Sound, Music, and Signal Processing: Reference

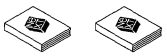
NeXT Developer's Library

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: Reference

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

Manual written by Doug Fulton
Edited by Caroline Rose, Roy West, Helen Casabona, Kathy Walrath, and Gary Miller
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 provides detailed descriptions of the Objective-C language classes, C functions, data formats, and other programming elements that make up the sound, music, and DSP software.

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
Header Files

All header files can be viewed on-line. They are located in **/usr/include** and its subdirectories. Each of the kits and the common classes have subdirectories—**appkit**, **soundkit**, **musickit**, and **objc**. Other subdirectories of interest include **streams**, **sound**, **dpsclient**, **dsp**, **sys**, and **servers**.

Chapter 2
Class Specifications

Chapter 2
Class Specifications

This chapter provides protocol information about the classes defined in the Sound Kit™ and Music Kit™. Each class is contained in a separate section wherein the class' instance variables and methods are listed and described. Familiarity with the concepts introduced in Volume 1 is assumed. A detailed explanation on how to read a class description is given in Chapter 2 of the *NeXTstep® Reference* manual.

Sound Kit Classes

The class specifications for the Sound Kit describe three classes:

Sound
SoundMeter
SoundView

The Sound class inherits from Object. SoundMeter and SoundView inherit from the Application Kit's View class.

Sound

INHERITS FROM Object

DECLARED IN soundkit.h

CLASS DESCRIPTION

Sound objects represent and manage sounds. Designed primarily to provide recording, playback, and editing of sampled sounds, a single Sound object can accommodate a number of different sound formats, including nested or multiple sounds and DSP sound synthesis program code.

The sound encapsulated in a Sound object can be recorded from CODEC microphone input, read from a soundfile or from the application's Mach-O sound segment, retrieved from the pasteboard, or created algorithmically. Whatever the source of sound, playback is usually transparent. Conversion to the format and sampling rate expected by the playback hardware is performed automatically for most Sounds; the Sound formats and sampling rates are listed below.

Both playback and recording are performed by background threads, allowing your application to proceed in parallel. Usually, an application expects a playback or recording to be immediate. The latency between sending a **play:** or **record:** message and the start of the playback or recording, while within the tolerance demanded by most applications, can be further decreased by first reserving the sound facilities that you wish to use. This is done by calling the **SNDReserve**() C function (described in Chapter 3, "C Functions").

The Sound class provides an application-wide name table called the *named Sound list* that lets you identify and locate sounds by a unique string name.

A Sound object can have a delegate, to which messages are sent when the object begins or ends playback or recording. The following messages are sent to the Sound delegate:

Message	Motivation
willPlay:	Sent just before the Sound begins playing
didPlay:	Sent when the Sound finishes playing
willRecord:	Sent before recording
didRecord:	Sent after recording
hadError:	Sent if the playback or recording generates an error

The argument for a delegate method is, of course, the Sound object that caused it to be sent.

A number of editing methods are provided, such as **insertSample:at:**, and **deleteSamplesAt:count:**. As the names imply, the editing methods only apply to Sound objects that contain sampled sound data (as opposed to DSP program code). The

isEditable method is provided as a test to determine whether the object contains editable data. Only if **isEditable** returns **YES** should an editing method be sent to the object.

To minimize data movement (and thus save time), an edited Sound may become fragmented; in other words, its sound data might become discontiguous in memory. While playback of a fragmented Sound object is transparent, it does incur some additional overhead. If you perform a number of edits—particularly near the beginning of the sound data—you may want to return the Sound to its natural, contiguous state by sending it the **compactSamples** message before you play it. However, a large Sound may take a long time to compact, so a judicious and well-timed use of **compactSamples** is advised. A fragmented Sound is ascertained by invoking the **needsToCompact** method; a return value of **YES** indicates that the receiver is fragmented. Note that a fragmented Sound is automatically compacted before it's copied to the pasteboard (through the **writeToPasteboard** method). Also, when you write a Sound to a soundfile, the data in the file is compact regardless of the state of the object.

A Sound object contains a SNDSoundStruct, the structure that describes and contains sound data and that's used as the soundfile format and the pasteboard sound type. The Sound object's **soundStruct** instance variable is a pointer to the object's SNDSoundStruct. Most of the methods defined in the Sound class are implemented so that you needn't be aware of this level of detail. However, if you wish to directly manipulate the sound data in a Sound object, you need to be familiar with the SNDSoundStruct architecture. This is described in Volume 1, Chapter 2 and outlined in the description of the **SNDAlloc()** function in Chapter 3, "C Functions."

The formats and sampling rates supported by the Sound object are the same as those defined for the SNDSoundStruct. The formats are represented as constants and fall into three groups: sampled data, DSP program code, and other formats. The sampled data formats describe the amplitude quantization of the sound data:

Sampled Data Formats	Quantization
SND_FORMAT_MULAW_8	8-bit mu-law
SND_FORMAT_MULAW_SQUELCH	8-bit mu-law with run-length encoding of silence
SND_FORMAT_LINEAR_8	8-bit linear
SND_FORMAT_LINEAR_16	16-bit linear
SND_FORMAT_LINEAR_24	24-bit linear
SND_FORMAT_LINEAR_32	32-bit linear
SND_FORMAT_FLOAT	32-bit floating point
SND_FORMAT_DOUBLE	64-bit floating point
SND_FORMAT_DSP_DATA_8	8-bit fixed point
SND_FORMAT_DSP_DATA_16	16-bit fixed point
SND_FORMAT_DSP_DATA_24	24-bit fixed point
SND_FORMAT_DSP_DATA_32	32-bit fixed point

Sound data for a DSP program consists of commands and data that can be sent to the DSP for sound synthesis. The C function **SNDReadDSPfile()** is provided to read a DSP program from a file into a sound structure.

DSP Program Formats	Meaning
SND_FORMAT_DSP_CORE	The core image for a DSP program

There are four other formats:

Other Formats	Meaning
SND_FORMAT_DISPLAY	Used to represent reduced data for display
SND_FORMAT_INDIRECT	Fragmented sampled data
SND_FORMAT_NESTED	Multiple sound structures
SND_FORMAT_UNSPECIFIED	Unknown format

The SND_FORMAT_DISPLAY format is used primarily by SoundView objects. You can't play display data.

The data in both fragmented and nested sounds contain any number of sub-structures. In a fragmented sound, the dataLocation field of the SNDSoundStruct points to a contiguous block of ordered addresses (the address list is terminated by **NULL**) each of which points to a SNDSoundStruct that contains one of the sound fragments. The data in a nested sound contains any number of (possibly fragmented) sounds. Each sound has its own SNDSoundStruct; the sound format, sampling rate, and number of channels can vary from one sound to the next. When you play a nested sound, the sounds are played back in order.

NeXT reserves the integer constants 0 through 255 to represent sound formats. You can provide your own formats represented by positive integers greater than 255. For example, you can create a format to identify data reduced models of a sound, or to store graphic information used to display reduced sound data. Personalized formats are particularly useful in a nested sound, wherein you can store original sound data using one of the Sound Kit formats along with your own versions of the data. Non-Sound Kit formats, as well as the SND_FORMAT_UNSPECIFIED format, are ignored during playback.

A Sound's data format is returned by the **dataFormat** method. Note that for a fragmented sound, the format of the actual data is returned (all the fragments have the same format, sampling rate, and number of channels). In other words, **dataFormat** never returns SND_FORMAT_INDIRECT.

The recording and playback hardware support three sampling rates, represented by the following floating point constants:

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)

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Sound	SNDSoundStruct	*soundStruct;
	int	soundStructSize;
	int	priority;
	id	delegate;
	int	status;
	char	*name;

soundStruct	The object's sound data structure.
soundStructSize	The length of **soundStruct** in bytes.
priority	The object's recording and playback priority.
delegate	The target of notification messages.
status	What the object is currently doing.
name	The object's name.

METHOD TYPES

Creating and freeing a Sound object	– free + new + newFromMachO: + newFromPasteboard + newFromSoundfile:
Finding and naming the object	+ addName:fromMachO: + addName:fromSoundfile: + addName:sound: + findSoundFor: + removeSoundForName:
Reading and writing sound data	– readSoundfile: – writeSoundfile: – writeToPasteboard
Modifying the object	– convertToFormat:samplingRate:channelCount: – setDataSize:dataFormat:samplingRate: channelCount:infoSize: – setDelegate: – setName:

Querying the object	– channelCount – compatibleWith: – data – dataFormat – dataSize – delegate – info – infoSize – isEditable – isEmpty – name – needsCompacting – sampleCount – samplesProcessed – samplingRate – soundStruct – soundStructSize – status
Recording and playing the object	– pause – pause: – play – play: – record – record: – resume – resume: – stop – stop:
Editing the sound data	– compactSamples – copySamples:at:count: – copySound: – deleteSamples – deleteSamplesAt:count: – insertSamples:at:
Archiving the object	– finishUnarchiving – read: – write:
Accessing the delegate	– tellDelegate:
Accessing the sound hardware	+ getVolume:: + setVolume:: + isMuted + setMute:

addName:fromMachO:

+ **addName:**(char *)*name* **fromMachO:**(char *)*sectionName*

Creates a Sound object from section *sectionName* in the application's Mach-O sound segment, assigns the name *name* to the object, and adds it to the named Sound list. The sound data is copied into the new Sound. If *name* is already in use, or if the section isn't found or its data can't be copied, the Sound isn't created and **nil** is returned. Otherwise, the new Sound is returned.

addName:fromSoundfile:

+ **addName:**(char *)*name* **fromSoundfile:**(char *)*filename*

Creates a Sound object from the soundfile *filename*, assigns the name *name* to the object, and adds it to the named Sound list. If *name* is already in use, or if *filename* isn't found or can't be read, the Sound isn't created and **nil** is returned. Otherwise, the new Sound is returned.

addName:sound:

+ **addName:**(char *)*name* **sound:***aSound*

Assigns the name *name* to the Sound *aSound* and adds it to the named Sound list. Returns *aSound*, or **nil** if *name* is already in use.

findSoundFor:

+ **findSoundFor:**(char *)*aName*

Finds and returns the named Sound object. First the named Sound list is searched; if the sound isn't found, then the method looks for *aName*.snd in the application's Mach-O sound segment. Finally, *aName*.snd is searched for in the following directories (in order):

- ~/Library/Sounds
- /LocalLibrary/Sounds
- /NextLibrary/Sounds

where ~ represents the user's home directory. If the Sound eludes the search, **nil** is returned.

getVolume::

+ **getVolume:**(float *)*left* **:**(float *)*right*

Returns, by reference, the stereo output levels as floating-point numbers between 0.0 and 1.0.

isMuted

+ (BOOL)**isMuted**

Returns YES if the sound output level is currently muted.

new

+ **new**

Creates and returns an empty, unnamed Sound object.

newFromMachO:

+ **newFromMachO:**(char *)*sectionName*

Creates and returns an unnamed Sound object from section *sectionName* in the application's Mach-O sound segment. The sound data is copied into the new Sound. If the section isn't found or its data can't be copied, the Sound isn't created and **nil** is returned.

newFromPasteboard

+ **newFromPasteboard**

Creates and returns an unnamed Sound object from the sound found on the pasteboard (the pasteboard can have only one sound entry at a time). The sound data is copied into the new Sound. If the Pasteboard doesn't currently contain a sound entry, the Sound isn't created and **nil** is returned.

newFromSoundfile:

+ **newFromSoundfile:**(char *)*filename*

Creates and returns an unnamed Sound object from the soundfile *filename*. The file name must be a complete UNIX path name that includes the ".snd" extension. If the file isn't be found or can't be read, the Sound isn't created and **nil** is returned.

removeSoundForName:

+ **removeSoundForName:**(char *)*name*

Removes the named Sound from the named Sound list. If the Sound isn't found, returns **nil**; otherwise returns the Sound.

setMute:

+ **setMute:**(BOOL)*aFlag*

Mutes and unmutes the sound output level as *aFlag* is YES or NO, respectively.

setVolume::

+ **setVolume:**(float)*left* **:**(float)*right*

Sets the stereo output levels. These affect the volume of the stereo signals sent to the built-in speaker and headphone jacks. *left* and *right* must be floating-point numbers between 0.0 (minimum) and 1.0 (maximum).

INSTANCE METHODS

channelCount

– (int)**channelCount**

Returns the number of channels in the receiver.

compactSamples

– (int)**compactSamples**

The receiver's sound is compacted into a contiguous block of data, undoing the fragmentation that can occur during editing. If the receiver's data isn't fragmented (its format isn't SND_FORMAT_INDIRECT), then this method does nothing. Compacting a large sound can take a long time; keep in mind that when you copy a Sound to the pasteboard, the object is automatically compacted before it's copied. Also, the soundfile representation of a Sound contains contiguous data so there's no need to compact a Sound before writing it to a soundfile simply to ensure that the file representation will be compact. An error code is returned.

compatibleWith:

– (BOOL)**compatibleWith:***aSound*

Returns **YES** if the format, sampling rate, and channel count of *aSound*'s sound data is the same as that of receiver's sound data. If one (or both) of the Sounds doesn't contain a sound (its **soundStruct** is **nil**) then the objects are declared compatible and **YES** is returned. Also, if one (or both) of the Sounds is fragmented (and so its format is SND_FORMAT_INDIRECT), then this method retrieves and compares the formats of the actual sound data.

convertToFormat:samplingRate:channelCount:

– (int)**convertToFormat:**(int)*aFormat*
samplingRate:(double)*aRate*
channelCount:(int)*aChannelCount*

Convert the receiver's sound to the given format, sampling rate, and number of channels. An error code is returned.

copySamples:at:count:

– (int)**copySamples:***aSound*
at:(int)*startSample*
count:(int)*sampleCount*

Replaces the receiver's sound data with a copy of a portion of *aSound*'s data. The copied portion starts at *aSound*'s *startSample*'th sample (zero-based) and extends over *sampleCount* samples. The receiver must be editable and the two Sounds must be compatible. If the specified portion of *aSound* is fragmented, the receiver will be fragmented. An error code is returned.

copySound:

– (int)**copySound:***aSound*

Replaces the receiver's sound data with a copy of *aSound*'s data. The receiver needn't be editable, nor must the two Sounds be compatible. An error code is returned.

data

– (unsigned char *)**data**

Returns a pointer to the receiver's sound data. You can use the pointer to examine, create, and modify the data. To intelligently manipulate the data, you need to be aware of its size, format, sampling rate, and the number of channels that it contains (a query method for each of these attributes is provided by the Sound class). The size of the data, in particular, must be respected; it's set when the receiver is created or given a new sound (through **readSoundfile:**, for example) and can't be changed directly. To resize the data, you should invoke one of the editing methods such as **insertSamples:at:** or **deleteSamplesAt:count:**. To start with a new, unfragmented sound with a determinate length, invoke the **setDataSize:dataFormat:samplingRate:channelCount:infoSize:** method. Manipulation of sound data that contains a DSP program (SND_FORMAT_DSP_CORE and SND_FORMAT_MK_DSP_CORE formats) isn't recommended. Keep in mind that the sound data in a fragmented sound is a pointer to a **NULL** terminated list of pointers to SNDSoundStructs, one for each fragment. To examine or manipulate the samples in a fragmented sound, you must understand the SNDSoundStruct structure (documented under the **SNDAlloc()** C function in Chapter 3, "C Functions").

dataFormat

– (int)**dataFormat**

Returns the format of the receiver's sound data. If the data is fragmented, the format of the samples is returned (in other words, SND_FORMAT_INDIRECT is never returned by this method).

dataSize

– (int)**dataSize**

Return the size (in bytes) of the receiver's sound data. If you modify the data (through the pointer returned by the **data** method) you must be careful not to exceed its length. If the sound is fragmented, the value returned by this method is the size of the receiver's **soundStruct** and doesn't include the actual data itself.

delegate

– **delegate**

Returns the receiver's delegate.

deleteSamples

– (int)**deleteSamples**

Deletes all the samples in the receiver's sound data. The receiver must be editable. An error code is returned.

deleteSamplesAt:count:

– (int)**deleteSamplesAt:**(int)*startSample* **count:**(int)*sampleCount*

Deletes a range of samples from the receiver: *sampleCount* samples are deleted starting with the *startSample*'th sample (zero-based). The receiver must be editable and may become fragmented. An error code is returned.

finishUnarchiving

– **finishUnarchiving**

You never invoke this method. It's invoked automatically by the **read:** method to tie up loose ends after unarchiving the receiver.

free

– **free**

Frees the receiver and deallocates its sound data. The receiver is removed from the named Sound list and its name made eligible for reuse.

info

– (char *)**info**

Returns a pointer to the receiver's info string.

infoSize

– (int)**infoSize**

Returns the size (in bytes) of the receiver's info string.

insertSamples:at:

– (int)**insertSamples:***aSound* **at:**(int)*startSample*

Pastes the sound data in *aSound* into the receiver, starting at the receivers *startSample*'th sample (zero-based). The receiver doesn't lose any of its original sound data—the samples greater than or equal to *startSample* are moved to accommodate the inserted sound data. The receiver must be editable and the two Sounds must be compatible (as determined by **isCompatible:**). If the method is successful, the receiver is fragmented. An error code is returned.

isEditable

– (BOOL)**isEditable**

Returns **YES** if the receiver's format indicates that it can be edited, otherwise returns **NO**. In general, an editable Sound contains sampled data; all Sound Kit-defined formats are editable except SND_FORMAT_DSP_CORE, SND_FORMAT_MK_DSP_CORE, and SND_FORMAT_UNSPECIFIED.

isEmpty

– (BOOL)**isEmpty**

Returns **YES** if the receiver doesn't contain any sound data, otherwise returns **NO**. This always returns **NO** if the receiver isn't editable (as determined by sending it the **isEditable** message).

name

– (const char *)**name**

Returns the receiver's name.

needsCompacting

– (BOOL)**needsCompacting**

Returns **YES** if the receiver's data is fragmented (its format is SND_FORMAT_INDIRECT). Otherwise returns **NO**.

pause

– (int)**pause**

Pauses the receiver during recording or playback.

pause:

– **pause:***sender*

Action method that pauses the receiver during recording or playback.

play

– (int)**play**

Initiates playback of the sound. The method returns immediately while the playback continues asynchronously in the background. The playback ends when the receiver receives the **stop** message, or when its data is exhausted.

When playback starts, **willPlay:** is sent to the receiver's delegate; when it stops, **didPlay:** is sent. Returns the receiver.

An error code is returned.

play:

– **play:***sender*

Action method that plays the receiver. Other than the argument and the return type, this is the same as the **play** method.

read:

– **read:**(NXTypedStream *)*stream*

Reads archived sound data from *stream* into the receiver. Returns the receiver.

readSoundfile:

– (int)**readSoundfile:**(char *)*filename*

Replaces the receiver's sound with that in the soundfile *filename*. The file name is a complete UNIX path name that must include the ".snd" extension. An error code is returned.

record

– (int)**record**

Initiate recording of a sound into the receiver. To record from the CODEC microphone, the receiver's format, sampling rate, and channel count must be SND_FORMAT_MULAW_8, SND_RATE_CODEC, and 1, respectively. If this information isn't set (if the receiver is a newly created object, for example), it defaults to accommodate a CODEC recording. If the receiver's format is SND_FORMAT_DSP_DATA_16, the recording is from the DSP.

The method returns immediately while the recording continues asynchronously in the background. The recording stops when the receiver receives the **stop** message or when the maximum recording time limit has elapsed (precisely ten minutes).

When the recording begins, **willRecord:** is sent to the receiver's delegate; when the recording stops, **didRecord:** is sent. Returns the receiver.

An error code is returned.

record:

– **record:***sender*

Action method that initiates a recording. Other than the argument and return type, this is the same as the **record** method.

resume

– (int)**resume**

Resumes the paused receiver's activity.

resume:

– **resume:***sender*

Action method that resumes the paused receiver.

sampleCount

– (int)**sampleCount**

Returns the number of sample frames, or channel count-independent samples, in the receiver.

samplesProcessed

– (int)**samplesProcessed**

If the receiver is currently playing or recording, this returns the number of sample frames that have been played or recorded so far. Otherwise, the number of sample frames in the receiver is returned.

samplingRate

– (double)**samplingRate**

Returns the receiver's sampling rate.

setDataSize:dataFormat:samplingRate:channelCount:infoSize:

– (int)**setDataSize:**(int)*newDataSize*
dataFormat:(int)*newDataFormat*
samplingRate:(double)*newSamplingRate*
channelCount:(int)*newChannelCount*
infoSize:(int)*newInfoSize*

Allocates new, unfragmented sound data for the receiver, as described by the arguments. The receiver's previous data is freed. This method is useful for setting a determinate data length prior to a recording or for creating a scratch pad for algorithmic sound creation. An error code is returned.

setDelegate:

– **setDelegate:***anObject*

Sets the receiver's delegate to *anObject*. The delegate may implement the following methods:

- willPlay:
- didPlay:
- willRecord:
- didRecord:
- hadError:

Returns the receiver.

setName:

– **setName:**(const char *)*theName*

Sets the receiver's name to *theName*. If *theName* is already being used, then the receiver's name isn't set and **nil** is returned; otherwise returns the receiver.

soundStruct

– (SNDSoundStruct *)**soundStruct**

Returns a pointer to the receiver's sound structure (its **soundStruct** variable). Use of the pointer requires a knowledge of the SNDSoundStruct architecture.

soundStructSize

– (int)**soundStructSize**

Returns the size (in bytes) of the receiver's sound structure (its **soundStruct** variable). Use of this value requires a knowledge of the SNDSoundStruct architecture.

status

– (int)**status**

Return the receiver's current status, one of the following integer constants:

- SK_STATUS_STOPPED
- SK_STATUS_RECORDING
- SK_STATUS_PLAYING
- SK_STATUS_INITIALIZED
- SK_STATUS_RECORDING_PAUSED
- SK_STATUS_PLAYING_PAUSED
- SK_STATUS_RECORDING_PENDING
- SK_STATUS_PLAYING_PENDING
- SK_STATUS_FREED

stop

– (int)**stop**

Terminates the receiver's playback or recording. If the receiver was recording, the **didRecord:** message is sent to the delegate; if playing, **didPlay:** is sent. Returns the receiver.

An error code is returned.

stop:

– **stop:***sender*

Action method that stops the receiver's playback or recording. Other than the argument and the return type, this is the same as the **stop** method.

tellDelegate:

– **tellDelegate:**(SEL)*theMessage*

Sends *theMessage* to the receiver's delegate (only sent if the delegate implements *theMessage*). You never invoke this method directly; it's invoked automatically as the result of activities such as recording and playing. However, you can use it in designing a subclass of Sound.

Returns the receiver.

write:

– **write:**(NXTypedStream *)*stream*

Archives the receiver by writing its data to *stream*, which must be open for writing. Returns the receiver.

writeSoundfile:

– (int)**writeSoundfile:**(char *)*filename*

Writes the receiver's sound to the soundfile filename. The file name is a complete UNIX path name that should include a ".snd" extension. An error code is returned.

writeToPasteboard

– (int)**writeToPasteboard**

Puts a copy of the receiver's sound on the pasteboard. If the receiver is fragmented, it's compacted before the copy is created. An error code is returned.

METHODS IMPLEMENTED BY THE DELEGATE

didPlay:

– **didPlay:***sender*

Sent to the delegate when the Sound stops playing.

didRecord:

– **didRecord:***sender*

Sent to the delegate when the Sound stops recording.

hadError:

– **hadError:***sender*

Sent to the delegate if an error occurs during recording or playback.

willPlay:

– **willPlay:***sender*

Sent to the delegate when the Sound begins to play.

willRecord:

– **willRecord:***sender*

Sent to the delegate when the Sound begins to record.

SoundMeter

INHERITS FROM View : Responder : Object

DECLARED IN soundkit.h

CLASS DESCRIPTION

A SoundMeter object, when associated with a Sound object, displays the level of sound recording and playback.

Objects of this class, when attached to a sampled sound, can display the average output level and peak hold in a bar-graph-style display.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct __vFlags	vFlags;
Declared in SoundMeter	id	sound;
	int	currentSample;
	float	currentValue;
	float	currentPeak;
	float	minValue;
	float	maxValue;
	float	holdTime;
	float	backgroundGray;
	float	foregroundGray;
	float	peakGray;
	struct {	
	unsigned int	running:1;
	unsigned int	bezeled:1;
	}	smFlags;

sound — The object's Sound.

currentSample — The Sound sample currently being displayed.

currentValue — The value of the current sample.

currentPeak	The current value of the peak bubble.
minValue	The minimum sample value so far.
maxValue	The maximum sample value so far.
holdTime	The hold duration of the peak bubble.
backgroundGray	The background color.
foregroundGray	The foreground (average bar) color.
peakGray	The peak bubble color.
smFlags.running	Is the object currently running?
smFlags.bezeled	Is the frame bezeled?

METHOD TYPES

Creating and freeing a	+ newFrame:
Modifying the object	– setBezeled: – setFloatValue: – setHoldTime: – setSound:
Querying the object	– backgroundGray – floatValue – foregroundGray – holdTime – isBezeled – isRunning – maxValue – minValue – peakGray – peakValue – setBackgroundGray: – setForegroundGray: – setPeakGray: – sound
Operating the object	– run: – stop:
Drawing the object	– drawCurrentValue – drawSelf::

Archiving and unarchiving the object
– read:
– write:

CLASS METHODS

newFrame:

+ **newFrame:**(const NXRect *)*frameRect*

Creates and returns a new, initialized SoundMeter object.

INSTANCE METHODS

backgroundGray

– (float)**backgroundGray**

Returns the receiver's background color. The default is black.

drawCurrentValue

– **drawCurrentValue**

Draws the receiver's running bar and peak bubble. You never invoke this method directly; it's invoked by **drawSelf::**, **setFloatValue**, and by the animation code while the receiver is running. You can override this method in a subclass to change the look of the running bar and peak bubble.

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Draws all the components of the receiver (frame, running bar, and peak bubble). You never invoke this method directly; however, you can override it in a subclass to change the way the receiver is displayed.

floatValue

– (float)**floatValue**

Returns the current running value.

foregroundGray

– (float)**foregroundGray**

Returns the receiver's foreground (average bar) color. The default is white.

holdTime

– (float)**holdTime**

Returns the receiver's peak value hold time in seconds.

isBezeled

– (BOOL)**isBezeled**

Returns YES if the receiver has a bezel.

isRunning

– (BOOL)**isRunning**

Returns YES if the receiver is currently running.

maxValue

– (float)**maxValue**

Returns the maximum running value so far. You can invoke this method after you stop this receiver to retrieve the overall maximum value for the previous performance. The maximum value is cleared when you restart the receiver.

minValue

– (float)**minValue**

Returns the minimum running value so far. You can invoke this method after you stop this receiver to retrieve the overall minimum value for the previous performance. The minimum value is cleared when you restart the receiver.

peakGray

– (float)**peakGray**

Returns the receiver's peak bubble color. The default is dark gray.

peakValue

– (float)**peakValue**

Returns the current peak value.

read:

– **read:**(NXTypedStream *)*aStream*

Unarchives the receiver by reading it from *aStream*.

run:

– **run:***sender*

Starts the receiver running. The receiver's Sound must either be playing or recording in order for any meter activity to occur. Note that this method only affects the state of the receiver—it doesn't trigger any activity in the Sound.

setBackgroundGray:

– **setBackgroundGray:**(float)*aValue*

Sets the receiver's background color.

setBezeled:

– **setBezeled:**(BOOL)*aFlag*

If *aFlag* is YES, a bezelled frame is drawn around the receiver. If *aFlag* is NO and the receiver has a frame, the frame is removed.

setFloatValue:

– **setFloatValue:**(float)*aValue*

Sets the current running value to *aValue*. If *aValue* is greater than the current peak value, or if the peak hold time has elapsed, then the peak value is set to *aValue* as well. If autoDisplay is on, the view is updated. You never invoke this method directly; it's invoked automatically when the receiver is running. However, you can reimplement this method in a subclass of SoundMeter.

setForegroundGray:

– **setForegroundGray:**(float)*aValue*

Sets the receiver's foreground (average bar) color.

setHoldTime:

– **setHoldTime:**(float)*seconds*

Sets the receiver's peak value hold time in seconds. This is the amount of time the peak bubble holds its value before decaying to the current average.

setPeakGray:

– **setPeakGray:**(float)*aValue*

Sets the receiver's peak bubble color.

setSound:

– **setSound:***aSound*

Sets the receiver's Sound object. *aSound* must contain sampled data ([*aSound* **isEditable**] must return TRUE).

sound

– **sound**

Returns the Sound object that the receiver is metering.

stop:

– **stop:***sender*

Stops the receiver's metering activity and sets its display to a default (zero signal) state. Note that this method only affects the state of the receiver—it doesn't trigger any activity in the Sound.

write:

– **write:**(NXTypedStream *)*aStream*

Archives the receiver by writing it to *aStream*.

SoundView

INHERITS FROM View : Responder : Object

DECLARED IN soundkit.h

CLASS DESCRIPTION

A SoundView object creates a view in which it displays a Sound object's sound data. A hairline cursor is provided for use in pointing and selecting. Only sampled sounds can be displayed in a SoundView.

Sounds are displayed on a two-dimensional graph. The amplitudes of individual samples are measured vertically and plotted against time, which proceeds left to right along the horizontal axis. A SoundView is always scaled vertically so that the full amplitude matches the height of the view with 0.0 amplitude in the center.

For most complete sounds, the length of the sound data in samples is greater than the horizontal length of the view 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.

Lines are drawn between the chosen values to yield a more continuous shape. Two drawing modes are provided:

- In SK_DISPLAY_WAVE mode, the drawing is rendered in an oscilloscopic fashion.
- In SK_DISPLAY_MINMAX mode, two lines are drawn, one to connect the maximum values, and one to connect the minimum values.

As you zoom in (as the reduction factor decreases), the two drawing modes become indistinguishable.

A mechanism is provided for selecting an area of the view. You can set the selected area through the method **setSelection:size:** or the user can make the selection by dragging the mouse. The playback, recording, and editing methods provided by SoundView operate on the selection.

When a SoundView's sound data changes (due to editing or recording), the manner in which the SoundView is redisplayed depends on its **autoscale** flag. With autoscaling disabled, the SoundView's frame grows or shrinks (horizontally) to the new sound data while the reduction factor is unchanged. If autoscaling is enabled, the reduction factor is automatically recomputed to maintain a constant frame size. By default, autoscaling is disabled; this is to accommodate the use of a SoundView object as the document of

a ScrollingView, allowing the ScrollingView to pan along the data displayed in the SoundView. As such, maintaining a constant reduction factor (or level of detail) across a change is more useful than maintaining a constant SoundView frame size. Note, however, that changing the reduction factor when autoscaling is disabled is useful for zooming in and out.

In order to provide greater efficiency, a SoundView creates its own Sound object, stored in its **reduction** instance variable, that contains only the samples from its **sound** instance variable that are actually displayed. Methods to set and retrieve the reduction are provided; however, you should only invoke these methods if you're creating an advanced application or if you're designing a subclass of SoundView.

SoundView implements the Application Kit's delegate paradigm, allowing messages to be sent to a delegate object when actions, such as playing, editing, or selecting a portion of the SoundView, are performed.

INSTANCE VARIABLES

Inherited from Object

```
Class           isa;
```

Inherited from Responder

```
id              nextResponder;
```

Inherited from View

```
NXRect          frame;
NXRect          bounds;
id              superview;
id              subviews;
id              window;
struct __vFlags vFlags;
```

Declared in SoundView

```
id              sound;
id              reduction;
id              delegate;
NXRect          selectionRect;
int             displayMode;
float           backgroundGray;
float           foregroundGray;
float           reductionFactor;
struct {
    unsigned int    disabled:1;
    unsigned int    continuous:1;
    unsigned int    calcDrawInfo:1;
    unsigned int    selectionDirty:1;
    unsigned int    autoscale:1;
    unsigned int    bezeled:1;
}               svFlags;
```

sound — The object's Sound.

reduction — The data reduced version of the object's Sound.

delegate	The object's delegate.
selectionRect	The object's current selection.
displayMode	Display mode; SK_DISPLAY_MINMAX by default.
backgroundGray	Background color; NX_WHITE by default.
foregroundGray	Foreground color; NX_BLACK by default.
reductionFactor	The ratio of sound samples to display units.
svFlags.disabled	Does the object (not) respond to mouse events?
svFlags.continuous	Does the object respond to mouse dragged events?
svFlags.calcDrawInfo	Does drawing info need to be recalculated?
svFlags.selectionDirty	Has the object changed (but not been played)?
svFlags.autoscale	Does it rescale the display when the sound data changes?
svFlags.bezeled	Does the object have a bezeled border?

METHOD TYPES

Creating and freeing a SoundView	– free + newFrame:
Modifying the object	– scaleToFit – setBackgroundGray: – setBezeled: – setContinuous: – setDelegate: – setDisplayMode: – setEnabled: – setForegroundGray: – setReduction: – setSound: – sizeToFit

Querying the object	– backgroundGray – delegate – displayMode – foregroundGray – getSelection:size: – isAutoScale – isBezeled – isContinuous – isEnabled – reduction – reductionFactor – sound
Selecting and editing the sound data	– copy: – cut: – delete: – mouseDown: – paste: – selectAll: – setSelection:size:
Modifying the display coordinates	– setAutoscale: – setReductionFactor:
Drawing the object	– calcDrawInfo – drawSelf:: – hideCursor – showCursor – sizeTo::
Responding to events	– acceptsFirstResponder – becomeFirstResponder – resignFirstResponder
Performing the sound data	– play: – record: – soundBeingProcessed – stop:
Archiving the object	– read: – write:
Accessing the delegate	– didPlay: – didRecord: – hadError: – tellDelegate: – willPlay: – willRecord:

CLASS METHODS

newFrame:

+ **newFrame:**(const NXRect *)*aRect*

Creates and returns a SoundView with the frame *aRect*. The new SoundView doesn't contain any sound data.

INSTANCE METHODS

acceptsFirstResponder

– (BOOL)**acceptsFirstResponder**

If the receiver is enabled, this returns YES, allowing the view to become the first responder. This method is automatically invoked by objects defined by the Application Kit; you should never need to invoke it directly.

backgroundGray

– (float)**backgroundGray**

Returns the receiver's background gray value (NX_WHITE by default).

becomeFirstResponder

– **becomeFirstResponder**

Promotes the receiver to first responder allowing user actions to be directed to the receiver. Returns the receiver.

calcDrawInfo

– **calcDrawInfo**

Calculates the receiver's internal drawing information. This method is automatically invoked when needed—when the receiver's sound data changes, for example. A subclass should invoke this method from any method that changes the receiver. The return value is ignored.

copy:

– **copy:***sender*

Copies the current selection to the pasteboard. Returns the receiver.

cut:

– **cut:***sender*

Deletes the current selection from the receiver, copies it to the pasteboard, and sends the **soundChanged:** message to the delegate. The insertion point is positioned to where the selection used to start. The sound data becomes fragmented. Returns the receiver.

delegate

– **delegate**

Returns the receiver's delegate object.

delete:

– **delete:***sender*

Deletes the current selection from the receiver's Sound and sends the **soundChanged:** message to the delegate. The deletion isn't placed on the pasteboard. The sound data becomes fragmented. Returns the receiver.

didPlay:

– **didPlay:***sender*

Used to redirect delegate messages from the receiver's Sound object; you never invoke this method directly.

didRecord:

– **didRecord:***sender*

Used to redirect delegate messages from the receiver's Sound object; you never invoke this method directly.

displayMode

– (int)**displayMode**

Returns the receiver's display mode, one of SK_DISPLAY_WAVE (oscilloscopic display) or SK_DISPLAY_MINMAX (minimum maximum display; this is the default).

drawSelf::

– **drawSelf:**(const NXRect *)*rects* **:**(int)*rectCount*

Displays the receiver's sound data. The selection is highlighted and the cursor is drawn (if it isn't currently hidden).

The SoundView class implements this method as a subclass responsibility inherited from View. You never send the **drawSelf::** message directly to a SoundView object. Instead, use one of the **display** methods defined in the View class.

foregroundGray

– (float)**foregroundGray**

Returns the receiver's foreground gray value (NX_BLACK by default).

free

– **free**

Frees the receiver but not its Sound object nor its delegate. The **willFree:** message is sent to the delegate.

getSelection:size:

– **getSelection:**(int *)*firstSample* **size:**(int *)*sampleCount*

Returns the selection by reference. The index of the selection's first sample (counting from 0) is returned in *firstSample*. The size of the selection in samples is returned in *sampleCount*. The method itself returns the receiver.

hadError:

– **hadError:***sender*

Used to redirect delegate messages from the receiver's Sound object; you never invoke this method directly.

hideCursor

– **hideCursor**

Hides the receiver's cursor. This is usually handled automatically. Returns the receiver.

isAutoScale

– (BOOL)**isAutoScale**

Returns YES if the receiver is in autoscaling mode, otherwise returns NO.

isBezeled

– (BOOL)**isBezeled**

Returns YES if the display features a bezeled border, otherwise returns NO (the default).

isContinuous

– (BOOL)**isContinuous**

Returns YES if the receiver responds to mouse dragged events (as set through **setContinuous:**). The default is NO.

isEnabled

– (BOOL)**isEnabled**

Returns YES if the receiver is enabled, otherwise returns NO. The mouse has no effect in a disabled SoundView. By default, a SoundView is enabled.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Allows a selection to be defined by clicking and dragging the mouse. This method takes control until a mouse-up occurs. While dragging, the selected region is highlighted. On mouse up, the delegate is sent the **selectionChanged:** message. If **isContinuous** is YES, **selectionChanged:** messages are also sent while the mouse is being dragged. Returns the receiver.

paste:

– **paste:***sender*

Replaces the current selection with a copy of the sound data currently on the pasteboard. If there is no selection the pasteboard data is inserted at the cursor position. The pasteboard data must be compatible with the receiver's data, as determined by the Sound method **compatibleWith:**. If the paste is successful, the **soundChanged:** message is sent to the delegate. The receiver's sound data becomes fragmented. Returns the receiver.

play:

– **play:***sender*

Play the current selection by invoking Sound's **play:** method. If there is no selection, the receiver's entire Sound is played. The **willPlay:** message is sent to the delegate before the selection is played; **didPlay:** is sent when the selection is done playing. Returns the receiver.

read:

– **read:**(void *)*stream*

Unarchives the receiver by reading it from *stream*.

record:

– **record:***sender*

Replaces the receiver's current selection with newly recorded material. If there is no selection, the recording is injected at the insertion point. The **willRecord:** message is sent to the delegate before the recording is started; **didRecord:** is sent after the recording has completed. Currently, the recorded data is always taken from the CODEC microphone input. Returns the receiver.

reduction

– **reduction**

Returns the receiver's display reduction Sound object. Provided for display optimization, the object returned by this method shouldn't be treated like a "normal" Sound—for example, it can't be played. The receiver owns the reduction object and may free it at any time.

reductionFactor

– (float)**reductionFactor**

Returns the receiver's reduction factor, computed as

```
reductionFactor = sampleCount / displayUnits
```

resignFirstResponder

– **resignFirstResponder**

Resigns the position of first responder. Returns the receiver.

scaleToFit

– **scaleToFit**

Recomputes the receiver's reduction factor to fit the sound data (horizontally) within the current frame. Invoked automatically when the receiver's data changes and the receiver is in autoscale mode. If the receiver isn't in autoscale mode, **sizeToFit** is invoked when the data changes. You never invoke this method directly; a subclass can reimplement this method to provide specialized behavior.

selectAll:

– **selectAll:***sender*

Creates a selection over the receiver's entire Sound. Returns the receiver.

setAutoscale:

– **setAutoscale:**(BOOL)*aFlag*

Sets the receiver's automatic scaling mode, used to determine how the receiver is redisplayed when its data changes. With autoscaling enabled (*aFlag* is YES), the receiver's reduction factor is recomputed so the sound data fits within the view frame. If it's disabled (*aFlag* is NO), the frame is resized and the reduction factor is unchanged. If the receiver is in a ScrollingView, autoScaling should be disabled (autoscaling is disabled by default). Returns the receiver.

setBackgroundGray:

– **setBackgroundGray:**(float)*aGray*

Sets the receiver's background gray value to *aGray*; the default is NX_WHITE. Returns the receiver.

setBezeled:

– **setBezeled:**(BOOL)*aFlag*

If *aFlag* is YES, the display is given a bezeled border. By default, the border of a SoundView display isn't bezeled. If autodisplaying is enabled, the Sound is automatically redisplayed. Returns the receiver.

setContinuous:

– **setContinuous:**(BOOL)*aFlag*

Sets the state of continuous action messages. If *aFlag* is YES, **selectionChanged:** messages are sent to the delegate as the mouse is being dragged. If NO, the message is sent only on mouse up. The default is NO. Returns the receiver.

setDelegate:

– **setDelegate:***anObject*

Sets the receiver's delegate to *anObject*. The delegate is sent messages when the user changes or acts on the selection. Returns the receiver.

setDisplayMode:

– **setDisplayMode:**(int)*aMode*

Sets the receiver's display mode, either SK_DISPLAY_WAVE or SK_DISPLAY_MINMAX (the default). If autodisplaying is enabled, the Sound is automatically redisplayed.

setEnabled:

– **setEnabled:**(BOOL)*aFlag*

Enables or disables the receiver as *aFlag* is YES or NO. The mouse has no effect in a disabled SoundView. By default, a SoundView is enabled. Returns the receiver.

setForegroundGray:

– **setForegroundGray:**(float)*aGray*

Sets the receiver's foreground gray value to *aGray*. The default is NX_BLACK. Returns the receiver.

setReduction:

– **setReduction:***aDisplayReduction*

Sets the receiver's display reduction Sound object to *aDisplayReduction*. An advanced application can set the display reduction directly to optimize or eliminate the recalculation of the display; this may be useful, for example, for repeated editing of extremely large sounds. The number of samples in the reduction must be exactly 1/reductionFactor times the number of samples of the current sound. The receiver owns the reduction and may free it at any time. Use of this method is optional; if the display reduction isn't set through this method, it's calculated automatically.

