSP56001, especially the host interface port, is assumed.

DSP details, including the D-15 connector pinouts, DSP memory map, and DSP instruction summary, can be found in *Reference*.

DSP Hardware

The hardware associated with the DSP includes:

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

Booting the DSP

The DSP is in the reset state while not in use by some task. When first accessed, it exits the reset state awaiting a bootstrap program. You can write a DSP program to be fed to the DSP during the bootstrap sequence. This and the following sections describe how to create, load, and communicate with your own DSP bootstrap program.

The DSP bootstrap program must be a single contiguous program segment starting at location p:0 in on-chip program memory, and it must not exceed 512 words in length (the size of on-chip program RAM). Below is an example DSP bootstrap program:

```
; boot56k.asm - Example DSP bootstrap program that goes into an
; infinite loop as it reads HRX, right-shifts one place, and
; writes HTX

        org p:0
reset   jmp >rcv_buz

        dup $40-2  ; output must be a contiguous segment
        nop
        endm

        org p:$40  ; typical starting address

rcv_buz jclr #0,x:$FFE9,rcv_buz  ; wait for data from host
        move x:$FFEB,A1

        LSR A  ; right-shift one place

xmt_buz jclr #1,x:$FFE9,xmt_buz  ; send shifted word to host
        move A1,x:$FFEB

        jmp rcv_buz

    end $40
```

DSP Assembly

To assemble this file (named **boot56k.asm**) and create the object file **boot56k.lod**, type

```
asm56000 -b -a -l -os,so boot56k
```

in a Terminal window. **asm56000** is the Motorola DSP assembler program. The command-line arguments used in the example are:

- **-b** option tells the program to create an object file.
- **-a** indicates that the object file should be absolute, or nonrelocatable. Absolute files are indicated by the extension ".lod". If **-a** is omitted, a relocatable file (extension ".lnk") is created. You can convert a ".lnk" file to a ".lod" file by using the DSP linker program, **lnk56000**.
- **-l** produces a listing file (extension ".lst"). The file contains a script of the operations performed by the assembler and is useful while debugging.
- **-os,so** causes the assembler to put symbol information in the object file. This information is used by Bug56™, the DSP symbolic debugger. Bug56 can be found in the directory **/NextDeveloper/Apps**.

DSP Tools Documentation

There are three UNIX manual pages documenting the DSP assembler **asm56000**, linker **lnk56000**, and librarian **lib56000**. Complete documentation of these tools is also provided in the directory **/NextLibrary/Documentation/Motorola**. The Bug56 application has an extensive help facility (available in the main menu), and is documented in the directory **/NextLibrary/Documentation/Ariel**.

Binary DSP Object File Format

The ".lod" object file written by the assembler is in a machine-independent ASCII format. NeXT supports a more efficient binary ".dsp" format. A ".lod" file can be converted to a ".dsp" file using the program **dspimg**. For example,

```
dspimg boot56k.lod boot56k.dsp
```

converts **boot56k.lod** to ".dsp" format. As a convenience, the extension of the input file name defaults to ".lod," and the output file name, when omitted, is derived from the input file name, so this example can also appear as

```
dspimg boot56k
```

Loading the DSP Bootstrap Program

After you've prepared the bootstrap program **boot56k.lod** or **boot56k.dsp**, you can load it into the DSP from a C program by calling the **DSPBootFile()** function. As shown in the following example, the function takes the name of the bootstrap program file as its single argument.

```
/*
 * test_boot56k.c - read and load boot56k.dsp (bootstrap file)
 * To compile and link:
 *     cc test_boot56k.c -ldsp_s -lsys_s
 */
#include <dsp/dsp.h>   /* needed by programs that use the DSP */
main()
{
    DSPBootFile("boot56k.dsp");  /* "boot56k.lod" works, too */
    /* Communicate with DSP program here */
}
```

After the call to **DSPBootFile()**, the DSP remains open and can be accessed via the Mach driver or by simple host interface programming, as discussed in the next section.

Software Access to the DSP

There are two basic ways to access the DSP:

- Reading and writing the host interface registers
- Sending and receiving Mach messages to and from the sound/DSP driver

These access modes are described below.

Host Interface Access

Reading and writing the eight bytes of the DSP host interface is the simplest and most general way to access the DSP. In this access mode, you have complete control over the DSP software. There are no programming conventions to obey and no reserved DSP resources. The disadvantages are that you must write your own DSP communication services, and the DSP can't interrupt the host processor. Lack of interrupt capability implies lack of direct memory access (DMA) transfer between the DSP and host memory. The Mach driver interface must be used to field DSP device interrupts.