If the size of *aDisplayReduction* (in samples) isn't correct, **nil** is returned; otherwise returns the receiver.

setReductionFactor:

– **setReductionFactor:**(float)*reductionFactor*

If the receiver is in autoscale mode, this does nothing and immediately returns the receiver. (Keep in mind that in autoscaling mode, the reduction factor is automatically recomputed when the sound data changes—see **scaleToFit:**.) With autoscaling disabled, *reductionFactor* is used to recompute the size of the receiver's frame (in display units) according to the formula

displayUnits = sampleCount / reductionFactor

Increasing the reduction factor zooms out, decreasing zooms in on the data.

If autodisplaying is enabled, the Sound is automatically redisplayed. Returns the receiver.

setSelection:size:

– **setSelection:**(int)*firstSample* **size:**(int)*sampleCount*

Sets the selection to be *sampleCount* samples wide, starting with sample *firstSample* (samples are counted from 0). Returns the receiver.

setSound:

– **setSound:***aSound*

Sets the receiver's Sound object to *aSound*. If autoscaling is enabled, the drawing coordinate system is adjusted so *aSound*'s data fits within the current frame. Otherwise, the frame is resized to accommodate the length of the data. If autodisplaying is enabled, the receiver is automatically redisplayed. Returns the receiver.

showCursor

– **showCursor**

Displays the receiver's cursor. This is usually handled automatically. Returns the receiver.

sizeTo::

– **sizeTo:**(NXCoord)*width* **:**(NXCoord)*height*

Sets the width and height of the receiver's frame. If autodisplaying is enabled, the receiver is automatically redisplayed. Returns the receiver.

sizeToFit

– **sizeToFit**

Resizes the receiver's frame (horizontally) to maintain a constant reduction factor. This method is invoked automatically when the receiver's data changes and the receiver isn't in autoscale mode. If the receiver is in autoscale mode, **scaleToFit** is invoked when the data changes. You never invoke this method directly; a subclass can reimplement this method to provide specialized behavior.

sound

– **sound**

Returns a pointer to the receiver's Sound object.

soundBeingProcessed

– **soundBeingProcessed**

Returns the **id** of the Sound object that's currently being played or recorded into. Note that the actual Sound object that's being performed isn't necessarily the receiver's **sound** (the object returned by the **sound** method); for efficiency, SoundView creates a private performance Sound object. While this is generally an implementation detail, this method is supplied in case the SoundView's delegate needs to know exactly which object will be/was performed.

stop:

– **stop:***sender*

Stops the receiver's current recording or playback. Returns the receiver.

tellDelegate:

– **tellDelegate:**(SEL)*theMessage*

Sends *theMessage* to the receiver's delegate with the receiver as the argument. If the delegate doesn't respond to the message, then it isn't sent. You normally never invoke this method; it's invoked automatically when an action, such as playing or editing, is performed. However, you can invoke it in the design of a SoundView subclass. Returns the receiver.

willPlay:

– **willPlay:***sender*

Used to redirect delegate messages from the receiver's Sound object; you never invoke this method directly.

willRecord:

– **willRecord:***sender*

Used to redirect delegate messages from the receiver's Sound object; you never invoke this method directly.

write:

– **write:**(void *)*stream*

Archives the receiver by writing it to *stream.*

METHODS IMPLEMENTED BY THE DELEGATE

didPlay:

– **didPlay:***sender*

Sent to the delegate just after the SoundView is played.

didRecord:

– **didRecord:***sender*

Sent to the delegate just after the SoundView is recorded into.

hadError:

– **hadError:***sender*

Sent to the delegate if an error is encountered during recording or playback of the SoundView's data.

selectionChanged:

– **selectionChanged:***sender*

Sent to the delegate whenever the SoundView's selection changes.

soundDidChange:

– **soundDidChange:***sender*

Sent to the delegate whenever the SoundView's sound data changes.

willFree:

– **willFree:***sender*

Sent to the delegate when the SoundView is freed.

willPlay:

– **willPlay:***sender*

Sent to the delegate just before the SoundView is played.

willRecord:

– **willRecord:***sender*

Sent to the delegate just before the SoundView is recorded into.

Music Kit Classes

The class specifications for the Music Kit describe the following classes:

Conductor
Envelope
FilePerformer
FileWriter
Instrument
Midi
Note
NoteFilter
NoteReceiver
NoteSender
Orchestra
Part
Partials
PartPerformer
PartRecorder
PatchTemplate
Performer
Samples
Score
ScorefilePerformer
ScorefileWriter
ScorePerformer
ScoreRecorder
SynthData
SynthInstrument
SynthPatch
TuningSystem
UnitGenerator
WaveTable
Add2UG
Allpass1UG
AsympUG
ConstantUG
DelayUG
DswitchtUG
DswitchUG
InterpUG
Mul1add2UG
Mul2UG
OnepoleUG
OnezeroUG
OscgafUG
OscgafiUG
OscgUG
Out1aUG

Out1bUG
Out2sumUG
ScaleUG
Scl1add2UG
Scl2add2UG
SnoiseUG
UnoiseUG

Conductor

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

The Conductor class defines the mechanism that controls the timing of a Music Kit performance. A Conductor's most important tasks are to schedule the sending of Notes by Performers (and Midi), and to control the timing of Envelope objects during DSP synthesis. Even in the absence of Performers and Envelopes, you may want to use a Conductor to take advantage of the convenient scheduling mechanism that it provides.

The Message Request Queue

Each instance of Conductor contains a message request queue, a list of messages that are to be sent to particular objects at specific times. To enqueue a message request with a Conductor, you invoke the **sel:to:atTime:argCount:** or **sel:to:withDelay:argCount:** method. Once you have 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 remove a request—you should use the following C functions:

- **MKNewMsgRequest()** creates and returns a new message request structure.
- **MKScheduleMsgRequest()** places a previously created message request in a Conductor's message request queue.
- **MKRepositionMsgRequest()** repositions a message request within a Conductor's queue.
- **MKCancelMsgRequest()** removes a message request.
- **MKRescheduleMsgRequest()** is a convenience function that cancels a request and then creates a new one.

For more information on these functions, see Chapter 3, "C Functions."

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

A Music Kit performance starts when the Conductor class receives the **startPerformance** message. At that time, 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 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 isn'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.

The rate at which a Conductor object processes its message request queue can be set through either of two methods:

- **setTempo:** sets the rate as beats per minute.
- **setBeatSize:** sets the size of an individual beat, in seconds.

You can change a Conductor's tempo anytime, 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.

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. If your application sends a message (or calls a C function) in response to an asynchronous event, it must first update the Conductors' notions of time by sending **lockPerformance** to the Conductor class. For example, if your application sends a Note directly to an Instrument, you should send **lockPerformance** immediately before the Note is sent. Every invocation of **lockPerformance** should be balanced by an invocation of **unlockPerformance**.

Conductors and Performers

Conductors and Performers have a special relationship: Every Performer object is controlled by an instance of Conductor, as set through Performer's **setConductor:** method. While a Performer can be controlled by only one Conductor, a single Conductor can control any number of Performers. As a Performer acquires successive

Notes, it enqueues, with its associated Conductor, requests for the Notes to be sent to its connected Instruments. This enqueuing is performed automatically through a mechanism defined by the Performer class. As a convenience, the Music Kit automatically creates an instance of Conductor called the *defaultConductor*; if you don't set a Performer's Conductor directly, it's controlled by the defaultConductor. You can retrieve the defaultConductor (in order to set its tempo or to enqueue message requests, for example) by sending the **defaultConductor** message to the Conductor class.

Conductors and Envelopes

The Music Kit also creates an instance of Conductor called the *clockConductor*, which you can retrieve through the **clockConductor** class method. The clockConductor has an unchangeable tempo of 60.0 beats per minute and it can't be paused (at least not by itself; you can pause the clockConductor as you pause the entire performance through the **pausePerformance** class method). While the clockConductor can be used to control Performers, its most important task is to control the timing of Envelope objects during DSP synthesis. Envelope breakpoints are fed to the DSP through messages that are enqueued automatically with the clockConductor.

The clockConductor's queue is treated like any other queue: You can enqueue message requests with the clockConductor just as you would with any other Conductor. This also means that the clockConductor's queue contributes to a determination of whether all the queues are empty.

Fine-tuning a Performance

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 time stamps. When the Conductor class is clocked, a running Application object must be present.

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 a MIDI synthesizer—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.

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 queue, thus allowing music to play with greater independence from your application's other computations. However, this means that the synchronization between music and graphics (for example) will be loosened.

In addition, you can set the Mach-scheduling priority of the performance thread (whether or not it's separate) through the **setThreadPriority:** method. 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
```

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Conductor	double	time;
	double	nextMsgTime;
	double	beatSize;
	double	timeOffset;
	BOOL	isPaused;
	id	delegate;

time	Current time in beats.
nextMsgTime	Time, in seconds, when the object is scheduled to send its next message.
beatSize	The duration of a single beat, in seconds.
timeOffset	Performance time offset, in seconds.
isPaused	YES if this object is currently paused.
delegate	The object's delegate.

METHOD TYPES

Allocating and initializing a Conductor instance	+ alloc + allocFromZone: – init
Copying and freeing the object	– copy – copyFromZone:
Acquiring extant Conductors	+ clockConductor + currentConductor + defaultConductor
Setting up a performance	+ setClocked: + useSeparateThread: + setThreadPriority: + setFinishWhenEmpty: + isClocked + performanceThread + finishWhenEmpty
Controlling a performance	+ startPerformance + pausePerformance + resumePerformance + finishPerformance + lockPerformance + lockPerformanceNoBlock + unlockPerformance + inPerformance + isEmpty + isPaused + time
Modifying a Conductor	– setBeatSize: – setTempo: – setTimeOffset: – pause – pauseFor: – resume – setDelegate:
Querying a Conductor	– beatSize – tempo – timeOffset – isPaused – delegate – isCurrentConductor – predictTime: – time

Enqueueing message requests	– sel:to:atTime:argCount: – sel:to:withDelay:argCount: – emptyQueue + afterPerformanceSel:to:argCount: + beforePerformanceSel:to:argCount:
Archiving	– finishUnarchiving: – read – write

CLASS METHODS

afterPerformanceSel:to:argCount:

+ (MKMsgStruct *)**afterPerformanceSel:**(SEL)*aSelector*
to:*toObject*
argCount:(int)*argCount,...*

Enqueues a request for *aSelector* to be sent to *toObject* immediately after the current or next performance ends. *argCount* specifies the number of four-byte arguments to *aSelector* followed by the arguments themselves separated by commas (two arguments, maximum). The messages are sent in the order that the requests are enqueued. Returns a pointer to a message request structure that can be passed to a C function such as **MKCancelMsgRequest()**.

See also: + **beforePerformanceSel:to:argCount:**, – **sel:to:atTime:argCount:**, – **sel:to:withDelay:argCount:**, **MKNewMsgRequest()**

alloc

+ **alloc**

Returns a new, initialized Conductor allocated in the default zone. If a performance is in progress, does nothing and returns **nil**.

See also: + **allocFromZone:**, – **init**

allocFromZone:

– **allocFromZone:**(NXZone *)*zone*

Returns a new, initialized Conductor allocated in *zone*. If a performance is in progress, does nothing and returns **nil**.

See also: + **alloc**, – **init**

beforePerformanceSel:to:argCount:

+ (MKMsgStruct *)**beforePerformanceSel:**(SEL)*aSelector*
to:*toObject*
argCount:(int)*argCount*,...

Enqueues a request for *aSelector* to be sent to *toObject* at the beginning of the next performance. *argCount* specifies the number of four-byte arguments to *aSelector* followed by the arguments themselves separated by commas (two arguments, maximum). The messages are sent in the order that the requests are enqueued. Returns a pointer to a message request structure that can be passed to a C function such as **MKCancelMsgRequest()**.

See also: **+ afterPerformanceSel:to:argCount:**, **– sel:to:atTime:argCount:**, **– sel:to:withDelay:argCount:**, **MKNewMsgRequest()**

clockConductor

+ clockConductor

Returns the clockConductor, the ever-present instance of Conductor that has an immutable tempo of 60.0 beats per minute. You can't free this object.

currentConductor

+ currentConductor

Returns the Conductor instance that's currently sending a message, or **nil** if no message is being sent.

defaultConductor

+ defaultConductor

Returns the defaultConductor, the ever-present instance of Conductor that, by default, is used to enqueue Note-sending messages generated by Performer objects. You can't free this object.

finishPerformance

+ finishPerformance

Ends the performance; the after-performance messages are sent and all other enqueued messages are removed (except from the before-performance queue). If **finishWhenEmpty** is YES, this message is automatically sent when all message queues are exhausted. Returns **nil**.

See also: **+ startPerformance**, **+ pausePerformance**, **+ resumePerformance**

finishWhenEmpty

+ (BOOL)**finishWhenEmpty**

Returns YES if the performance will automatically finish when all Conductors' message queues are empty, otherwise returns NO.

See also: **+ setFinishWhenEmpty:**, **+ finishPerformance**

inPerformance

+ (BOOL)**inPerformance**

Returns YES if a performance is currently taking place (even if it's paused), otherwise returns NO.

See also: **+ startPerformance**

isClocked

+ (BOOL)**isClocked**

Returns YES if the performance is clocked, NO if it isn't. In a clocked performance (the default), messages from the message request queues are sent at the times indicated by their timestamps. In an unclocked performance, the messages are sent one after another as quickly as possible.

See also: **+ setClocked:**, **MKSetDeltaT()**

isEmpty

+ (BOOL)**isEmpty**

Returns YES if a performance is in progress and all the Conductor instances' message request queues are empty, otherwise returns NO.

See also: **+ setFinishWhenEmpty:**

isPaused

+ (BOOL)**isPaused**

Returns YES if the performance is paused, otherwise returns NO.

See also: **+ pause**, **+ resume**, **– pause**, **– resume**

lockPerformance

+ **lockPerformance**

Waits for the availability of, and then acquires, the Music Kit performance lock. This updates all Conductors' notions of the current time. Returns **self**.

In a separate-threaded performance, you should lock the Music Kit before sending a message (or group of messages) to a Music Kit object. In a performance that isn't separate-threaded, you only need to lock the performance if the message that you're sending depends on the Conductors' notions of time being current. After you've successfully locked the performance and have sent the desired messages, you must invoke **unlockPerformance**. Note that acquisitions of the Music Kit lock can be nested; if you send **lockPerformance** twice, you must send **unlockPerformance** twice to release the lock.

See also: + **unlockPerformance**, + **lockPerformanceNoBlock**

lockPerformanceNoBlock

+ **lockPerformanceNoBlock**

This is the same as **lockPerformance**, except it doesn't wait for the Music Kit lock to become available. Returns **nil** if the lock is thereby unsuccessful, otherwise returns **self**.

See also: + **lockPerformance**, + **unlockPerformance**

pausePerformance

+ **pausePerformance**

Pauses the performance; all Conductor instances suspend their message-sending activity until the Conductor class receives the **resumePerformance** message. You can't pause an unclocked performance; returns **nil** in this case, otherwise returns **self**. This message is ignored if a performance isn't in progress or if it's already paused.

See also: + **resumePerformance**, + **isPaused**, – **pause**, – **resume**

performanceThread

+ (cthread_t) **performanceThread**

Returns the separate thread in which the performance is running. If a performance isn't in progress, or if it isn't in a separate thread, returns NO_CTHREAD.

See also: + **useSeparateThread:**

resumePerformance

+ **resumePerformance**

Resumes a performance, allowing it to continue from where it was paused. You can't pause, and so can't resume, an unclocked performance; returns **nil** in this case. Otherwise returns **self**, although the method does nothing if a performance isn't in progress or if it isn't currently paused.

See also: + **pausePerformance**, + **isPaused**, – **pause**, – **resume**

setClocked:

+ **setClocked:**(BOOL)*yesOrNo*

If a performance is in progress, this does nothing and returns **nil**. Otherwise, it sets a performance to be clocked or unclocked as *yesOrNo* is YES or NO and returns **self**.

In a clocked performance (the default), messages from the message request queues are sent at the times indicated by their timestamps. In an unclocked performance, the messages are sent one after another as quickly as possible. Note, however, that if the performance is unclocked and isn't in a separate thread, a subsequent **startPerformance** message won't return until the performance is over, thus disabling the user interface for the duration of the performance.

See also: + **isClocked**, **MKSetDeltaT()**

setFinishWhenEmpty:

+ **setFinishWhenEmpty:**(BOOL)*yesOrNo*

If *yesOrNo* is YES, the performance is ended when all the Conductors' message request queues are empty. If NO, the performance continues until the **finishPerformance** message is sent to the Conductor class. By default, a performance finishes when the queues are empty.

See also: + **finishWhenEmpty**, + **finishPerformance**

setThreadPriority:

+ **setThreadPriority:**(float)*priority*

Sets the Mach scheduling priority of the performance thread for all subsequent performances. The priority change takes effect when the **startPerformance** method is invoked; it reverts to its original level between performances. The value of *priority* is limited to fall between 0.0 and 1.0, where 0.0 is normal priority and 1.0 is the maximum.

See also: + **useSeparateThread:**

startPerformance

+ **startPerformance**

Starts a performance; the messages in the before-performance queue are sent and then all Conductor instances begin processing their individual queues. If the performance is clocked, isn't in a separate thread, and a running Application object isn't present, this does nothing and returns **nil**. In all other cases, **self** is returned. Note, however, that if the performance is unclocked, the method doesn't return until the performance is over. If a performance is in progress, it isn't interrupted.

You can delay the begin time of individual Conductors through the **setTimeOffset:** instance method.

See also: + **finishWhenEmpty**, + **finishPerformance**, – **setTimeOffset:**

time

+ (double)**time**

Returns the current performance time, in seconds. This doesn't include time that the performance has been paused, nor does it include the performance's delta time. If a performance isn't in progress, MK_NODVAL is returned (use **MKIsNoDVal()** to check for this value). Performance time is normally updated only when a Conductor sends a message; however, you can force time to be updated by sending **lockPerformance** to the Conductor class.

See also: + **lockPerformance**, **MKIsNoDVal()**

unlockPerformance

+ **unlockPerformance**

Releases the Music Kit's performance lock; as a convenience, this also sends **flushTimedMessages** to the Orchestra class, causing any buffered synthesis commands to be flushed to the DSP. Note that the Music Kit locks can be nested; if you send **lockPerformance** twice, you must send **unlockPerformance** twice before the lock is released.

See also: + **lockPerformance**, + **lockPerformanceNoBlock**

useSeparateThread:

+ **useSeparateThread:**(BOOL)*yesOrNo*

If *yesOrNo* is YES, all subsequent performances will be run in a separate thread, allowing more independence between the performance and the rest of your application. If NO, the performance is run in the same thread that invokes **startPerformance**; this is the default. Does nothing and returns **nil** if a performance is in progress, otherwise returns **self**.

Keep in mind that when you run your performance in a separate thread, you must bracket all messages to Music Kit objects (sent from your application) with **lockPerformance** and **unlockPerformance**.

See also: **+ performanceThread**

INSTANCE METHODS

beatSize

– (double)**beatSize**

Returns the size of the Conductor's beat in seconds. The default is 1.0.

See also: **– setBeatSize:**, **– setTempo:**, **– tempo**

copy

– **copy**

Returns a new Conductor created through **[[Conductor alloc] init]**.

See also: **+ alloc**, **+ allocFromZone:**, **– init**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new Conductor created through **[[Conductor allocFromZone:***zone***] init]**.

See also: **+ alloc**, **+ allocFromZone:**, **– init**

delegate

– delegate

Returns the Conductor's delegate object, as set through the **setDelegate:** method. A Conductor's delegate is alerted when the Conductor is paused and resumed.

See also: – **setDelegate:**

emptyQueue

– emptyQueue

Removes all message requests from the Conductor's message request queue. Returns **self**.

finishUnarchiving

– finishUnarchiving

You never invoke this method directly; to read an archived Conductor, call the **NXReadObject**() C function. This method is invoked by **NXReadObject**() which returns the value returned by this method, as follows: If the unarchived Conductor was the clockConductor (when it was archived), this method frees the unarchived object and returns the current clockConductor. If a performance is in progress, the unarchived object is freed and the defaultConductor is returned. If a performance isn't in progress and the unarchived object was the defaultConductor, the current defaultConductor takes the new object's tempo and time offset, the unarchived object is freed, and the defaultConductor is returned. Otherwise, the unarchival is successful and **nil** is returned.

init

– init

Initializes a new Conductor object by setting its tempo to 60.0 beats per minute. Returns **self**.

See also: **+ alloc**, **– copy**

isCurrentConductor

– (BOOL)**isCurrentConductor**

Returns YES if the Conductor is currently sending a message from its message request queue, otherwise returns NO.

isPaused

– (BOOL)**isPaused**

Returns YES if the receiver is paused, otherwise returns NO.

See also: – **pause**, – **pauseFor:**, – **resume**

pause

– **pause**

Pauses the performance of the Conductor and sends **conductorDidPause:** to its delegate. Pausing a Conductor causes the object to stop sending messages in its message request queue (message requests can still be enqueued). The suspension is restricted to the present performance. You invoke **resume** to unpause a Conductor.

You can't pause the clockConductor through this method; returns **nil** in this case (and the delegate message isn't sent). Otherwise returns the receiver. Note that you can pause a Conductor object before a performance begins.

See also: **– pauseFor:**, **– resume**, **– setTimeOffset:**, **+ pausePerformance**

pauseFor:

– **pauseFor:**(double)*seconds*

Pauses the performance of the Conductor, sends **conductorDidPause:** to its delegate, and schedules a request for **resume** to be sent to the receiver in *seconds* seconds. If the receiver is currently paused through a previous invocation of this method, the current **resume** request supercedes the previous one. The effect is restricted to the present performance.

You can't pause the clockConductor through this method; returns **nil** in this case (and the delegate message isn't sent). Otherwise returns the receiver. Note that you can invoke this method before a performance begins; the **resume** message is enqueued to be sent *seconds* seconds after the performance starts.

See also: **– pause**, **– resume**, **– setTimeOffset:**, **+ pausePerformance**

predictTime:

– (double)**predictTime:**(double)*beatTime*

Returns the time, in seconds, when beat *beatTime* should occur given the Conductor's tempo. If *beatTime* is less than the Conductor's current time, 0.0 is returned.

See also: **– time**

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived Conductor, call the **NXReadObject**() C function.

See also: – **finishUnarchiving**

resume

– **resume**

Resumes the Conductor's performance, sends **conductorDidResume:** to its delegate, and returns **self**. If the receiver isn't currently paused, this has no effect.

A resumed Conductor's notion of time is frozen while it's paused. For example, if the Conductor was paused 1 beat before it was scheduled to send its next message, the message is sent 1 beat after the Conductor is resumed.

See also: – **pause**, – **pauseFor:**

sel:to:atTime:argCount:

– **sel:**(SEL)*aSelector*
to:*toObject*
atTime:(double)*beats*
argCount:(int)*argCount,...*

Places, in the Conductor's message request queue, a request for *aSelector* to be sent to *toObject* at time *beats* beats from the beginning of the receiver's performance. To ensure that the Conductor's notion of time is up to date, you should send **lockPerformance** before invoking this method. *argCount* specifies the number of four-byte arguments to *aSelector* followed by the arguments themselves, separated by commas (two arguments, maximum).

See also: – **sel:to:withDelay:argCount:**

sel:to:withDelay:argCount:

– **sel:**(SEL)*aSelector*
to:*toObject*
withDelay:(double)*beats*
argCount:(int)*argCount,...*

Places, in the receiver's message request queue, a request for *aSelector* to be sent to *toObject* at time *beats* beats from the receiver's notion of the current time. To ensure that the receiver's notion of time is up to date, you should send **lockPerformance** before invoking this method. *argCount* specifies the number of four-byte arguments to *aSelector* followed by the arguments themselves, separated by commas (two arguments, maximum).

See also: – **sel:to:atTime:argCount:**

setBeatSize:

– (double)**setBeatSize:**(double)*newBeatSize*

Sets the Conductor's tempo by changing the size of a beat to *newBeatSize*, measured in seconds. The default beat size is 1.0 (one second). Attempts to set the tempo of the clockConductor are ignored. Returns the previous beat size.

See also: **–beatSize**, **– setTempo:**, **– tempo**

setDelegate:

– **setDelegate:***delegate*

Sets the Conductor's delegate object to *delegate* and returns **self**. The delegate is sent **conductorDidPause:** and **conductorDidResume:** as the Conductor is paused and resumed, respectively.

See also: **– delegate**, **– pause**, **– pauseFor:**, **– resume:**

setTempo:

– (double)**setTempo:**(double)*newTempo*

Sets the Conductor's tempo to *newTempo*, measured in beats per minute. Attempts to set the tempo of the clockConductor are ignored. Returns the Conductor's old tempo.

See also: **– tempo**, **– setBeatSize:**, **– beatSize**

setTimeOffset:

– (double)**setTimeOffset:**(double)*newTimeOffset*

Sets the Conductor's performance time offset to *newTimeOffset* seconds. Keep in mind that since the offset is measured in seconds, it's not affected by the Conductor's tempo. Attempts to set the offset of the clockConductor are ignored. Returns the old time offset.

See also: **– timeOffset**, **– pause**, **– pauseFor:**

tempo

– (double)**tempo**

Returns the Conductor's tempo in beats per minute.

See also: **– setTempo:**, **– setBeatSize:**, **– beatSize**

time

– (double)**time**

Returns the number of beats that the Conductor has spent in active performance. This excludes time that it or the entire performance has been paused and also excludes the Conductor's performance time offset.

See also: **– predictTime:**

timeOffset

– (double)**timeOffset**

Returns the Conductor's performance time offset in seconds.

See also: **– setTimeOffset:**, **– pause**, **– pauseFor:**

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive a Conductor, call the **NXWriteObject**() C function. An archived Conductor remembers its tempo and time offset when its unarchived.

See also: **– read:**, **– finishUnarchiving**

conductorDidPause:

– **conductorDidPause:***conductor*

Sent to the delegate when *conductor* is paused through a **pause** or **pauseFor:** message. Pausing an entire performance (through **pausePerformance**) doesn't cause this message to be sent.

conductorDidResume:

– **conductorDidResume:***conductor*

Sent to the delegate when *conductor* is resumed through a **resume** message; keep in mind that **pauseFor:** automatically schedules a **resume** message. You should also note that when this message *isn't* sent when a Conductor exhausts its time offset and so begins its performance.

Envelope

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

An Envelope object represents a two-dimensional (Cartesian) coordinate system in which you can order a series of *breakpoints*, each of which is located as a pair of x and y values. An Envelope is defined (primarily) by two arrays: One contains a series of increasing x values, the other contains the corresponding y values.

Envelopes are most often used to control musical attributes during DSP synthesis. This is achieved by associating an Envelope with an AsympUG UnitGenerator (through methods defined by the AsympUG class). The AsympUG produces a continuous signal that follows the shape defined by connecting the Envelope's successive breakpoints with asymptotic curves.

In addition to an Envelope's x and y arrays, you can also provide an array of *smoothing* values. Smoothing is used by an AsympUG to define the slope of the segment into a particular breakpoint (the smoothing value of the first breakpoint is ignored). Smoothing values must be positive and are usually no greater than 1.0. A smoothing of 1.0, the default, provides the gentlest slope possible: The full amount of time between breakpoints is used to travel from one y value to the next. As you decrease the smoothing for a breakpoint, the y value is attained in less time; a smoothing of 0.0 causes the AsympUG to generate the breakpoint's y value instantaneously and constantly until the next breakpoint.

While you must always supply an array of y values when defining an Envelope, the same isn't true for x and smoothing. Rather than provide an x array, you can specify a sampling period that's used as an x increment: The x value of the first breakpoint is 0.0, and successive x values are integer multiples of the sampling period value. Similarly, you can supply a constant smoothing value rather than provide a smoothing array. In the presence of both an x array and a sampling period, or both a smoothing array and a default smoothing, the array takes precedence.

Envelopes are described as having three parts: attack, sustain, and release. You can set the sustain portion of an Envelope by designating one of its breakpoints as the *stickpoint*. Everything up to the stickpoint is the Envelope's attack; everything after the stickpoint is its release. When the stickpoint is reached during DSP synthesis, its y value is sustained until a noteOff arrives to signal the release.

An Envelope object can be set as the value of a Note's parameter through Note's **setPar:toEnvelope:** method. Parameters that accept Envelope objects are usually associated with other, constant-valued parameters that interpret the Envelope by scaling and offsetting the Envelope's x and y values. For example, the **MK_ampEnv** parameter takes an Envelope as its value; **MK_amp0** and **MK_amp1** are

constant-valued parameters that scale and offset the y values in **MK_ampEnv** according to the formula

(*scale** y) + *offset*

where *scale* is calculated as **MK_amp1** – **MK_amp0** and offset is simply the value of **MK_amp0**. In other words, **MK_amp0** defines the interpreted value when y is 0.0 and **MK_amp1** is the interpreted value when y is 1.0. Similarly, the **MK_ampAtt** and **MK_ampRel** parameters are scalers on the x values in the attack and in the release portion of the Envelope, respectively (you can't offset x values).

While Envelope objects are most useful in DSP synthesis, they can also be used to return a discrete value of y for a given x, as provided in the method **lookupYForX:**. If the x value doesn't correspond precisely to a breakpoint in the Envelope, the method does a linear interpolation between the immediately surrounding breakpoints. When used for discrete-value lookup, an Envelope's smoothing values and stickpoint are ignored.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Envelope	double	defaultSmoothing;
	double	samplingPeriod;
	double	*xArray;
	double	*yArray;
	double	*smoothingArray;
	int	stickPoint;
	int	pointCount;

defaultSmoothing	Smoothing for all breakpoints (used in the absence of the smoothing array).
samplingPeriod	Constant x-increment (used in the absence of the x array).
xArray	Array of x values.
yArray	Arrays of y values
smoothingArray	Array of smoothing values.
stickPoint	The object's stickpoint.
pointCount	Number of breakpoints in the object.

METHOD TYPES

Copying, initializing, and freeing an Envelope	– copy – copyFromZone: – init – free
Defining and retrieving breakpoints	– setPointCount:xArray:orSamplingPeriod: yArray:smoothingArray: orDefaultSmoothing: – setPointCount:xArray:yArray: – getNth:x:y:smoothing: – pointCount – xArray – samplingPeriod – yArray – smoothingArray – defaultSmoothing
Attack, sustain, and release	– setStickPoint: – stickPoint – attackDur – releaseDur
Interpolating y values	– lookupYForX:
Writing the Envelope	– writeScorefileStream:
Archiving the Envelope	– write: – read:

INSTANCE METHODS

attackDur

– (double)**attackDur**

Returns the duration of the attack portion of the Envelope. This the difference between the x value of the first breakpoint and the x value of the stickpoint. If the Envelope doesn't have a stickpoint (or if the stickpoint is out of bounds), the duration of the entire Envelope is returned.

See also: – **setStickPoint:**, **releaseDur**

copy

– **copy**

Creates and returns a new Envelope object as a copy of the receiving Envelope.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Same as **copy**, but the new Envelope is allocated in *zone*.

See also: – **copy**

defaultSmoothing

– (double)**defaultSmoothing**

Returns the Envelope's default smoothing value, or MK_NODVAL if there's a smoothing array (use **MKIsNoDVal()** to check MK_NODVAL).

See also: – **setPointCount:...**, – **smoothingArray**

free

– **free**

Frees the Envelope and its contents and removes its name (if any) from the Music Kit name table.

getNth:x:y:smoothing:

– (MKEnvStatus)**getNth:**(int)*n*
x:(double *)*xPtr*
y:(double *)*yPtr*
smoothing:(double *)*smoothingPtr*

Returns, by reference, the x, y, and smoothing values for the *n*'th breakpoint in the Envelope counting from breakpoint 0. The method's return value is a constant that describes the position of the *n*'th breakpoint:

Position	Constant
last point in the Envelope	MK_lastPoint
stickpoint	MK_stickPoint
point out of bounds	MK_noMorePoints
any other point	MK_noEnvError

If the Envelope's y array is NULL, or its x array is NULL and its sampling period is 0.0, **MK_noMorePoints** is returned.

See also: – **setPointCount:...**, – **pointCount**, – **xArray**, – **yArray**, – **smoothingArray**

init

– **init**

Initializes the Envelope by setting its default smoothing to 1.0, its sampling period to 1.0, and its stickpoint to MAXINT. You never invoke this method directly. A subclass implementation should send **[super init]** before performing its own initialization. Returns **self**.

lookupYForX:

– (double)**lookupYForX:**(double)*xVal*

Returns the y value that corresponds to *xVal*. If *xVal* doesn't fall precisely on one of the Envelope's breakpoints, the return value is computed as a linear interpolation between the y values of the nearest breakpoints on either side of *xVal*. If *xVal* is out of bounds, this returns the first or last y value, depending on which boundary was exceeded. If the Envelope's y array is NULL, this returns MK_NODVAL (use **MKIsNoDVal**() to check MK_NODVAL).

pointCount

– (int)**pointCount**

Returns the number of breakpoints in the Envelope.

See also: – **setPointCount:...**

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived Envelope, call the **NXReadObject**() C function.

See also: – **write:**

releaseDur

– (double)**releaseDur**

Returns the duration of the release portion of the Envelope. This is the difference between the x value of the stickpoint and the x value of the final breakpoint. Returns 0.0 if the Envelope doesn't have a stickpoint, or if the stickpoint is out of bounds.

See also: – **setStickPoint:**, **releaseDur**

samplingPeriod

– (double)**samplingPeriod**

Returns the sampling period, or MK_NODVAL if there's an x array (use **MKIsNoDVal()** to check MK_NODVAL).

See also: – **setPointCount:...**, – **xArray**

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

– **setPointCount:**(int)*count*
xArray:(double *)*xPtr*
orSamplingPeriod:(double)*period*
yArray:(double *)*yPtr*
smoothingArray:(double *)*smoothingPtr*
orDefaultSmoothing:(double)*smoothing*

Fills the Envelope with data by copying the first *count* values from *xPtr*, *yPtr*, and *smoothingPtr*. If *xPtr* is NULL, the Envelope's sampling period is set to *period* (otherwise *period* is ignored). Similarly, *smoothing* is used as the Envelope's default smoothing in the absence of *smoothingPtr*. If *yPtr* is NULL, the Envelope's y array is unchanged. Returns **self**.

See also: – **setPointCount:xArray:yArray:**, – **pointCount**, – **xArray**, – **yArray**, – **smoothingArray**, – **samplingPeriod**, – **defaultSmoothing**

setPointCount:xArray:yArray:

– **setPointCount:**(int)*count*
xArray:(double *)*xPtr*
yArray:(double *)*yPtr*

This is a cover for the more complete **setPointCount:xArray:orSamplingPeriod:...** method. The Envelope's smoothing specification is unchanged (smoothing is initialized to a constant 1.0). If *xPtr* or *yPtr* is NULL, the Envelope's x or y array is unchanged, respectively. Returns **self**.

See also: – **setPointCount:xArray:orSamplingPeriod:...**, – **pointCount**, – **xArray**, – **yArray**

setStickPoint:

– **setStickPoint:**(int)*index*

Sets the Envelope's stickpoint to the *index*'th breakpoint, counting from 0. Returns **self**, or **nil** if *index* is out of bounds.

See also: – **stickPoint**

smoothingArray

– (double *)**smoothingArray**

Returns a pointer to the Envelope's smoothing array, or NULL if none.

See also: – **setPointCount:...**, – **defaultSmoothing**

stickPoint

– (int)**stickPoint**

Returns the index of the stickpoint, or MAXINT if none.

See also: – **setStickPoint:**

writeScorefileStream:

– **writeScorefileStream:**(NXStream *)*aStream*

Writes the Envelope to the stream *aStream* in scorefile format. The stream must already be open. The Envelope's breakpoints are written, in order, as (*x, y, smoothing*) with the stickpoint followed by a vertical bar. For example, a simple three-breakpoint Envelope describing an arch might look like this (the second breakpoint is the stickpoint):