Host Interface Programming Example

The following example illustrates communication with the example bootstrap program **boot56k.asm** given above.

```
/* To compile and link: cc test_getput.c -ldsp_s -lsys_s */
#include <dsp/dsp.h>
main() {
    int tval,reply;

    DSPBootFile("boot56k");

    DSPSetHF0();  /* Set HF0 */
    DSPSetHF1();  /* Set HF1 */
    DSPReadICR(&reply);       /* 8 bits, right-justified */
    printf("\n\ticr = 0x%X\n",reply);

    DSPReadISR(&reply);       /* 8 bits, right-justified */
    printf("\tisr = 0x%X\n",reply);

    tval = 0xBBCCDD;          /* test value */
    DSPWriteTX(tval);         /* 24 bits, right-justified in 32 */
```

```
    /* Wait for the data to become ready (RXDF on). */
    while (!DSPDataIsAvailable());
    while (DSPDataIsAvailable()) {
        DSPReadRX(&reply); /* 24 bits, right-justified */
        printf("\trx = 0x%X\n",reply); }

    if (reply != tval>>1)
        printf("ERROR: rx should be = 0x%X\n",tval>>1);

    DSPClose();
}
```

The output of this program should be

```
icr = 0x18
isr = 0x6
rx = 0x5DE66E
```

Note: Since IVR isn't used in the DSP host interface, there are no functions for reading and writing that register.

For convenience (and efficiency), whole arrays can be written to the transmit registers using

DSPWriteTXArray(*intArray, numberOfInts*)

and read from the receive registers of the DSP using

DSPReadRXArray(*intArray, numberOfInts*)

Each word of the transfer is conditioned on TXDE for **DSPWriteTXArray**() and on RXDF for **DSPReadRXArray**().

DSP Error Handling

Most of the DSP system functions (prefix "DSP") return an integer error code, where 0 indicates success and nonzero indicates failure. These functions also write a string describing the error to the file name passed as the single argument to **DSPEnableErrorFile**().

Restrictions on Host Interface Programming

Certain restrictions apply to the operation of the write/read primitives (which are implemented using the Mach driver):

- To write the ICR, you use the functions **DSPSetHF1()**, **DSPSetHF0()**, **DSPClearHF1()**, and **DSPClearHF0()**.

- To write the CVR, you pass a host command identifier (an integer) to the **DSPHostCommand()** function. The host commands are described in the *Motorola DSP56001 User's Manual.*

These restrictions are necessary because the DSP driver uses TREQ and HREQ for its own purposes. In particular, HREQ causes an interrupt that causes the driver to read all available words from the DSP into a kernel buffer. A call to **DSPGetRX()** actually fetches words from this buffer rather than from the DSP directly.

Host Interface Access Functions

Functions that support simple host interface programming, such as **DSPReadRX()**, are currently documented in the procedure prototypes defined in the following header files in the directory **/usr/include/dsp**:

- **dsp.h** is a master header file that pulls in all function prototypes for the DSP library (**libdsp_s.a**).

- **DSPError.h** contains prototypes for the DSP error handling functions.

- **DSPConversion.h** prototypes the functions that convert data between type DSPFix24 and **int**, **float**, and **double**.

- **DSPObject.h** prototypes the low-level DSP interface functions.

DSPObject.h contains many functions that are useful only in conjunction with the array processing or Music Kit monitors. The following sections list the functions that are useful regardless of the DSP monitor that you use.

Orienting the Host

- **DSPGetDSPCount()** returns the number of DSPs in your cube.

- **DSPSetCurrentDSP**(int *index*) and **DSPGetCurrentDSP()** set and return, respectively, the zero-based index of the DSP upon which subsequent DSP functions will act.

- **DSPSetMessagePriority**(int *priority*) and **DSPGetMessagePriority**() set and return, respectively, the priority of messages sent to the current DSP. There are three priorities, represented as the constants DSP_MSG_HIGH, DSP_MSG_MED, and DSP_MSG_LOW.

- **DSPSetOpenPriority**(int *priority*) and **DSPGetOpenPriority**() set and return, respectively, the priority with which a subsequent call to **DSPOpenNoBoot**() opens the DSP. There are two priorities: 0 is low and 1 is high. With high priority, a process can gain access to the DSP even if it has already been opened by another process. This is used mostly by the DSP debugger. The original process should be frozen while the new process steps in and looks around.

- **DSPEnableHostMsg**() and **DSPDisableHostMsg**() enable and disable, respectively, DSP host message protocol. **DSPHostMsgIsEnabled**() returns the current state of the protocol. With this protocol, DSP error messages are sent on the DSP error port. Otherwise, all messages arrive on the DSP message port.

Opening and Closing the DSP

- **DSPInit**() opens the DSP and loads a minimal, generic DSP boot program. This is the function that's most commonly called to open the DSP.

- **DSPOpenNoBoot**() opens the DSP without loading a boot program.