```
(0.0, 0.0, 0.0) (0.3, 1.0, 0.05) | (0.5, 0.0, 0.2)
```

Returns **nil** if the Envelope's y array is NULL. Otherwise returns **self**.

xArray

– (double *)**xArray**

Returns a pointer to the Envelope's x array, or NULL if none.

See also: – **setPointCount:...**, – **samplingPeriod**

yArray

– (double *)**yArray**

Returns a pointer to the Envelope's y array, or NULL if none.

See also: – **setPointCount:...**

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive an Envelope, call the **NXWriteObject()** C function.

See also: – **read**

FilePerformer

INHERITS FROM Performer : Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A FilePerformer is an abstract class that provides methods for performing time-ordered music data from a file or a stream. The Music Kit includes a single subclass, ScorefilePerformer, that reads and performs data from a scorefile.

You establish a FilePerformer object's source of data through one of two methods (but never both):

- The **setFile:** method associates a FilePerformer with a file name. The object opens and closes the file for you as a performance begins and ends. The file name is remembered between performances.

- The **setStream:** method associates a FilePerformer with an NXStream. Opening and closing the stream is the responsibility of your application. The FilePerformer's stream pointer is set to NULL after each performance so you must send another **setStream:** message to replay the stream.

You can restrict the data that the FilePerformer will perform through the **setFirstTimeTag:** and **setLastTimeTag:** methods. As a FilePerformer fashions Notes from its source, it only performs those Notes that have time tags within the given range.

The FilePerformer class declares **nextNote** and **performNote:**, both of which are invoked automatically during a performance, as subclass responsibilities:

- A subclass implementation of **nextNote** reads data from the **stream** instance variable and from it creates either a Note object or a time tag for the following Note (regardless of how the source of data was declared—whether through **setFile:** or **setStream:**—the **stream** variable is guaranteed to be open for reading while a performance is in progress). It returns the Note that it creates, or, in the case of a time tag, it sets the instance variable **fileTime** to this value and returns **nil** (the **fileTime** variable supercedes the **nextPerform** variable inherited from Performer—FilePerformers never set **nextPerform** directly). When **stream** has been wrung dry, **nextNote** should set **fileTime** to MK_ENDOFTIME; this will cause the FilePerformer to be deactivated.

- You implement **performNote:** to perform the Note that's passed as its argument (this method supercedes the **perform** method declared as a subclass responsibility by Performer—a subclass of FilePerformer needn't implement **perform**). Typically, this means passing the Note as the argument to **sendNote:**, sent to the FilePerformer's NoteSenders (creation of a FilePerformer's NoteSenders is left to the subclass).

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Performer	id	conductor;
	MKPerformerStatus	status;
	int	performCount;
	double	timeShift;
	double	duration;
	double	time;
	double	nextPerform;
	id	noteSenders;
	id	delegate;
Declared in FilePerformer	char	*filename;
	double	fileTime;
	NXStream	*stream;
	double	firstTimeTag;
	double	lastTimeTag;

filename	The object's file name, if set
fileTime	The current time in the file, in beats
stream	The object's NXStream pointer
firstTimeTag	The FilePerformer's least time tag value
lastTimeTag	The FilePerformer's greatest time tag value

METHOD TYPES

Copying and Initializing a FilePerformer	
	– copyFromZone:
	– init
Defining a subclass	+ fileExtensions
	– activateSelf
	– initializeFile
	– nextNote
	– performNote:
	– deactivateSelf
	– finishFile
Accessing the object's data	– setFile:
	– setStream:
	– file
	– stream

Restricting the object's data	– setFirstTimeTag: – setLastTimeTag: – firstTimeTag – lastTimeTag
Archiving the object	– read: – write:

CLASS METHODS

fileExtensions

+ (char **)**fileExtensions**

You can implement this method in a subclass to return a NULL-terminated array of file name extensions that your subclass recognizes. When a FilePerformer is activated, these extensions are appended, one-by-one, to the given file name (as set through **setFile:**) until a match is found. The unadorned file name taken literally as the argument to **setFile:** is always searched for first. Files set through **setStream:** are exempt from all this mucking around: The file name appendix is manipulated *only* if the file is set through **setFile:**.

INSTANCE METHODS

activateSelf

– **activateSelf**

Prepares the FilePerformer for a performance by doing the following:

1. If the object's data source was set through **setFile:**, the file is located (see **fileExtensions**) and the **stream** instance variable is opened to the file.

2. The **initializeFile** message is sent to **self**.

3. **nextNote** is invoked until it returns a Note with a time tag equal to or greater than the FilePerformer's first time tag value.

If **stream** can't be opened, if **initializeFile** returns **nil**, or if an appropriate Note isn't found, the FilePerformer is deactivated. You never invoke this method; it's invoked automatically by the **activate** method inherited from Performer.

copyFromZone

– **copyFromZone:**(NXZone *)*zone*

Creates and returns a FilePerformer as a copy of the receiving FilePerformer. The new object copies the receiver's NoteSenders and file name, its **stream** variable is set to NULL, and it's inactive.

deactivateSelf

– **deactivateSelf**

Deactivates the FilePerformer by invoking **finishFile** and setting **stream** to NULL. You never invoke this method; it's invoked automatically when the FilePerformer receives the **deactivate** message.

file

– (char *)**file**

Returns the FilePerformer's file name, as set through **setFile:**.

finishFile

– **finishFile**

You never invoke this method; it's invoked automatically by **deactivateSelf**. A subclass can implement this method for post-performance operations. You shouldn't close the **stream** pointer as part of this method. The default implementation does nothing. The return value is ignored.

firstTimeTag

– (double)**firstTimeTag**

Returns the least time tag value that the FilePerformer considers for performance, as set through **setFirstTimeTag:**.

init

– **init**

Initializes a recently allocated FilePerformer. A subclass implementation should send **[super init]** before performing its own initialization. Returns **self**.

initializeFile

– **initializeFile**

Invoked automatically by **activateSelf**, a subclass can implement this method to perform file initialization; the file is guaranteed to be open and accessible through the

stream instance variable. If **nil** is returned, the FilePerformer is deactivated. The default implementation does nothing and returns **self**.

lastTimeTag

– (double)**lastTimeTag**

Returns the greatest time tag value that the FilePerformer considers for performance, as set through **setLastTimeTag:**.

nextNote

– **nextNote**

This is a subclass responsibility that's expected to read data from the **stream** instance variable and from it fashion a Note object or a time tag value, as explained in detail in the class description, above. You never invoke this method; it's invoked automatically by the **perform** method.

perform

– **perform**

You never invoke this method, nor should you reimplement it in a subclass. It defines a FilePerformer's general performance instructions, as required by the Performer class. To wit: It invokes **nextNote** until that method returns **nil** and passes each Note returned by **nextNote** as the argument in a **performNote:** message sent to **self**.

performNote:

– **performNote:***aNote*

This is a subclass responsibility that's expected to perform its argument, *aNote*, as explained in detail in the class description, above. You never invoke this method; it's invoked automatically by the **perform** method. The return value is ignored.

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived FilePerformer, call the **NXReadObject()** C function.

setFile:

– **setFile:**(char *)*aName*

Associates the FilePerformer with the file named *aName*. The file is opened when the FilePerformer is activated and closed when it's deactivated. While it's open, the file can be read through the **stream** instance variable. If the FilePerformer is active, this does

nothing and returns **nil**, otherwise returns **self**. Invoking this method invalidates a previous invocation of **setStream:**. A FilePerformer remembers its file name between performances (unlike its amnesia with regard to its stream).

setFirstTimeTag:

– **setFirstTimeTag:**(double)*aTimeTag*

Sets the smallest time tag considered for performance to *aTimeTag* and returns **self**. If the FilePerformer is active, does nothing and returns **nil**.

setLastTimeTag:

– **setLastTimeTag:**(double)*aTimeTag*

Sets the largest time tag considered for performance to *aTimeTag* and returns **self**. If the FilePerformer is active, does nothing and returns **nil**.

setStream:

– **setStream:**(NXStream *)*aStream*

Sets the FilePerformer's stream pointer (the **stream** instance variable) to *aStream*, which must already be open for reading. If the FilePerformer is active, this does nothing and returns **nil**, otherwise returns **self**. Invoking this method invalidates a previous invocation of **setFile:**. Keep in mind that the stream variable is set to NULL after each performance; to perform the same stream twice, you must resend **setStream:** before each performance.

stream

– (NXStream *)**stream**

Returns the FilePerformer's stream pointer. If you set the FilePerformer's file through **setStream:**, the value returned here is the value passed as the argument to that method. If you set the file through **setFile:**, this method returns a stream pointer to the file only if the FilePerformer is active.

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive a FilePerformer, you call the **NXWriteRootObject**() C function.

FileWriter

INHERITS FROM	Instrument : Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

A FileWriter is an Instrument that realizes Notes by writing them to a file on the disk. An abstract class, FileWriter implements methods that locate, open, and close files; it's left to the FileWriter subclass to define the format in which the Notes are written. This task is met in the subclass' implementation of **realizeNote:fromNoteFileWriter:**. The Music Kit's only FileWriter subclass, ScorefileWriter, writes Notes in scorefile format.

You identify a FileWriter's file either by the file's name or as an open NXStream. If the file is identified by name (through the **setFile:** method) the FileWriter object opens and closes the file for you: The file is opened for writing when the object first receives the **realizeNote:fromNoteReceiver:** message and closed after the performance. A FileWriter remembers its file's name between performances, but the file is overwritten each time it's opened.

The **setStream:** method sets the FileWriter's file to an NXStream. Opening and closing the stream is the responsibility of the application. A FileWriter forgets the designated stream between performances; if you want to write to the same stream on successive performances, you must send **setStream:** before each.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Instrument	id	noteFileWriters;
Declared in FileWriter	MKTimeUnit	timeUnit;
	char	*filename;
	NXStream	*stream;
	double	timeShift;

timeUnit	Either MK_second or MK_beat; used to compute a Note's time tag and duration.
filename	The object's file name.
stream	The object's stream pointer.
timeShift	Optional time tag value offset.

METHOD TYPES

Copying and initializing a FileWriter	– copyFromZone: – init
Defining a subclass	+ fileExtension
Accessing the file	– setFile: – file – setStream: – stream
Initializing and finishing the file	– initializeFile – firstNote: – afterPerformance – finishFile
Interpreting time	– setTimeShift: – timeShift – setTimeUnit: – timeUnit
Archiving the object	– read: – write:

CLASS METHODS

fileExtension

+ (char *)**fileExtension**

Returns the file name extension that's used by the class. The value returned by this method is automatically appended to the names of the files that are written by a FileWriter (even if the name already contains an extension). The default implementation returns NULL. A subclass may override this method to return its own extension (the return value *shouldn't* include the initial ".").

INSTANCE METHODS

afterPerformance

– **afterPerformance**

You never invoke this method; it's invoked automatically just after a performance. It closes the FileWriter's **stream** variable (if the FileWriter opened it itself in the **firstNote:** method) and sets it to NULL.

copyFromZone:

– **copyFromZone:**(NXZone)*zone*

Creates and returns a copy of the FileWriter. The new object's file is undefined.

file

– (char *)**file**

Returns the FileWriter's file name, as set through **setFile:**.

finishFile

– **finishFile**

This can be overridden by a subclass to perform post-performance activities. However, the implementation shouldn't close the FileWriter's NXStream pointer. You never send the **finishFile** message directly to a FileWriter; it's invoked automatically after each performance. The return value is ignored.

firstNote:

– **firstNote:***aNote*

You never invoke this method; it's invoked automatically just before the FileWriter writes its first Note. It opens the FileWriter's file (if set through **setFile:**) and then sends **initializeFile** to **self**.

init

– **init**

Initializes the FileWriter.

initializeFile

– **initializeFile**

You never invoke this method directly; it's invoked automatically (from within the **firstNote:** method) when the FileWriter receives its first Note. A subclass can override this method to perform file initialization, such as writing a header. When this method is invoked, the file is guaranteed to be open for writing and can be accessed through the **stream** instance variable. The return value is ignored.

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived FileWriter, call the **NXReadObject()** C function.

setFile:

– **setFile:**(char *)*aName*

Associates the FileWriter with the file *aName*. The file is opened when the first Note is realized (written to the file) and closed at the end of the performance. If the FileWriter is already in a performance, this does nothing and returns **nil**; otherwise it returns **self**.

setStream:

– **setStream:**(NXStream *)*aStream*

Sets the FileWriter's file as the stream *aStream*. You must open and close the stream yourself. If the FileWriter is already in a performance, this does nothing and returns **nil**; otherwise it returns **self**.

setTimeShift:

– **setTimeShift:**(double)*timeShift*

Sets the FileWriter's performance time offset, in seconds, to *timeShift*. The offset, which can be negative, is added to the time tag value of each Note that's written by the FileWriter.

setTimeUnit:

– **setTimeUnit:**(MKTimeUnit)*aTimeUnit*

Sets the unit in which the FileWriter measures time, thus affecting the time tag and duration values of the Notes it writes. The argument can be MK_second for measurement in seconds, or MK_beat for beats. The default is MK_second.

stream

– (NXStream *)**stream**

Returns the FileWriter's stream pointer, or NULL if it isn't set. The pointer is set to NULL after each performance.

timeShift

– (double)**timeShift**

Returns the FileWriter's performance time offset, in seconds, as set through **setTimeShift:**. The default is 0.0.

timeUnit

– (MKTimeUnit)**timeUnit**

Returns the unit in which the FileWriter measures time, either MK_second or MK_beat. The default is MK_second.

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive a FileWriter, call the **NXWriteRootObject()** C function. A FileWriter archives its **filename**, **timeUnit**, and **timeShift** instance variables (as well as the instance variables defined in Instrument).

Instrument

INHERITS FROM Object

DECLARED IN musickit.hu

CLASS DESCRIPTION

Instrument is an abstract class that defines the general mechanism for receiving and realizing Notes during a Music Kit performance. An Instrument receives Notes through its NoteReceivers, auxiliary objects that are typically connected to a Performer's NoteSenders. The manner in which an Instrument realizes Notes is defined in its implementation of **realizeNote:fromNoteReceiver:**. This method is automatically invoked by an Instrument's NoteReceivers, when such objects receive **receiveNote:** messages.

An Instrument is considered to be in performance from the time that one of its NoteReceivers invokes the **realizeNote:fromNoteReceiver:** method until the Conductor class receives the **finishPerformance** message. There are two implications regarding an Instrument's involvement in a performance:

- An Instrument's **firstNote:** and **afterPerformance** methods are invoked as the Instrument begins and finishes its performance, respectively. These methods can be implemented in a subclass to provide specialized initialization and post-performance cleanup.

- Some Instrument methods can't be invoked during a performance. For example, you can't add or remove NoteReceivers while the Instrument is performing.

Creating and adding NoteReceivers to an Instrument object is generally the obligation of the Instrument subclass; most subclasses dispose of this duty in their **init** methods. However, instances of some subclasses are born with no NoteReceivers—they expect these objects to be added by your application. You should visit the class description of the Instrument subclass that you're using to determine just what sort of varmint you're dealing with.

The Music Kit defines a number of Instrument subclasses. Notable among these are: SynthInstrument, which synthesizes Notes on the DSP; PartRecorder adds Notes to a designated Part; ScorefileWriter writes them to a scorefile; and NoteFilter, an abstract class that acts as a Note conduit, altering the Notes that it receives before passing them on to other Instruments. In addition, the Midi class can be used as an Instrument to realize Notes on an external MIDI synthesizer.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Instrument	id	noteReceivers;
noteReceivers	The object's List of NoteReceivers.	

METHOD TYPES

Creating and freeing an Instrument	– copy – copyFromZone: – init – free
Manipulating NoteReceivers	– addNoteReceiver: – noteReceiver – noteReceivers – isNoteReceiverPresent: – removeNoteReceiver: – removeNoteReceivers – freeNoteReceivers
Performing	– firstNote: – afterPerformance – inPerformance
Realizing Notes	– realizeNote:fromNoteReceiver:
Archiving	– write: – read:

INSTANCE METHODS

addNoteReceiver:

– **addNoteReceiver:***aNoteReceiver*

Adds *aNoteReceiver* to the Instrument, first removing it from its current Instrument, if any. If the receiving Instrument is in performance, this does nothing and returns **nil**, otherwise returns *aNoteReceiver.*

See also: – **removeNoteReceiver:**, – **noteReceivers**, – **isNoteReceiverPresent:**

afterPerformance

– **afterPerformance**

You never invoke this method; it's automatically invoked when the performance is finished. A subclass can implement this method to do post-performance cleanup. The default implementation does nothing; the return value is ignored.

See also: – **firstNote:**, – **inPerformance**

copy

– **copy**

Creates and returns a new Instrument as a copy of the receiving Instrument. The new object has its own NoteReceiver collection that contains copies of the Instrument's NoteReceivers. The new NoteReceivers' connections (see the NoteReceiver class) are copied from the NoteReceivers in the receiving Instrument.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

This is the same as **copy**, but the new object is allocated from *zone*.

See also: – **copy**

firstNote:

– **firstNote:***aNote*

You never invoke this method; it's invoked just before the Instrument realizes its first Note. A subclass can implement this method to perform pre-realization initialization. The argument is the Note that the Instrument is about to realize; it's provided as a convenience and can be ignored in a subclass implementation. The Instrument is considered to be in performance after this method returns. The return value is ignored.

See also: – **afterPerformance**, – **inPerformance**

free

– **free**

Frees the Instrument and its NoteReceivers. If the Instrument is in performance, this does nothing and returns **self**, otherwise returns **nil**.

See also: – **freeNoteReceivers**

freeNoteReceivers

– freeNoteReceivers

Disconnects, removes, and frees the Instrument's NoteReceivers. No checking is done to determine if the Instrument is in performance. Returns **self**.

See also: **– removeNoteReceivers:**

inPerformance

– (BOOL)**inPerformance**

Returns YES if the Instrument is in performance. Otherwise returns NO. An Instrument is considered to be in performance from the time that one of its NoteReceivers invokes **realizNote:fromNoteReceiver:**, until the time that the Conductor class receives **finishPerformance.**

See also: **– firstNote:**, **– afterPerformance**

init

– init

Initializes an Instrument that was created through **allocFromZone:**. Returns **self**.

isNoteReceiverPresent:

– (BOOL)**isNoteReceiverPresent:***aNoteReceiver*

Returns YES if *aNoteReceiver* is in the Instrument's NoteReceiver List. Otherwise returns NO.

See also: **– noteReceiver**, **– noteReceivers**

noteReceiver

– noteReceiver

Returns the first NoteReceiver in the Instrument's NoteReceiver List. This is useful if you want to send a Note directly to an Instrument, but you don't care which NoteReceiver does the receiving:

```
[[anInstrument noteReceiver] receiveNote:aNote]
```

See also: **– addNoteReceiver**, **– noteReceivers**, **– isNoteReceiverPresent**

noteReceivers

– **noteReceivers**

Creates and returns a List that contains the Instrument's NoteReceivers. It's the sender's responsibility to free the List.

See also: – **addNoteReceiver**, – **noteReceiver**, – **isNoteReceiverPresent**

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived Note, call the **NXReadObject**() C function.

See also: – **write:**

realizeNote:fromNoteReceiver:

– **realizeNote:***aNote* **fromNoteReceiver:***aNoteReceiver*

You implement this method in a subclass to define the manner in which the subclass realizes Notes. *aNote* is the Note that's to be realized; *aNoteReceiver* is the NoteReceiver that received it. The default implementation does nothing; the return value is ignored.

You never invoke this method from your application; it should only be invoked by the Instrument's NoteReceivers as they are sent **receiveNote:** messages. Keep in mind that you can send **receiveNote:** directly to a NoteReceiver.

removeNoteReceiver:

– **removeNoteReceiver:***aNoteReceiver*

Removes *aNoteReceiver* from the Instrument's NoteReceiver List, but neither disconnects the NoteReceiver from its connected NoteSenders nor frees the NoteReceiver. If the Instrument is in performance, this does nothing and returns **nil**; otherwise returns *aNoteReceiver*.

See also: – **removeNoteReceivers**, – **addNoteReceiver**, – **noteReceivers**, – **isNoteReceiverPresent**

removeNoteReceivers

– **removeNoteReceivers**

Removes all the Instrument's NoteReceivers but neither disconnects nor frees them. Returns **self**.

See also: – **removeNoteReceiver**, – **addNoteReceiver**, – **noteReceivers**, – **isNoteReceiverPresent**

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive an Instrument, call the **NXWriteRootObject()** C function. The Instrument's NoteReceivers are archived through **NXWriteObject()**.

See also: – **read:**

Midi

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A Midi object provides a simple interface to the MIDI device driver. The Midi class also provides a mechanism that automatically converts MIDI messages into Note objects and vice versa, allowing you to incorporate MIDI data into a Music Kit application with a minimum of effort. A Midi object is specified as it corresponds uniquely to a serial port; since there are only two serial ports, there can be but two distinct Midi objects within your application.

The Midi class emulates Performer and Instrument in that it's instances contain NoteSenders and NoteReceivers: As a Midi object receives messages from the MIDI driver, it fashions Note objects and issues these Notes into a performance through its NoteSenders. Analogously, a Midi object receives Notes through its NoteReceivers, turns them into MIDI messages, and sends the messages to the MIDI driver.

A Midi object automatically creates 17 NoteSenders. The first (NoteSender 0) corresponds to the channel used for MIDI System and Channel Mode Messages. The other 16 (NoteSenders 1 through 16) correspond to the 16 MIDI Voice Channels. The NoteSender through which a Midi object issues a particular Note corresponds to the channel on which it was received. Alternatively, you can tell a Midi object to issue all Notes through NoteSender 0 by sending it the **setMergeInput:YES** message. In this case, each Note is given a MK_midiChan parameter that indicates the original channel.

NoteReceiver 0 is the analog of NoteSender 0 in merge-input mode: When it receives a Note on NoteReceiver 0, a Midi object reads the Note's MK_midiChan parameter and realizes the Note on that channel. The other 16 NoteReceivers correspond to the 16 MIDI Voice Channels.

Before a Midi object can receive or send MIDI messages, it must be opened and started: **open** establishes communication between the object and the MIDI driver, **run** starts the driver's clock ticking. Balancing these two methods are **stop**, which stops the driver's clock, and **close**, which breaks communication between the object and the MIDI driver. These methods change the state of the Midi object:

- MK_devOpen. The Midi object is open but not running.
- MK_devRunning. The object is open and running.
- MK_devStopped. The object has been running but is now stopped.
- MK_devClosed. The object is closed.

As you start, pause, resume, and stop a performance, you should similarly control your Midi objects, as described by the following table:

To the Conductor class	To your Midi objects
startPerformance	run
pausePerformance	stop
resumePerformance	run
finishPerformance	close

The MIDI driver has its own clock that's more reliable than the Conductor's clock. To take advantage of this, the Conductor's clock is synched to the driver's clocked—this is particularly beneficial when you're recording MIDI. However, if you're receiving MIDI data and processing it in real time, it's better to decouple the Conductor from the MIDI driver by sending **setUseInputTimeStamps:NO** to the Midi object.

As a Midi object initiates an outgoing MIDI message it gives the message a timestamp that indicates when the message should be sent into the real world by the MIDI driver. By sending **setOutputTimed:NO** to a Midi object, you can specify that the driver is to ignore the timestamps and send all messages as soon as it receives them. This improves real-time response, but at the expense of possible rhythmic unsteadiness.

MIDI to Note conversion

When a Midi object receives a MIDI message, it creates a Note object of a particular note type and with particular parameters according to the following rules:

- If the message is a MIDI Note On with a Key Velocity greater than 0, a noteOn is created and given key number (MK_keyNum) and velocity (MK_velocity) parameters that correspond to the message's Key Number and Key Velocity values. The note tag is reckoned from the message's Channel Number and Key Number.

- A MIDI Note Off, or a Note On with 0 Key Velocity prompts a noteOff. If the Note Off contains a Release Key Velocity value greater than 0, the noteOff's MK_relVelocity parameter will reflect this. The note tag is as with a noteOn.

- A MIDI Channel Voice message other than Note On and Note Off prompts a noteUpdate that contains one of MK_keyPressure, MK_afterTouch, MK_controlChange, MK_pitchBend, or MK_programChange, depending on the MIDI message. If it contains MK_controlChange, the Note will also contain a MK_controlVal parameter. If MK_keyPressure, the Note is given a note tag.

- Notes created from any other message, such as Channel Mode and System messages, are mutes. Its parameters are described below.

The parameters in a mute are devised as follows:

- MIDI Channel Mode messages dissolve into two parameters: MK_basicChan and MK_chanMode. The former records the Basic Channel while the latter takes one of the following values: MK_resetControllers, MK_localControlModeOn, MK_localControlModeOff, MK_allNotesOff, MK_omniModeOff, MK_omniModeOn, MK_monoMode, and MK_polyMode.

- Parameters that correspond to MIDI System Common messages are MK_timeCodeQ (MIDI time code, quarter frame), MK_songPosition, MK_songSelect, and MK_tuneRequest.

- A MIDI System Real Time message heralds a MK_sysRealTime parameter; it's possible values are MK_sysClock, MK_sysStart, MK_sysContinue, MK_sysStop, MK_sysActiveSensing, and MK_sysReset.

- The parameter MK_sysExclusive corresponds to a MIDI system exclusive message. Its value is a C string, with each MIDI byte encoded as a pair of hexadecimal digits delimited by a comma.

Note to MIDI conversion

- If two successive noteOns have the same note tag and the same MK_keyNum value, an intervening Note Off message is generated and sent.

- If two successive noteOns have the same note tag but different MK_keyNum values, the second Note On message is immediately followed by a Note Off with the Key Number of the first Note On (in other words, the first Note On is silenced).

- If a noteOn has no MK_keyNum parameter, a value is generated from the MK_freq parameter, if any, otherwise a default value of 64 is used.

- A noteDur is split into noteOn/noteOff pair and the separate Notes are processed.

- A noteOn or noteOff without a note tag, a noteOff with an inactive note tag, or an MK_keyPressure noteUpdate with an inactive or missing note tag is ignored.

- A noteOff with no MK_relVelocity parameter prompts a Note On with 0 Velocity.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Midi	id	noteSenders;
	id	noteReceivers;
	MKDeviceStatus	deviceStatus;
	char	*midiDev;
	BOOL	useInputTimeStamps;
	BOOL	outputIsTimed;
	double	localDeltaT;

noteSenders	The object's collection of NoteSenders.
noteReceivers	The object's collection of NoteReceivers
deviceStatus	The object's status.

midiDev	Midi device port name.
useInputTimeStamps	YES if Conductor is updated by the driver.
outputIsTimed	YES if the driver's clock is used for output.
localDeltaT	Offset added to MIDI output time stamps.

METHOD TYPES

Creating and freeing a Midi object	– free + new + newOnDevice:
Querying the object	– channelNoteReceiver: – channelNoteSender: – conductor – deviceStatus – localDeltaT – noteReceiver – noteReceivers – noteSender – noteSenders – outputIsTimed – useInputTimeStamps
Modifying the object	– acceptSys: – ignoreSys: – setLocalDeltaT: – setMergeInput: – setOutputTimed: – setUseInputTimeStamps:
Opening and running the object	– abort – close – open – openInputOnly – openOutputOnly – run – stop

CLASS METHODS

alloc
allocFromZone:

You never invoke these methods; Midi overrides them to generate errors.

new

+ new

If a Midi object for the device "midi1" (the default MIDI device name) doesn't already exist, this creates such an object. Otherwise, returns the existing object.

newOnDevice:

+ newOnDevice:(char *)*devName*

If a Midi object for the device *devName* doesn't already exist, this creates such an object. Otherwise, returns the existing object. The argument must be either "midi0" or "midi1".

INSTANCE METHODS

abort

– abort

Immediately stops and closes the Midi object, sets its status to MK_devClosed, and releases the device port. A more graceful approach is to invoke **stop** and **close**.

acceptSys:

– acceptSys:(MKMidiParVal)*param*

Instructs the Midi object to accept incoming MIDI messages that set the parameter MK_sysRealTime to the value specified in *param*, which must be one of MK_sysStop, MK_sysStart, MK_sysContinue, MK_sysClock, or MK_sysActiveSensing.

channelNoteReceiver:

– channelNoteReceiver:(unsigned)*n*

Returns the NoteReceiver for MIDI channel *n*, as explained in the class description.

channelNoteSender:

– channelNoteSender:(unsigned)*n*

Returns the NoteSender for MIDI channel *n*, as explained in the class description.

close

– close

Waits for the Midi object's output queue to empty and then closes the Midi object, sets its status to MK_devClosed, and releases the device port. Returns **self**.

conductor

– **conductor**

Always returns the clockConductor.

deviceStatus

– (MKDeviceStatus)**deviceStatus**

Returns the Midi object's device status, as listed in the class description.

free

– **free**

Closes and frees the Midi object and frees its NoteSenders and NoteReceivers.

ignoreSys:

– **ignoreSys:**(MKMidiParVal)*param*

Instructs the Midi object to ignore messages that set the MK_sysRealTime parameter to *param*. The list of values that are ignored by default is given in **acceptSys:**.

init

You never invoke this method; Midi overrides it to generate an error.

localDeltaT

– (double)**localDeltaT**

Returns the Midi object's local delta time, in seconds, as set through **setLocalDeltaT:**. The local delta time is added to the global delta time, as set through **MKSetDeltaT**(), and the sum is added into each MIDI driver timestamp. This has no effect if the Midi object isn't timed. The default is 0.0.

noteReceiver

– **noteReceiver**

Returns the Midi object's first NoteReceiver (NoteReceiver 0).

noteReceivers

– **noteReceivers**

Returns a List containing the Midi object's NoteReceivers.

noteSender

– **noteSender**

Returns the Midi object's first NoteSender (NoteSender 0).

noteSenders

– **noteSenders**

Returns a List containing the Midi object's NoteSenders.

open

– **open**

Opens the Midi object for two-way communication with the MIDI driver. The object's status is set to MK_devOpen. If the object is open in only one direction, **close** is first invoked. Returns **self**, or **nil** if the object can't be opened.

openInputOnly

– **openInputOnly**

Opens the Midi object for input from the MIDI driver. If the Midi object is open in both directions or for output only, **close** is first invoked. Returns **self**, or **nil** if the object can't be opened.

openOutputOnly

– **openOutputOnly**

Opens the Midi object for output to the MIDI driver. If the object is open in both directions or for input only, **close** is first invoked. Returns **self**, or **nil** if the object can't be opened.

outputIsTimed

– (BOOL)**outputIsTimed**

Returns YES if the messages sent by the Midi object to the MIDI driver are given timestamps, otherwise returns NO. The default is YES.

run

– **run**

Opens the Midi object (if necessary), starts its clock, and sets the Midi object's status to MK_devRunning. Returns **self**, or **nil** if it's closed and can't be opened.

setLocalDeltaT:

– **setLocalDeltaT:**(double)*seconds*

Sets the Midi object's local delta time, in seconds, to *seconds*; the default is 0.0. The local delta time is added to the global delta time, as set through **MKSetDeltaT()**, and the sum is added into each timestamp before it's passed to the MIDI driver. This has no effect if the Midi object isn't timed. Returns **self**.

setMergeInput:

– **setMergeInput:**(BOOL)*yesOrNo*

If *yesOrNo* is YES, each Note fashioned by the Midi object from a MIDI message is given an MK_midiChan parameter with a value set to the channel on which the Note was received. All Notes are then sent to the Midi object's NoteSender 0. By default, the input isn't merged.

setOutputTimed:

– **setOutputTimed:**(BOOL)*yesOrNo*

Establishes whether MIDI messages are sent to the MIDI driver with or without timestamp values, as *yesOrNo* is YES or NO. If the Midi object is timed, messages are stamped with the Conductor's notion of the current time plus the global and local delta times. If it's untimed, the timestamps are always 0, indicating to the MIDI driver that the messages should be sent immediately. The default is timed.

setUseInputTimeStamps:

– **setUseInputTimeStamps:**(BOOL)*yesOrNo*

If *yesOrNo* is YES the Conductor's clock is synched to the MIDI driver's clock as the Midi object receives MIDI messages. If the Midi object isn't closed, this does nothing and returns **nil**; otherwise returns self. The two clocks are synched by default.

stop

– **stop**

Stops the Midi object's clock and sets it's status to MK_devStopped. Returns **self**.

useInputTimeStamps

– (BOOL)**useInputTimeStamps**

Returns YES or NO as the Conductor's clock and the MIDI driver's clock are synchronized, as set through **setUseInputTimeStamps:**. The default is YES.

Note

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

Note objects are containers of musical information. The amount and type of information that a Note can hold is practically unlimited; however, you should keep in mind that Notes haven't the ability to act on this information, but merely to store it. It's left to other objects to read and process the information in a Note. Most of the other Music Kit classes are designed around Note objects, treating them as common currency. For example, Part objects store Notes, Performers acquire them and pass them to Instruments, Instruments read the contents of Notes and apply the information therein to particular styles of realization, and so on.

The information that comprises a Note defines the attributes of a particular musical event. Typically, an object that uses Notes plucks from them just those bits of information in which it's interested. Thus you can create Notes that are meaningful in more than one application. For example, a Note object that's realized as synthesis on the DSP would contain many particles of information that are used to drive the synthesis machinery; however, this doesn't mean that the Note can't also contain graphical information, such as how the Note would be rendered when drawn on the screen. The objects that provide the DSP synthesis realization (SynthPatch objects, as defined by the Music Kit) are designed to read just those bits of information that have to do with synthesis, and ignore anything else the Note contains. Likewise, a notation application would read the attributes that tell it how to render the Note graphically, and ignore all else. Of course, some information, such as the pitch and duration of the Note, would most likely be read and applied in both applications.

Most of the methods defined by the Note class are designed to let you set and retrieve information in the form of *parameters*. A parameter consists of a tag, a name, a value, and a data type:

- A parameter tag is a unique integer used to catalog the parameter within the Note; the Music Kit defines a number of parameter tags such as MK_freq (for frequency) and MK_amp (for amplitude).

- The parameter's name is used primarily to identify the parameter in a scorefile. The names of the Music Kit parameters are the same as the tag constants, but without the "MK_" prefix. You can also use a parameter's name to retrieve its tag by passing the name to Note's **parName:** class method. (As explained in its descriptions below, it's through this method that you create your own parameter tags.)

- A parameter's value can be a **double**, **int**, string (**char ***), or an object (**id**). The method you invoke to set a parameter value depends on the type of the value. To set a **double** value, for example, you would invoke the **setPar:toDouble:** method. Analogous methods exist for the other types. You can retrieve the value of a **double**-, **int**-, or string-valued parameter as any of these three types, regardless of the actual type of the value. For example, you can set the frequency of a Note as a **double**, thus:

  ```
  [aNote setPar:MK_freq toDouble:440.0]
  ```

  and then retrieve it as an **int**:

  ```
  int freq = [aNote parAsInt:MK_freq]
  ```

  The type conversion is done automatically.

- Object-valued parameters are treated differently from the other value types. The only Music Kit objects that are designed to be used as parameter values are Envelopes and WaveTables (and the WaveTable descendants Partials and Samples). Special methods are provided for setting and retrieving these objects. Other objects, most specifically objects of your own classes, are set through the **setPar:toObject:** method. While an instance of any class may be set as a parameter's value through this method, you should note well that only those objects that respond to the **writeASCIIStream:** and **readASCIIstream:** messages can be written to and read from a scorefile. None of the Music Kit classes implement these methods and so their instances can't be written to a scorefile as parameter values (Envelopes and WaveTables are written and read through a different mechanism).

- The parameter's data type is set when the parameter's value is set; thus the data type is either a **double**, **int**, string, Envelope, WaveTable, or (other) object. These are represented by constants, as given in the description of **parType:**, the method that retrieves a parameter's data type.

A parameter is said to be present within a Note once its value has been set. You can determine whether a parameter is present in one of four ways:

- The easiest way is to invoke the boolean method **isParPresent:**, passing the parameter tag as the argument. An equivalent C function, **MKIsNoteParPresent()** is also provided for greater efficiency.

- At a lower lever, you can invoke the **parVector:** method to retrieve one of a Note's "parameter bit vectors," integers that the Note uses internally to indicate which parameters are present. You query a parameter bit vector by masking it with the parameter's tag:

```
/* A Note may have more then one bit vector to accommodate all
 * its parameters.
 */
int parVector = [aNote parVector:(MK_amp/32)];

/* If MK_amp is present, the predicate will be true. */
if (parVector & (1 << (MK_amp % 32)))
```

- If you plan on retrieving the value of the parameter after you've checked for the parameter's presence, then it's generally more efficient to go ahead and retrieve the value and *then* determine if the parameter is actually set by comparing its value to the appropriate parameter-not-set value, as given below:

Retrieval type	**No-set value**
int	MAXINT
double	MK_NODVAL (but see below)
char *	""
id	**nil**

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 */
```

- If you're looking for and processing a large number of parameters in one block, then you should make calls to the **MKNextParameter()** C function, which returns the values of a Note's extant parameters only. See the function's description in Chapter 2 for more details.

A Note has two special timing attributes: A Note's time tag corresponds, conceptually, to the time during a performance that the Note is performed. Time tags are set through the **setTimeTag:** method. The other timing attribute is the Note's duration, a value that indicates how long the Note will endure once it has been struck. It's set through **setDur:**. A single Note can have only one time tag and one duration. Keep in mind, however, that not all Notes need a time tag and a duration. For example, if you realize a Note by sending it directly to an Instrument, then the Note's time tag—indeed, whether it even has a time tag—is of no consequence; the Note's performance time is determined by when the Instrument receives it (although see the ScorefileWriter, ScoreRecorder, and PartRecorder class descriptions for alternatives to this edict). Similarly, a Note that merely initiates an event, relying on a subsequent Note to halt the festivities (as described in the discussion of *note types*, below) doesn't need and actually mustn't be given a duration value.

During a performance, time tag and duration values are measured in time units called *beats*. The size of a beat is determined by the tempo of the Note's Conductor. However, you never set a Note's Conductor directly; instead, it's identified as the Conductor of the Performer (or Midi) that last performed the Note. Therefore, to determine its Conductor, a Note must know its most recent Performer. To this end, the Note is informed, whenever it's performed, of the Performer that's performing it; this informing is done automatically by the Performer itself. If a Note hasn't been performed by a Performer—if you've sent it directly to an Instrument, for example—then its Conductor is the *defaultConductor*, which has a default (but not immutable) tempo of 60.0 beats per minute. Keep in mind that if you send a Note directly to an Instrument, then the Note's time tag is (usually) ignored, as described above, but its duration may be considered and employed by the Instrument.

A Note has a *note type* that casts it into one of five roles:

- A noteDur represents an entire musical note (a note with a duration).
- A noteOn establishes the beginning of a note.
- A noteOff establishes the end of a note.
- A noteUpdate represents the middle of a note (it updates a sounding note).
- A mute makes no sound.

Only noteDurs may have duration values; the very act of setting a Note's duration changes it to a noteDur.

You match the two Notes in a noteOn/noteOff pair by giving them the same *note tag* value; a note tag is an integer that identifies two or more Notes as part of the same musical event or phrase. In addition to coining noteOn/noteOff pairs, note tags are used to associate a noteUpdate with a noteDur or noteOn that's in the process of being performed. The C function **MKNoteTag()** is provided to generate note tag values that are guaranteed to be unique across your entire application—you should never create a new note tag except through this function.

Instead of or in addition to being actively realized, a Note object can be stored. In a running application, Notes are stored within Part objects through the **addToPart:** method. A Note can be added to only one Part at a time; adding it to a Part automatically removes it from its previous Part. Within a Part object, Notes are sorted according to their time tag values.

For long-term storage, Notes can be written to a scorefile. There are two "safe" ways to write a scorefile: You can add a Note-filled Part to a Score and then write the Score to a scorefile, or you can send Notes during a performance to a ScorefileWriter Instrument. The former of these two methods is generally easier and more flexible since it's done statically and allows random access to the Notes within a Part. The latter allows Note objects to be reused since the file is written dynamically; it also lets you record interactive performances.

You can also write individual Notes in scorefile format to an open stream by sending **writeScorefileStream:** to the Notes. This can be convenient while debugging, but keep in mind that the method is designed primarily for use by Score and ScorefileWriter objects; if you write Notes directly to a stream that's open to a file, the file isn't guaranteed to be recognized by methods that read scorefiles, such as Score's **readScorefile:**.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Note	MKNoteType	noteType;
	int	noteTag;
	id	performer;
	id	part;
	double	timeTag;

noteType — The Note's note type.

noteTag — The Note's note tag.

performer — The Performer that most recently performed this Note.

part — The Part that this Note is a member of.

timeTag — The Note's time tag.

METHOD TYPES

Creating and freeing a Note	– copy – copyFromZone: – init – initWithTimeTag: – split:: – free
Storing the object	– addToPart: – part – removeFromPart – writeScorefileStream: – read: – write:
Querying the object	– compare: – conductor – performer
Modifying parameters	– setPar:toDouble: – setPar:toInt: – setPar:toString: – setPar:toEnvelope: – setPar:toWaveTable: – setPar:toObject: – copyParsFrom: – removePar:
Querying parameters	– parAsDouble: – parAsInt: – parAsString: – parAsStringNoCopy: – parAsEnvelope: – parAsWaveTable: – parAsObject: – freq – keyNum – isParPresent: – parType: + parName: – parVector: – parVectorCount
Time tag and duration	– setTimeTag: – setDur: – timeTag – dur

Note type and note tag	– setNoteType: – setNoteTag: – noteType – noteTag

CLASS METHODS

parName:

+ (int)**parName:**(char *)*aName*

Returns the integer that identifies the parameter named *aName*. If the named parameter doesn't have an identifier, one is created and thereafter associated with the parameter.

See also: – **setPar:toDouble:** (etc.), – **isParPresent:**

INSTANCE METHODS

addToPart:

– **addToPart:***aPart*

Removes the Note from the Part that it's currently a member of and adds it to *aPart*. Returns the Note's old Part, if any.

This method is equivalent to Part's **addNote:** method.

See also: –**part**, – **removeFromPart**

compare:

– (int)**compare:***aNote*

Returns a value that indicates which of the receiving Note and the argument Note would appear first if the two Notes were sorted into the same Part:

- –1 indicates that the receiving Note is first.
- 1 means that the argument, *aNote*, is first.
- 0 is returned if the receiving Note and *aNote* are the same object.

Keep in mind that the two Notes needn't actually be members of the same Part, nor must they be members of Parts at all. Naturally, the comparison is judged first on the relative values of the two Notes' time tags; changing one or both of the Notes' time tags invalidates the result of a previous invocation of this method.

conductor

– **conductor**

Returns the Conductor of the Performer that most recently performed the Note. If the Note hasn't been performed (by a Performer), then this returns the defaultConductor. A Note's Conductor is used primarily by Instrument objects that split noteDurs into noteOn/noteOff pairs; performance of the noteOff is scheduled with the Conductor that's returned by this method.

See also: – **performer**

copy

– **copy**

Creates and returns a new Note object as a copy of the receiving Note. The receiving Note's parameters, time tag, duration, note type, and note tag are copied into the new Note. Object-valued parameters are shared by the two Notes. The new Note isn't a member of a Part, regardless of the membership of the original Note. However, the new Note's Performer is that of the original Note, even though the new Note hasn't actually been performed. This imposture is necessary so that an Instrument can copy the Notes that it receives (prior to altering them, for example) without sacrificing access to the appropriate Conductor (more specifically, to the Conductor's tempo), which is retrieved through the Note's Performer.

See also: – **copyParsFrom:**, – **copyFromZone:**, – **split::**

copyParsFrom:

– **copyParsFrom:***aNote*

Copies *aNote*'s parameters into the receiving Note. Object-valued parameters are shared by the two Notes. Returns **self**.

See also: – **copy**, – **copyFromZone:**, – **split::**

copyFromZone:

– **copyFromZone:**(NXZone *)*aZone*

The same as **copy**, but the new Note is allocated in *aZone*.

See also: – **copy**, – **copyParsFrom:**, – **split::**

dur

– (double)**dur**

Returns the Note's duration, or MK_NODVAL if it isn't set (use the function **MKIsNoDVal()** to check for MK_NODVAL).

See also: – **setDur:**

free

– **free**

Removes the Note from its Part and then frees the Note (the Note's object-valued parameters aren't freed).

freq

– (double)**freq**

This method returns the Note's frequency, measured in Hertz (or cycles-per-second). If the frequency parameter MK_freq is present, its value is returned; otherwise, the frequency is converted from the key number value given by the MK_keyNum parameter. In the absence of both MK_freq and MK_keyNum, MK_NODVAL is returned (use the function **MKIsNoDVal()** to check for MK_NODVAL). The correspondence between key numbers and frequencies is given in Appendix A, "Music Tables."

Frequency and key number are the only two parameters whose values are retrieved through specialized methods. All other parameter values should be retrieved through one of the **parAs***Type***:** methods.

See also: – **keyNum**, – **setPar:toDouble:**

init

– **init**

Initializes a Note by setting its note type to MK_mute. Returns **self**.

See also: – **initWithTimeTag:**

initWithTimeTag:

– **init:**(double)*aTimeTag*

The same as **init**, but also sets the Note's time tag to *aTimeTag*.

See also: – **init**

isParPresent:

– (BOOL)**isParPresent:**(int)*parameterTag*

Returns **YES** if the parameter identified by *parameterTag* is present in the Note (in other words, if its value has been set), and **NO** if it isn't.

See also: – **parVector:**, **MKIsNoteParPresent()**, **MKNextParameter()**, + **parName:**, – **parType:**, – **setPar:toDouble:** (etc), – **parAsDouble:** (etc)

keyNum

– (int)**keyNum**

This method returns the key number of the Note. Key numbers are integers that enumerate discrete pitches; they're provided primarily to accommodate MIDI. If the MK_keyNum parameter is present, its value is returned; otherwise, the key number that corresponds to the value of the MK_freq parameter, if present, is returned. In the absence of both MK_keyNum and MK_freq, MAXINT is returned. The correspondence between key numbers and frequencies is given in Appendix A, "Music Tables."

Frequency and key number are the only two parameters whose values are retrieved through specialized methods. All other parameter values should be retrieved through one of the **parAs***Type***:** methods.

See also: – **freq**, – **setPar:toInt:**

noteTag

– (int)**noteTag**

Return the Note's note tag, or MAXINT if it isn't set.

See also: – **setNoteTag:**, **MKNoteTag()**

noteType

– (MKNoteType)**noteType**

Returns the Note's note type, one of MK_noteDur, MK_noteOn, MK_noteOff, MK_noteUpdate, or MK_mute. The note type describes the character of the Note, whether it represents an entire musical note (or event), the beginning, middle, or end of a note, or no note (no sound). A newly created Note is a mute. A Note's note type can be set through **setNoteType:**, although **setDur:** and **setNoteTag:** may also change it as a side effect.

See also: – **setNoteType:**, – **setDur:**, – **setNoteTag:**

parAsDouble:

– (double)**parAsDouble:**(int)*parameterTag*

Returns a **double** value converted from the value of the parameter identified by *parameterTag*. If the parameter isn't present or if its value is an object, returns MK_NODVAL (use the function **MKIsNoDVal()** to check for MK_NODVAL). You should use the **freq** method if you want to retrieve the frequency of the Note.

See also: **MKGetNoteParAsDouble()**, – **setPar:toDouble:** (etc), – **parType:**, – **isParPresent:**

parAsEnvelope:

– **parAsEnvelope:**(int)*parameterTag*

Returns the Envelope value of *parameterTag*. If the parameter isn't present, or if its value isn't an Envelope, returns **nil**.

See also: **MKGetNoteParAsEnvelope()**, – **setPar:toDouble:** (etc), – **parType:**, – **isParPresent:**

parAsInt:

– (int)**parAsInt:**(int)*parameterTag*

Returns an **int** value converted from the value of the parameter identified by *parameterTag*. If the parameter isn't present, or if its value is an object, returns MAXINT.

See also: **MKGetNoteParAsInt()**, **– setPar:toDouble:** (etc), **– parType:**, **– isParPresent:**

parAsObject:

– parAsObject:(int)*parameterTag*

Returns the object value of the parameter identified by *parameterTag*. If the parameter isn't present, or if its value isn't an object, returns **nil**. This method can be used to return Envelope and WaveTable objects in addition to non-Music Kit objects.

See also: **MKGetNoteParAsObject()**, **– setPar:toDouble:** (etc), **– parType:**, **– isParPresent:**

parAsString:

– (char *)**parAsString:**(int)*parameterTag*

Returns a string converted from a copy of the value of the parameter identified by *parameterTag*. If the parameter isn't present, or if its value is an object, returns an empty string.

See also: **MKGetNoteParAsString()**, **– setPar:toDouble:** (etc), **– parType:**, **– isParPresent:**

parAsStringNoCopy:

– (char *)**parAsStringNoCopy:**(int)*parameterTag*

Returns a string converted from the value of the parameter identified by *parameterTag*. If the parameter was set as a string, then this returns a pointer to the actual string itself; you should neither delete nor alter the value returned by this method. If the parameter isn't present, or if its value is an object, returns an empty string.

See also: **MKGetNoteParAsStringNoCopy()**, **– setPar:toDouble:** (etc), **– parType:**, **– isParPresent:**

parAsWaveTable:

– **parAsWaveTable:**(int)*parameterTag*

Returns the WaveTable value of the parameter identified by *parameterTag*. If the parameter isn't present, or if it's value isn't a WaveTable, returns **nil**.

parType:

– (MKDataType)**parType:**(int)*parameterTag*

Returns the data type of the value of the parameter identified by *parameterTag*. The data type is set when the parameter's value is set; the specific data type of the value, one of the MKDataType constants listed below, depends on which method you used to set it:

Method	Data type
setPar:toInt:	MK_int
setPar:toDouble	MK_double
setPar:toString:	MK_string
setPar:toWaveTable:	MK_waveTable
setPar:toEnvelope:	MK_envelope
setPar:toObject:	MK_object

If the parameter's value hasn't been set, MK_noType is returned.

See also: **MKGetNoteParAsWaveTable()**, – **setPar:toDouble:** (etc), – **parType:**, – **isParPresent:**

parVector:

– (unsigned)**parVector:**(unsigned)*index*

Returns an integer bit vector that indicates the presence of the *index*'th set of parameters. Each bit vector represents 32 parameters. For example, if *index* is 1, the bits in the returned value indicate the presence of parameters 0 through 31, where 1 means the parameter is present and 0 means that it's absent. An *index* of 2 returns a vector that represents parameters 32 through 63, and so on. To query for the presence of a particular parameter, use the following predicate formula:

```
[aNote parVector:(parameterTag/32)] & (1<<(parameterTag%32))
```

In this formula, *parameterTag* identifies the parameter that you're interested in. Keep in mind that the parameter bit vectors only indicate the presence of a parameter, not its value.

See also: – **parVectorCount**, – **isParPresent:**

parVectorCount

– (int)**parVectorCount**

Returns the number of parameter bit vectors that the Note is using to accommodate all its parameters identifiers. Normally you only need to know this if you're iterating over the parameter vectors.

See also: – **parVector**

part

– **part**

Returns the Part that contains the Note, or **nil** if none. By default, a Note isn't contained in a Part.

See also: – **addToPart:**, – **removeFromPart**

performer

– **performer**

Returns the Performer that most recently performed the Note. This is provided, primarily, as part of the implementation of the **conductor** method.

See also: – **conductor**

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived Note, call the **NXReadObject**() C function.

See also: – **write:**

removeFromPart

– **removeFromPart**

Removes the Note from its Part. Returns the Part, or **nil** if none.

See also: – **addToPart:**, – **part**

removePar:

– **removePar:**(int)*parameterTag*

Removes the parameter identified by *parameterTag* from the Note; in other words, this sets the parameter's value to indicate that the parameter isn't set. If the parameter was present, then the Note is returned; if not, **nil** is returned.

See also: + **parName:**, – **isParPresent:**, – **setPar:toDouble:** (etc)

setDur:

– (double)**setDur:**(double)*value*

Sets the Note's duration to *value* beats and sets its note type to MK_noteDur. If *value* is negative the duration isn't set, the note type isn't changed, and MK_NODVAL is returned (use the function **MKIsNoDVal()** to check for MK_NODVAL); otherwise returns *value*.

See also: – **dur**, – **conductor**

setNoteTag:

– **setNoteTag:**(int)*newTag*

Sets the Note's note tag to *newTag*; if the note type is MK_mute, it's changed to MK_noteUpdate. Returns **self**.

Note tags are used to associate different Notes with each other, thus creating an identifiable (by the note tag value) "Note stream." For example, you create a noteOn/noteOff pair by giving the two Notes identical note tag values. Also, you can associate any number of noteUpdates with a single noteDur, or with a noteOn/noteOff pair, through similarly matching note tags. While note tag values are arbitrary, they should be unique across an entire application; to ensure this, you should never create noteTag values but through the **MKNoteTag()** C function.

See also: – **noteTag**, **MKNoteTag()**

setNoteType:

– **setNoteType:**(MKNoteType)*newNoteType*

Sets the Note's note type to *newNoteType*, one of:

- MK_noteDur; represents an entire musical note.
- MK_noteOn; represents the beginning of a note.
- MK_noteOff; represents the end of a note.
- MK_noteUpdate; represents the middle of a note.
- MK_mute; makes no sound.

Returns **self**, or **nil** if *newNoteType* isn't a valid note type.

You should keep in mind that the **setDur:** method automatically sets a Note's note type to MK_noteDur; **setNoteTag:** changes mutes into noteUpdates.

See also: – **noteType**, – **setNoteTag:**, – **setDur:**

setPar:toDouble:

– **setPar:**(int)*parameterTag* **toDouble:**(double)*aDouble*

Sets the value of the parameter identified by *parameterTag* to *aDouble*, and sets its data type to MK_double. Returns **self**.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsDouble:**

setPar:toEnvelope:

– **setPar:**(int)*parameterTag* **toEnvelope:***anEnvelope*

Sets the value of the parameter identified by *parameterTag* to *anEnvelope*, and sets its data type to MK_envelope. Returns **self**.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsEnvelope:**

setPar:toInt:

– **setPar:**(int)*parameterTag* **toInt:**(int)*anInteger*

Sets the value of the parameter identified by *parameterTag* to *anInteger*, and sets its data type to MK_int. Returns **self**.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsInteger:**

setPar:toObject:

– **setPar:**(int)*parameterTag* **toObject:***anObject*

Sets the value of the parameter identified by *parameterTag* to *anObject*, and sets its data type to MK_object. Returns **self**.

While you can use this method to set the value of a parameter to any object, it's designed, principally, to allow you to use an instance of one of your own classes as a parameter value. If you want the object to be written to and read from a scorefile, it must respond to the messages **writeASCIIStream:** and **readASCIIStream:**. While response to these messages isn't a prerequisite for an object to be used as the argument to this method, if you try to write a Note that contains a parameter that doesn't respond to **writeASCIIStream:**, an error is generated.

If you're setting the value as an Envelope or WaveTable object, you should use the **setPar:toEnvelope:** or **setPar:toWaveTable:** method, respectively.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsObject:**

setPar:toString:

– **setPar:**(int)*parameterTag* **toString:**(char *)*aString*

Sets the value of the parameter identified by *parameterTag* to *aString*, and sets its data type to MK_string. Returns **self**.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsString:**

setPar:toWaveTable:

– **setPar:**(int)*parameterTag* **toWaveTable:***aWaveTable*

Sets the value of the parameter identified by *parameterTag* to *aWaveTable*, and sets its data type to MK_waveTable. Returns **self**.

See also: + **parName:**, – **parType:**, – **isParPresent:**, – **parAsWaveTable:**

setTimeTag:

– (double)**setTimeTag:**(double)*newTimeTag*

Sets the Note's time tag to *newTimeTag* or 0.0, whichever is greater (a time tag can't be negative). The old time tag value is returned; a return value of MK_ENDOFTIME indicates that the time tag hadn't been set. Time tags are used to sort the Notes within a Part; if you change the time tag of a Note that's been added to a Part, the Note is automatically resorted.

See also: – **timeTag**, – **addToPart:**, –**sort** (Part)

split::

– **split:**(id *)*aNoteOn* **:**(id *)*aNoteOff*

This method splits a noteDur into a noteOn/noteOff pair, as described below. The new Notes are returned by reference in the arguments. The noteDur itself is left unchanged. If the receiving Note isn't a noteDur, this does nothing and returns **nil**, otherwise it returns **self**.

The receiving Note's MK_relVelocity parameter, if present, is copied into the noteOff. All other parameters are copied into (or, in the case of object-valued parameters, referenced by) the noteOn. The noteOn takes the receiving Note's time tag value; the noteOff's time tag is that of the Note plus its duration. If the receiving Note has a note tag, it's copied into the noteOn and noteOff; otherwise a new note tag is generated for them. The new Notes are added to the receiving Note's Part, if any.

Keep in mind that while this method replicates the noteDur within the noteOn/noteOff pair, it doesn't replace the former with the latter. To do this, you must free the noteDur yourself.

timeTag

– (double)**timeTag**

Returns the Note's time tag. If the time tag isn't set, MK_ENDOFTIME is returned. Time tag values are used to sort the Notes within a Part.

See also: – **setTimeTag:**

writeScorefileStream:

– **writeScorefileStream:**(NXStream *)*aStream*

Writes the Note, in scorefile format, to the stream *aStream*. The stream must be open for writing. You rarely invoke this method yourself; it's invoked from the scorefile writing methods defined by Score and ScorefileWriter. Returns **self**.

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive a Note, call the **NXWriteRootObject**() C function. The Note's parameters, note type, note tag, and time tag are archived directly. Its Performer and Part are archived through **NXWriteObjectReference**().

See also: – **read:**

NoteFilter

INHERITS FROM	Instrument : Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

NoteFilter is an abstract class that combines the Note-receiving protocol it inherits from Instrument with the Note-sending protocol defined by the Performer class. You interpose a series of NoteFilter objects between a Performer and an Instrument to create a Note processing pipeline.

Having created a set of NoteSenders and NoteReceivers, a NoteFilter object receives Notes through its NoteReceivers, modifies them, and then sends them to its NoteSenders. Each subclass provides a unique system for modifying Notes in its implementation of **realizeNote:fromNoteReceiver:**, a subclass responsibility inherited from Instrument and passed on to the NoteFilter subclasses. When designing a NoteFilter subclass, you should keep in mind that the responsibility of sending Notes to the NoteSenders falls to the subclass itself. A NoteFilter subclass implementation of **realizeNote:fromNoteReceiver:** should include an invocation of NoteSender's **sendNote:** method (or one of its sister methods; see the NoteSender class description).

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Instrument	id	noteReceivers;
Declared in NoteFilter	id	noteSenders;
noteSenders	Collection of NoteSenders.	

METHOD TYPES

Creating and freeing a NoteFilter	– copy – copyFromZone: – free – init
Modifying the object	– addNoteSender: – freeNoteSenders – removeNoteSender: – removeNoteSenders

Querying the object	– isNoteSenderPresent: – noteSender – noteSenders
Archiving the object	– read: – write:

INSTANCE METHODS

addNoteSender:

– **addNoteSender:***aNoteSender*

Removes *aNoteSender* from its present owner (if any) and adds it to the receiving NoteFilter. If the NoteFilter is in performance, or if *aNoteSender* is already owned by the NoteFilter, this does nothing and returns **nil**; otherwise returns *aNoteSender*.

copy

– **copy**

Creates and returns a NoteFilter as a copy of the receiving NoteFilter. The new object contains copies of the receiving NoteFilter's NoteSenders and NoteReceivers.

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Same as **copy**, but uses the specified *zone*.

free

– **free**

Frees the NoteFilter and its NoteSenders and NoteReceivers.

freeNoteSenders

– **freeNoteSenders**

Removes and frees the NoteFilter's NoteSenders. Returns **self**.

init

– **init**

Initializes a new NoteFilter. A subclass implementation should first send **[super init]**.

isNoteSenderPresent:

– (BOOL)**isNoteSenderPresent:***aNoteSender*

Returns YES if *aNoteSender* is one of the NoteFilter's NoteSenders. Otherwise returns NO.

noteSender

– **noteSender**

Returns the NoteFilter's first NoteSender. This method should only by invoked if the NoteFilter contains only one NoteSender or if you don't care which NoteSender you get.

noteSenders

– **noteSenders**

Creates and returns a List of the NoteFilter's NoteSenders. It's the sender's responsibility to free the List.

removeNoteSender:

– **removeNoteSender:***aNoteSender*

Removes *aNoteSender* from the NoteFilter. If the NoteFilter is in a performance, this does nothing and returns **nil**; otherwise it returns the argument.

removeNoteSenders

– **removeNoteSenders**

Removes all the NoteFilter's NoteSenders and returns **self**.

NoteReceiver

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

NoteReceiver is an auxiliary class that completes the implementation of Instrument. Instances of NoteReceiver are owned by Instrument objects to provide the following:

- It's part of the link between a Performer and an Instrument. NoteReceiver's **connect:** method connects a NoteReceiver to a NoteSender, which is owned by a Performer in much the same way that a NoteReceiver is owned by an Instrument. When a NoteReceiver is connected to a NoteSender, their respective owners are said to be connected. NoteSender defines an equivalent **connect:** method—it doesn't matter which of the two objects is the receiver and which is the argument when sending a **connect:** message.

- NoteReceiver's **receiveNote:** method defines the mechanism by which an Instrument obtains Notes. When a NoteReceiver receives the **receiveNote:** message, it forwards the argument (a Note object) to its owner by invoking the Instrument method **realizeNote:fromNoteReceiver:**. The **receiveNote:** method itself is sent when a connected NoteSender receives a **sendNote:** message from its owner; you can also send **receiveNote:** (or one of its five sister methods) directly to a NoteReceiver from your application. You can toggle a NoteReceiver's ability to pass Notes to its owner through the **squelch** and **unsquelch** methods; a NoteReceive won't send **realizeNote:fromNoteReceiver:** messages while it's squelched.

Unlike NoteSenders, which are generally expected to be created by the Performers that own them, NoteReceivers can be created either by their owners or by your application. For example, each SynthInstrument object creates and adds to itself a single NoteReceiver. ScorefileWriter objects, on the other hand, don't create any NoteReceivers; it's left to your application to create and add them. A NoteReceiver is created through the **new** class method and added to an Instrument through the latter's **addNoteReceiver:**.

A NoteReceiver can be owned by only one Instrument at a time; however, it can be connected to any number of NoteSenders. In addition, two different NoteReceivers can be connected to the same NoteSender. Thus the connections between Performers and Instruments can describe an arbitrarily complicated network. To retrieve the NoteReceivers that are owned by a particular Instrument, you invoke the Instrument's **noteReceiver** or **noteReceivers** method.

NoteReceivers are also created, owned, and used by Midi objects as part of their assumption of the role of Instrument.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NoteReceiver	id	noteSenders;
	BOOL	isSquelched;
	id	owner;

noteSenders — List of connected NoteSenders.

isSquelched — YES if the NoteReceiver is squelched. No by default.

owner — Instrument (or Midi) that owns the NoteReceiver.

METHOD TYPES

Creating a NoteReceiver	– copy – copyFromZone: – init – free
The object's owner	– owner
Connecting the object	– connect: – disconnect: – disconnect – connections – connectionCount – isConnected:
Squelching the object	– squelch – unsquelch – isSquelched
Receiving Notes	– receiveNote: – receiveNote:atTime: – receiveNote:withDelay: – receiveAndFreeNote: – receiveAndFreeNote:atTime: – receiveAndFreeNote:withDelay:
Archiving the object	– write: – read:

INSTANCE METHODS

connect:

– **connect:***aNoteSender*

Connects *aNoteSender* to the NoteReceiver; if the argument isn't a NoteSender, the connection isn't made. Returns **self**.

See also: – **disconnect:**, – **isConnected**, – **connections**

connectionCount

– (unsigned int)**connectionCount**

Returns the number of NoteSenders that are connected to the NoteReceiver.

See also: – **connect:**, – **disconnect:**, – **isConnected**, – **connections**

connections

– **connections**

Creates and returns a List of the NoteSenders that are connected to the NoteReceiver. It's the sender's responsibility to free the List.

See also: – **connect:**, – **disconnect:**, – **isConnected**

copy

– **copy**

Creates and returns an unowned NoteReceiver that's connected to the same NoteSenders as the receiver of this message. If the receiving NoteReceiver is squelched, so, too, shall be the copy.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

This is the same as **copy**, but the new object is allocated from *zone*.

See also: – **copy**

disconnect

– **disconnect**

Severs the connections between the NoteReceiver and all the NoteSenders it's connected to. Returns **self**.

See also: – **disconnect:**, – **connect:**, – **isConnected:**, – **connections**

disconnect:

– **disconnect:***aNoteSender*

Severs the connection between the NoteReceiver and *aNoteSender*; if the NoteSender isn't connected, this does nothing. Returns **self**.

See also: – **disconnect**, – **connect:**, – **isConnected:**, – **connections**

free

– **free**

Severs the connections between the NoteReceiver and all its connected NoteSenders and then frees the NoteReceiver.

See also: – **disconnect**

init

– **init**

Initializes a NoteReceiver that was created through **allocFromZone:**. Returns **self**.

isConnected:

– (BOOL)**isConnected:***aNoteSender*

Returns YES if *aNoteSender* is connected to the NoteReceiver, otherwise returns NO.

See also: – **connect**, – **disconnect**, – **connections**, – **connectionCount**

isSquelched

– (BOOL)**isSquelched**

Returns YES if the NoteReceiver is squelched, otherwise returns NO. A squelched NoteReceiver won't invoke its owner's **realizeNote:fromNoteReceiver:** method.

See also: – **squelch**, – **unsquelch**

owner

– **owner**

Returns the Instrument (or Midi object) that owns the NoteReceiver.

See also: – **addNoteReceiver:** (Instrument, Midi)

read:

– **read:**(NXTypedStream *)*stream*

Unarchives the NoteReceiver by reading it from *stream*. You never invoke this method directly; to read an archived NoteReceiver, call the **NXReadObject**() C function.

See also: – **write:**

receiveAndFreeNote:

– **receiveAndFreeNote:***aNote*

Sends the message **receiveNote:***aNote* to **self** and then frees *aNote*. Returns **self**.

See also: – **receiveNote:**, – **receiveAndFreeNote:atTime:**,
– **receiveAndFreeNote:withDelay:**

receiveAndFreeNote:atTime:

– **receiveAndFreeNote:***aNote* **atTime:**(double)*time*

Enqueues, with the appropriate Conductor, a request for **receiveAndFreeNote:***aNote* to be sent to **self** at time *beatsSinceStart*, measured in beats from the beginning of the Conductor's performance. See **receiveNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **receiveNote:**, – **receiveAndFreeNote:**,
– **receiveAndFreeNote:withDelay:**

receiveAndFreeNote:withDelay:

– **receiveAndFreeNote:***aNote* **withDelay:**(double)*delayTime*

Enqueues, with the appropriate Conductor, a request for **receiveAndFreeNote:***aNote* to be sent to **self** after *delayBeats*. See **receiveNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **receiveNote:**, – **receiveAndFreeNote:**, – **receiveAndFreeNote:atTime:**

receiveNote:

– **receiveNote:***aNote*

Sends the message **realizeNote:***aNote* **fromNoteReceiver:self** to the NoteReceiver's owner. If the NoteReceiver is squelched, the message isn't sent. This method is invoked automatically as the NoteReceiver's connected NoteSenders receive **sendNote:** messages; you can also invoke this method directly. Returns **self**.

See also: – **receiveAndFreeNote:**, – **receiveNote:withDelay:**, – **receiveNote:atTime:**

receiveNote:atTime:

– **receiveNote:***aNote* **atTime:**(double)*time*

Enqueues, with the Conductor object described below, a request for **receiveNote:***aNote* to be sent to **self** at time *beatsSinceStart*, measured in beats from the beginning of the Conductor's performance. If *beatsSinceStart* has already passed, the enqueued message is sent immediately. Returns **self**.

The request is enqueued with the object that's returned by [*aNote* **conductor**]. If the Note was received from a NoteSender, this is the Conductor of the Performer that originally sent *aNote* into the performance. If you invoke this method (or any of the **receiveNote:** methods that enqueue a message request) directly, or if Midi is the originator of the Note, then the default Conductor is used.

See also: – **receiveNote:**, – **receiveNote:withDelay:**

receiveNote:withDelay:

– **receiveNote:***aNote* **withDelay:**(double)*delayTime*

Enqueues, with the appropriate Conductor, a request for **receiveNote:***aNote* to be sent to **self** after *delayBeats*. See **receiveNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **receiveNote:**, – **receiveNote:atTime:**

squelch

– **squelch**

Disables the NoteReceiver's ability to send **realizeNote:fromNoteReceiver:** messages to its owner. Returns **self**.

See also: – **isSquelched**, – **unsquelch**

unsquelch

– **unsquelch**

Enables the NoteReceiver's ability to send **realizeNote:fromNoteReceiver:** messages to its owner, undoing the effect of a previous **squelch** message. Returns **self**.

See also: – **isSquelched**, – **squelch**

write:

– **write:**(NXTypedStream *)*stream*

Archives the NoteReceiver by writing it to *stream*. The NoteReceiver's connections and owner are archived by reference. You never invoke this method directly; to archive a NoteSender, call the **NXWriteRootObject**() C function.

See also: – **read:**

NoteSender

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

NoteSender is an auxiliary class that completes the implementation of Performer. Instances are created and owned by Performer objects, normally when the Performer itself is initialized. A NoteSender object performs two functions:

- It's part of the link between a Performer and an Instrument. NoteSender's **connect:** method connects a NoteSender to a NoteReceiver, which is owned by an Instrument in much the same way that a NoteSender is owned by a Performer. When a NoteSender is connected to a NoteReceiver, their respective owners are said to be connected. NoteReceiver defines an equivalent **connect:** method—it doesn't matter which of the two objects is the receiver and which is the argument when sending a **connect:** message.

- NoteSender's **sendNote:** method defines the mechanism by which a Performer relays a Note to a set of Instruments. When a NoteSender receives a **sendNote:** message, it sends **receiveNote:** to its connected NoteReceivers which, in turn, send **realizeNote:fromNoteReceiver:** to their owners (Instrument objects). You can toggle a NoteSender's ability to send Notes through the **squelch** and **unsquelch** methods; a NoteSender won't send **receiveNote:** messages while it's squelched.

There's a fundamental difference between these two tasks in that while you connect NoteSenders to NoteReceivers from your application, sending Notes is a Performer's responsibility: Subclasses of Performer should invoke **sendNote:** as part of their implementations of the **perform** method.

A NoteSender can be owned by only one Performer at a time; however, it can be connected to any number of NoteReceivers. In addition, two different NoteSenders can be connected to the same NoteReceiver. Thus the connections between Performers and Instruments can describe an arbitrarily complicated network. To retrieve the NoteSenders that are owned by a particular Performer—to connect them to NoteReceivers, or to squelch and unsquelch them—you invoke the Performer's **noteSender** or **noteSenders** method.

NoteSenders are also owned and used by NoteFilter and Midi objects. Neither of these classes inherits from Performer, but they both require the Note-sending mechanism that NoteSenders provide.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in NoteSender	id	*noteReceivers;
	int	connectionCount;
	BOOL	isSquelched;
	id	owner;

noteReceivers — List of connected NoteReceivers.

connectionCount — Number of connections.

isSquelched — YES if the NoteSender is squelched. NO by default.

owner — Performer (or NoteFilter or Midi) that owns the NoteSender.

METHOD TYPES

Creating a NoteSender	– copy – copyFromZone: – init – free
The object's owner	– owner
Connecting the object	– connect: – disconnect: – disconnect – connections – connectionCount – isConnected:
Squelching the object	– squelch – unsquelch – isSquelched
Sending Notes	– sendNote: – sendNote:atTime: – sendNote:withDelay: – sendAndFreeNote: – sendAndFreeNote:atTime: – sendAndFreeNote:withDelay:
Archiving the object	– write: – read:

INSTANCE METHODS

connect:

– **connect:***aNoteReceiver*

Connects *aNoteReceiver* to the NoteSender; if the argument isn't a NoteReceiver, the connection isn't made. Returns **self**.

See also: – **disconnect:**, – **isConnected**, – **connections**

connectionCount

– (unsigned int)**connectionCount**

Returns the number of NoteReceivers that are connected to the NoteSender.

See also: – **connect:**, – **disconnect:**, – **isConnected**, – **connections**

connections

– **connections**

Creates and returns a List of the NoteReceivers that are connected to the NoteSender. It's the sender's responsibility to free the List.

See also: – **connect:**, – **disconnect:**, – **isConnected**

copy

– **copy**

Creates and returns an unowned NoteSender that's connected to the same NoteReceivers as the receiver of this message. If the receiving NoteSender is squelched, so, too, shall be the copy.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

This is the same as **copy**, but the new object is allocated from *zone*.

See also: – **copy**

disconnect

– disconnect

Severs the connections between the NoteSender and all the NoteReceivers it's connected to. Returns **self**.

See also: **– disconnect:**, **– connect:**, **– isConnected:**, **– connections**

disconnect:

– disconnect:*aNoteReceiver*

Severs the connection between the NoteSender and *aNoteReceiver*; if the NoteReceiver isn't connected, does nothing. Returns **self**.

See also: **– disconnect**, **– connect:**, **– isConnected:**, **– connections**

free

– free

Severs the connections between the NoteSender and all its connected NoteReceivers, and then frees the NoteSender. You can't free a NoteSender that's in the process of sending a Note—specifically, an Instrument shouldn't invoke this method as part of its **realizeNote:fromNoteReceiver:** method.

See also: **– disconnect**

init

– init

Initializes the NoteSender and returns **self**.

isConnected:

– (BOOL)**isConnected:***aNoteReceiver*

Returns YES if *aNoteReceiver* is connected to the NoteSender, otherwise returns NO.

See also: **– connect**, **– disconnect**, **– connections**, **– connectionCount**

isSquelched

– (BOOL)**isSquelched**

Returns YES if the NoteSender is squelched (its Note-sending ability is disabled), otherwise returns NO.

See also: **– squelch**, **– unsquelch**

owner

– **owner**

Returns the Performer (or NoteFilter or Midi object) that owns the NoteSender.

See also: – **addNoteSender:** (Performer, NoteFilter, Midi)

read:

– **read:**(NXTypedStream *)*stream*

Unarchives the NoteSender by reading it from *stream*. You never invoke this method directly; to read an archived NoteSender, call the **NXReadObject()** C function.

See also: – **write:**

sendAndFreeNote:

– **sendAndFreeNote:***aNote*

Sends the message **sendNote:***aNote* to **self** and then frees *aNote*. Returns **self**.

See also: – **sendNote:**, – **sendAndFreeNote:atTime:**,
– **sendAndFreeNote:withDelay:**,

sendAndFreeNote:atTime:

– **sendAndFreeNote:***aNote* **atTime:**(double)*beatsSinceStart*

Enqueues, with the appropriate Conductor, a request for **sendAndFreeNote:***aNote* to be sent to **self** at time *beatsSinceStart*, measured in beats from the beginning of the Conductor's performance. See **sendNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **sendNote:**, – **sendAndFreeNote:**, – **sendAndFreeNote:withDelay:**

sendAndFreeNote:withDelay:

– **sendAndFreeNote:***aNote* **withDelay:**(double)*delayBeats*

Enqueues, with the appropriate Conductor, a request for **sendAndFreeNote:***aNote* to be sent to **self** after *delayBeats*. See **sendNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **sendNote:**, – **sendAndFreeNote:**, – **sendAndFreeNote:atTime:**

sendNote:

– **sendNote:***aNote*

Sends the message **receiveNote:***aNote* to the NoteReceivers that are connected to the NoteSender. If the NoteSender is squelched, the messages aren't sent. Normally, this method is only invoked by the NoteSender's owner. Returns **self**.

See also: – **sendAndFreeNote:**, – **sendNote:withDelay:**, – **sendNote:atTime:**

sendNote:atTime:

– **sendNote:***aNote* **atTime:**(double)*beatsSinceStart*

Enqueues, with the Conductor object described below, a request for **sendNote:***aNote* to be sent to **self** at time *beatsSinceStart*, measured in beats from the beginning of the Conductor's performance. If *beatsSinceStart* has already passed, the enqueued message is sent immediately. Returns **self**.

The request is enqueued with the object returned by [*aNote* **conductor**]. Normally—if the owner is a Performer—this is the owner's Conductor. However, if the owner is a NoteFilter, the request is enqueued with the Conductor of the Performer (or Midi) that originally sent *aNote* into the performance (or the defaultConductor if the NoteFilter itself created the Note).

See also: – **sendNote:**, – **sendNote:withDelay:**

sendNote:withDelay:

– **sendNote:***aNote* **withDelay:**(double)*delayBeats*

Enqueues, with the appropriate Conductor, a request for **sendNote:***aNote* to be sent to **self** after *delayBeats*. See **sendNote:atTime:** for a description of the Conductor that's used. Returns **self**.

See also: – **sendNote:**, – **sendNote:atTime:**

squelch

– **squelch**

Disables the NoteSender's ability to send **receiveNote:** to its NoteReceivers. Returns **self**.

See also: – **isSquelched**, – **unsquelch**

unsquelch

– unsquelch

Enables the NoteSender ability to send Notes, undoing the effect of a previous squelch message. Returns **self**.

See also: **– isSquelched**, **– squelch**

write:

– write:(NXTypedStream *)*stream*

Archives the NoteSender by writing it to *stream*. The NoteSender's connections and owner are archived by reference. You never invoke this method directly; to archive a NoteSender, call the **NXWriteRootObject**() C function.

See also: **– read:**

Orchestra

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

The Orchestra class manages DSP resources used in music synthesis. Each instance of Orchestra represents a single DSP that's identified by **orchIndex**, a zero-based integer index. In the basic NeXT configuration, there's only one DSP so there's only one Orchestra instance.

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 the methods defined here. 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 those defined by Orchestra. You need to allocate synthesis objects directly only if you want to assemble sound-making modules at a low level.

Before you can do anything with an Orchestra—particularly, before you can allocate synthesis objects—you must create and open it. You create an Orchestra through the **new** or **newOnDSP:** method (you *don't* use **alloc** and **init**). To open an Orchestra, you send it the **open** message. Opening an Orchestra gains access to the DSP that it represents, allowing you to allocate synthesis objects through methods such as **allocSynthPatch:** and **allocUnitGenerator:**. To start the synthesis running, you send **run** to the Orchestra. The **stop** method halts synthesis and **close** surrenders control of the DSP. The state of an Orchestra object with respect to these methods is described as its device status:

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

Note that these are the same methods and MKDeviceStatus values used to control and describe the status of a Midi object.

As you start, pause, resume, and stop a performance, you should similarly control your Orchestra objects, as described by the following table:

When you send this to the Conductor class	Send this to your Orchestra objects
startPerformance	run
pausePerformance	stop
resumePerformance	run
finishPerformance	close

When the Orchestra is running, the allocated UnitGenerators produce 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 cause 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). You can also set the Orchestra to write a soundfile that contains DSP commands by invoking the **setOutputCommandsFile:** method. A DSP commands soundfile is usually much smaller than the analogous sample data soundfile

Every command that's sent to the DSP is given a timestamp indicating when the command should be executed. The manner in which the DSP regards these timestamps depends on whether its Orchestra is timed or untimed, as set through the **setTimed:** method. In a timed Orchestra, commands are executed at the time indicated by its timestamp. If the Orchestra is untimed, the DSP ignores the timestamps, executing commands as soon as it receives them. By default, an Orchestra is timed.

The DSP is a separate real-time processor with its own clock and its own notion of the current time. Since the DSP can be dedicated to a single task—in this case, generating sound—its clock is generally more reliable than the main processor, which may be controlling any number of other tasks. If your application is generating Notes without user-interaction, then you should set the Music Kit performance to be unclocked, through the Conductor's **setClocked:** method, and the Orchestra to be timed. This allows the Music Kit to process Notes and send timestamped commands to the DSP as quickly as possible, relying on the DSP's clock to synthesize the Notes at the correct time. However, if your application must respond to user-initiated actions with as little latency as possible, then the Conductor must be clocked. In this case, you can set the Orchestra to be untimed. A clocked Conductor and an untimed Orchestra yields the best possible response time, but at the expense of possible rhythmic irregularity.

If your application responds to user actions but can sustain some latency between an action and its effect, then you may want to set the Conductor to be clocked and the DSP to be timed and use the C function **MKSetDeltaT()** to set your application's *delta time*. Delta time is an imposed latency that allows the Music Kit to run slightly ahead of the DSP. Any rhythmic irregularities created by the Music Kit's dependence on the CPU's clock are evened out by the utter dependability of the DSP's clock.

With regard to DSP resources, the Orchestra makes an educated estimate as to how much of the DSP is needed to synthesize a Note—for various reasons, it can't know for sure exactly how much it needs—and will deny allocation requests that exceed this estimate. 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:** 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. If you're writing samples to a soundfile with the DAC disabled, headroom is ignored.

While the speed of the DSP makes real-time synthesis approachable, there's always a sound output time delay that's equal to the size of the buffer used to collect 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 synthesis computation and, again, fewer simultaneous voices may result. You can also improve response time by using the high sampling rate (44100 samples per second) although this, too, attenuates the synthesis power of the DSP. By default, the Orchestra's sampling rate is 22050 samples per second.

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 and are 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 consider only 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.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Orchestra	double	computeTime;
	double	samplingRate;
	id	stack;
	char	*outputSoundfile;
	char	*outputCommandsFile;
	id	xZero;
	id	yZero;
	id	xSink;
	id	ySink;
	id	sineROM;
	id	muLawROM;
	MKDeviceStatus	deviceStatus;
	unsigned short	orchIndex;
	BOOL	isTimed;
	BOOL	useDSP;
	BOOL	soundOut;
	BOOL	SSISoundOut;
	BOOL	isLoopOffChip;
	BOOL	fastResponse;
	double	localDeltaT;
	short	onChipPatchPoints;

computeTime	Time in seconds to compute one sample.
samplingRate	Sampling rate.
stack	List of UnitGenerators in order as they appear in DSP memory.
outputSoundfile	Soundfile name to which output samples are written.
outputCommandsFile	Soundfile name to which DSP commands are written.
xZero	Special x memory patchpoint that always holds 0.
yZero	Special y memory patchpoint that always holds 0.
xSink	Special x memory patchpoint that's never read.
ySink	Special y memory patchpoint that's never read.
sineROM	Special read-only SynthData that represents the sine ROM.

muLawROM	Special read-only SynthData that represents the mu-law ROM.
deviceStatus	The object's status.
orchIndex	Index to the DSP that's managed by this instance.
isTimed	YES if DSP commands are timed.
useDSP	YES if running on a DSP.
soundOut	YES if sound is being sent to the DAC.
SSISoundOut	YES if sound is being sent to the DSP port.
isLoopOffChip	YES if the orchestra loop is running partially off-chip.
fastResponse	YES if response latency should be minimized.
localDeltaT	Offset in seconds added to output timestamps.
onChipPatchPoints	Number of on-chip patchpoints.

METHOD TYPES

Creating and freeing an Orchestra	– free + free + new + newOnDSP:
Modifying the object	+ flushTimedMessages – setOnChipMemoryConfigDebug:patchPoints: – setOffChipMemoryConfigXArg:yArg: – setSamplingRate: + setSamplingRate: – sharedObjectFor: – trace:msg:

Querying the object	+ DSPCount – computeTime – deviceStatus – fastResponse – headroom – index – isTimed – localDeltaT + nthOrchestra: – outputSoundfile – outputCommandsFile – simulatorFile – samplingRate – peekMemoryResources: – segmentName: – soundOut
Adjusting DSP computation and timing	– beginAtomicSection – endAtomicSection + setFastResponse: – setFastResponse: + setHeadroom: – setHeadroom: + setLocalDeltaT: – setLocalDeltaT: + setTimed: – setTimed:
Setting the output destination	– setOutputSoundfile: – setOutputCommandsFile: – setSimulatorFile: – setSoundOut:
Opening and running the DSP	– abort + abort – close + close – flushTimedMessages – open + open – run + run + stop – stop

Allocating synthesis objects	– allocPatchpoint: + allocPatchpoint: – allocSynthData:length: + allocSynthData:length: – allocSynthPatch: + allocSynthPatch: – allocSynthPatch:patchTemplate: + allocSynthPatch:patchTemplate: – allocUnitGenerator: + allocUnitGenerator: – allocUnitGenerator:after: – allocUnitGenerator:before: – allocUnitGenerator:between:: – dealloc: + dealloc: – muLawROM – segmentSink: – segmentZero: – sineROM
Accessing the shared data table	– installSharedObject:for: – installSharedSynthDataWithSegment:for: – installSharedSynthDataWithSegment AndLength:for: – sharedObjectFor:segment: – sharedObjectFor:segment:length:

CLASS METHODS

DSPCount

+ (unsigned short)**DSPCount**

Returns the number of DSPs on your computer.

abort

+ **abort**

Sends **abort** to each of the Orchestra instances and sets each to MK_devClosed. If any of the Orchestras responds to the **abort** message by returning **nil**, so, too, does this method return **nil**. Otherwise returns the receiver.

allocPatchpoint:

+ **allocPatchpoint:**(MKOrchMemSegment)*segment*

Allocates a patchpoint in segment *segment*. Returns the patchpoint (a SynthData object), or **nil** if the object couldn't be allocated.

allocSynthData:length:

+ **allocSynthData:**(MKOrchMemSegment)*segment* **length:**(unsigned)*size*

Allocates a SynthData object. The allocation is on the first Orchestra that can accommodate *size* words in segment *segment*. Returns the SynthData, or **nil** if the object couldn't be allocated.

allocSynthPatch:

+ **allocSynthPatch:***aSynthPatchClass*

This is the same as **allocSynthPatch:patchTemplate:** but uses the default template.

allocSynthPatch:patchTemplate:

+ **allocSynthPatch:***aSynthPatchClass* **patchTemplate:***p*

Allocates a SynthPatch with a PatchTemplate of *p* on the first Orchestra with sufficient resources. Returns the SynthPatch or **nil** if it couldn't be allocated.

allocUnitGenerator:

+ **allocUnitGenerator:***classObj*

Allocates a UnitGenerator of class *classObj*. The object is allocated on the first Orchestra that can accomodate it. Returns the UnitGenerator, or **nil** if the object couldn't be allocated.

close

+ **close**

Sends **close** to each of the Orchestra instances and sets each to MK_devClosed. If any of the Orchestras responds to the **close** message by returning **nil**, so, too, does this method return **nil**. Otherwise returns **self**.

dealloc:

+ **dealloc:***aSynthResource*

Deallocates the argument, which must be a previously allocated SynthPatch, UnitGenerator, or SynthData, by sending it the **dealloc** message.

flushTimedMessages

+ **flushTimedMessages**

Flushes all currently buffered DSP commands by invoking the **flushTimedMessages** instance method for each Orchestra.

free

+ **free**

Frees all the existing Orchestra instances.

new

+ **new**

If an Orchestra object exists for the default DSP, returns that object. Otherwise, creates and initializes a new Orchestra for the default DSP.

newOnDSP:

+ **newOnDSP:**(unsigned short)*index*

Creates and returns an Orchestra instance for the *index*'th DSP. If an Orchestra object already exists for the specified DSP, the existing object is returned. Returns **nil** if *index* is out of bounds or if the *index*'th DSP isn't available.

nthOrchestra:

+ **nthOrchestra:**(unsigned short)*index*

Returns the Orchestra of the *index*'th DSP. If *index* is out of bounds, or if an Orchestra hasn't been created for the specified DSP, **nil** is returned.

open

+ **open**

Sends **open** to each of the Orchestra instances and sets each to MK_devOpen. If any of the Orchestras responds to the **open** message by returning **nil**, so, too, does this method return **nil**. Otherwise returns **self**.

run

+ **run**

Sends **run** to each of the Orchestra instances and sets each to MK_devRunning. If any of the Orchestras responds to the **run** message by returning **nil**, so, too, does this method return **nil**. Otherwise returns **self**.

setFastResponse:

+ **setFastResponse:**(BOOL)*yesOrNo*

Sends **setFastResponse:***yesOrNo* to all existing Orchestra objects and returns **self**. This also sets the default used by subsequently created Orchestras.

setHeadroom:

+ **setHeadroom:**(double)*headroom*

Sets the headroom of all Orchestra instances to *headroom*. Returns **self**.

setLocalDeltaT:

+ **setLocalDeltaT:**(double)*val*

Sets the local delta time for all Orchestras and changes the default, which is otherwise 0.0.

setSamplingRate:

+ **setSamplingRate:**(double)*newSRate*

Sets the sampling rate of all Orchestra instances by sending **setSamplingRate:***newSRate* to all closed Orchestras. This method also changes the default sampling rate; when a new Orchestra is subsequently created, it also gets set to *newSRate*. Returns **self**.

setTimed:

+ **setTimed:**(BOOL)*areOrchsTimed*

Sends **setTimed:***areOrchsTimed* to each Orchestra instance. If *areOrchsTimed* is YES, the DSP processes the commands that it receives at the times specified by the commands' timestamps. If it's NO, DSP commands are processed as quickly as possible. By default, an Orchestra is timed; this method sets the default to *areOrchsTimed*.

stop

+ **stop**

Sends **stop** to each of the Orchestra instances and sets each to MK_devStopped. If any of the Orchestras responds to the **run** message by returning **nil**, so, too, does this method return **nil**. Otherwise returns **self**.