- **DSPOpenNoBootHighPriority**() performs a high-priority open without loading a boot program. This is normally used only by the DSP debugger.

- **DSPReset**() resets the DSP (which must already be open). A reset DSP is awaiting a bootstrap program.

- **DSPBootFile**(char **filename*) opens (if necessary) and boots the DSP from the given program file.

- **DSPBoot**(DSPLoadSpec **system*) opens (if necessary) and boots the DSP from the given program. DSPLoadSpec is defined in **/usr/lib/include/dsp_structs.h**.

- **DSPClose**() and **DSPRawClose**() close the DSP; the Raw close doesn't clean up the device.

- **DSPCloseSaveState**() and **DSPRawCloseSaveState**() are like the previous functions, but the state of the open modes are retained and used in a subsequent reopening of the DSP.

DSP Ownership Information

- **DSPIsOpen**() returns nonzero if the DSP is open.
- **DSPGetOwnerString**() returns a pointer to a string that contains information about the process that currently owns the DSP. It's in a form exemplified by the following:

```
DSP opened in PID 351 by me on Sun Jun 18 17:50:46 1989
```

- **DSPOpenWhoFile**() registers the current owner of the DSP in the DSP log file. This is called implicitly by the functions that open the DSP.
- **DSPCloseWhoFile**() deletes the DSP log file. This is called implicitly by the functions that close the DSP.

Reading and Writing DSP/Host Interface Flags

- **DSPSetHF0**(), **DSPClearHF0**(), and **DSPGetHF0**() set, clear, and return the state of HF0 (host flag 0), respectively.

An analogous set of functions is provided for HF1, and a **DSPGet...** function (only) is provided for HF2 and HF3.

Reading and Writing Interface Registers

- **DSPReadICR**(int **registerValuePtr*) reads the DSP Interrupt Control Register into the integer pointed to by the argument (8 bits, right-justified).
- **DSPGetICR**() returns the ICR register.

An analogous set of functions is provided for the Command Vector Register (CVR), and the Interrupt Status Register (ISR).

Reading and Writing Commands and Data

- **DSPHostCommand**(int *cmd*) issues the given host command.
- **DSPWriteTX**(DSPFix24 *word*) writes the low-order 24 bits of *word* into the DSP Transmit Byte registers.

- **DSPWriteTXArray**(DSPFix24 **dataPtr*, int *n*) writes *n* words from *dataPtr* into the DSP Transmit Byte registers.

- **DSPWriteTXArrayB**(DSPFix24 **dataPtr*, int *n*) writes the data backwards.

- **DSPReadRX**(DSPFix24 **wordPtr*) reads the next word from the DSP Receive Byte registers into the 24-bit word (right-justified) pointed to by the argument. The function waits for the time limit returned by **DSPDefaultTimeLimit**() to expire before giving up (and returning an error code).

- **DSPReadRXArray**(DSPFix24 **dataPtr*, int *n*) reads the next *n* words from the RX registers into *dataPtr.*

- **DSPGetRX**() returns the RX register.

Synchronization

- **DSPAwaitHC**(int *msTimeLimit*) waits for the HC bit to clear. This happens when the next instruction to be executed on the DSP is the first word of the Host Command interrupt vector. *msTimeLimit* is the maximum wait time; 0 means wait forever.

- **DSPAwaitTRDY**(int *msTimeLimit*) waits for the TRDY bit to be set.

- **DSPAwaitData**(int *msTimeLimit*) waits for the DSP to send data to the host.

- **DSPDataIsAvailable**() returns nonzero if data from the DSP is available.

Ports

The following functions return a **port_t** value; the **port_t** data type is defined in **/usr/include/sys/message.h**. The DSP must be open before you call any of these functions.

- **DSPGetOwnerPort**() returns the port that conveys DSP and sound-out ownership.

- **DSPGetHostMessagePort**() returns the port that's used to send host messages to the DSP.

- **DSPGetDSPMessagePort**() returns the port that's used to receive DSP messages sent from the DSP to the host.

- **DSPGetErrorPort**() returns the port that receives error messages from the DSP.

The Simulator

The Motorola DSP56001 simulator, **sim56000**, isn't provided by NeXT; it can be obtained directly from Motorola.

- **DSPIsSimulated**() returns nonzero if the DSP is being simulated.
- **DSPIsSimulatedOnly**() returns nonzero if the DSP simulator output is open while the DSP is closed.
- **DSPOpenSimulatorFile**(char **filename*) opens *filename* for simulator output.
- **DSPCloseSimulatorFile**() closes the simulator output file.
- **DSPStartSimulatorFP**(FILE **filePtr*) starts the simulator, with output *filePtr*.
- **DSPGetSimulatorFP**() returns a pointer to the simulator output file.
- **DSPStopSimulator**() stops the simulator.

Index

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.
2910.00
12/90