INSTANCE METHODS

abort

– **abort**

This is the same as *close*, except it doesn't wait for enqueued DSP commands to be executed. Returns **nil** if an error occurs, otherwise returns **self**.

allocPatchpoint:

– **allocPatchpoint:**(MKOrchMemSegment)*segment*

Allocates and returns a SynthData to be used as a patchpoint in the specified segment (MK_xPatch or MK_yPatch). Returns **nil** if an illegal segment is requested.

allocSynthData:length:

– **allocSynthData:**(MKOrchMemSegment)*segment* **length:**(unsigned)*size*

Allocates and returns a new SynthData object with the specified length, or **nil** if the Orchestra doesn't have sufficient resources, if *size* is 0, or if an illegal segment is requested. *segment* should be MK_xData or MK_yData.

allocSynthPatch:

– **allocSynthPatch:***aSynthPatchClass*

Same as **allocSynthPatch:patchTemplate:** but uses the default template.

allocSynthPatch:patchTemplate:

– **allocSynthPatch:***aSynthPatchClass* **patchTemplate:***p*

Allocates and returns a SynthPatch for PatchTemplate *p*. The Orchestra first tries to find an idle SynthPatch; failing that, it creates and returns a new one. If a new one can't be built, this method returns **nil**.

allocUnitGenerator:

– **allocUnitGenerator:***class*

Allocates and returns a UnitGenerator of the specified class, creating a new one if necessary.

allocUnitGenerator:after:

– **allocUnitGenerator:***class* **after:***aUnitGeneratorInstance*

Allocates and returns a UnitGenerator of the specified class. The newly allocated object will execute after *aUnitGeneratorInstance*.

allocUnitGenerator:before:

– **allocUnitGenerator:***class* **before:***aUnitGeneratorInstance*

Allocates and returns a UnitGenerator of the specified class. The newly allocated object will execute before *aUnitGeneratorInstance*.

allocUnitGenerator:between::

– **allocUnitGenerator:***class*
between:*aUnitGeneratorInstance*
:*anotherUnitGeneratorInstance*

Allocates and returns a UnitGenerator of the specified class. The newly allocated object will execute after *aUnitGeneratorInstance* and before *anotherUnitGenerator.*

beginAtomicSection

– **beginAtomicSection**

Marks the beginning of a section of DSP commands that are sent as a unit; this method should be balanced by **endAtomicSection**. Returns **self**. You should use this method when you are sending a block of DSP commands that, if broken up, would leave the DSP in an inconsistent state. Atomic sections are recursive: If you send **beginAtomicSection** twice, you must send **endAtomicSection** twice.

close

– **close**

Severs communication with the DSP, allowing other processes to claim it. Before closing, all enqueued DSP commands are executed. The SynthPatch-allocated UnitGenerators and SynthInstrument-allocated SynthPatches are freed. All SynthPatches must be idle and non-SynthPatch-allocated UnitGenerators must be deallocated before sending this message. Returns **nil** if an error occurs, otherwise returns **self**.

computeTime

– (double)**computeTime**

Returns the compute time estimate currently used by the Orchestra, in seconds-per-sample.

dealloc:

– **dealloc:***aSynthResource*

Deallocates *aSynthResource* by sending it the **dealloc** message. *aSynthResource* may be a UnitGenerator, a SynthData, or a SynthPatch.

deviceStatus

– (MKDeviceStatus)**deviceStatus**

Returns the Orchestra status, one of

- MK_devClosed
- MK_devOpen
- MK_devRunning
- MK_devStopped

The Orchestra states are explained in the class description, above.

endAtomicSection

– **endAtomicSection**

Marks the end of a section of DSP commands that are sent as a unit, as begun by **beginAtomicSection**. Returns **self**.

fastResponse

– (BOOL)**fastResponse**

Returns YES if the Orchestra is using small sound-out buffers to minimize response latency. Otherwise returns NO.

flushTimedMessages

– **flushTimedMessages**

Sends buffered DSP commands to the DSP. This is usually done for you by the Conductor; however, if your application sends messages directly to a SynthPatch or UnitGenerator without the assistance of a Conductor, you must invoke this method yourself (after sending the synthesis messages). Returns **self**.

free

– **free**

Frees the Orchestra and its UnitGenerators, clears all SynthPatch allocation lists, and releases the DSP. All SynthPatches must be idle and non-SynthPatch-allocated UnitGenerators must be deallocated before sending this message. Returns **nil** if an error occurs, otherwise returns **self**.

headroom

– (double)**headroom**

Returns the Orchestra's headroom, as set through the **setHeadroom:** method. Headroom should be a value between –.0 and 1.0. The default is 0.1.

index

– (unsigned short)**index**

Returns the (zero-based) index of the DSP associated with the Orchestra.

installSharedObject:for:

– **installSharedObject:***aSynthObj* **for:***aKeyObj*

Places *aSynthObj* on the shared object table and sets its reference count to 1. *aKeyObj* is used to index the shared object. Does nothing and returns **nil** if the *aSynthObj* is already present in the table. Also returns **nil** if the Orchestra isn't open. Otherwise, returns **self**.

This method differs from **installSharedObjectWithSegmentAndLength:for:** in that the length and segment are wild cards.

installSharedSynthDataWithSegment:for:

– **installSharedSynthDataWithSegment:***aSynthDataObj* **for:***aKeyObj*

Places *aSynthDataObj* on the shared object table in the segment specified by *aSynthDataObj* and sets its reference count to 1. Does nothing and returns **nil** if the *aSynthObj* is already present in the table. Also returns **nil** if the Orchestra isn't open. Otherwise, returns **self**.

This method differs from **installSharedObjectWithSegmentAndLength:for:** in that the length is a wild card.

installSharedSynthDataWithSegmentAndLength:for:

– **installSharedSynthDataWithSegmentAndLength:***aSynthDataObj* **for:***aKeyObj*

Places *aSynthDataObj* on the shared object table in the segment of aSynthDataObj with the specified length and sets its reference count to 1. *aKeyObj* is used to index the shared object. Does nothing and returns **nil** if the *aSynthDataObj* is already present in the table. Also returns **nil** if the Orchestra isn't open. Otherwise, returns **self**.

isTimed

– (BOOL)**isTimed**

Returns YES if the Orchestra is timed, NO if it's untimed.

localDeltaT

– (double)**localDeltaT**

Returns the value set through **setLocalDeltaT:**.

muLawROM

– muLawROM

Returns a SynthData object representing the MuLawROM. You should never deallocate this object.

open

– open

Opens the Orchestra's DSP and sets the Orchestra's status to MK_devOpen. Returns **nil** if the DSP can't be opened, otherwise returns **self**.

outputCommandsFile

– (char *)**outputCommandsFile**

Returns a pointer to the name of the Orchestra's DSP commands format soundfile, or NULL if none.

outputSoundfile

– (char *)**outputSoundfile**

Returns a pointer to the name of the Orchestra's output soundfile, or NULL if none.

peekMemoryResources:

– (MKOrchMemStruct *)**peekMemoryResources:**(MKOrchMemStruct *)*peek*

Returns the available resources in *peek*, which must be a pointer to a valid MKOrchMemStuct. The returned value is the available memory for each segment; however, xData, yData and pSubr compete for the same memory. You should interpret the returned value with appropriate caution.

run

– run

Starts the clock on the Orchestra's DSP, thus allowing the processor to begin executing commands, and sets the Orchestra's status to MK_devRunning. This opens the DSP if it isn't already open. Returns **nil** if the DSP couldn't be opened or run, otherwise returns **self**.

samplingRate

– (double)**samplingRate**

Returns the Orchestra's sampling rate. The default is 22050.0.

segmentName:

– (char *)**segmentName:**(int)*whichSegment*

Returns a pointer to the name of the specified MKOrchMemSegment.

segmentSink:

– **segmentSink:**(MKOrchMemSegment)*segment*

Returns a special pre-allocated patchpoint (a SynthData) in the specified segment from which, by convention, data is never read. It's commonly used as a place to send the output of idle UnitGenerators. The patchpoint shouldn't be deallocated. The argument must be either MK_xPatch or MK_yPatch.

segmentZero:

– **segmentZero:**(MKOrchMemSegment)*segment*

Returns a special pre-allocated patchpoint (a SynthData) in the specified segment that always holds 0 and to which, by convention, nothing is ever written. The patchpoint shouldn't be deallocated. The argument must be either MK_xPatch or MK_yPatch.

setFastResponse:

– **setFastResponse:**(BOOL)*yesOrNo*

Sets the size of the sound output buffer; two sizes are possible. If *yesOrNo* is YES, the smaller size is used, thereby improving response time but somewhat decreasing the DSP's synthesis power. If it's NO, the larger buffer is used. By default, an Orchestra uses the larger buffer. Returns **self**.

setHeadroom:

– **setHeadroom:**(double)*headroom*

Sets the Orchestra's computation headroom, adjusting the tradeoff between processing power and reliability. The argument should be in the range -.0 to 1.0. As you increase an Orchestra's headroom, the risk of falling out of real time decreases, but synthesis power is also weakened. The default, 0.1, is a conservative estimate and can be decreased in many cases without heightening the risk of falling out of real time.

The effective sampling period—the amount of time the Orchestra thinks the DSP has to produce a sample—is based on the formula

(1.0/**samplingRate**) * (1.0 – **headroom**).

Returns **self**.

setLocalDeltaT:

– **setLocalDeltaT:**(double)*val*

Sets the offset, in seconds, that's added to the timestamps of commands sent to the Orchestra's DSP. The offset is added to the delta time that's set with **MKSetDeltaT()**. This has no effect if the receiver isn't timed. Returns **self**.

setOffChipMemoryConfigXArg:yArg:

– **setOffChipMemoryConfigXArg:**(float)*xPercentage* **yArg:**(float)*yPercentage*

Reserves percentages of off-chip memory for X and Y memory arguments. The arguments must be between 0.0 and 1.0. An argument of 0.0 causes the default percentage to be used for that segment. If *xPercentage* + *yPercentage* is greater than 1.0, the settings are ignored and the method returns **nil**. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

setOnChipMemoryConfigDebug:patchPoints:

– **setOnChipMemoryConfigDebug:**(BOOL)*debugIt* **patchPoints:**(short)*count*

Sets configuration of on-chip memory. If *debugIt* is YES, a partition is reserved for the DSP debugger; *count* is the number of on-chip patchpoint locations that are reserved. By default, the debugger isn't used and 11 patchpoints are reserved. If *count* is 0, the default is used. By implication, this also sets the number of UnitGenerator arguments that can be set in L memory: As more patchpoints are requested, fewer UnitGenerator arguments are possible. Attempts to set the patchpoint count, such that no room is left for L arguments, are ignored. Returns **self**, or **nil** if the configuration is unsuccessful.

setOutputCommandsFile:

– **setOutputCommandsFile:**(char *)*fileName*

Sets the DSP commands format soundfile to which DSP commands are written. The file can be played as a soundfile using any of the standard soundfile-playback functions, utilities, or applications. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

Writing samples to a commands file (through **setOutputCommandsFile:**), a soundfile (through **setOutputSoundfile:**), a simulator file (through **setSimulatorFile:**), and to the DAC (through **setSoundOut:**) are mutually exclusive operations.

setOutputSoundfile:

– **setOutputSoundfile:**(char *)*fileName*

Sets the soundfile to which sound samples are written. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

Writing samples to a commands file (through **setOutputCommandsFile:**), a soundfile (through **setOutputSoundfile:**), a simulator file (through **setSimulatorFile:**), and to the DAC (through **setSoundOut:**) are mutually exclusive operations.

setSamplingRate:

– **setSamplingRate:**(double)*newSRate*

Sets the Orchestra's sampling rate to *newSRate*, taken as samples per second. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

setSimulatorFile:

– **setSimulatorFile:**(char *)*fileName*

Sets the name of a file to which simulator output is sent. The file is in a format that can be passed directly to the Motorola Simulator. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

Writing samples to a commands file (through **setOutputCommandsFile:**), a soundfile (through **setOutputSoundfile:**), a simulator file (through **setSimulatorFile:**), and to the DAC (through **setSoundOut:**) are mutually exclusive operations.

setSoundOut:

– **setSoundOut:**(BOOL)*yesOrNo*

Sets whether the Orchestra sends its sound signal to the DAC, as *yesOrNo* is YES or NO. All Orchestras send to the DAC by default. The Orchestra must be closed when you invoke this method: Returns **nil** if it's open, otherwise returns **self**.

Writing samples to a commands file (through **setOutputCommandsFile:**), a soundfile (through **setOutputSoundfile:**), a simulator file (through **setSimulatorFile:**), and to the DAC (through **setSoundOut:**) are mutually exclusive operations.

setTimed:

– **setTimed:**(BOOL)*isOrchTimed*

If *isOrchTimed* is YES, the Orchestra's DSP executes the commands it receives according to their timestamps. If it's NO, the DSP ignores the timestamps and processes the commands immediately. By default, an Orchestra is timed.

sharedObjectFor:

– **sharedObjectFor:***aKeyObj*

Returns, from the Orchestra's shared object table, the SynthData, UnitGenerator, or SynthPatch object that's indexed by *aKeyObj*. If the object is found, *aKeyObj*'s

reference count is incremented. If it isn't found, or if the Orchestra isn't open, returns **nil**.

sharedObjectFor:segment:

– **sharedObjectFor:***aKeyObj* **segment:**(MKOrchMemSegment)*whichSegment*

Returns, from the Orchestra's shared data table, the SynthData, UnitGenerator, or SynthPatch object that's indexed by *aKeyObj*. The object must be allocated in the specifed segment. If the object is found, *aKeyObj*'s reference count is incremented. If it isn't found, or if the Orchestra isn't open, returns **nil**.

sharedObjectFor:segment:length:

– **sharedObjectFor:***aKeyObj*
segment:(MKOrchMemSegment)*whichSegment*
length:(int)*length*

Returns, from the Orchestra's shared data table, the SynthData, UnitGenerator, or SynthPatch object that's indexed by *aKeyObj*. The object must be allocated in the specifed segment and have a length of *length*. If the object is found, *aKeyObj*'s reference count is incremented. If it isn't found, or if the Orchestra isn't open, returns **nil**.

simulatorFile

– (char *)**simulatorFile**

Returns a pointer to the name of the Orchestra's simulator file, or NULL if none.

sineROM

– **sineROM**

Returns a SynthData object representing the SineROM. You should never deallocate this object.

stop

– **stop**

Stops the clock on the Orchestra's DSP, thus halting execution of commands, and sets the Orchestra's status to MK_devStopped. This opens the DSP if it isn't already open. Returns **nil** if an error occurs, otherwise returns **self**.

trace:msg:

– **trace:**(int)*typeOfInfo* **msg:**(char *)*fmt*,...

Used to print debugging information. The arguments to the **msg:** keyword are like those to **printf**(). If the *typeOfInfo* trace is set, prints to standard error.

Part

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A Part is a sorted collection of Notes that can be edited and performed. Parts are typically grouped together in a Score.

A Note can belong to only one Part at a time, and a Part to only one Score. When you add a Note to a Part, it's automatically removed from its old Part. Similarly, adding a Part to a Score removes it from its previous Score.

You can add Notes to a Part either by invoking one of Part's **addNote:** methods, or by "recording" them with a PartRecorder, a type of Instrument that realizes Notes by adding copies of them to a specified Part. A Part is added to a Score through Part's **addToScore:** method (or the equivalent Score method **addPart:**).

Within a Part, Notes are ordered by their time tag values, lowest to highest. To move a Note within a Part, you simply change the Note's time tag (through Note's **setTimeTag:** method). For efficiency, a Part sorts itself only when its Notes are retrieved or when a Note is moved within the Part (or removed altogether). In other words, adding a Note to a Part won't cause the Part to sort itself; but keep in mind that since adding a Note to a Part automatically removes it from its current Part, the act will cause the moved-from Part to sort itself. You can force a Part to sort itself by sending it a **sort** message.

A Part can be a source of Notes in a performance through association with a PartPerformer. During a performance, the PartPerformer reads the Notes in the Part, performing them in order. While you shouldn't free a Part or any of its Notes while an associated PartPerformer is active, you can add Notes to and remove Notes from the Part at any time without affecting the PartPerformer's performance.

To each Part you can give an info Note, a sort of header for the Part that can contain any amount and type of information. Info Notes are typically used to describe a performance setup; for example, an info Note might contain, as a parameter, the name of the SynthPatch subclass on which the Notes in the Part are meant to be synthesized. Keep in mind that a Part's info Note, like any other Note, must be retrieved and its parameters applied by some other object (or your application) for it to have an effect. A few parameters defined by the Music Kit are designed specifically to be used in a Part's info Note. These are listed in the description of the **setInfo:** method, below. The info Note is stored separately from the Notes in the body of the Part; most of the Note-accessing methods, such as **empty**, **nth:**, and **next:**, don't apply to the info Note. The exceptions—the methods that *do* affect the info Note—are so noted in their descriptions below.

Parts are commonly given string name identifiers, through the **MKNameObject()** C function. The most important use of a Part's name is to identify the Part in a scorefile.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Part	id	score;
	id	notes;
	id	info;
	int	noteCount;
	BOOL	isSorted;

score	The Score the Part is a member of.
notes	The Part's List of Notes.
info	The Part's info Note.
noteCount	Number of Notes in the Part.
isSorted	YES if the Part is currently sorted.

METHOD TYPES

Creating and freeing a Part	– copy – copyFromZone: – init – free – freeSelfOnly
Storing the object	– addToScore: – score – removeFromScore – read: – write: – awake
Adding Notes	– addNote: – addNoteCopy: – addNotes:timeShift: – addNoteCopies:timeShift: – noteCount – setInfo:
Removing Notes	– removeNote: – removeNotes: – empty – isEmpty – freeNotes

Retrieving Notes	– notes – notesNoCopy – atTime: – atTime:nth: – atOrAfterTime: – atOrAfterTime:nth: – firstTimeTag:lastTimeTag: – nth: – next: – containsNote: – info
Manipulating and Sorting Notes	– combineNotes – splitNotes – shiftTime: – sort – isSorted

INSTANCE METHODS

addNote:

– **addNote:***aNote*

Moves *aNote* from its present Part to the receiving Part. Returns *aNote*'s old Part, or **nil** if none.

See also: – **addNoteCopy:**, – **addNotes:timeShift:**, – **removeNote:**

addNoteCopy:

– **addNoteCopy:***aNote*

Adds a copy of *aNote* to the Part. Returns the new Note.

See also: – **addNote:**, – **addNoteCopies:timeShift:**, – **removeNote:**

addNotes:timeShift:

– **addNotes:***aNoteList* **timeShift:**(double)*shift*

Moves each Note in *aNoteList* from its present Part to the receiving Part, adding *shift* to each Note's time tag in the process. The List argument is typically generated through Part's **notes** or **firstTimeTag:lastTimeTag:** method. In this way, all or a portion of one Part can be merged into another. Returns **self**, or **nil** if *aNoteList* is **nil**.

See also: – **addNoteCopies:timeShift:**, – **shiftTime:**

addNoteCopies:timeShift:

– **addNoteCopies:***aNoteList* **timeShift:**(double)*shift*

Copies each Note in *aNoteList* into the Part, adding *shift* to each new Note's time tag in the process (the Notes in the List are unaffected). The List argument is typically generated through Part's **notes** or **firstTimeTag:lastTimeTag:** method. In this way, all or a portion of one Part can be copied into another. Returns **self**, or **nil** if *aNoteList* is **nil**.

See also: – **addNotes:timeShift:**, – **shiftTime:**

addToScore:

– **addToScore:***aScore*

Moves the Part from its present Score, if any, to *aScore*. This is equivalent to Score's **addPart:** method. Returns **self**.

See also: – **removeFromScore**, – **score**

atOrAfterTime:

– **atOrAfterTime:**(double)*timeTag*

Returns the first Note with a time tag equal to or greater than *timeTag*, or **nil** if none.

See also: – **atTime:**, – **atOrAfterTime:nth:**, – **next:**

atOrAfterTime:nth:

– **atOrAfterTime:**(double)*timeTag* **nth:**(unsigned)*n*

Returns the *n*th Note (zero-based) in the Part that has a time tag equal to or greater than *timeTag*, or **nil** if none.

See also: – **atTime:**, – **atOrAfterTime:**, – **next:**

atTime:

– **atTime:**(double)*timeTag*

Returns the (first) Note in the Part that has a time tag of *timeTag*, or **nil** if none. Invokes Note's **compareNotes:** method if the Part contains more than one such Note.

See also: – **atOrAfterTime:**, – **atTime:nth:**, – **next:**

atTime:nth:

– **atTime:**(double)*timeTag* **nth:**(unsigned)*n*

Returns the *n*th Note (zero-based) in the Part that has a time tag of *timeTag*, or **nil** if none.

See also: – **atTime:**, – **atOrAfterTime:**, – **next:**

combineNotes

– **combineNotes**

Creates and adds a single noteDur for each noteOn/noteOff pair in the Part. A noteOn/noteOff pair is identified by pairing a noteOn with the earliest subsequent noteOff that has a matching note tag. The parameters from the two Notes are merged in the noteDur. If the noteOn and its noteOff have different values for the same parameter, the value from the noteOn takes precedence. The noteDur's duration is the time tag difference between the two original Notes. After the noteDur is created and added to the Part, the noteOn and noteOff are removed and freed. Returns **self**.

See also: – **splitNotes**

containsNote:

– (BOOL)**containsNote:***aNote*

Returns YES if the Part contains *aNote*, otherwise returns NO.

See also: – **isEmpty**, – **noteCount**

copy

– **copy**

Creates and returns a new Part as a copy of the receiving Part. The new Part contains copies of the receiving Part's Notes (including the info Note) and is added to the same Score as the receiving Part, but is left unnamed.

See also: – **copyFromZone:**

copyFromZone:

– **copyfromZone:**(NXZone *)*aZone*

This is the same as **copy**, but the new Note is allocated in *aZone*.

See also: – **copy**

empty

– empty

Removes the Part's Notes but not its info Note. Returns **self**.

See also: **– removeNote:**, **– freeNotes**

firstTimeTag:lastTimeTag:

– firstTimeTag:(double)*firstTimeTag* **lastTimeTag:**(double)*lastTimeTag*

Creates and returns a List of the Part's Notes that have time tag values between *firstTimeTag* and *lastTimeTag*, inclusive. The sender is responsible for freeing the List. The object returned by this method is useful as the List argument in methods such as **addNotes:** (sent to another Part), **addNotes:timeShift:**, and **removeNotes:**.

free

– free

Frees the Part, its Notes, and its info Note. Removes the Part's name from the Music Kit name table. If the Part has an active PartPerformer associated with it, this method does nothing.

See also: **– freeSelfOnly**, **– freeNotes**

freeNotes

– freeNotes

Removes and frees the Part's Notes and its info Note. If the Part has an active PartPerformer associated with it, this does nothing and returns **nil**; otherwise returns **self**.

See also: **– empty**, **– removeNotes:**

freeSelfOnly

– freeSelfOnly

Frees the Part but not its Notes. The Part is removed from its Score, if any. You can free a Part while it's being performed by a PartPerformer—it's the Part's Notes, not the Part itself, that's performed.

See also: **– empty**, **– removeNotes:**

info

– **info**

Returns the Part's info Note.

See also: – **setInfo:**

isEmpty

– (BOOL)**isEmpty**

Returns YES if the Part contains no Notes (not including the info Note), otherwise returns NO.

See also: – **noteCount**

isSorted

– (BOOL)**isSorted**

Returns YES if the Part's Notes are currently guaranteed to be in time tag order, otherwise returns NO.

See also: – **sort**

next:

– **next:***aNote*

Returns the Note immediately following *aNote*, or **nil** if *aNote* isn't a member of the Part, or if it's the last Note in the Part.

See also: – **nth:**, – **atTime:**, – **atOrAfterTime:**

noteCount

– (unsigned)**noteCount**

Returns the number of Notes in the Part (not counting the info Note).

See also: – **notes**, – **isEmpty**

notes

– **notes**

Creates and returns a List of the Part's Notes. The Part is sorted before the List is created. The sender is responsible for freeing the List.

See also: – **notesNoCopy**, – **noteCount**

notesNoCopy

– **notesNoCopy**

Returns the List object that contains the Part's Notes. The List isn't guaranteed to be sorted.

See also: – **notes**, – **noteCount**

nth:

– **nth:**(unsigned)*n*

Returns the *n*th Note (0-based), or **nil** if *n* is out of bounds (negative or greater than the Part's Note count).

See also: – **notes**, – **noteCount**, – **atTime:**

removeFromScore

– **removeFromScore**

Removes the Part from its present Score. This is equivalent to Score's **removePart:** method. Returns **self**, or **nil** if it isn't part of a Score.

See also: – **addToScore:**, – **score**

removeNote:

– **removeNote:***aNote*

Removes *aNote* from the Part. Returns the Note or **nil** if it isn't found.

See also: – **removeNotes:**, – **empty**, – **addNote**

removeNotes:

– **removeNotes:***aList*

Removes all the Notes the Part has in common with *aList*. Returns **self**.

See also: – **removeNote:**, – **empty**, – **addNote:**, – **firstTimeTag:lastTimeTag:**

score

– **score**

Returns the Score the Part is a member of, or **nil** if none.

See also: – **addToScore:**, – **removeFromScore**

setInfo:

– **setInfo:***aNote*

Sets the Part's info Note to *aNote* and returns **self**. The info Note can be given information (as parameters) that helps define how the Part should be interpreted; in particular, special Music Kit parameters (more accurately, parameter tags) are designed to be used in a Part info Note. Listed below, these parameters pertain to the manner in which the Notes in the Part are synthesized, although as with any Note, the info Note's parameters must be read and applied by some other object (or your application) in order for them to have an effect. Keep in mind that the info Note is by no means restricted to containing only these parameters.

Parameter Tag	Expected Value	Typical Use
MK_synthPatch	SynthPatch subclass	Argument to SynthInstrument's **setSynthPatchClass:** method.
MK_synthPatchCount	integer	Argument to SynthInstrument's **setSynthPatchCount:** method.
MK_midiChan	integer	Argument to Midi's **channelNoteReceiver:** method.
MK_track	integer	Automatically set when a midifile is read into a Score.

The info Note is stored separately from the Part's main body of Notes; methods such as **empty** don't affect it.

See also: – **info**

shiftTime:

– **shiftTime:**(double)*shift*

Shifts the Part's contents by adding *shift* to each of the Notes' time tags. Returns **self**.

sort

– **sort**

Causes the Part to sort itself if it's currently unsorted. Normally, a Part sorts itself only when Notes are accessed, moved, or removed. Returns **self**.

See also: – **isSorted**

splitNotes

– splitNotes

Splits the Part's noteDurs into noteOn/noteOff pairs. Each noteDur's note type is set to noteOn and a noteOff is created (and added) to complement it. The original parameters and note tag are divided between the two Notes as described in Note's **split::** method. Returns **self**.

See also: **– combineNotes:**, **– split::** (Note)

Partials

INHERITS FROM WaveTable : Object

DECLARED IN musickit.h

CLASS DESCRIPTION

The Partials class lets you define a sound waveform by adding together a number of sine wave components. Partials are used to provide musical timbres in DSP synthesis, primarily by the SynthPatch classes that provide wave table synthesis—classes such as Wave1vi and DBWave1vi. Partials' sister class, Samples, lets you define a waveform as a series of sound samples, through association with a Sound object or soundfile.

Each of the sine waves in a Partials object is characterized by a frequency ratio, an amplitude ratio, and an initial phase. The frequency ratios are taken as integer multiples of a fundamental frequency—in other words, a ratio of 1.0 is the fundamental frequency, 2.0 is twice the fundamental, 3.0 is three times the fundamental, and so on. The fundamental frequency itself is defined in the frequency parameters of the Note objects that use the Partials. The amplitude ratios are relative to each other: A sine wave component with an amplitude ratio of 0.5 has half the amplitude of a component with an amplitude ratio of 1.0. The initial phase determines the point in the sine curve at which a particular component starts. Phase is specified in degrees; a phase of 360.0 is the same as a phase of 0.0. While phase information has been found to have little significance in the perception of timbre, it can be important in other uses. For example, if you're creating a waveform that's used as a sub-audio control signal—most notably for vibrato—you will probably want to randomize or stagger the phases of the sine waves.

All the component information for a Partials object is set through the **setPartialCount:freqRatios:ampRatios:phases:orDefaultPhase:** method. The first argument, an **int**, is the number of sine waves in the object. The next three arguments are pointers to arrays of **double**s that provide corresponding lists of frequency, amplitude, and phase information. The additional **orDefaultPhase:** keyword is provided in recognition of phase's slim contribution to the scheme: Rather than create and set an array of initial phases, you can pass NULL to **phases:** and set all the sine wave components to a common initial phase as the argument to **orDefaultPhase:**. The following example demonstrates how to create a simple, three component Partials object.

```
double freqs    = {1.0, 2.0, 3.0 };
double amps     = {1.0, 0.5, 0.25 };
id aPartials    = [Partials new];

[aPartials setPartialCount:3 freqRatios:freqs
ampRatios:amps phases:NULL orDefaultPhase:0.0];
```

The elements in the arrays are matched by index order: The first sine wave is defined by the first element of **freqs** and the first element of **amps**; the second elements of the arrays define the second sine wave; the third elements define the third sine wave. Since the phase array is specified as NULL, all three sine waves are given an initial phase of 0.0.

In a scorefile, Partials are defined as curly-bracketed value pairs—or triplets if you want to specify phase—and the entire Partials definition is enclosed in square brackets. If a phase value is missing, the phase of the previous component is used; the default phase is 0.0. You can define a Partials object in-line as the value of a parameter or, more typically, in a global **waveTable** statement. The previous example could be defined in a scorefile as

```
waveTable simpleSound = [{1.0, 1.0}{2.0, 0.5}{3.0, 0.25}];
```

where **simpleSound** is used to identify the object in subsequent Note statements:

```
partName (1.0) ... waveform:simpleSound ...;
```

When this scorefile is read into an application, the Partials object will be given the string name "simpleSound". The object itself can be retrieved by passing this string to the **MKGetNamedObject()** C function.

If you're creating a Partials object in an application and writing it to a scorefile, you should always name the object through **MKNameObject()**. This allows the object to be defined once (albeit in-line, not in the header) in a **waveTable** statement and then referred to by name in subsequent Notes. Without a name, a Partials object is defined in-line in every Note statement that refers to it.

As a convenience, the Partials class lets you assign a range of fundamental frequencies for which an object is valid. This can be useful in avoiding foldover, a bitter reality of digitally-generated sounds that occurs when you try to create a frequency greater than half the sampling rate. The **setFreqRangeLow:high:** method specifies the range. **freqWithinRange:** returns YES or NO as the argument frequency is within the valid range. However, keep in mind that it's the responsibility of the application or SynthPatch to check the frequency range.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from WaveTable	int	length;
	double	scaling;
	DSPDatum	*dataDSP;
	double	*dataDouble;

Declared in Partials

```
double      *ampRatios;
double      *freqRatios;
double      *phases;
int         partialCount;
double      defaultPhase;
double      minFreq;
double      maxFreq;
```

ampRatios	Array of amplitude ratios
freqRatios	Array of frequency ratios.
phases	Arrays of initial phases.
partialCount	Number of sine waves in the object.
defaultPhase	Common phase, used in the absence of **phases**.
minFreq	Optional frequency minimum.
maxFreq	Optional frequency maximum.

METHOD TYPES

Creating and freeing a Partials	– copy
Defining the sine waves	– setPartialCount:freqRatios:ampRatios:phases: orDefaultPhase:
Modifying the object	– free – init – setFreqRangeLow:high: – writeScorefileStream:
Querying the object	– ampRatios – defaultPhase – freqRatios – freqWithinRange: – getPartial:freqRatio:ampRatio:phase: – highestFreqRatio – maxFreq – minFreq – partialCount – phases
Computing the waveform	– fillTableLength:scale:

ampRatios

– (double *)**ampRatios**

Returns a pointer to the receiver's amplitude ratios array. You should neither free nor alter the array.

copy

– **copy**

Creates and returns a Partials as a copy of the receiver. The returned object contains copies of the receiver's arrays.

defaultPhase

– (double)**defaultPhase**

Returns the receiver's default phase.

fillTableLength:scale:

– **fillTableLength:**(int)*aLength* **scale:**(double)*aScaling*

Computes the sampled waveform from the sine wave components.

This method is a subclass responsibility that's invoked automatically by the data retrieval methods inherited from the WaveTable class—you needn't invoke this method yourself. Returns the receiver, or **nil** if an error occurs.

free

– **free**

Frees the receiver and its arrays.

freqRatios

– (double *)**freqRatios**

Returns a pointer to the receiver's frequency ratios array. You should neither free nor alter the array.

freqWithinRange:

– (BOOL)**freqWithinRange:**(double)*freq*

Returns YES if *freq* is within the range of fundamental frequencies ordinarily associated with this timbre, as set by **setFreqRangeLow:high:**.

getPartial:freqRatio:ampRatio:phase:

– (int)**getPartial:**(int)*n*
freqRatio:(double *)*fRatio*
ampRatio:(double *)*aRatio*
phase:(double *)*phase*

Returns, by reference, the frequency ratio, amplitude ratio, and initial phase of the *n*th sine wave component (counting from 0). The amplitude ratio value is scaled by the current value of the **scaling** instance variable inherited from WaveTable.

If the *n*th sine wave is the last in the receiver, the method returns MK_lastValue. If *n* is out of bounds, –1 is returned. Otherwise 0 is returned unless *n* is the last point, in which case 2 is returned.

highestFreqRatio

– (double)**highestFreqRatio**

Returns the highest frequency ratio in the receiver. This can be useful in determining if the receiver will generate a waveform that will fold over.

init

– **init**

Initializes the receiver. A subclass implementation should send **[super init]** before performing its own initialization. The return value is ignored.

maxFreq

– (double)**maxFreq**

Returns the maximum fundamental frequency at which this timbre is ordinarily used.

minFreq

– (double)**minFreq**

Returns the minimum fundamental frequency at which this timbre is ordinarily used.

partialCount

– (int)**partialCount**

Returns the number of sine wave components.

phases

– (double *)**phases**

Returns a pointer to the receiver's phase array. You should neither free nor alter the array.

setFreqRangeLow:high:

– **setFreqRangeLow:**(double)*freq1*
high:(double)*freq2*

Sets the frequency range associated with this timbre.

setPartialCount:freqRatios:ampRatios:phases:orDefaultPhase:

– **setPartialCount:**(int)*count*
freqRatios:(double *)*freqRats*
ampRatios:(double *)*ampRats*
phases:(double *)*phases*
orDefaultPhase:(double)*defaultPhase*

Defines the receiver's sine wave components. *count* is the number of sine waves components; *freqRats, ampRats*, and *phases* are pointers to arrays that define the frequency ratios, amplitude ratios, and initial phases, respectively, of the sine wave components (the arrays are copied into the receiver). The elements of the arrays are matched by index order: The nth sine wave is configured from the nth element in each array.

If *phases* is NULL, the value of *defaultPhase* is used as the initial phase for all the components. If *freqRats* or *ampRats* is NULL, the corresponding extant array, if any, is unchanged.

Note that this method sets the **length** instance variable to 0, forcing a recompute in a subsequent data array retrieval (through the **dataDSP:...** and **dataDouble:...** methods) as explained in the WaveTable class.

Returns the receiver.

writeScorefileStream:

– **writeScorefileStream:**(NXStream *)*aStream*

Writes the receiver in scorefile format on the specified stream. Returns **nil** if **ampRatios** or **freqRatios** is NULL, otherwise returns the receiver.

PartPerformer

INHERITS FROM Performer : Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A PartPerformer object performs the Notes in a particular Part. The association between a Part and a PartPerformer is made through PartPerformer's **setPart:** method. While a single PartPerformer can only be associated with one Part, any number of PartPerformers can be associated with the same Part.

When you activate a PartPerformer the object copies its Part's list of Notes, but it doesn't copy the Notes themselves. When it's performed, the PartPerformer sequences over its copy of the list, allowing you to edit the Part (by adding or removing Notes) without disturbing the performance—changes made to a Part during a performance are not seen by the PartPerformer. However, since only the list of Notes is copied but not the Notes themselves, you should never free a Part's Notes during a performance, and you should alter them with discretion (in particular, it's not a good idea to change a Note's time tag while the Part that contains it is being performed).

A PartPerformer remembers its Part between performances, but it generates a new copy of the Part's Note list each time it's activated. Thus, if you edit a Part during a performance, you don't have to resend **setPart:** before the next performance to get the PartPerformer to recognize the changes; the modifications will be seen when the PartPerformer is reactivated.

In addition to the time shift and duration variables that it inherits from Performer, a PartPerformer contains variables that define the first time tag and last time tag values that it considers valid for a performance. Consider the following:

```
[aPartPerf setTimeShift:3.0];
[aPartPerf setFirstTimeTag:5.0];
```

The PartPerformer will perform, as its initial Note, the first Note that it finds in its Part that has a time tag value greater than or equal to 5.0, and it performs it at beat 3.0 plus the difference between 5.0 and the Note's actual time tag value.

A PartPerformer's duration and its last time tag value compete with each other; you shouldn't set both variables. For example, the following:

```
[aPartPerf setDuration:4.0];
```

is equivalent to

```
[aPartPerf setLastTimeTag:[aPartPerf firstTimeTag] + 4.0];
```

A PartPerformer creates a single NoteSender through which it sends all its Notes. It won't recognize any NoteSenders that you add from your application.

If you're performing a Score, you can use a ScorePerformer to automatically create PartPerformers for you, one for each Part in the Score. In addition, if you design your own subclass of PartPerformer, you can tell a ScorePerformer to create instances of your subclass through ScorePerformer's **setPartPerformerClass:** method.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Performer	id	conductor;
	MKPerformerStatus	status;
	int	performCount;
	double	timeShift;
	double	duration;
	double	time;
	double	nextPerform;
	id	noteSenders;
Declared in PartPerformer	id	nextNote;
	id	noteSender;
	id	part;
	double	firstTimeTag;
	double	lastTimeTag;

nextNote — The next Note to perform.

noteSender — The PartPerformer's only NoteSender.

part — The Part associated with this object.

firstTimeTag — Least time tag value considered for performance.

lastTimeTag — Greatest time tag value considered for performance.

METHOD TYPES

Initializing and freeing PartPerformer
- init
- free

Accessing the Part
- setPart:
- part

Performing the object	– activateSelf – deactivateSelf – perform
Accessing the timing variables	– setFirstTimeTag: – firstTimeTag – setLastTimeTag: – lastTimeTag
Archiving the object	– write: – read: – awake

INSTANCE METHODS

activateSelf

– **activateSelf**

Prepares the PartPerformer for a performance by creating a copy of its Part's list of Notes. It then zips through the Note list copy until a Note with a time tag greater than or equal to the PartPerformer's first time tag value (as set through **setFirstTimeTag:**) is found. The **nextPerform** instance variable is set such that the Note will be performed at a time that reflects the difference between the Note's time tag and the value of the PartPerformer's first time tag, plus the PartPerformer's performance time offset (as set through the **setTimeShift:** method inherited from Performer).

You never invoke this method directly; it's invoked as part of the **activate** method inherited from Performer. It returns **nil** (and the PartPerformer isn't activated) if the Part hasn't been set, if the Part doesn't contain any Notes, or if none of its Notes has a time tag between the PartPerformer's first time tag and last time tag values. Otherwise it returns **self**.

See also: – **activate** (Performer)

awake

– **awake**

Prepares a recently unarchived PartPerformer for use by creating and adding a NoteSender.

See also: – **write:**, – **read:**

deactivateSelf

– **deactivateSelf**

Frees the PartPerformer's Note list. You never invoke this method directly; it's invoked as part of the **deactivate** method inherited from Performer.

See also: – **deactivate** (Performer)

firstTimeTag

– (double)**firstTimeTag**

Returns the PartPerformer's first time tag value, as set through **setFirstTimeTag:**.

See also: – **setFirstTimeTag:**, – **setLastTimeTag:**

free

– **free**

This invokes **[super free]** with an additional check: If the PartPerformer was created by a ScorePerformer, it can't be freed through this method. It must be freed through a message to the ScorePerformer.

See also: – **free** (ScorePerformer), – **freePartPerformers** (ScorePerformer)

init

– **init**

Initializes the PartPerformer by creating and adding its single NoteSender. A subclass implementation should send **[super init]** before performing its own initialization. Returns **self**.

lastTimeTag

– (double)**lastTimeTag**

Returns the PartPerformer's last time tag value, as set through **setLastTimeTag:**.

See also: – **setFirstTimeTag:**, – **setLastTimeTag:**

part

– **part**

Returns the PartPerformer's Part object, as set through **setPart:**.

See also: – **setPart:**

perform

– perform

Performs a single Note from the Note list by sending it to the PartPerformer's NoteSender. You never invoke this method directly; it's automatically and sequentially invoked during a performance. If the PartPerformer's Note list is exhausted, or if the next Note in the list has a time tag greater than the PartPerformer's last time tag value, the PartPerformer is deactivated. Keep in mind that a PartPerformer reads one Note ahead: It sends the Note read during the previous invocation of **perform** (or **activateSelf** if this is the first invocation), retrieves the next Note, and schedules another **perform** message based on the Note's time tag. A subclass implementation should send **[super perform]**. The return value is ignored.

read:

– **read:**(NXTypedStream *)*stream*

Unarchives the PartPerformer by reading it from *stream*. You never invoke this method directly; to read an archived PartPerformer, call the **NXReadObject()** C function.

See also: – **write:**, – **awake**

setFirstTimeTag:

– **setFirstTimeTag:**(double)*firstTimeTag*

Sets the least time tag value that the PartPerformer considers for performance. When the PartPerformer is activated, it reads through its Note list, searching for the first Note that has a time tag value greater than or equal to *firstTimeTag*. It begins performing from that Note. Keep in mind that the difference between the Note's time tag value and the value set here is made up as a (seeming) delay before the Note is performed. However, this delay is added into the PartPerformer's notion of the current time (as returned by the **time** method, inherited from Performer). This is distinct from the real delay incurred by setting the object's performance time offset (through Performer's **setTimeShift:** method): The time offset *isn't* added into the PartPerformer's notion of the current time.

If the PartPerformer is in a performance, this does nothing and returns **nil**. Otherwise, it returns **self**.

See also: – **firstTimeTag**, – **setLastTimeTag:**, – **setTimeShift:** (Performer)

setLastTimeTag:

– **setLastTimeTag:**(double)*lastTimeTag*

Sets the greatest time tag value that the PartPerformer considers for performance. When, during its sequence of **perform** invocations, the PartPerformer reads a Note from its Note list that has a time tag value greater than *lastTimeTag*, the PartPerformer is immediately deactivated.

This method performs the same function as the **setDuration:** method inherited from Performer. You shouldn't invoke both.

If the PartPerformer is in a performance, this does nothing and returns **nil**. Otherwise, it returns **self**.

See also: – **lastTimeTag**, – **setFirstTimeTag:**, – **setDuration:** (Performer)

setPart:

– **setPart:***aPart*

Associates the PartPerformer with *aPart*. If the PartPerformer is active, this does nothing and returns **nil**. Otherwise it returns **self**.

See also: – **part**

write:

– **write:**(NXTypedStream *)*stream*

Archives the PartPerformer by writing it to *stream*. You never invoke this method directly; to archive a PartPerformer, call the **NXWriteRootObject()** C function. In addition to the archiving defined by the Performer class, the PartPerformer's first and last time tag values are archived directly and its Part and ScorePerformer (if any) are archived by reference.

See also: – **read:**, – **awake**

PartRecorder

INHERITS FROM	Instrument : Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

A PartRecorder is an Instrument that realizes Notes by adding copies of them to a Part. A PartRecorder's Part is set through the **setPart:** method. If the Part already contains Notes, the old Notes aren't removed or otherwise affected by recording into the Part—the recorded Notes are merged in.

Each PartRecorder contains a single NoteReceiver object. During a performance, a PartPerformer receives Notes from its NoteReceiver, copies them, and then adds them to its Part object. Each Note is given a new (but not necessarily different) timeTag; if the Note is a noteDur, it's also given a new duration. The timeTag and duration are computed either as beats or as seconds, depending on the value of the **timeUnit** instance variable. If **timeUnit** is set to MK_second, the default, the new values are in seconds from the beginning of the performance. If it's set to MK_beat, they're computed as beats.

You can create PartRecorders yourself, or you can use a ScoreRecorder object to create a group of them for you.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Instrument	id	noteReceivers;
Declared in PartRecorder	MKTimeUnit	timeUnit;
	id	noteReceiver;
	id	part;

timeUnit	The unit in which the object measures time.
noteReceiver	The object's single NoteReceiver.
part	The object's Part.

METHOD TYPES

Creating and freeing a PartRecorder	– copy
Modifying the object	– init – setPart:
Querying the object	– part
Realizing Notes	– realizeNote:fromNoteReceiver:
Accessing time	– setTimeUnit: – timeUnit

INSTANCE METHODS

copy

– copy

Creates and returns a new PartRecorder as a copy of the receiver. The new object has its own NoteReceiver object but adds Notes to the same Part as the receiver.

init

– init

Initializes the receiver by creating and adding its single NoteReceiver. You never invoke this method directly. A subclass implementation should send **[super init]** before performing its own initialization. The return value is ignored.

part

– part

Returns the receiver's Part object.

realizeNote:fromNoteReceiver:

– realizeNote:*aNote* **fromNoteReceiver:***aNoteReceiver*

Copies *aNote*, computes and sets the new Note's timeTag (and duration if it's a noteDur), and then adds the new Note to the receiver's Part. *aNoteReceiver* is ignored. Returns the receiver.

setPart:

– **setPart:***aPart*

Sets *aPart* as the receiver's Part. Returns the receiver.

setTimeUnit:

– **setTimeUnit:**(MKTimeUnit)*aTimeUnit*

Sets the receiver's **timeUnit** instance variable to *aTimeUnit,* one of MK_second or MK_beat. The default is MK_second.

timeUnit

– (MKTimeUnit)**timeUnit**

Returns the receiver's **timeUnit**, either MK_second or MK_beat.

PatchTemplate

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A PatchTemplate is a recipe for building a DSP synthesis patch. It contains specifications for the UnitGenerator and SynthData objects that are needed and instructions for connecting these objects together. PatchTemplate objects are created as part of a SynthPatch subclass design. Keep in mind that while each subclass of SynthPatch creates one or more PatchTemplates, you don't need to subclass PatchTemplate itself.

PatchTemplate's **addUnitGenerator:ordered:** and **addSynthData:length:** methods describe the objects that make up the SynthPatch. It's important to keep in mind that these methods don't add actual objects to the PatchTemplate. Instead, they specify the types of objects that will be created when the SynthPatch is constructed by the Orchestra.

A PatchTemplate's UnitGenerators are specified by their class, given as the first argument to the **addUnitGenerator:ordered:** method. The argument should be a UnitGenerator leaf class, not a master class (leaf and master classes are explained in the UnitGenerator class description).

The UnitGenerator is further described as being ordered or unordered, as the argument to the **ordered:** keyword is YES or NO. Ordered UnitGenerators are executed (on the DSP) in the order that they're added to the PatchTemplate; unordered UnitGenerators are executed in an undetermined order. Usually, the order in which UnitGenerators are executed is significant; for example, if the output of UnitGenerator A is read by UnitGenerator B, then A must be executed before B if no delay is to be incurred. As a convenience, the **addUnitGenerator:** method is provided to add UnitGenerators that are automatically declared as ordered. The advantage of unordered UnitGenerators is that their allocation is less constrained.

SynthDatas are specified by a DSP memory segment and a length. The memory segment is given as the first argument to **addSynthData:length:**. This can be either MK_xData, for x data memory, or MK_yData, for y data memory. Which memory segment to specify depends on where the UnitGenerators that access it expect it to be. The argument to the **length:** keyword specifies the size of the SynthData, or how much DSP memory it represents, and is given as DSPDatum (24-bit) words.

A typical use of a SynthData is to create a location called a *patchpoint* that's written to by one UnitGenerator and then read by another. A patchpoint, which is always 16 words long, is ordinarily the only way that two UnitGenerators can communicate. The **addPatchpoint:** method is provided as a convenient way to add SynthDatas that are

used as patchpoints. The argument to this method is either MK_xPatch or MK_yPatch, for x and y patchpoint memory, respectively.

The object-adding methods each return a unique integer that identifies the added UnitGenerator or SynthData.

Once you have added the requisite synthesis elements to a PatchTemplate, you can specify how they are connected. This is done through invocations of the **to:sel:arg:** method. The first argument is an integer that identifies a UnitGenerator (such as returned by **addUnitGenerator:**), the last argument is an integer that identifies a SynthData (or patchpoint). The argument to the **sel:** keyword is a selector that's implemented by the UnitGenerator and that takes a SynthData object as its only argument. Typical selectors are **setInput:** (the UnitGenerator reads from the SynthData) and **setOutput:** (it writes to the SynthData). Notice that you can't connect a UnitGenerator directly to another UnitGenerator.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in PatchTemplate	(none)	

METHOD TYPES

Creating a PatchTemplate object	– copy – init
Adding and connecting synthesis elements	– addPatchpoint: – addSynthData:length: – addUnitGenerator: – addUnitGenerator:ordered: – to:sel:arg:
Querying the object	– synthElementCount

INSTANCE METHODS

addPatchpoint:

– (unsigned)**addPatchpoint:**(MKOrchMemSegment)*segment*

Adds a patchpoint (SynthData) specification to the receiver. *segment* is the DSP memory segment from which the patchpoint is allocated. It can be either MK_xPatch or MK_yPatch. Returns an integer that identifies the patchpoint specification.

addSynthData:length:

– (unsigned)**addSynthData:**(MKOrchMemSegment)*segment* **length:**(unsigned)*len*

Adds a SynthData specification to the receiver. The SynthData has a length of *len* DSPDatum words and is allocated from the DSP segment *segment*, which should be either MK_xData or MK_yData. Returns an integer that identifies the SynthData specification.

addUnitGenerator:

– (unsigned)**addUnitGenerator:***aUGClass*

Adds an ordered UnitGenerator specification to the receiver. Implemented as **[self addUnitGenerator:***aUGClass* **ordered:YES]**. Returns an integer that identifies the UnitGenerator specification.

addUnitGenerator:ordered:

– (unsigned)**addUnitGenerator:***aUGClass* **ordered:**(BOOL)*isOrdered*

Adds a UnitGenerator specification to the receiver. The UnitGenerator is an instance of *aUGClass*, a UnitGenerator leaf class. If *isOrdered* is YES, then the order in which the specification is added (in relation to the receiver's other UnitGenerators) is the order in which the UnitGenerator, once created, is executed on the DSP.

copy

– copy

Creates and returns a PatchTemplate as a copy of the receiver.

synthElementCount

– (unsigned)**synthElementCount**

Returns the number of UnitGenerator and SynthData specifications (including patchpoints) that have been added to the receiver.

to:sel:arg:

– **to:**(unsigned)*anObjInt*
sel:(SEL)*aSelector*
arg:(unsigned)*anArgInt*

Specifies a connection between the UnitGenerator identified by *anObjInt* and the SynthData identified by *anArgInt*. The means of the connection are specified in the method *aSelector*, to which the UnitGenerator must respond. *anObjInt* and *anArgInt* are identifying integers returned by PatchTemplate's add methods. If either of these arguments are invalid identifiers, the method returns **nil**, otherwise it returns the receiver.

Performer

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

The Performer class supplies a mechanism for automatically sending a series of Notes into a Music Kit performance in a timely manner. The 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). 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 in turn 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." Analogously, Instruments contains 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 (an Instrument that synthesizes Notes on the DSP), you would do the following:

```
/* aPartPerformer and aSynthIns are assumed to exist. */
[[aPartPerformer noteSender] connect:[aSynthIns noteReceiver]];
```

The **noteSender** method returns one of a Performer's NoteSenders (the **noteReceiver** method operates analogously for an Instrument with regard to its NoteReceivers). If you're using instances of the Music Kit Performer subclasses, you should refer to their descriptions to determine if they create 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.

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.

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 data base 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 design and use of the **perform** method follows two principles:

- It should acquire one Note at a time.
- It's never invoked directly; instead, it's automatically invoked once for each Note. This is part of the Performer's scheduling mechanism, as described below.

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 only need one.

As part of its **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.

Keep in mind that the Performer class itself doesn't invoke **sendNote:** for you; you must include the invocation as part of your subclass' implementation of the **perform** method.

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.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Performer	id	conductor;
	MKPerformerStatus	status;
	int	performCount;
	double	timeShift;
	double	duration;
	double	time;
	double	nextPerform;
	id	noteSenders;

conductor — The Performer's Conductor.

status — The object's performance status.

performCount	Number of **perform** messages the object has received.
timeShift	Performance time offset in beats.
duration	Maximum number of beats the object will spend in performance.
time	The object's notion of the current time in beats.
nextPerform	Used to schedule invocations of **perform**.
noteSenders	The object's collection of NoteSenders.

METHOD TYPES

Copying and initializing a Performer	– copy – copyFromZone: – init
Freeing a Performer and its contents	– free – freeNoteSenders
Accessing NoteSenders	– addNoteSender: – removeNoteSender: – removeNoteSenders – isNoteSenderPresent: – noteSender – noteSenders
Accessing the Conductor	– setConductor: – conductor
Performance timing	– setTimeShift: – timeShift – setDuration: – duration – time

Performing the object	– perform – performCount – activate – activateSelf – deactivate – deactivateSelf – pause – pauseFor: – resume – status – inPerformance
Accessing the delegate	– setDelegate: – delegate
Archiving the object	– write: – read: – awake

INSTANCE METHODS

activate

– activate

Primes the Performer for the next (or the current) performance. It does this by performing the following duties:

- The **nextPerform** variable is to 0.0 and the performance count is cleared.
- The **activateSelf** method is invoked.
- The first **perform** invocation is scheduled to take place at the time indicated by **nextPerform** (which may have been modified by **activateSelf**).
- The Performer's status is set to MK_active.
- The message **performerDidActivate:** is sent to the Performer's delegate.

Upon completing these rounds, **self** is returned. However, successful activation depends on a number of conditions:

- If the Performer's status isn't MK_inactive (in other words, if it's already in performance, whether or not its currently paused), **self** is returned with no further ado.

- If the Performer's performance duration (as set through **setDuration:**) is less than or equal to 0.0, **nil** is returned and the rest is for naught.

- Should **activateSelf** return **nil**, immediately thereafter so, too, will this.

- If, after **activateSelf** is invoked, the value of **nextPerform** is greater than the Performer's duration, **nil** is returned and the Performer lies doggo.

Note that the **perform** invocation, the status change, and the delegate notification only happen if the method is entirely successful.

See also: – **activateSelf**, – **deactivate**, – **status**, – **inPerformance**

activateSelf

– **activateSelf**

You never invoke this method directly; it's invoked automatically from the **activate** method. A subclass can implement this method to perform pre-performance activities. In particular, if the subclass needs to alter the initial **nextPerform** value, it should be done here. When **activateSelf** is invoked, the Performer's status is MK_inactive, thus you can, for example, set its Conductor in the implementation of this method but you can't, for another example, pause the Performer. If **activateSelf** returns **nil**, the Performer isn't activated; the return value is otherwise insignificant. The default implementation does nothing and returns **self**.

See also: – **activate**

addNoteSender:

– **addNoteSender:***aNoteSender*

Removes *aNoteSender* from its current owner, if any, and adds it to the Performer. Both Performers (the old owner and the receiver of this message) must be inactive. Returns **nil** if either of the Performer is active, or if the NoteSender already belongs to the receiving Performer, otherwise returns **self**.

See also: – **noteSender**, – **noteSenders**, – **removeNoteSender:**, – **inPerformance**

awake

– **awake**

Prepares a recently unarchived Performer for use. The Performer's status is set to MK_inactive.

See also: – **write:**, – **read:**

conductor

– conductor

Returns the Performer's Conductor, as set through **setConductor:**. The Conductor's tempo influences the rate at which a Performer performs its Notes; in addition, you can control the performance activities of a group of Performers by sending messages, such as **pause** and **resume**, to their mutual Conductor. By default, the defaultConductor controls all Performers.

See also: **– setConductor:**

copy

– copy

Creates and returns a new, inactive Performer as a copy of the receiving Performer. The new object copies the time shift and duration values, and it adds to itself copies of the NoteSenders. The new object's **nextPerform** variable is set to 0.0.

copyFromZone:(NXZone)*zone*

– copyFromZone:(NXZone)*zone*

The same as **copy** but the new Performer is allocated in *zone*.

deactivate

– deactivate

Halts the Performer's performance. The Performer is removed from the performance, its **deactivateSelf** method is invoked (and the return value ignored), its status is set to MK_inactive, and the message **performerDidDeactivate:** is sent to the delegate. If the object has scheduled a **resume** message through a previous invocation of **pauseFor:**, that message is cancelled. Keep in mind that while you can send **deactivate** directly to a Performer, the method is also invoked automatically when the Performer's duration expires, if the entire performance is ended (through the Conductor class method **finishPerformance**), or (for certain subclasses) when the Performer runs out of Notes. Returns **self**.

See also: **– deactivateSelf**, **– activate**

deactivateSelf

– deactivateSelf

You never invoke this method directly; it's invoked automatically from the **deactivate** method. A subclass can implement this method to perform post-performance activities. You should note that at the time that this method is invoked, the Performer's status will not yet have been changed to MK_inactive. The default implementation does nothing; the return value is ignored (by the **deactivate** method).

See also: – **deactivate**

delegate

– delegate

Returns the Performer's delegate object, as set through **setDelegate:**. The delegate is sent messages as the Performer is activated, paused, resumed, and deactivated.

See also: – **setDelegate:**

duration

– (double)**duration**

Returns the Performer's performance time limit as set through **setDuration:**.

See also: – **setDuration:**, – **setTimeShift:**, – **time**

free

– free

Frees the Performer and its NoteSenders. The Performer must be inactive; this does nothing and returns **nil** if the Performer is currently in performance (even if it's paused).

See also: – **freeNoteSenders**

freeNoteSenders

– freeNoteSenders

Disconnects and frees the Performer's NoteSenders and returns **self**. The Performer must be inactive; this does nothing and returns **nil** if the Performer is currently in performance (even if it's paused).

See also: – **free**, – **addNoteSender:**, – **removeNoteSender:**, – **removeNoteSenders**

init

– init

Initializes the Performer. This method is often subclassed to create and add some number of NoteSenders to the Performer. Such an implementation should send **[super init]** before performing its own initialization. Returns **self**.

inPerformance

– (BOOL)**inPerformance**

Returns YES if the Performer is considered to be in performance, otherwise returns NO. The determination is based on the Performer's status, where the MK_active and MK_paused states indicate that the Performer is in performance, and MK_inactive means that it isn't.

See also: – **status**

isNoteSenderPresent:

– (BOOL)**isNoteSenderPresent:***aNoteSender*

Returns YES if *aNoteSender* is owned by the Performer, otherwise returns NO.

See also: – **addNoteSender:**, – **removeNoteSender:**, – **noteSenders**

noteSender

– noteSender

Returns one of the Performer's NoteSenders. This is convenient if the Performer owns but a single NoteSender, or if you don't care which NoteSender you retrieve. Given a stable set of NoteSenders, this method will always return the same one.

See also: – **noteSenders**, – **addNoteSender:**, – **removeNoteSender:**

noteSenders

– noteSenders

Creates and returns a List that contains the Performer's NoteSenders. It's the sender's responsibility to free this List.

See also: – **noteSender**, – **addNoteSender:**, – **removeNoteSender:**

pause

– pause

Suspends the Performer's performance for an indeterminate amount of time. To unpause a Performer, send it the **resume** message. While it's paused, the Performer is unable to receive **perform** messages; the **perform** message that's impending at the time that **pause** is received is rescheduled when the Performer is resumed. This method also sets the Performer's status to MK_paused and the **performerDidPause:** message is sent to its delegate. The time a Performer spends paused isn't deducted from its duration time limit. If the Performer is already paused, or if it isn't in a performance (in other words, if its status isn't MK_active), none of the above obtains. Returns **self**.

See also: – **pauseFor:**, – **resume**, – **setTimeShift:**

pauseFor:

– pauseFor:(double)*beats*

Suspends the Performer's performance for *beats* beats. A **resume** message is automatically sent to the Performer at the appointed time to rouse it from its slumber, or you can unpause the Performer ahead of schedule by sending it **resume** directly (the scheduled **resume** message won't be sent). While it's paused, the Performer is unable to receive **perform** messages; the **perform** message that's impending at the time that **pauseFor:** is received is rescheduled when the Performer is resumed. This method also sets the Performer's status to MK_paused and the **performerDidPause:** message is sent to its delegate.

You can send **pauseFor:** to a paused Performer and thereby reschedule the previously requested **resume** message. The delegate message is suppressed in this case.

If *beats* is less than or equal to 0.0, the Performer isn't paused and **nil** is returned. Similarly, if the Performer isn't in a performance it won't be paused, although in this case and in all others, **self** is returned.

See also: – **pause**, – **resume**, – **setTimeShift:**

perform

– perform

This is the soul of a Performer; its design embodies the character of the particular subclass that implements it. The Performer class itself declares this method as a subclass responsibility. The guidelines for implementing this method take up the greater share of the class description, above, and can be summarized as they fall into three basic tasks:

- A Note is acquired.
- The NoteSender method **sendNote:** is invoked with this Note as an argument.
- The **nextPerform** instance variable is set.

You never invoke **perform** directly; it's invoked sequentially and automatically during a performance. Note that the **perform**-message count (as retrieved through **performCount**) is incremented *before* **perform** is delivered to the Performer. Thus, an invocation of **performCount** from within an implementation of **perform** will always yield a value of 1 or greater. It's acceptable to send **pause**, **pauseFor:**, or **deactivate** to **self** as part of the implementation of **perform**. The value returned by **perform** is ignored.

See also: – **activateSelf**, – **performCount**

performCount

– (int)**performCount**

Returns the number of **perform** messages the Performer has received in the current performance, or how many it received in its previous performance if the Performer is currently inactive.

See also: – **perform**, – **performCount**

read:

– **read:**(NXTypedStream *)*stream*

Unarchives the Performer by reading it from *stream*. You never invoke this method directly; to read an archived Performer, call the **NXReadObject()** C function.

See also: – **write:**, – **awake**

removeNoteSender:

– **removeNoteSender:***aNoteSender*

Removes *aNoteSender* from the Performer. If the Performer is currently in performance, or if it doesn't own *aNoteSender*, the object isn't removed and **nil** is returned. Otherwise returns **self**.

See also: – **addNoteSender:**, – **removeNoteSenders**, – **isNoteSenderPresent:**

removeNoteSenders

– **removeNoteSenders**

Removes the Performer's NoteSenders. If the Performer is in a performance, the objects aren't removed. Returns **self**.

See also: – **addNoteSender:**, – **removeNoteSender:**

resume

– **resume**

Resumes a paused Performer. The Performer's status is set to MK_active and the **performerDidResume:** message is sent to its delegate. The **perform** message that was impending when the Performer was paused (whether through **pause** or **pauseFor:**) is rescheduled such that, to the Performer, time will have seemed to stand still while it was paused. If the Performer isn't paused, this does nothing. Returns **self**.

See also: – **pause**, – **pauseFor:**

setConductor:

– **setConductor:***aConductor*

Sets the Performer's Conductor. If *aConductor* is **nil** the Performer is associated with the defaultConductor. The Conductor's tempo influences the rate at which a Performer performs its Notes; in addition, you can control the performance activities of a group of Performers by sending messages, such as **pause** and **resume**, to their mutual Conductor. By default, the defaultConductor controls all Performers.

See also: – **conductor**

setDelegate:

– **setDelegate:***delegate*

Sets the Performer's delegate and returns **self**. Delegate messages are sent when you invoke the following Performer methods:

Method	Delegate Message
activate	performerDidActivate:
pause, pauseFor:	performerDidPause:
resume	performerDidResume:
deactivate	performerDidDeactivate:

As usual, a delegate message is sent only if the object responds to it.

See also: – **delegate**

setDuration:

– **setDuration:**(double)*dur*

Sets the Performer's duration to *dur* in beats. This value is the maximum amount of time, in beats, that the Performer is allowed to perform. The Performer is deactivated when, as part of its appointed duties in the **perform** method, it sets the **nextPerform** variable such that the next invocation of **perform** would fall outside the duration time limit. For example, if you set a Performer's duration to 10.0 and the Performer receives

the **perform** message five beats into the performance during which it sets **nextPerform** to 6.0, the Performer will be deactivated as soon as that invocation of **perform** finishes (even though it has five beats left in its duration).

The stopwatch on a Performer's duration time limit starts ticking when the Performer actually starts performing—in other words, after its time shift, if any, has expired. Time spent paused also doesn't count.

The Performer must be inactive. Returns **nil** if the Performer is currently in performance (and doesn't set the duration), otherwise returns **self**.

See also: – **duration**, – **setTimeShift:**, – **timeShift**

setTimeShift:

– **setTimeShift:**(double)*timeShift*

Imposes an initial delay of *timeShift* beats on the Performer's performance. The Performer must be inactive. If you pause a Performer before its time shift expires—in other words, if you send **pause** or **pauseFor:** within *timeShift* beats of sending **startPerformance** to the Conductor class—the balance of the time shift will be applied after the Performer is resumed. Returns **nil** if the Performer is currently in performance, otherwise returns **self**.

See also: – **timeShift**, – **setDuration:**, – **duration**

status

– (MKPerformerStatus)**status**

Returns the Performer's current performance status as one of the following integer constants:

Constant	Meaning
MK_active	In a performance (or activated in anticipation of a performance)
MK_paused	In a performance but currently paused
MK_inactive	Not in a performance

You can't set a Performer's status directly; it's set as the Performer is created (MK_inactive), activated (MK_active), paused (MK_paused), resumed (MK_active), and deactivated (MK_inactive).

See also: – **inPerformance**

time

– (double)**time**

Returns the amount of time the Performer has spent actually performing, in beats. This value doesn't include the Performer's time shift, nor any time it has spent paused. In addition, a Performer's time is updated only when it receives a **perform** message. If the Performer is inactive, this returns MK_ENDOFTIME.

timeShift

– (double)**timeShift**

Returns the Performer's time shift value, as set through **setTimeShift:**.

See also: – **setTimeShift:**

write:

– **write:**(NXTypedStream *)*stream*

Archives the Performer by writing it to *stream*. You never invoke this method directly; to archive a Performer, call the **NXWriteRootObject**() C function. The Performer's NoteSender List (but *not* the NoteSenders themselves), duration, and time shift are archived directly. Its Conductor and delegate are archived by reference.

See also: – **read:**, – **awake**

METHODS IMPLEMENTED BY THE DELEGATE

performerDidActivate:

– **performerDidActivate:***sender*

Sent to the delegate when *sender* is activated.

performerDidPause:

– **performerDidPause:***sender*

Sent to the delegate when *sender* is paused.

performerDidResume:

– **performerDidResume:***sender*

Sent to the delegate when *sender* is resumed.

performerDidDeactivate:

– **performerDidDeactivate:***sender*

Sent to the delegate when *sender* is deactivated.

Samples

INHERITS FROM	WaveTable : Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

A Samples object represents one complete cycle of a sound waveform as a series of samples. The data for a Samples object is established through association with a Sound object, defined by the Sound Kit. Two methods are provided to create this association:

- **setSound:** takes a Sound object as an argument, copies it, and associates the receiver with the copied Sound.
- **readSoundfile:** takes the name of a soundfile, creates a Sound object for the data contained therein, and associates the receiver with the newly created Sound.

The Sound object or soundfile must be one channel of 16-bit linear data (SND_FORMAT_LINEAR_16). The sampling rate is ignored; Samples objects are designed to be used as lookup tables for oscillator UnitGenerators in which use the sampling rate of the original data is of no consequence.

You can create a Samples object from a scorefile by quoting the name of a soundfile within curly brackets which are themselves enclosed by square brackets. The object can be given a name in a waveTable statement:

```
waveTable mySamples = [ {"samplesFile.snd" }];
```

A Samples object that's written to a scorefile is referred to by the name of the soundfile from which it was created. If a Sound object is used, a soundfile is created and the object is written to it, as explained in the method **writeScorefileStream:**. You should always name your Samples objects by calling the **MKNameObject()** C function.

Samples' sister class, Partials, lets you define a waveform by its sine wave components.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from WaveTable	int	length;
	double	scaling;
	DSPDatum	*dataDSP;
	double	*dataDouble;
Declared in Samples	id	sound;
	char	*soundfile;

sound	The object's Sound object.
soundfile	The name of the soundfile, if the Sound was set through **readSoundfile:**.

METHOD TYPES

Creating and freeing a Samples	– copy – free
Modifying the object	– fillTableLength:scale: – init – readSoundfile: – setSound:
Querying the object	– sound – soundfile
Writing the object	– writeScorefileStream:

INSTANCE METHODS

copy

– **copy**

Creates and returns a new Samples object as a copy of the receiver. The receiver's Sound is copied into the new object.

fillTableLength:scale:

– **fillTableLength:**(int)*aLength* **scale:**(double)*aScaling*

Copies *aLength* samples from the receiver's Sound into the **dataDSP** array (inherited from WaveTable) and scales the copied data by multiplying it by *aScaling*. If *aScaling* is 0.0, the data is scaled to fit perfectly within the range -1.0 to 1.0.

The **dataDouble** array (also from WaveTable) is reset. You ordinarily don't invoke this method; it's invoked from methods defined in WaveTable. Returns **self** or **nil** if there's a problem.

free

– **free**

Frees the receiver and its Sound.

init

– **init**

Sent automatically when the receiver is created, you can also invoke this method to reset a Samples object. It sets the receiver's **sound** variable to **nil** and **soundfile** to NULL. The receiver's previous Sound object, if any, is freed. A subclass implementation should send **[super init]**. Returns the receiver.

readSoundfile:

– **readSoundfile:**(char *)*aSoundfile*

Creates a new Sound object, reads the data from *aSoundfile* into the object, and then sends **setSound:** to the receiver with the new Sound as the argument. You shouldn't free the Sound yourself; it's automatically freed when the receiver is freed, initialized, or when a subsequent Sound is set. Returns **nil** if the **setSound:** message returns **nil**; otherwise returns the receiver.

setSound:

– **setSound:***aSound*

Sets the receiver's Sound to a copy of *aSound* (the receiver's current Sound is freed). *aSound* must be one-channel, 16-bit linear data. You shouldn't free the Sound yourself; it's automatically freed when the receiver is freed, initialized, or when a subsequent Sound is set. Returns **nil** if *aSound* is in the wrong format, otherwise returns the receiver.

sound

– **sound**

Returns the receiver's Sound object.

soundfile

– (char *)**soundfile**

Returns the name of the receiver's soundfile, or NULL if the receiver's Sound wasn't set through **readSoundfile:**. The name isn't copied; you shouldn't alter the returned string.

writeScorefileStream:

– **writeScorefileStream:**(NXStream *)*aStream*

Writes the receiver in scorefile format to the stream *aStream*. If the Sound wasn't set from a soundfile, a soundfile with the unique name "samples*Number*.snd" (where *Number* is added only if needed) is created and the Sound is written to it. The object remembers if its Sound has been written to a soundfile. If the receiver couldn't be written to the stream, returns **nil**, otherwise returns the receiver.

Score

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A Score is a collection of Part objects. Scores can be read from and written to a scorefile or Standard MIDI File, performed with a ScorePerformer, and can be used to record Notes from a ScoreRecorder.

Each Score has an info Note (a mute) that defines, in its parameters, information that can be useful in performing or otherwise interpreting the Score. Typical information includes tempo, DSP headroom (see the Orchestra Class), and sampling rate (the parameters MK_tempo, MK_headroom, and MK_samplingRate are provided to accommodate this utility).

When you read a scorefile into a Score, a Part object is created and added to the Score for each Part name in the file's **part** statement. ScoreFile **print** statements are printed as the scorefile is read. You can set the stream on which these messages are printed by invoking **setScorefilePrintStream:**.

Reading and Writing Standard MIDI Files

Reading and writing a Standard MIDI File is similar to receiving and sending MIDI messages through a Midi object—in particular, Notes are translated into MIDI data (and vice versa) according to the rules outlined in the Midi class description. In addition, the way a Score object reads MIDI File data depends on the file's format (familiarity with the Standard MIDI File specification is assumed in the following):

- When reading a format 0 MIDI File, a Part is created for each of the 16 MIDI Channels. Each of these Parts is given an info Note that contains an MK_midiChan parameter that specifies the channel number. An additional Part is created to store the channel-less MIDI messages in the file (this is the first Part in the Score).

- When reading a format 1 MIDI File, a Part is created for each track in the file; each of these Parts is given an info Note that contains an MK_track parameter and an MK_midiChan parameter. The value of a Part's MK_track parameter—if not specified in the file—is a monotonically increasing integer starting with track 1. The MK_midiChan value is the channel of the first Note for that track. An additional Part that corresponds to the tempo map track is added to the score (this is the first Part in the Score).

- When reading a format 2 MIDI File, a Part is created for each sequence in the file; each of these Parts is given an info Note that contains an MK_sequence parameter and an MK_midiChan parameter. The value of a MK_sequence parameter is similar to the format 1 MK_track parameter, except that the monotonically

increasing integer series starts with (sequence) 0. The value of MK_midiChan is the same as in format 1. No additional Part is added.

You can determine the format of a MIDI file that's read into a Score object by the presence or absence of the MK_track and MK_sequence parameters in the info Notes of the appropriate Parts.

For each Standard MIDI File meta-event, a parameter is created and added to a particular Note in the Score (except for the Track Name event, as explained in the following table). Such a Note is either the Score's info Note, a Part info Note, or a Note that's created whole cloth to accommodate the parameter, depending on the meta-event and the MIDI File format. The following table explains:

Meta-event	**Parameter**	**Note**
Sequence Number	MK_sequence	Formats 0 and 1: Score info Note Format 2: Info Note of the corresponding Part
Text Event	MK_text	Format 0: New Note added to the first Part Formats 1 and 2: New Note added to the corresponding Part
Lyric	MK_lyric	See Text Event
Cue Point	MK_cuePoint	See Text Event
Copyright Notice	MK_copyright	Score info Note
Sequence Name	MK_title	Formats 0 and 1: Score info Note Format 2: Interpreted as a Track Name event.
Track Name	(none)	Format 0: Interpreted as a Sequence Name event Formats 1 and 2: Corresponding Part is named through **MKNameObject()**
Marker	MK_marker	Formats 0 and 1: New Note added to the first Part Format 2: New Note added to the corresponding Part
Time Signature	MK_timeSignature	See Marker
Key Signature	MK_keySignature	See Marker
Set Tempo	MK_tempo	See Marker
End of Track	MK_track	Formats 0 and 1: New Note added to the corresponding Part (but see below)
	MK_sequence	Format 2: New Note added to the corresponding Part (but see below)
SMPTE Offset	MK_smpteOffset	Formats 0 and 1: Score info Note Format 2: Info Note of the corresponding Part

The End of Track meta-event is translated only if it's preceded by a time delay.

When writing a Score as a Standard MIDI File, the Music Kit writes the file in level 1 format, following these conventions:

- The Parts are written as separate tracks, in the order they appear in the Score (the track number encoded in the Part's info Note is ignored).
- If the Score info Note has a MK_title, MK_tempo, MK_copyright, MK_sequence, or MK_smpteOffset parameter, the corresponding meta-event (as given in the table, above) is written to the beginning of the file.
- The name of each Part, as determined by **MKGetObjectName**(), is used as the value of a Track Name meta-event .
- The time tag of the last Note in each Part is used as the value of an End of Track meta-event.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in Score	id	parts;
	NXStream	*scorefilePrintStream;
	id	info;

parts	The object's List of Parts.
scorefilePrintStream	The stream used by scorefile **print** statements.
info	The object's info Note.

METHOD TYPES

Creating and freeing a Score	– copy – copyFromZone: – init – free – freeNotes – freeSelfOnly
Modifying the object	– addPart: – empty – freeParts – freePartsOnly – removePart: – setInfo: – setScorefilePrintStream: – shiftTime:

Querying the object	– info – isPartPresent: – midiPart: – noteCount – partCount – parts – scorefilePrintStream
Reading and writing files	– readScorefile: – readScorefileStream: – readMidifile: – readMidifileStream: – readScorefile:firstTimeTag:lastTimeTag: timeShift: – readScorefileStream:firstTimeTag: lastTimeTag:timeShift: – readMidifile:firstTimeTag:lastTimeTag: timeShift: – readMidifileStream:firstTimeTag: lastTimeTag:timeShift: – writeScorefile: – writeScorefileStream: – writeOptimizedScorefile: – writeOptimizedScorefileStream: – writeMidifile: – writeMidifileStream: – writeScorefile:firstTimeTag:lastTimeTag: timeShift: – writeScorefileStream:firstTimeTag: lastTimeTag:timeShift: – writeOptimizedScorefile:firstTimeTag: lastTimeTag:timeShift: – writeOptimizedScorefileStream:firstTimeTag: lastTimeTag:timeShift: – writeMidifile:firstTimeTag:lastTimeTag: timeShift: – writeMidifileStream:firstTimeTag: lastTimeTag:timeShift:

INSTANCE METHODS

addPart:

– **addPart:***aPart*

Adds *aPart* to the Score. The Part is first removed from the Score that it's presently a member of, if any. Returns *aPart*, or **nil** if it's already a member of the Score.

copy

– **copy**

Creates and returns a new Score as a copy of the receiving Score. The Score's Part, Notes, and info Note are all copied.

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Same as **copy** but uses the specified zone.

empty

– **empty**

Removes the Score's Parts but doesn't free them. Returns **self**.

free

– **free**

Frees the Score and its contents.

freeNotes

– **freeNotes**

Removes and frees the Score's Notes (including the info Note). Returns **self**.

freeParts

– **freeParts**

Removes and frees the Score's Parts and the Notes contained therein. Doesn't free the Score's info Note. Parts that are currently being performed by a PartPerformer aren't freed. Returns **self**.

freePartsOnly

– **freePartsOnly**

Removes and frees the Score's Parts but doesn't free the Notes contained therein. Parts that are currently being performed by a PartPerformer aren't freed. Returns **self**.

freeSelfOnly

– **freeSelfOnly**

Frees the Score but not its Parts nor their Notes. The info Note isn't freed. Returns **self**.

info

– **info**

Returns the Score's info Note.

init

– **init**

Initializes a new Score.

isPartPresent:

– (BOOL)**isPartPresent:***aPart*

Returns YES if *aPart* has been added to the Score, otherwise returns NO.

midiPart:

– **midiPart:**(int)*aChan*

Returns the first Part object in the Score that has an info Note specifying *aChan* as its MK_midiChan parameter value.

noteCount

– (unsigned)**noteCount**

Returns the number of Notes in the Score (not counting the info Note).

partCount

– (unsigned)**partCount**

Returns the number of Parts contained in the Score.

parts

– **parts**

Creates and returns a List containing the Score's Parts. The Parts themselves aren't copied. It is the sender's responsibility to free the List.

readMidifile:

– **readMidifile:**(char *)*fileName*

Reads the Standard MIDI File *fileName* into the Score, creating a Part for each MIDI Channel represented in the file and a Note for each MIDI message. Returns **self**, or **nil** if the file couldn't be read.

readMidifile:firstTimeTag:lastTimeTag:timeShift:

– **readMidifile:**(char *)*aFileName*
firstTimeTag:(double)*firstTimeTag*
lastTimeTag:(double)*lastTimeTag*
timeShift:(double)*timeShift*

The same as **readMidifile:**, but only those Notes with time tags within the given boundaries (inclusive) are retained. *timeShift* is added to each Note's time tag.

readMidifileStream:

– **readMidifileStream:**(NXStream *)*aStream*

Reads the Standard MIDI File on *aStream*, converting the messages therein into Note objects. Returns **self**, or **nil** if the data couldn't be read.

readMidifileStream:firstTimeTag:lastTimeTag:timeShift:

– **readMidifileStream:**(NXStream *)*aStream*
firstTimeTag:(double)*firstTimeTag*
lastTimeTag:(double)*lastTimeTag*
timeShift:(double)*timeShift*

The same as **readMidifileStream:**, but only those Notes with time tags within the given boundaries (inclusive) are retained. *timeShift* is added to each Note's time tag.

readScorefile:

– **readScorefile:**(char *)*fileName*

Reads the named scorefile (regular or optimized) and merges its contents with the Score. Returns **self**, or **nil** if the file couldn't be read.

readScorefile:firstTimeTag:lastTimeTag:timeShift:

– **readScorefile:**(char *)*fileName*
firstTimeTag:(double)*firstTimeTag*
lastTimeTag:(double)*lastTimeTag*
timeShift:(double)*timeShift*

The same as **readScorefile:**, but only those Notes with time tags within the given boundaries (inclusive) are retained. *timeShift* is added to each Note's time tag.

readScorefileStream:

– **readScorefileStream:**(NXStream *)*stream*

Reads the scorefile (regular or optimized) pointed to by *stream* into the Score. The stream must be open for reading; the sender is responsible for closing the stream. Returns **self**, or **nil** if the stream couldn't be read.

readScorefileStream:firstTimeTag:lastTimeTag:timeShift:

– **readScorefileStream:**(NXStream *)*stream*
firstTimeTag:(double)*firstTimeTag*
lastTimeTag:(double)*lastTimeTag*
timeShift:(double)*timeShift*

The same as **readScorefileStream:**, but only those Notes with time tags within the given boundaries (inclusive) are retained. *timeShift* is added to each Note's time tag.

removePart:

– **removePart:***aPart*

Removes *aPart* from the Score. Returns *aPart*, or **nil** if it isn't a member of the Score.

scorefilePrintStream

– (NXStream *)**scorefilePrintStream**

Returns the Score's ScoreFile **print** statement stream.

setInfo:

– **setInfo:***aNote*

Sets the Score's info Note to a copy of *aNote*. The Score's previous info Note is removed and freed.

setScorefilePrintStream:

– **setScorefilePrintStream:**(NXStream *)*aStream*

Sets the stream used by ScoreFile **print** statements to *aStream*. Returns **self**.

shiftTime:

– **shiftTime:**(double)*shift*

Shifts the time tags of all Score's Notes by *shift* beats. Returns **self**.

writeMidifile:

– **writeMidifile:**(char *)*aFileName*

Writes the Score's Notes as a level 1 Standard MIDI File named *aFileName*. Returns **self**, or **nil** if the stream couldn't be written.

writeMidifile:firstTimeTag:lastTimeTag:timeShift:

– **writeMidifile:**(char *)*aFileName*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

Writes the Score's Notes, within the given time tag range and with the given *timeShift*, as a level 1 Standard MIDI File named *aFileName*. Returns **self**, or **nil** if the file couldn't be written.

writeMidifileStream:

– **writeMidifileStream:**(NXStream *)*aStream*

Writes the Score, as level 1 Standard MIDI File data, to *aStream*. Returns **self**, or **nil** if the stream couldn't be written.

writeMidifileStream:firstTimeTag:lastTimeTag:timeShift:

– **writeMidifileStream:**(NXStream *)*aStream*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

Write the Score, as level 1 Standard MIDI File data, to *aStream*. Only the Notes within the given time tag boundaries (inclusive) are written. The time tag are offset by the value provided in *timeShift*. Returns **self**, or **nil** if the stream couldn't be written.

writeScorefile:

– **writeScorefile:**(char *)*aFileName*

Writes the Score as a regular scorefile named *aFileName*. Returns **self**, or **nil** if the file couldn't be written.

writeScorefile:firstTimeTag:lastTimeTag:timeShift:

– **writeScorefile:**(char *)*aFileName*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

The same as **writeScorefile:**, but only those Notes with time tags in the specified range (inclusive) are written to the file. The written Notes' time tags are shifted by *timeShift* beats. Returns **self**, or **nil** if the file couldn't be written.

writeScorefileStream:

– **writeScorefileStream:**(NXStream *)*aStream*

Writes the Score as a regular scorefile to the stream pointed to by *aStream*. The stream must be open for writing; the sender is responsible for closing the stream. Returns **self**, or **nil** if the data couldn't be written.

writeScorefileStream:firstTimeTag:lastTimeTag:timeShift:

– **writeScorefileStream:**(NXStream *)*aStream*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

The same as **writeScorefileStream:**, but only those Notes with time tags in the specified range (inclusive) are written to the file. The written Notes' time tags are shifted by *timeShift* beats. Returns **self**, or **nil** if the data couldn't be written.

writeOptimizedScorefile:

– **writeOptimizedScorefile:**(char *)*aFileName*

Same as **writeScorefile:**, but writes the file in optimized format.

writeOptimizedScorefile:firstTimeTag:lastTimeTag:timeShift:

– **writeOptimizedScorefile:**(char *)*aFileName*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

Same as **writeScorefile:firstTimeTag:lastTimeTag:timeShift:**, but writes the file in optimized format.

writeOptimizedScorefileStream:

– **writeOptimizedScorefileStream:**(NXStream *)*aStream*

Same as **writeScorefileStream:**, but writes the data in optimized format.

writeOptimizedScorefileStream:firstTimeTag:timeShift:

– **writeOptimizedScorefileStream:**(NXStream *)*aStream*
firstTimeTag:(double)*firsttime tag*
lastTimeTag:(double)*lasttime tag*
timeShift:(double)*timeShift*

Same as **writeScorefileStream:firstTimeTag:lastTimeTag:timeShift:**, but writes the data in optimized format.

ScorefilePerformer

INHERITS FROM FilePerformer : Performer : Object

DECLARED IN musickit.h

CLASS DESCRIPTION

ScorefilePerformers are used to perform scorefiles. You can set a ScorefilePerformer's file either by name or as a stream, as explained in the FilePerformer class. If you set the file by name (through the **setFile:** method), the ScorefilePerformer searches for the following, in order:

1. The file as given literally by name as the argument to **setFile:**.
2. An optimized scorefile created by appending ".playscore" to the given name.
3. An editable (regular) scorefile created by appending ".score" to the given name.

When a ScorefilePerformer is activated, it reads its file's header and creates and names a NoteSender for each unique member of the **part** statement(s). The ScorefilePerformer also fashions some number of info Notes from the file: It creates a Score info Note that correponds to the file's **info** statement, and creates a Part info Note for each Part info statement. The former is retrieved through the **info** method; the latter are retrieved as they correspond to the object's NoteSenders, through **infoForNoteSender:**.

During a performance, a ScorefilePerformer reads successive Note and time statements from which it creates Note objects that it sends through its NoteSenders. As a Note belongs to a particular Part (as designated in the scorefile), so is the Note sent through the NoteSender that was created for that Part. When it reaches the end of the file, the ScorefilePerformer deactivates itself.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Performer	id	conductor;
	MKPerformerStatus	status;
	int	performCount;
	double	timeShift;
	double	duration;
	double	time;
	double	nextPerform;
	id	noteSenders;

Inherited from FilePerformer	char	*filename;
	double	fileTime;
	NXStream	*stream;
	double	firstTimeTag;
	double	lastTimeTag;
Declared in ScorefilePerformer	NXStream	*scorefilePrintStream;
	id	info;

scorefilePrintStream — The stream used for the scorefile's **print** statements.

info — Score info Note for the file.

METHOD TYPES

Creating a ScorefilePerformer	– copyFromZone: – init – free
Defining the class	+ fileExtensions
Retrieving the info Notes	– info – infoForNoteSender:
Manipulating the print stream	– setScorefilePrintStream: – scorefilePrintStream
Performing the object	– initializeFile – nextNote – performNote: – finishFile
Archiving the object	– read: – write:

CLASS METHODS

fileExtensions

+ (char **)**fileExtension**

You never need to invoke this method; it defines ScorefilePerformer's file name extension list.

copyFromZone

– **copyFromZone:**(NXZone *)*zone*

Creates and returns a ScorefilePerformer as a copy of the receiving ScorefilePerformer. The new object contains a copy of the receiving object's info Note.

finishFile

– **finishFile**

Defines ScorefilePerformer's post-performance operations. You never invoke this method directly; it's invoked automatically at the end of a performance.

free

– **free**

Frees the ScorefilePerformer, its NoteSenders, and its info Note. If the ScorefilePerformer is active, this does nothing and returns **self**. Otherwise, returns **nil**

info

– **info**

Returns the ScorefilePerformer's info Note, fashioned from the **info** statement in the header of the scorefile.

infoForNoteSender:

– **infoForNoteSender:***aNoteSender*

Returns the info Note associated with the given NoteSender, fashioned from a Part info statement in the ScorefilePerformer's scorefile. If *aNoteSender* isn't a contained in the ScorefilePerformer, this returns **nil**.

init

– **init**

Initializes the ScorefilePerformer. A subclass implementation should send **[super init]** before performing its own initialization.

initializeFile

– **initializeFile**

Defines ScorefilePerformer's pre-performance operations. You never invoke this method directly; it's invoked automatically at the beginning of a performance.

nextNote

– **nextNote**

Defines ScorefilePerformer's file-reading operations. You never invoke this method; it's invoked automatically during a performance.

performNote:

– **performNote:***aNote*

Defines ScorefilePerformer's in-performance operations. You never invoke this method; it's invoked automatically during a performance.

read:

– **read:**(NXTypedStream *)*stream*

You never invoke this method directly; to read an archived ScorefilePerformer, call the **NXReadObject()** C function.

setScorefilePrintStream:

– **setScorefilePrintStream:**(NXStream *)*aStream*

Sets the stream to which scorefile **print** statements are printed as the ScorefilePerformer reads its file. If you don't specify a stream, one to standard error is opened and used.

scorefilePrintStream

– (NXStream *)**scorefilePrintStream**

Sets the stream to which scorefile **print** statements, as set through **setScorefilePrintStream:**. If you don't specify a stream, one to standard error is opened and used.

write:

– **write:**(NXTypedStream *)*stream*

You never invoke this method directly; to archive a ScorefilePerformer, you call the **NXWriteRootObject()** C function.

ScorefileWriter

INHERITS FROM	FileWriter : Instrument : Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

A ScorefileWriter is an Instrument that realizes Notes by writing them to a scorefile. The name of the scorefile to which the Notes are written is set through methods inherited from FileWriter.

Each of a ScorefileWriter's NoteReceivers corresponds to a Part that will appear in the scorefile. Unlike most Instruments, the ScorefileWriter class doesn't add any NoteReceivers to a newly created object, they must be added by invoking the **addNoteReceiver:** method.

The names of the Parts represented in the scorefile are taken from the NoteReceivers for which they were created. You can name a NoteReceiver by calling the **MKNameObject**() function.

The header of the scorefile always includes a **part** statement that names the Parts represented in the Score, and a **tagRange** statement that states the range of noteTag values used in the Note statements. A ScorefileWriter can be given an info Note that's written as a Score info statement in the file; similarly, the ScorefileWriter's NoteReceivers can each contain a Part info Note. These, too, are written to the scorefile, each in a separate Part info statement.

You shouldn't change the name of a data object (such as an Envelope, WaveTable, or NoteReceiver) during a performance involving a ScorefileWriter.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Instrument	id	noteReceivers;
Inherited from FileWriter	MKTimeUnit	timeUnit;
	char	*filename;
	NXStream	*stream;
	double	timeShift;
Declared in ScorefileWriter	id	info;

info — The info Note to be written to the file.

METHOD TYPES

Creating a ScorefileWriter	– copy – init
Modifying the object	– setInfo: – setInfo:forNoteReceiver:
Querying the object	+ fileExtension – info – infoForNoteReceiver:
Performing	– finishFile – initializeFile – realizeNote:fromNoteReceiver:

CLASS METHODS

fileExtension

+ (char *)**fileExtension**

Returns "score", the default file extension for scorefiles. The string isn't copied.

INSTANCE METHODS

copy

– copy

Creates and returns a new ScorefileWriter as a copy of the receiver. The new object copies the receivers NoteReceivers and info Notes.

finishFile

– finishFile

You never invoke this method; it's invoked automatically at the end of a performance.

info

– info

Returns the receiver's info Note, as set through **setInfo:**.

infoForNoteReceiver:

– **infoForNoteReceiver:***aNoteReceiver*

Returns the info Note that's associated with a *NoteReceiver* (as set through **setInfo:forNoteReceiver:**).

init

– **init**

Initializes the receiver. You never invoke this method directly. A subclass implementation should send **[super init]** before performing its own initialization. The return value is ignored.

initializeFile

– **initializeFile**

Initializes the scorefile. You never invoke this method; it's invoked automatically just before the receiver writes its first Note to the scorefile.

realizeNote:fromNoteReceiver:

– **realizeNote:***aNote* **fromNoteReceiver:***aNoteReceiver*

Realizes *aNote* by writing it to the scorefile. The Note statement created from *aNote* is assigned to the Part that corresponds to *aNoteReceiver*.

setInfo:

– **setInfo:***aNote*

Sets the receiver's info Note, freeing a previously set info Note, if any. The Note is written, in the scorefile, as a Score info statement. Returns the receiver.

setInfo:forNoteReceiver:

– **setInfo:***aPartInfo* **forNoteReceiver:***aNoteReceiver*

Sets *aPartInfo* as the Note that's written as the info Note for the Part that corresponds to the NoteReceiver *aNoteReceiver*. The Part's previously set info Note, if any, is freed. If the receiver is in performance, or if *aNoteReceiver* doesn't belong to the receiver, does nothing and returns **nil**, otherwise returns the receiver.

ScorePerformer

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A ScorePerformer performs a Score object by creating a group of PartPerformers, one for each Part in the Score, and controlling the group's performance. A ScorePerformer's Score is set and its PartPerformers are created when it receives the **setScore:** message. If you add Parts to or remove Parts from the Score after sending the **setScore:** message, the changes will not be seen by the ScorePerformer.

The ScorePerformer class doesn't inherit from Performer but many of its methods, such as **activate**, **pause**, and **resume**, emulate Performer methods. When a ScorePerformer receives such a message, it forwards the message to each of its PartPerformer objects, which are true Performers.

A ScorePerformer also has a Performer-like status: It can be active, inactive, or paused. The status of a ScorePerformer is, in general, the same as the status of all its PartPerformers. For instance, when you send the **activate** message to a ScorePerformer, its status becomes MK_active as does the status of all its PartPerformers. However, you can access and control a PartPerformer independent of the ScorePerformer that created it. Thus, an individual PartPerformer's status can be different from that of the ScorePerformer.

By default, the PartPerformers that a ScorePerformer creates are direct instances of the PartPerformer class. You can specify a different class through the **setPartPerformerClass:** method; however, the class you set must inherit from PartPerformer.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in ScorePerformer	MKPerformerStatus	status;
	id	partPerformers;
	id	score;
	double	firstTimeTag;
	double	lastTimeTag;
	double	timeShift;
	double	duration;
	id	conductor;
	id	delegate;
	id	partPerformerClass;

status	The ScorePerformer's status.
partPerformers	The object's PartPerformer instances.
score	The Score with which the object is associated.
firstTimeTag	Smallest time tag value considered for performance.
lastTimeTag	Greatest time tag value considered for performance.
timeShift	Performance time offset in beats.
duration	Maximum performance duration in beats.
conductor	The object's Conductor.
delegate	The object's delegate.
partPerformerClass	The class the object uses to create its PartPerformers.

METHOD TYPES

Copying and initializing a ScorePerformer	– copy – copyFromZone: – init
Freeing a ScorePerformer and its contents	– free – freePartPerformers
Setting and retrieving the Score	– setScore: – score
Setting and retrieving the delegate	– setDelegate: – delegate
Accessing the PartPerformers	– setPartPerformerClass: – partPerformerClass – partPerformers – noteSenders – partPerformerForPart: – removePartPerformers

Broadcasting to the PartPerformers	– setConductor: – setDuration: – duration – setFirstTimeTag: – firstTimeTag – setLastTimeTag: – lastTimeTag – setTimeShift: – timeShift
Performing the object	– activate – activateSelf – deactivate – deactivateSelf – pause – resume – status
Archiving the object	– write: – read: – awake

INSTANCE METHODS

activate

– activate

Sends **activateSelf** to **self** and then sends the **activate** message to each of the ScorePerformer's PartPerformers. The ScorePerformer's status is set to MK_active if any of its PartPerformers could be activated (specifically, if any one of the PartPerformers returns non-**nil** when sent the activate message). If the ScorePerformer's Score isn't set, if the Score doesn't contain any Parts, or if the **activateSelf** message returns **nil**, this method returns **nil** (and the PartPerformers aren't activated); otherwise returns **self**.

See also: **– activateSelf**, **– deactivate**

activateSelf

– activateSelf

You never invoke this method directly; it's invoked as part of the **activate** method. A subclass implementation should send **[super activateSelf]**. If **activateSelf** returns **nil**, the ScorePerformer isn't activated. The default implementation does nothing and returns the ScorePerformer.

See also: **– activate**, **– deactivate**

awake

– awake

Prepares a recently unarchived ScorePerformer for use.

See also: **– write:**, **– read:**

copy

– copy

Creates and returns a new ScorePerformer that's a copy of the receiving ScorePerformer. The new object is associated with the same Score as the original, but it contains its own PartPerformers. The new object is inactive regardless of the status of the original.

See also: **– copyFromZone:**

copyFromZone:

– copyFromZone:(NXZone)*zone*

The same as **copy**, but the new ScorePerformer is allocated in the specified zone.

See also: **– copy**

deactivate

– deactivate

Sends **deactivateSelf** to **self**, sets the ScorePerformer's status to MK_inactive, sends **deactivate** to each of the PartPerformers, and then sends **performerDidDeactivate:** to the delegate. Returns **self**.

See also: **– deactivateSelf**, **– activate**

deactivateSelf

– deactivateSelf

You never invoke this method directly; it's invoked as part of the **deactivate** method. The default does nothing; a subclass can implement this method to perform post-performance activites. The return value is ignored.

See also: **– deactivate**, **– activate**

delegate

– delegate

Returns the ScorePerformer's delegate object, as set through **setDelegate:**. The delegate is sent messages as the ScorePerformer is activated, paused, resumed, and deactivated.

See also: **– setDelegate:**

duration

– (double)**duration**

Returns the ScorePerformer's maximum performance duration in beats, as set through **setDuration:**. By default, a ScorePerformer has an unlimited performance duration.

See also: **– setDuration:**, **– setTimeShift:**, **– setFirstTimeTag:**, **– setLastTimeTag:**

firstTimeTag

– (double)**firstTimeTag**

Returns the smallest time tag value considered for performance, as set through **setFirstTimeTag:**. By default, the first time tag is 0.0.

See also: **– setFirstTimeTag:**, **– setLastTimeTag:**, **– setDuration:**, **– setTimeShift:**

free

– free

If the ScorePerformer is currently performing (it's status isn't MK_inactive), this immediately returns **nil**. Otherwise frees the ScorePerformer and its PartPerformers.

See also: **– freePartPerformers**

freePartPerformers

– freePartPerformers

Removes and frees the ScorePerformer's PartPerformers and sets the ScorePerformer's Score to **nil**. Returns the ScorePerformer.

init

– init

Initializes the ScorePerformer and returns **self**.

lastTimeTag

– (double)**lastTimeTag**

Returns the greatest time tag value considered for performance, as set through **setLastTimeTag:**. By default, the greatest time tag is (virtually) infinity.

See also: **– setFirstTimeTag:**, **– setLastTimeTag:**, **– setDuration:**, **– setTimeShift:**

noteSenders

– **noteSenders**

Creates and returns a List containing the NoteSender objects that belong to the ScorePerformer's PartPerformers. The List will be empty if the Score hasn't been set. By default, each PartPerformer creates a single NoteSender, although you may change this by subclassing PartPerformer and setting the new class through the **setPartPerformerClass:** method. The sender is responsible for freeing the List that's created by this method.

See also: **– setScore:**, **– partPerformers**, **– setPartPerformerClass:**

partPerformerForPart:

– **partPerformerForPart:***aPart*

Returns the PartPerformer that's associated with *aPart*, where *aPart* must be a Part in the ScorePerformer's Score; returns **nil** if the Part isn't found or if the Score isn't set. Keep in mind that it's possible for a Part to have more than one PartPerformer; this method returns only the PartPerformer that was created by the receiver of this message.

See also: **– setScore:**, **– partPerformers**

partPerformers

– **partPerformers**

Creates and returns a List containing the PartPerformers that were created by the ScorePerformer. The List will be empty if the Score hasn't been set. The sender is responsible for freeing the List.

See also: **– setScore:**, **– partPerformerForPart:**

pause

– **pause**

Suspends the ScorePerformer's performance: The **pause** message is sent to each of the PartPerformers, the ScorePerformer's status is set to MK_paused, and **performerDidPause:** is sent to the ScorePerformer's delegate. Returns **self**.

Note: No check is made to ensure that the ScorePerformer is currently active. Because of this, a ScorePerformer will erroneously consider itself to be in a performance (but paused) if you send it a **pause** message before you actually activate it. By the same omission, the delegate message is sent every time you invoke this method, whereas the correct behavior would see that the message is sent only if the ScorePerformer is active when this method is invoked. The Performer class *does* perform a status check from within its **pause** method, so this unfortunate exuberance isn't exhibited by the ScorePerformer's PartPerformers.

See also: **– resume**

read:

– read:(NXTypedStream *)*stream*

Unarchives the ScorePerformer by reading it from *stream*. You never invoke this method directly; to read an archived ScorePerformer, call the **NXReadObject**() C function.

See also: **– write:**

removePartPerformers

– removePartPerformers

Removes the ScorePerformer's PartPerformers (but doesn't free them) and sets the ScorePerformer's Score to **nil**. Returns **self**.

See also: **– partPerformers, – freePartPerformers**

resume

– resume

Resumes the ScorePerformer's performance: The **resume** message is sent to each of the PartPerformers, the ScorePerformer's status is set to MK_active, and **performerDidResume:** is sent to the delegate. Returns **self**.

Note: No check is made to ensure that the ScorePerformer is currently paused. Because of this, a ScorePerformer will erroneously consider itself to be in a performance if you send it a **resume** message before you actually activate it. By the same omission, the delegate message is sent every time you invoke this method, whereas the correct behavior would see that the message is sent only if the ScorePerformer is paused when this method is invoked. The Performer class *does* perform a status check from within its **resume** method, so this unfortunate exuberance isn't exhibited by the ScorePerformer's PartPerformers.

See also: **– pause**

score

– **score**

Returns the ScorePerformer's Score.

See also: – **setScore:**

setConductor:

– **setConductor:***aConductor*

If the ScorePerformer's Score has been set, this sends the message **setConductor:***aConductor* to each of the ScorePerformer's PartPerformers. Otherwise, *aConductor* is cached and the messages are sent when **setScore:** is invoked. By default, the PartPerformers follow the baton of the defaultConductor; they continue to do so if *aConductor* is **nil**. If the ScorePerformer is in a performance (if its status isn't MK_inactive), this does nothing and returns **nil**. Otherwise, **self** is returned.

setDelegate:

– **setDelegate:***delegate*

Sets the ScorePerformer's delegate and returns **self**. Delegate messages are sent when you invoke the following ScorePerformer methods:

Method	Delegate Message
activate	performerDidActivate:
pause	performerDidPause:
resume	performerDidResume:
deactivate	performerDidDeactivate:

As usual, a delegate message is sent only if the object responds to it.

See also: – **delegate**

setDuration:

– **setDuration:**(double)*aDuration*

Sets the maximum number of beats that the ScorePerformer can spend in performance to *aDuration* beats. This accomplished by sending **setDuration:***aDuration* to each of the ScorePerformer's PartPerformers. If the ScorePerformer's Score hasn't been set, *aDuration* is cached and the PartPerformer messages are sent when **setScore:** is invoked. If the ScorePerformer is in a performance (if its status isn't MK_inactive), this does nothing and returns **nil**. Otherwise **self** is returned.

See also: – **duration**, – **setTimeShift:**, – **setFirstTimeTag:**, – **setLastTimeTag:**

setFirstTimeTag:

– **setFirstTimeTag:**(double)*aTimeTag*

Sets the smallest time tag value considered for performance by sending **setFirstTimeTag:***aTimeTag* to each of the ScorePerformer's PartPerformers. If the ScorePerformer's Score hasn't been set, *aTimeTag* is cached and the messages are sent when **setScore:** is invoked. If the ScorePerformer is in a performance (if its status isn't MK_inactive), this does nothing and returns **nil**. Otherwise **self** is returned.

See also: **– firstTimeTag**, **– setLastTimeTag:**, **– setTimeShift:**, **– setDuration:**

setLastTimeTag:

– **setLastTimeTag:**(double)*aTimeTag*

Sets the greatest time tag value considered for performance by sending **setLastTimeTag:***aTimeTag* to each of the ScorePerformer's PartPerformers. If the ScorePerformer's Score hasn't been set, *aTimeTag* is cached and the messages are sent when **setScore:** is invoked. If the ScorePerformer is in a performance (if its status isn't MK_inactive), this does nothing and returns **nil**. Otherwise **self** is returned.

See also: **– lastTimeTag**, **– setFirstTimeTag:**, **– setTimeShift:**, **– setDuration:**

setScore:

– **setScore:***aScore*

Sets the ScorePerformer's Score to *aScore* and creates a PartPerformer object for each of the Score's Parts. The PartPerformers are instances of the class set through **setPartPerformer:**; by default, they're instances of the PartPerformer class. This method also sets the ScorePerformer's Subsequent changes to *aScore* (by adding or removing Parts) won't be seen by the ScorePerformer. The PartPerformers from a previously set Score (if any) are removed and freed. Returns **self**.

setTimeShift:

– **setTimeShift:**(double)*aTimeShift*

Sets the ScorePerformer's performance time offset by sending **setTimeShift:***aTimeShift* to each of its PartPerformers. If the ScorePerformer's Score hasn't been set, *aTimeTag* is cached and the messages are sent when **setScore:** is invoked. If the ScorePerformer is in a performance (if its status isn't MK_inactive), this does nothing and returns **nil**. Otherwise **self** is returned.

See also: **– timeShift**, **– setDuration:**, **– lastTimeTag**, **– setFirstTimeTag:**

status

– (MKPerformerStatus)**status**

Returns the ScorePerformer's current performance status as one of the following integer constants:

Constant	Meaning
MK_active	In a performance (or activated in anticipation of a performance)
MK_paused	In a performance but currently paused
MK_inactive	Not in a performance

You can't set a ScorePerformer's status directly; it's set as the ScorePerformer is created (MK_inactive), activated (MK_active), paused (MK_paused), resumed (MK_active), and deactivated (MK_inactive).

timeShift

– (double)**timeShift**

Returns the ScorePerformer's performance time offset, as set through **setTimeShift:**. By default, the offset is 0.0.

See also: **– setTimeShift:**, **– setFirstTimeTag:**, **– setLastTimeTag:**, **– setDuration:**

write:

– **write:**(NXTypedStream *)*stream*

Archives the ScorePerformer by writing it to *stream*. You never invoke this method directly; to archive a NoteSender, call the **NXWriteRootObject()** C function. The ScorePerformer's PartPerformer List, PartPerformer class, first and last time tag variables, duration, and time shift are archived directly. Its Score, Conductor, and delegate are archived by reference.

See also: **– read:**

METHODS IMPLEMENTED BY THE DELEGATE

performerDidActivate:

– performerDidActivate:*sender*

Sent to the delegate when *sender* is activated.

performerDidPause:

– **performerDidPause:***sender*

Sent to the delegate when *sender* is paused.

performerDidResume:

– **performerDidResume:***sender*

Sent to the delegate when *sender* is resumed.

performerDidDeactivate:

– **performerDidDeactivate:***sender*

Sent to the delegate when *sender* is deactivated.

ScoreRecorder

INHERITS FROM	Object
DECLARED IN	musickit.h

CLASS DESCRIPTION

A ScoreRecorder is a pseudo-Instrument that adds Notes to the Parts in a given Score. It does this by creating a PartRecorder, a true Instrument, for each of the Score's Part objects. A ScoreRecorder's Score is set through the **setScore:** method. If you add Parts to or remove Parts from the Score after sending the **setScore:** message, the changes will not be seen by the ScoreRecorder.

A ScoreRecorder is said to be in performance from the time any of its PartRecorders receives a Note until the performance is finished.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in ScoreRecorder	id	partRecorders;
	id	score;
	MKTimeUnit	timeUnit;
	id	partRecorderClass;

partRecorders	The object's Set of PartRecorders.
score	The object's Score.
timeUnit	Unit the object's PartRecorders use to measure time; one of MK_second or MK_beat.
partRecorderClass	Class used to create PartRecorder objects (must inherit from PartRecorder).

METHOD TYPES

Creating and freeing a ScoreRecorder	– copyFromZone: – free – init
Modifying the object	– freePartRecorders – partRecorders – removePartRecorders – setPartRecorderClass: – setScore: – setTimeUnit:
Querying the object	– noteReceivers – partRecorderForPart: – partRecorderClass – score – timeUnit
Performing the object	– afterPerformance – firstNote: – inPerformance

INSTANCE METHODS

afterPerformance

– **afterPerformance**

You never invoke this method; it's invoked automatically at the end of the performance.

copy

– **copy**

Creates and returns a ScoreRecorder as a copy of the receiving ScoreRecorder. The new object has the same Score as the original, but contains its own set of PartRecorders.

firstNote:

– **firstNote:***aNote*

You never invoke this method; it's invoked automatically when the first Note is received by any of the ScoreRecorder's PartRecorders.

free

– **free**

Frees the ScoreRecorder and its PartRecorders. If you want to free the ScoreRecorder only, send **removePartRecorders** to the object before invoking this method.

freePartRecorders

– **freePartRecorders**

Frees the ScoreRecorder's PartRecorders and sets its Score to **nil**. Returns **self**.

inPerformance

– (BOOL)**inPerformance**

Returns YES if the ScoreRecorder is in performance, otherwise returns NO.

noteReceivers

– **noteReceivers**

Returns a List object that contains the ScoreRecorder's NoteReceivers (the NoteReceivers of the ScoreRecorder's PartRecorders). It's the sender's responsibility to free the List.

partRecorderForPart:

– **partRecorderForPart:***aPart*

Returns the PartRecorder that corresponds to *aPart*, or **nil** if not found.

partRecorders

– **partRecorders**

Returns a List object that contains the ScoreRecorder's PartRecorders. It's the sender's responsibility to free the List.

partRecorderClass

– **partRecorderClass**

Returns the class object that the ScoreRecorder uses to create its PartRecorder objects, as set through **setPartRecorderClass:**. By default, the ScoreRecorder creates instances directly from PartRecorder.

removePartRecorders

– **removePartRecorders**

Removes the ScoreRecorder's PartRecorders and sets its Score to **nil**. The PartRecorder objects aren't freed. Returns **self**.

score

– **score**

Returns the ScoreRecorder's Score, as set through **setScore:**.

setPartRecorderClass:

– **partRecorderClass:***classObject*

Sets the class object that the ScoreRecorder uses to create its PartRecorder objects. The argument must inherit from PartRecorder. By default, the ScoreRecorder creates instances directly from PartRecorder.

setScore:

– **setScore:***aScore*

Removes and frees the ScoreRecorder's PartRecorders, sets its Score to *aScore*, and creates and adds a PartRecorder for each Part in the Score. Subsequent changes to *aScore* (adding or removing Parts) aren't seen by the ScoreRecorder. If the receiver is in performance, this does nothing and returns **nil**, otherwise it returns **self**.

If you want to set the Score without freeing the current PartRecorders you should send **removePartRecorders** before invoking this method; the PartRecorders are then removed but not freed.

setTimeUnit:

– **setTimeUnit:**(MKTimeUnit)*aTimeUnit*

Sets the ScoreRecorder's time unit to *aTimeUnit*, one of MK_beat and MK_second, and forwards the **setTimeUnit:***aTimeUnit* message to the ScoreRecorder's PartRecorders. If the ScoreRecorder is in performance, this does nothing and returns **nil**. Otherwise returns **self**.

timeUnit

– (MKTimeUnit)**timeUnit**

Returns the ScoreRecorder's time unit, either MK_second or MK_beat.

SynthData

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

SynthData objects represent DSP memory that's used in music synthesis. For example, you can use a SynthData object to load predefined data for wavetable synthesis or to store DSP-computed data to create a digital delay. Perhaps the most common use of SynthData is to create a location through which UnitGenerators can pass data. This type of SynthData object is called a *patchpoint*. For example, in frequency modulation an oscillator UnitGenerator writes its output to a patchpoint which can then be read by another oscillator as its frequency input.

You never create SynthData objects directly in an application, they can only be created by the Orchestra through its **allocSynthData:length:** or **allocPatchpoint:** methods. A SynthData object is typically allocated and owned by a SynthPatch, an object that configures a set of SynthData and UnitGenerator objects into a DSP software instrument.

The methods **setData:** and **setConstant:** are used to download data to the DSP memory that's represented by a SynthData object. The former downloads an array of values, the latter fills the entire memory with a constant value. These methods are simple versions of the more thorough methods **setData:length:offset:** and **setConstant:length:offset:**, which allow you to load an arbitrary amount of data into any portion of the SynthData's memory. The data in a SynthData object, like all DSP data used in music synthesis, is 24-bit fixed point words (data type DSPDatum). Similar methods are provided that let you download arrays of 16-bit (short) data. You can declare a SynthData to be read-only by sending it the message **setReadOnly:YES**. You can't change the data in a read-only SynthData object.

An instance of SynthData knows the address of its DSP memory, but it doesn't contain a copy of the data that it downloads.

DSP memory allocation and management is explained in the Orchestra class description; many of the return types used here, such as DSPAddress and DSPMemorySpace, are described in Orchestra. In general, the design of the Orchestra makes intimate knowledge of the details of the DSP unnecessary.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in SynthData	id	synthPatch;
	id	orchestra;
	int	length;
	MKOrchAddrStruct	orchAddr;
	BOOL	readOnly;

synthPatch	The SynthPatch that owns this object (if any).
orchestra	The orchestra on which the object is allocated.
length	Length of allocated memory in words.
orchAddr	Structure that directly represents DSP memory.
readOnly	YES if the object's data is read-only.

METHOD TYPES

Filling the SynthData	– setData: – setData:length:offset: – setToConstant: – setToConstant:length:offset: – setShortData: – setShortData:length:offset: – clear
Dealloacating the object	– dealloc
Modifying the object	– setReadOnly:
Querying the object	– address – isAllocated – isFreeable – length – memorySpace – orchAddrPtr – orchestra – readOnly – referenceCount – synthPatch
Unit Generator-compatibility	– finish – idle – run

INSTANCE METHODS

address

– (DSPAddress)**address**

Returns the DSP address of the SynthData's memory block.

clear

– **clear**

Clears the SynthData's memory but doesn't deallocate it.

dealloc

– **dealloc**

If the SynthData isn't part of a SynthPatch, it's deallocated. If it's part of a SynthPatch that can be freed, the entire SynthPatch is deallocated. Otherwise does nothing and returns **nil**.

finish

– (double)**finish**

This does nothing and returns 0.0. It's provided for compatibility with UnitGenerator; specifically, it allows a SynthPatch to send **finish** to all its SynthElement objects without regard for their class.

idle

– **idle**

This does nothing and returns the SynthData. It's provided for compatibility with UnitGenerator; specifically, it allows a SynthPatch to send **idle** to all its SynthElement objects without regard for their class.

isAllocated

– (BOOL)**isAllocated**

Provided for compatibility with UnitGenerator. Always returns YES, since deallocated SynthDatas are freed immediately.

isFreeable

– (BOOL)**isFreeable**

Invoked by the Orchestra to determine whether the SynthData may be freed. Returns YES if it can, NO if it can't. (A SynthData can be freed if its a member of a Synthpatch that can be freed.)

length

– (int)**length**

Returns the size (in words) of the SynthData's memory block.

memorySpace

– (DSPMemorySpace)**memorySpace**

Returns the DSP space in which the SynthData's memory block is allocated.

orchAddrPtr

– (MKOrchAddrStruct *)**orchAddrPtr**

Returns a pointer to the SynthData's address structure.

orchestra

– **orchestra**

Returns the SynthData's Orchestra object.

readOnly

– (BOOL)**readOnly**

Returns YES if the SynthData is read-only.

referenceCount

– (int)**referenceCount**

If the SynthData is installed in its Orchestra's shared object table, this returns the number of objects that have allocated it. Otherwise returns 1.

run

– **run**

This does nothing and returns the SynthData. It's provided for compatibility with UnitGenerator; specifically, it allows a SynthPatch to send **run** to all its SynthElement objects without regard for their class.

setData:

– **setData:**(DSPDatum *)*dataArray*

Loads *dataArray* into the SynthData's memory. Implemented as (and returns the value of)

```
[self setData:dataArray length:length offset:0];
```

where the second argument is the SynthData's allocated length.

setData:length:offset:

– **setData:**(DSPDatum *)*dataArray*
length:(int)*len*
offset:(int)*off*

Loads (at most) *len* words of data from *dataArray* into the SynthData's memory, starting at location *off* words from the beginning of the SynthData's memory block. If *off* + *len* is greater than the SynthData's length (as returned by the **length** method), or if the data couldn't otherwise be loaded, the error MK_synthDataLoadErr is generated and **nil** is returned. Otherwise returns **self**.

setReadOnly:

– **setReadOnly:**(BOOL)*readOnlyFlag*

Sets the SynthData to read-only if *readOnlyFlag* is YES and read-write if it's NO. The default access for a SynthData object is read-write. Returns the SynthData. The Orchestra automatically creates some read-only SynthData objects (SineROM, MuLawROM, and the zero and sink patchpoints) that ignore this method.

setShortData:

– **setData:**(char *)*dataArray*

Loads *dataArray* into the SynthData's memory. Implemented as (and returns the value of)

```
[self setShortData:dataArray length:length offset:0];
```

where the second argument is the SynthData's allocated length.

setShortData:length:offset:

– **setShortData:**(short *)*dataArray*
length:(int)*len*
offset:(int)*off*

Loads (at most) *len* words of 16-bit data from *dataArray* into the SynthData's memory, right-justified, starting at location *off* words from the beginning of the SynthData's memory block. The data in dataArray is If *off* + *len* is greater than the SynthData's length (as returned by the **length** method), or if the data couldn't otherwise be loaded, the error MK_synthDataLoadErr is generated and **nil** is returned. Otherwise returns **self**.

setToConstant:

– **setToConstant:**(DSPDatum)*value*

Fills the SynthData's memory with the constant *value*. Implemented as (and returns the value of)

```
[self setToConstant:value length:length offset:0];
```

where the second argument is the instance variable **length**.

setToConstant:length:offset:

– **setToConstant:**(DSPDatum)*value*
length:(int)*len*
offset:(int)*off*

Similar to **setData:length:offset:**, but loads the constant *value* rather than an array; see **setData:length:offset:** for details.

synthPatch

– **synthPatch**

Returns the SynthPatch that the SynthData is part of, if any.

SynthInstrument

INHERITS FROM Instrument : Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A SynthInstrument realizes Notes by synthesizing them on the DSP. It does this by forwarding each Note it receives to a SynthPatch object, which translates the parameter information in the Note into DSP instructions. A SynthInstrument can manage any number of SynthPatch objects (limited by the speed and size of the DSP). However, all of its SynthPatches are instances of the same SynthPatch subclass. You assign a particular SynthPatch subclass to a SynthInstrument through the latter's **setSynthPatchClass:** method. A SynthInstrument can't change its SynthPatch class during a performance.

Each SynthPatch managed by the SynthInstrument corresponds to a particular note tag. As the SynthInstrument receives Notes, it compares the Note's note tag to the note tags of the SynthPatches that it's managing. If a SynthPatch already exists for the note tag, the Note is forwarded to that object; otherwise, the SynthInstrument either asks the Orchestra to allocate another SynthPatch, or it preempts an allocated SynthPatch to accommodate the Note. Which action it takes depends on the SynthInstrument's allocation mode and the available DSP resources (as explained later in this description).

Every SynthInstrument maintains an *update state* into which it merges parameters from note tag-less noteUpdates. When a Note that signals a new SynthPatch arrives, the parameters in the update state are merged with the new Note (the parameters of the new Note take precendence if there's a conflict). Thus the update state defines a set of "sticky" parameters for the SynthInstrument.

A SynthInstrument can either be in automatic allocation mode (MK_AUTOALLOC) or manual mode (MK_MANUALALLOC). In automatic mode, SynthPatches are allocated directly from the Orchestra as Notes are received by the SynthInstrument and released when it's no longer needed. Automatic allocation is the default.

In manual mode, the SynthInstrument pre-allocates a fixed number of SynthPatch objects through the **setSynthPatchCount:** method. If it receives more simultaneously sounding Notes than it has SynthPatches, the SynthInstrument preempt its oldest running SynthPatch (by sending it the **preemptFor:** message).

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Inherited from Instrument	id	noteReceivers;

Declared in SynthInstrument	id	synthPatchClass;
	unsigned short	allocMode;
	id	taggedPatches;
	id	controllerTable;
	id	updates;
	BOOL	retainUpdates
	id	orchestra

synthPatchClass	Class used to create SynthPatch instances.
allocMode	The object's allocation mode; either MK_MANUALALLOC or MK_AUTOALLOC.
taggedPatches	HashTable of allocated SynthPatches.
controllerTable	Part of the update state that contains MIDI controller values.
updates	The rest of the update state.
retainUpdates	Is the update state retained between performances?
orchestra	The Orchestra object from which SynthPatches are allocated.

METHOD TYPES

Creating and freeing a SynthInstrument	– copy – copyFromZone: – free
Modifying the object	– autoAlloc – init – mute: – clearUpdates – setRetainUpdates: – setSynthPatchClass: – setSynthPatchClass:orchestra:
Querying the object	– activeSynthPatches: – allocMode – orchestra – doesRetainUpdates – getUpdates:controllerValues: – synthPatchClass

Allocating SynthPatch objects	– preemptSynthPatchFor:patches: – setSynthPatchCount: – setSynthPatchCount:patchTemplate: – synthPatchCount – synthPatchCountForPatchTemplate:
Performing the object	– realizeNote:fromNoteReceiver: – abort

INSTANCE METHODS

abort

– **abort**

Sends the **noteEnd** message to all running (or finishing) SynthPatches managed by the SynthInstrument. This causes all SynthPatches to immediately become available.

activeSynthPatches:

– **activeSynthPatches:***aTemplate*

Returns the first in the sequence of the SynthInstrument's SynthPatches, created from the specified PatchTemplate, that are currently sounding. If *aTemplate* is **nil**, the default PatchTemplate is used. The sequence is ordered by the begin times of the SynthPatches' current phrases, from the earliest to the latest. You step down the sequence by sending **next** to the objects returned by this method. If there aren't any active SynthPatches with the specified template, **nil** is returned.

allocMode

– (unsigned short)**allocMode**

Returns the SynthInstrument's allocation mode, one of MK_AUTOALLOC or MK_MANUALALLOC.

autoAlloc

– **autoAlloc**

Sets the SynthInstrument's allocation mode to MK_AUTOALLOC and releases any manually allocated SynthPatch objects. If the SynthInstrument is in performance and isn't already in MK_AUTOALLOC mode, this does nothing and returns **nil**. Otherwise returns the receiver.

clearUpdates

– **clearUpdates**

Clears the SynthInstrument's update state.

copy

– **copy**

Creates and returns a new SynthInstrument as a copy of the receiver. The copy has the same (NoteReceiver) connections but has no SynthPatches allocated.

doesRetainUpdates

– (BOOL)**doesRetainUpdates**

If the SynthInstrument retains its update state between preformances, this returns YES, otherwise returns NO. By default, a SynthInstrument doesn't retain its update state.

free

– **free**

If the SynthInstrument isn't in performance, this frees the SynthInstrument. Otherwise does nothing and returns **self**.

getUpdates:controllerValues:

– **getUpdates:**(Note **)*aNoteUpdate* **controllerValues:**(HashTable **)*controllers*

Returns the SynthInstrument's update state as it's split between the MIDI controller value parameters (given in the *controllers* HashTable) and all other parameters (the *aNoteUpdate* Note).

init

– **init**

Initializes the SynthInstrument.

mute:

– **mute:***aMute*

You never invoke this method; it's invoked automatically when the SynthInstrument receives a mute Note. Mutes aren't normally forwarded to SynthPatches since they usually don't produce sound. The default implementation does nothing. A subclass can implement this method to examine *aMute* and act accordingly.

orchestra

– **orchestra**

Returns the orchestra object from which SynthPatches are allocated, as set with **setSynthPatchClass:orchestra:**. If this method returns the Orchestra class, then SynthPatches are allocated from the first available Orchestra.

preemptSynthPatchFor:patches:

– **preemptSynthPatchFor:***aNote* **patches:***firstPatch*

You never invoke this method. It's invoked automatically when the SynthInstrument is in manual mode and all SynthPatches are in use, or when it's in auto mode and the DSP resources needed to build another SynthPatch aren't available. The return value is taken as the SynthPatch to preempt in order to accommodate the latest request. *firstPatch* is the first in a sequence of ordered active SynthPatches, as returned by the **activeSynthPatches:** method. The default implementation simply returns *firstPatch*, the SynthPatch with the oldest phrase. A subclass can reimplement this method to provide a different scheme for determining which SynthPatch to preempt.

realizeNote:fromNoteReceiver:

– **realizeNote:***aNote* **fromNoteReceiver:***aNoteReceiver*

Synthesizes *aNote*.

setRetainUpdates:

– **setRetainUpdates:**(BOOL)*yesOrNo*

If *yesOrNo* is YES, the SynthInstrument's update state is retained between performances. Otherwise, it's cleared after each performance.

setSynthPatchClass:

– **setSynthPatchClass:***aSynthPatchClass*

Sets the SynthInstrument's SynthPatch class to *aSynthPatchClass*. Returns **nil** if the argument isn't a subclass of SynthPatch or the SynthInstrument is in a performance (the class isn't set in this case). Otherwise returns **self**.

setSynthPatchClass:orchestra:

– **setSynthPatchClass:***aSynthPatchClass* **orchestra:***anOrchestra*

Like **setSynthPatchClass:**, but also sets the Orchestra object from which SynthPatches will be allocated. If *anOrchestra* is **nil**, SynthPatches are allocated on the first available Orchestra.

setSynthPatchCount:

– (int)**setSynthPatchCount:**(int)*voices*

Attempts to allocate *voices* SynthPatch objects. Implemented as

```
[self setSynthPatchCount:voices template:nil];
```

Returns the number of objects that were actually allocated.

setSynthPatchCount:patchTemplate:

– (int)**setSynthPatchCount:**(int)*voices* **patchTemplate:***aTemplate*

Attempts to allocate *voices* SynthPatch objects using the patch template *aTemplate* (the Orchestra must be open). This puts the SynthInstrument in manual mode. If *aTemplate* is **nil**, the value returned by the message

```
[synthPatchClass defaultPatchTemplate]
```

is used. Returns the number of objects that were allocated (it may be less than the number requested). If the SynInstrument is in performance and it isn't already in manual mode, this message is ignored and 0 is returned.

If you decrease the number of manually allocated SynthPatches during a performance, the extra SynthPatches aren't deallocated until they become inactive. In other words, reallocating downward won't interrupt active SynthPatches.

synthPatchClass

– **synthPatchClass**

Returns the SynthInstrument's SynthPatch class.

synthPatchCount

– (int)**synthPatchCount**

Returns the number of allocated SynthPatch objects created with the default PatchTemplate.

synthPatchCountForPatchTemplate:

– (int)**synthPatchCountForPatchTemplate:***aTemplate*

Returns the number of allocated SynthPatch objects created with the PatchTemplate *aTemplate*.

SynthPatch

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

SynthPatch is an abstract class, each subclass of which defines a unique configuration of UnitGenerator and SynthData objects that work as a sound synthesis module. SynthPatch objects aren't created directly; rather, they're allocated through messages, such as **allocSynthPatch:**, that you send to an Orchestra object. Once you've allocated a SynthPatch, you feed it Notes through the **noteOn:**, **noteUpdate:**, and **noteOff:** methods, where the type of the Note is assumed to correspond to the name of the method. To halt a SynthPatch, you send it **noteEnd**. Alternatively, and more commonly, you can create and use instances of SynthInstrument to allocate and distribute Notes to SynthPatch objects for you.

The Music Kit includes a number of SynthPatch subclasses that you can use in your application. The design of a SynthPatch subclass is examined in detail in Chapter 4 of *Concepts*.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in SynthPatch	id	synthElements;
	id	synthInstrument;
	int	noteTag;
	MKSynthStatus	status;
	id	patchTemplate;
	BOOL	isAllocated;
	id	orchestra;

synthElements — List of UnitGenerator and SynthData objects.

synthInstrument — The SynthInstrument object that owns the object, if any.

noteTag — The object's current note tag (used by its SynthInstrument).

status — The object's status.

patchTemplate — The object's PatchTemplate.

isAllocated — YES if the object is allocated.

orchestra	Orchestra on which the object is allocated.

METHOD TYPES

Creating and freeing a SynthPatch	– dealloc – init – free
Defining a subclass	+ defaultPatchTemplate + patchTemplateFor: – controllerValues: – noteOnSelf: – noteOffSelf: – noteUpdateSelf: – noteEndSelf – freeSelf – phraseStatus – synthElementAt: – preemptFor: – moved:
Performing the object	– noteOn: – noteOff: – noteUpdate: – noteEnd
Querying the object	– synthInstrument – isFreeable – next – noteTag – patchTemplate – status

CLASS METHODS

defaultPatchTemplate

+ defaultPatchTemplate

Returns the default PatchTemplate for the class.

patchTemplateFor:

+ patchTemplateFor:*currentNote*

Returns an appropriate PatchTemplate with which to create a SynthPatch instance that will adequately synthesize *currentNote*. This method is invoked by SynthInstrument whenever it needs to allocate a new SynthPatch instance. It may also be sent by an

application to obtain the template to be used as the second argument to SynthInstrument's **setSynthPatchCount:patchTemplate:** method, or to Orchestra's **allocSynthPatch:patchTemplate:** method. Implementation of this method is a subclass responsibility. If *currentNote* is **nil**, the default template should be returned.

INSTANCE METHODS

controllerValues:

– **controllerValues:***controllers*

You never invoke this method; it's sent by the SynthPatch's SynthInstrument when a new Note stream begins, before the **noteOn:** message is sent. *controllers* is a HashTable that describes the state of the MIDI controllers by mapping integer controller numbers to integer controller values. A subclass can implement this method to examine the argument and act accordingly. The default implementation does nothing. The return value is ignored.

dealloc

– **dealloc**

You can use this method to deallocate a SynthPatch that you allocated directly from an Orchestra. It sends **noteEnd** to the receiver, deallocates the SynthPatch, and returns **nil**. If the SynthPatch is owned by a SynthInstrument, this does nothing and returns **nil**.

free

– **free**

You never invoke this method; only the Orchestra can free a SynthPatch. If a subclass needs to do anything special when the receiver is freed, it should override **freeSelf**.

freeSelf

– **freeSelf**

Sent just before the SynthPatch is freed by the Orchestra. A subclass can implement this method to provide specialized behavior.

init

– **init**

You never invoke this method; it's sent by the Orchestra when a new SynthPatch has just been allocated, but before its UnitGenerators are connected. A subclass may override this method to provide additional initialization. A return of **nil** aborts the creation and frees the new SynthPatch. The default implementation does nothing and returns **self**.

isFreeable

– (BOOL)**isFreeable**

Returns YES if the SynthPatch may be freed; otherwise returns NO. A SynthPatch may only be freed if it's idle and it isn't owned by a manually allocated SynthInstrument.

moved:

– **moved:***aUG*

Sent when the Orchestra moves a SynthPatch's UnitGenerator during DSP memory compaction. *aUG* is the object that was moved. A subclass can override this method to provide specialized behavior. The default implementation does nothing.

next

– **next**

This method is used in conjunction with a SynthInstrument's **preemptSynthPatchFor:patches:** method. It returns the next SynthPatch in a List of active SynthPatches owned by the SynthInstrument. The objects in the List are in the order in which they began synthesizing their current phrases (oldest first).

noteEnd

– **noteEnd**

Causes the SynthPatch to become idle. The message **noteEndSelf** is sent to the SynthPatch and its status is set to MK_idle. Returns **self**.

noteEndSelf

– **noteEndSelf**

You never invoke this method; it's invoke automatically by the **noteEnd** method. A subclass may override this to do what it needs to do to ensure that the SynthPatch produces no output. Usually, the subclass implementation sends the **idle** message to its output UnitGenerator. The default implementation does nothing; the return value is ignored.

noteOff:

– (double)**noteOff:***aNote*

Concludes a Note stream by sending **noteOffSelf:***aNote* to the SynthPatch and setting the SynthPatch's status to MK_finishing. Returns the value returned by **noteOffSelf:**.

noteOffSelf:

– (double)**noteOffSelf:***aNote*

You never invoke this method; it's invoked automatically by the **noteOff**: method. A subclass may provide an implementation that describes its response to a noteOff. The return value is the amount of time to wait, in seconds, before the SynthPatch can be released (in other words, before **noteEnd** can be sent). The default implementation returns 0.0.

noteOn:

– **noteOn:***aNote*

This starts or rearticulates a Note stream by sending **noteOnSelf:***aNote* to the SynthPatch. If **noteOnSelf:** returns **self**, the SynthPatch's status is set to MK_running and **self** returned. If **noteOnSelf:** returns **nil**, **noteEnd** is sent to the SynthPatch and **nil** is returned.

noteOnSelf:

– **noteOnSelf:***aNote*

You never invoke this method; it's invoked automatically by the **noteOn**: method. A subclass may provide an implementation that describes its response to a noteOn. The method should return **self**, or **nil** if you want the SynthPatch to immediately become idle. The default implementation returns **self**.

noteTag

– (int)**noteTag**

Returns the note tag associated with the Note stream the SynthPatch is currently playing. A SynthPatch's note tag is used as an identifier by its SynthInstrument.

noteUpdate:

– **noteUpdate:***aNote*

Updates a Note stream by sending **noteUpdateSelf:***aNote* to the SynthPatch. Returns **nil** if the SynthPatch is idle, otherwise returns **self**

noteUpdateSelf:

– **noteUpdateSelf:***aNote*

You never invoke this method; it's invoked automatically by the **noteUpdate**: method. A subclass may provide an implementation that describes its response to a noteUpdate. The return value is ignored.

orchestra

– **orchestra**

Returns the Orchestra object on which the SynthPatch is allocated. All the UnitGenerator and SynthData objects in the SynthPatch are allocated on the same Orchestra.

patchTemplate

– **patchTemplate**

Returns the PatchTemplate that was used to allocate the SynthPatch.

phraseStatus

– (MKPhraseStatus)**phraseStatus**

This is a convenience method for SynthPatch subclass implementors. The return value gives a precise account of the current status of the SynthPatch with regard to the Note stream. This method can only be invoked from within **noteOnSelf:**, **noteOffSelf:**, **noteUpdateSelf:**, and **noteEndSelf**, and returns a value as follows:

noteOnSelf:: If the SynthPatch is beginning a new phrase and isn't being preempted, MK_phraseOn is returned. If the SynthPatch is being preempted to begin a new phrase, MK_phraseOnPreempt is returned. If the phrase is already in progress, MK_phraseRearticulate is returned.

noteOffSelf:: Always returns MK_phraseOff.

noteUpdateSelf:: If the SynthPatch is finishing, MK_phraseOffUpdate is returned. Otherwise, MK_phraseUpdate is returned.

noteEndSelf: Always returns MK_phraseEnd.

Called from outside a SynthPatch, MK_noPhraseActivity is returned.

preemptFor:

– **preemptFor:***aNote*

Sent when the SyntPatch is running or finishing and is preempted by its SynthInstrument. The default implementation does nothing and returns **self**. Normally, a time equal to the value returned by **MKPreemptDuration()** is allowed to elapse before the preempting Note begins. A subclass can specify that the new Note happen immediately by returning **nil**.

status

– (int)**status**

Returns the status of the SynthPatch, one of MK_running, MK_finishing, and MK_idle.

synthElementAt:

– **synthElementAt:**(unsigned)*anIndex*

Returns the UnitGenerator or SynthData at the specified index or **nil** if *anIndex* is out of bounds. *anIndex* is zero-based.

synthInstrument

– **synthInstrument**

Returns the SynthInstrument that owns the SynthPatch, if any.

TuningSystem

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A TuningSystem object represents a musical tuning system by mapping the 128 key numbers to the frequencies that you specify. The frequencies in a TuningSystem object don't have to increase as the key numbers increase—you can even create a TuningSystem that descends in pitch as the key numbers ascend the scale.

The TuningSystem class maintains a master system called the *installed tuning system*. By default, the installed tuning system is set to 12-tone equal-temperament with A above middle C set to 440 Hz. A key number that doesn't reference a TuningSystem object takes its frequency value from the installed tuning system. The frequency value of a pitch variable is also taken from the installed system. The difference between key numbers and pitch variables is explained in *Concepts*. The entire map of key numbers, pitch variables, and frequency values in the default 12-tone equal-tempered system is given in Appendix B, "Music Tables."

You can install a tuning system by sending the **install** message to a TuningSystem object. Keep in mind that this doesn't install the object itself, it simply copies its key number-frequency map. Subsequent changes to the object won't affect the installed tuning system (unless you again send the object the **install** message).

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in TuningSystem	id	frequencies;
frequencies	Array of frequencies, indexed by key number.	

METHOD TYPES

Creating and freeing a TuningSystem

- copy
- copyFromZone:
- init
- initFromInstalledTuningSystem
- free

Tuning the object	– setKeyNum:toFreq: – setKeyNumAndOctaves:toFreq: – setTo12ToneTempered – transpose:
Querying the object	– freqForKeyNum:
Tuning the installed tuning system	– install + setKeyNum:toFreq: + setKeyNumAndOctaves:toFreq: + transpose:
Querying the installed tuning system	+ freqForKeyNum:
Archiving a TuningSystem	– read: – write:

CLASS METHODS

freqForKeyNum:

+ (double)**freqForKeyNum:**(MKKeyNum)*aKeyNum*

Returns the installed frequency for the key number *aKeyNum*. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal()** to check for this value).

See also: + **setKeyNum:toFreq:**

setKeyNum:toFreq:

+ **setKeyNum:**(MKKeyNum)*aKeyNum* **toFreq:**(double)*freq*

Maps the *aKeyNum* key number to the frequency *freq* in the installed tuning system. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal()** to check for this value); otherwise returns **self**.

See also: + **setKeyNumAndOctaves:toFreq:**, + **freqForKeyNum:**

setKeyNumAndOctaves:toFreq:

+ **setKeyNumAndOctaves:**(MKKeyNum)*aKeyNum* **toFreq:**(double)*freq*

Maps the *aKeyNum* key number to the frequency *freq* in the installed tuning system, then tunes all octaves of *aKeyNum* to octaves of *freq*. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal()** to check for this value); otherwise returns **self**.

See also: + **setKeyNum:toFreq:**, + **freqForKeyNum:**

transpose:

+ **transpose:**(double)*semitones*

Transposes the installed tuning system by *semitones* half-steps. (The half-step used here is always 100 cents.) If *semitones* is positive, the transposition is up, if it's negative, the transposition is down. You can transpose the tuning system by increments smaller than a half-step by supplying a fractional argument. Returns **self**.

See also: + **setKeyNumAndOctaves:toFreq:**, + **freqForKeyNum:**

INSTANCE METHODS

copy

– **copy**

Creates and returns a new TuningSystem as a copy of the receiving TuningSystem.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

The same as **copy**, but the new object is allocated in the specified zone.

See also: – **copy**

free

– **free**

Frees the TuningSystem.

freqForKeyNum:

– (double)**freqForKeyNum:**(MKKeyNum)*aKeyNum*

Returns the TuningSystem's frequency for the key number *aKeyNum*. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal()** to check for this value).

See also: – **setKeyNum:toFreq:**

init

– **init**

Initializes a newly allocated TuningSystem to a 12-tone equal-tempered scale.

initFromInstalledTuningSystem

– **initFromInstalledTuningSystem**

Initializes a new TuningSystem object and tunes it to the installed tuning system.

install

– **install**

Uses the TuningSystem's key number-frequency map as the installed tuning system. The TuningSystem object itself isn't installed, so subsequent changes to the object won't affect the installed tuning system. Returns **self**.

setKeyNum:toFreq:

– **setKeyNum:**(MKKeyNum)*aKeyNum* **toFreq:**(double)*freq*

Maps the TuningSystem's *aKeyNum* key number to *freq*. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal**() to check for this value); otherwise returns **self**.

See also: – **setKeyNumAndOctaves:toFreq:**, – **freqForKeyNum:**

setKeyNumAndOctaves:toFreq:

– **setKeyNumAndOctaves:**(MKKeyNum)*aKeyNum* **toFreq:**(double)*freq*

Maps the TuningSystem's *aKeyNum* key number to the frequency *freq*, then tunes all octaves of *aKeyNum* to octaves of *freq*. If *aKeyNum* is out of bounds, returns MK_NODVAL (use **MKIsNoDVal**() to check for this value); otherwise returns **self**.

See also: – **setKeyNum:toFreq:**, – **freqForKeyNum:**

setTo12ToneTempered

– **setTo12ToneTempered**

Sets the TuningSystem's tuning to 12-tone equal-tempered, with A above middle C equal to 440 Hz. Returns **self**.

transpose:

– **transpose:**(double)*semitones*

Transposes the TuningSystem by *semitones* half-steps. (The half-step used here is always 100 cents.) If *semitones* is positive, the transposition is up, if it's negative, the transposition is down. You can transpose the object by increments smaller than a half-step by supplying a fractional argument. Returns **self**.

UnitGenerator

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

UnitGenerators are the building blocks of DSP music synthesis. Each UnitGenerator subclass represents a DSP program called a *unit generator* that provides a particular synthesis operation, such as waveform generation, filtering, and mixing. Sound is synthesized by downloading unit generators to the DSP, interconnecting them, and making them run. You can allocate and use UnitGenerators directly in your application, but they're most commonly allocated—and more easily controlled—as part of the design of a SynthPatch subclass, as explained in the SynthPatch class description.

To download a copy of a particular unit generator to the DSP, you send the **allocUnitGenerator:** message to an open Orchestra object, passing the class of the UnitGenerator that represents the unit generator. For example, to download a copy of the unoise unit generator (which generates white noise), you allocate an instance of the UnoiseUG class:

```
/* Create an Orchestra and a variable for the UnitGenerator. */
id anOrch = [[Orchestra alloc] init];
id aNoise;
/* Open the Orchestra; check for failure. */
if (![anOrch open])
. . .

/* The UnitGenerator object is created at the same time that the
 * unit generator program is download to the DSP.
 */
aNoise = [anOrch allocUnitGenerator:[UnoiseUGx class]];
```

Notice that the receiver of the **class** message in the final line of the example is UnoiseUGx. The "x" is explained later in this class description.

To connect two UnitGenerators together, you allocate a *patchpoint* through which they can communicate. A patchpoint is a type of SynthData object that's designed to be used for just this purpose, to communicate data from the output of one UnitGenerator to the input of another. For example, to connect our UnoiseUGx object to a sound-output UnitGenerator, such as Out1aUGx, a patchpoint must be allocated and then passed as the argument in an invocation of UnoiseUGx's **setOutput:** method and Out1aUGx's **setInput:** method. But in order to do this, you have to understand a little bit about DSP memory spaces.

The DSP's memory is divided into three sections, P, X, and Y: P memory holds program data; X and Y contain data. Unit generator programs are always downloaded

to P memory; the memory represented by a SynthData object is allocated in either X or Y, as the argument to Orchestra's **allocSynthData:** method is MK_xData or MK_yData. In general, there's no difference between the two data memory spaces; dividing data memory into two partitions allows the DSP to be used more efficiently.

Each of a UnitGenerator's inputs and outputs (or *memory arguments*) are represented by either an "x" or a "y" at the end of its name, indicating the memory space from which or to which the UnitGenerator reads or writes data. Thus, the UnoiseUGx object used above writes its output data to X memory. Similarly, the Out1aUGx object reads from X memory. Therefore, the patchpoint that connects them should be allocated in X memory.

The UnoiseUGx and Out1aUGx object used in the example are called *leaf* classes. They inherit from the *master* classes UnoiseUG and Out1aUG. For each master class, a complete set of leaf classes are created such that every permutation of X and Y memory is provided. For example, in addition to UnoiseUGx, there is also a UnoiseUGy leaf class. Some UnitGenerators have more than one memory argument; an OscgafiUG object (an oscillator), for example, has three inputs and one input. To accommodate 4 memory arguments, 16 permutations of X and Y memory are possible, so there are 16 leaf classes of the OscgafiUG master class. Both the master class and the complete set of leaf classes are created automatically from a unit generator source through the **dspwrap** utility.

You never create an instance of a master class; UnitGenerator objects are always instances of leaf classes.

Most of the methods defined in the UnitGenerator class are subclass responsibilities or are provided to help define the functionality of a subclass. The most important of these are **runSelf**, **idleSelf**, and **finishSelf**. These methods implement the subclass-specific behavior of the object in response to the **run**, **finish**, and **idle** messages, respectively. In addition to implementing the subclass responsibility methods, you should also provide methods for poking values into the memory arguments of the DSP unit generator that the UnitGenerator represents. For example, an oscillator UnitGenerator would provide a **setFreq:** method to set the frequency of the unit generator that's running on the DSP.

The UnitGenerator master classes are listed and described in a separate section. The descriptions include a list of the UnitGenerator's memory arguments and how they correspond to the leaf class names. The unit generator programs provided by NeXT are given as source code in **/usr/lib/dsp/ugsrc**.

INSTANCE VARIABLES

Inherited from Object	Class	isa;

Declared in UnitGenerator	id	synthPatch;
	id	orchestra;
	BOOL	isAllocated;
	MKUGArgStruct	*args;
	MKSynthStatus	status;
	MKOrchMemStruct	relocation;

synthPatch	The SynthPatch that owns this object, if any.
orchestra	The Orchestra on which the object is allocated.
isAllocated	YES if allocated
args	The object's DSP memory arguments
status	The object's status
relocation	The object's relocation information

METHOD TYPES

Designing a UnitGenerator subclass	– setAddressArg:to: – setAddressArg:toInt: – setAddressArgToSink: – setAddressArgToZero: + enableErrorChecking: + shouldOptimize: – setDatumArg:to: – setDatumArg:toLong: – init – moved – runSelf – finishSelf – freeSelf – idleSelf
Using a UnitGenerator	– run – finish – idle – dealloc

Querying the object	+ argCount – argCount + argName: + argSpace: + classInfo – classInfo – isAllocated – isFreeable + masterUGPtr – orchestra – referenceCount – relocation – resources – runsAfter: – status – synthPatch

CLASS METHODS

alloc

You never invoke this method; it's overridden to generate an error. To create a UnitGenerator, you must allocate it through an Orchestra object.

allocFromZone:

You never invoke this method; it's overridden to generate an error. To create a UnitGenerator, you must allocate it through an Orchestra object.

argCount

+ (unsigned)**argCount**

Returns the number of memory arguments as declared by the UnitGenerator subclass' DSP unit generator source code.

argName:

+ (char *)**argName:**(unsigned)*argNum*

Returns a pointer to the name of the UnitGenerator subclass' *argNum*'th memory argument, as declared in the DSP unit generator source code.

argSpace:

+ (DSPMemorySpace)**argSpace:**(unsigned)*argNum*

Returns the memory space to or from which the address-valued argument *argNum* reads or writes. If *argNum* isn't an address-valued argument, returns DSP_MS_N.

classInfo

+ (MKLeafUGStruct *)**classInfo**

Returns the leaf class structure defined by the UnitGenerator subclass. A subclass responsibility, this method is automatically generated by **dspwrap**.

copy

You never invoke this method; it's overridden to generate an error. To create a UnitGenerator, you must allocate it through an Orchestra object.

copyFromZone:

You never invoke this method; it's overridden to generate an error. To create a UnitGenerator, you must allocate it through an Orchestra object.

enableErrorChecking:

+ **enableErrorChecking:**(BOOL)*yesOrNo*

Sets whether various error checks are done, such as verifying that UnitGenerator arguments are correct. The default is NO. You should send **enableErrorChecking:YES** only when you are debugging.

masterUGPtr

+ (MKMasterUGStruct *)**masterUGPtr**

Returns the master class structure defined by the UnitGenerator subclass. A subclass responsibility, this method is automatically generated by **dspwrap**.

shouldOptimize:

+ (BOOL)**shouldOptimize:**(unsigned)*arg*

A subclass can override this method to reduce the command stream on an argument-by-argument basis, returning YES if *arg* should be optimized, NO if it shouldn't. The default implementation always returns NO.

Optimization of an argument means that if the argument is set to the same value twice, the second setting is suppressed. You should never optimize an argument that the unit generator DSP code itself might change.

Argument optimization applies to the entire class—all instances of the UnitGenerator's leaf classes inherit an argument's optimization. Optimization of an argument can't be changed while the UnitGenerator is in use.

INSTANCE METHODS

argCount

– (unsigned)**argCount**

Returns the number of memory arguments defined by the UnitGenerator. The same value is returned by the **argCount** class method.

classInfo

– (MKLeafUGStruct *)**classInfo**

Returns a pointer to the UnitGenerator's leaf structure. The same structure pointer is returned by the **classInfo** class method.

dealloc

– **dealloc**

Deallocates the UnitGenerator and frees its SynthPatch, if any. Returns **nil**.

finish

– (double)**finish**

Finishes the UnitGenerator's activity by sending **finishSelf** and then sets its status to MK_finishing. You never subclass this method; **finishSelf** provides subclass finishing instructions. Returns the value of **[self finishSelf]**, which is taken as the amount of time, in seconds, before the UnitGenerator can be idled.

finishSelf

– (double)**finishSelf**

A subclass may override this method to provide instructions for finishing. Returns the amount of time needed to finish; the default returns 0.0.

free

– **free**

Only the Orchestra may free a UnitGenerator. This method is overridden to do nothing.

freeSelf

– **freeSelf**

You never invoke this method directly, it's invoked automatically when a UnitGenerator is freed by the Orchestra. A subclass may implement this method to provide specialized behavior.

idle

– **idle**

Idles the UnitGenerator by sending **[self idleSelf]** and then sets its status to MK_idle. You never subclass this method; **idleSelf** provides subclass idle instructions. When a UnitGenerator is idle, it produces no output signal.

idleSelf

– **idleSelf**

A subclass may override this method to provide instructions for idling. The default does nothing and returns the receiver. Most UnitGenerator subclasses implement **idleSelf** to patch their outputs to sink, a location that nobody reads. UnitGenerators that have inputs, such as Out2sumUG, implement **idleSelf** to patch their inputs to zero, a location that always holds the value 0.0.

init

– **init**

You never explicitly create a UnitGenerator. Therefore, you never invoke this method; it's sent when the UnitGenerator is created by the Orchestra, after its DSP code is loaded. If this method returns **nil**, the UnitGenerator is automatically freed by the Orchestra. A subclass implementation should send **[super init]** before doing its own initialization and should immediately return **nil** if **[super init]** returns **nil**. The default implementation returns **self**.

isAllocated

– (BOOL)**isAllocated**

Returns YES if the UnitGenerator has been allocated (by its Orchestra), NO if it hasn't.

isFreeable

– (BOOL)**isFreeable**

Invoked by the Orchestra to determine whether the UnitGenerator may be freed. Returns YES if it can, NO if it can't. (A UnitGenerator can be freed if it isn't currently allocated or if its SynthPatch can be freed.)

moved

– **moved**

You never invoke this method. It's automatically invoked by the Orchestra if the UnitGenerator is moved during compaction. A subclass can override this method to perform specialized behavior. The default does nothing; the return value is ignored.

orchestra

– **orchestra**

Returns the Orchestra object on which the UnitGenerator is allocated.

referenceCount

– (int)**referenceCount**

If the UnitGenerator is installed in its Orchestra's shared object this table, returns the number of objects that are using it. Otherwise returns 1 if it's allocated, 0 if not.

relocation

– (MKOrchMemStruct *)**relocation**

Returns a pointer to the structure that describes the UnitGenerator's location on the DSP. You can access the fields of the structure without caching it, for example:

```
[aUnitGenerator relocation]->pLoop
```

returns the starting location of the receiver's pLoop code.

resources

– (MKOrchMemStruct *)**resources**

Return a pointer to the structure that describes the UnitGenerator's memory requirements. Each field of the structure represents a particular Orchestra memory segment; its value represents the number of words that the segment requires.

run

– **run**

Starts the UnitGenerator by sending **[self runSelf]** and then sets its status to MK_running. You never subclass this method, you implement **runSelf** to provide subclass instructions. A UnitGenerator must be sent **run** before it can be used.

runSelf

– **runSelf**

A subclass implementation of this method provides instructions for making the UnitGenerator's DSP code run. You never invoke this method directly, it's invoked automatically by the **run** method. The default does nothing and returns the receiver.

runsAfter:

– (BOOL)**runsAfter:***aUnitGenerator*

Returns YES if the UnitGenerator is executed after *aUnitGenerator*. Execution order is determined by comparing the objects' pLoop addresses.

setAddressArg:to:

– **setAddressArg:**(unsigned)*argNum* **to:***memoryObj*

Sets the address-valued argument indexed by *argNum* to *memoryObj*. If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

setAddressArg:toInt:

– **setAddressArg:**(unsigned)*argNum* **toInt:**(int)*dspAddress*

Sets the address-valued argument indexed by *argNum* to *dspAddress* in DSP memory. If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

setAddressArgToSink:

– **setAddressArgToSink:**(unsigned)*argNum*

Sets the address-valued argument indexed by *argNum* to the sink patchpoint. (The sink patchpoint is a location which, by convention, is never read.) If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

setAddressArgToZero:

– **setAddressArgToZero:**(unsigned)*argNum*

Sets the address-valued argument *argNum* to the zero patchpoint. (The zero patchpoint is a location which, by convention, is never written.) If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

setDatumArg:to:

– **setDatumArg:**(unsigned)*argNum* **to:**(DSPDatum)*val*

Sets the datum-valued argument indexed by *argNum* to *val*. If *argNum* is an L-space argument (two 24-bit words), its high-order word is set to *val* and its low-order word is cleared. If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

setDatumArg:toLong:

– **setDatumArg:**(unsigned)*argNum* **toLong:**(DSPLongDatum *)*val*

Sets the datum-valued argument *argNum* to *val*. If *argNum* isn't an L-space argument (it can't accommodate a 48-bit value) its value is set to the high 24-bits of *val*. If *argNum* is out of bounds, an error is generated and **nil** is returned. Otherwise returns **self**. This is ordinarily only invoked in the implementation of a subclass.

status

– (int)**status**

Returns the UnitGenerator's status, one of MK_idle, MK_running, and MK_finishing. You never set the status directly. A newly allocated UnitGenerator is idle; its status changes automatically as it receives the **run**, **finish**, and **idle** messages.

synthPatch

– **synthPatch**

Returns the UnitGenerator's SynthPatch, if any.

WaveTable

INHERITS FROM Object

DECLARED IN musickit.h

CLASS DESCRIPTION

A WaveTable represents a single period of a sound waveform as a series of samples. WaveTable is an abstract class that's succeeded by two inheriting classes: Samples and Partials. The Samples subclass lets you define a WaveTable through association with a Sound object or soundfile; Partials lets you build a waveform by adding sine wave components. If you're interested in using WaveTables to create a library of timbres you should refer to the descriptions of the Samples and Partials subclasses. Detailed familiarity with the WaveTable class, in this case, isn't necessary.

WaveTable objects are designed to be used as *lookup tables* for oscillator UnitGenerators such as OscgafiUG. When it's instructed to run, the oscillator downloads the WaveTable's data to a portion of memory on the DSP and then cycles over the data to generate a timbre that's defined by the shape of the waveform that the data represents. To assist this process, a WaveTable object maintains two separate arrays of data pointed to by the **dataDSP** and **dataDouble** instance variables:

- **dataDSP** contains values of type DSPDatum, the type used by the Music Kit to represent the DSP's 24-bit fixed-point format. It's this array that's downloaded to the DSP by an oscillator.

- **dataDouble** contains doubles. It's provided as means for representing WaveTable data on the host without the loss of precision implied by the DSPDatum type.

Subclasses of WaveTable are responsible for filling at least one of these arrays with data; the values in both arrays are assumed to be within the range (–1.0, 1.0). The mechanism for filling the chosen array is defined by the subclass in its implementation of **fillTableLength:scale:**. Which array to fill (or whether to fill both) is at the discretion of the subclass designer. For example, the Partials subclass fills the **dataDouble** array only; Samples, on the other hand, fills both arrays.

The **fillTableLength:scale:** method is never invoked directly; instead, it's invoked as needed when a WaveTable object receives a request for its data. WaveTable defines two fundamental methods, **dataDSP** and **dataDouble**, that return pointers to their namesake arrays. Additional methods let you scale and size the data:

- **dataDSPScale:** lets you specify, as a double, the amplitude scaling factor of the dataDSP array.

- **dataDSPLength:** lets you specify the length, in samples, of the dataDSP array.

- **dataDSPLength:scale:** scales and sizes the dataDSP array.

An analogous set of methods scales and sizes the **dataDouble** array. Conversion between the **dataDSP** and **dataDouble** arrays is provided by these methods; for example, if you invoke one of the **dataDSP** retrieval methods before the array has been filled with data, the method automatically fills **dataDSP** with data converted from the **dataDouble** array. If neither array has been filled, **fillTableLength:scale:** is invoked.

WaveTables are usually used as parameter values in Note objects as set through Note's **setPar:toWaveTable:** method. The Music Kit defines a number of parameters that take WaveTables as values.

INSTANCE VARIABLES

Inherited from Object	Class	isa;
Declared in WaveTable	int	length;
	double	scaling;
	DSPDatum	*dataDSP;
	double	*dataDouble;

length	Length of the data arrays, in elements (samples).
scaling	Amplitude scaling factor; 0.0 indicates normalization.
dataDSP	Array of 24-bit fixed-point data.
dataDouble	Array of double-precision floating-point data.

METHOD TYPES

Creating a WaveTable instance	– copy – free
Modifying the instance	– init
Querying the instance	– length – scaling
Computing the waveform	– fillTableLength:scale:
Retrieving data	– dataDSP – dataDSPLength: – dataDSPLength:scale: – dataDSPScale: – dataDouble – dataDoubleLength: – dataDoubleLength:scale: – dataDoubleScale:

Archiving the instance – read:
– write:

INSTANCE METHODS

copy

– **copy**

Creates and returns a new WaveTable as a copy of the receiver

dataDSP

– (DSPDatum *)**dataDSP**

Returns a pointer to the receiver's **dataDSP** array. Implemented as an invocation of **dataDSPLength:scale:**, with the **length** and **scaling** instance variables as arguments.

dataDSPLength:

– (DSPDatum *)**dataDSPLength:**(int)*aLength*

Returns a pointer to the receiver's **dataDSP** array. Implemented as an invocation of **dataDSPLength:scale:**, with *aLength* and the **scaling** instance variable as arguments.

dataDSPLength:scale:

– (DSPDatum *)**dataDSPLength:**(int)*aLength* **scale:**(double)*aScaling*

Returns a pointer to the receiver's **dataDSP** array, recomputing the data if necessary (as defined in the class description). The array is sized and scaled according to the arguments and the **length** and **scaling** instance variables are set to these values. If the receiver can't fill the array, NULL is returned. You should neither modify nor free the data returned by this method.

dataDSPScale:

– (DSPDatum *)**dataDSPScale:**(double)*aScaling*

Returns a pointer to the receiver's **dataDSP** array. Implemented as an invocation of **dataDSPLength:scale:**, with the **length** instance variable and *aScaling* as arguments.

dataDouble

– (double *)**dataDouble**

Returns a pointer to the receiver's **dataDouble** array. Implemented as an invocation of **dataDoubleLength:scale:**, with the **length** and **scaling** instance variables as arguments.

dataDoubleLength:

– (double *)**dataDoubleLength:**(int)*aLength*

Returns a pointer to the receiver's **dataDouble** array. Implemented as an invocation of **dataDoubleLength:scale:**, with *aLength* and the **scaling** instance variable as arguments.

dataDoubleLength:scale:

– (double *)**dataDoubleLength:**(int)*aLength* **scale:**(double)*aScaling*

Returns a pointer to the receiver's **dataDouble** array, recomputing the data if necessary (as defined in the class description). The array is sized and scaled according to the arguments and the **length** and **scaling** instance variables are set to these values. If the array can't be filled, NULL is returned. You should neither modify nor free the data returned by this method.

dataDoubleScale:

– (double *)**dataDoubleScale:**(double)*aScaling*

Returns a pointer to the receiver's **dataDouble** array. Implemented as an invocation of **dataDoubleLength:scale:**, with the **length** instance variable and *aScaling* as arguments.

fillTableLength:scale:

– **fillTableLength:**(int)*aLength* **scale:**(double)*aScaling*

Computes the receiver's data, sizing and scaling according to the arguments. This is a subclass responsibility method; a subclass can implement the method to fill the **dataDSP** array, the **dataDouble** array, or both. If only one of the arrays is computed and filled, the other should be freed and its pointer set to NULL. If the data can't be computed, both arrays should be freed and **nil** returned. Otherwise, the receiver should be returned.

free

– **free**

Frees **dataDSP** and **dataDouble**, and then frees the receiver itself. This method also removes the receiver's name, if any, from the Music Kit name table.

init

– **init**

Initializes the receiver. If you override this method in a subclass, you should include **[super init]** in the implementation. Returns the receiver.

length

– (int)**length**

Returns the length, in elements, of the data arrays (the two arrays should always contain the same number of elements). A return value of 0 indicates that the arrays haven't been filled, or that the data needs to be recomputed.

scaling

– (double)**scaling**

Returns the factor by which the values (sample amplitudes) in the data arrays are scaled. A return value of 0.0, the default, indicates that the values are normalized, or scaled to fit perfectly within the range –1.0 to 1.0.

Add2UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Add2UG produces the sum of two input signals:

output = *input1* + *input2*

MEMORY SPACES

Add2UG*abc*

a output
b input 1
c input 2

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

Allpass1UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Allpass1UG is a one-pole, one-zero filter. The value of the filter coefficient is set directly. The filter's transfer function is given as

$$H(z) = \frac{bb0 + 1/z}{1 + (bb0)/z}$$

where *bb0* is the filter coefficient. The Allpass1UG filter uses a one-sample delay in its computation.

MEMORY SPACES

Allpass1UG*ab*

a output
b input

INSTANCE METHODS

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **self**, or **nil** if the argument isn't a patchpoint.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **self**, or **nil** if the argument isn't a patchpoint.

setBB0:

– **setBB0:**(double)*bb0*

Sets the filter coefficient to *bb0*. For stability, the coefficient should be within the bounds

$$-1.0 < bb0 < 1.0$$

Returns **self**.

clear

– **clear**

Sets the value of the delay memory to 0.0. Returns **self**.

AsympUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

AsympUG creates an exponential signal that approaches a limit (the "target") at a particular rate, where the rate expresses the proportion of the remaining journey that's taken with each step, where 0.0 is no progress and 1.0 is the whole thing:

output = *previousOutput* + (*rate* * (*target* – *previousOutput*))
previousOutput = *output*

Methods are provided that let you set the rate directly as it would be used in the formula above. You can also set it indirectly as a time limit (referred to as "T60") that defines the amount of time, in seconds, that it should take for the target to be perceptually reached.

AsympUG objects are normally used to provide dynamic scaling of a musical attribute. To this end, the output of an AsympUG is typically connected to the frequency or amplitude input of an OscgafUG or OscgafiUG object. In addition, instances of AsympUG are usually used in association with Envelope objects; the C function **MKUpdateAsymp**() is provided to take care of the rather messy business of setting and managing an AsympUG's attributes according to a given Envelope and a set of Note parameters. By using **MKUpdateAsymp**(), you need only set the AsympUG's output patchpoint; all other methods are invoked for you.

MEMORY SPACES

AsympUG*a*
a output

INSTANCE METHODS

abortEnvelope

– abortEnvelope

Disassociates the AsympUG from its Envelope. If the AsympUG is running, it stops reading breakpoints, although it isn't otherwise interrupted (it continues to follow its current trajectory). Returns **self**.

envelope

– envelope

Returns the Envelope that's associated with the AsympUG, or **nil** if none.

envelopeStatus

– (MKEnvStatus)**envelopeStatus**

Returns the type of the most recently acquired Envelope breakpoint. There are three types of breakpoints: the stickpoint (represented by MK_stickPoint), the final breakpoint in the Envelope (MK_lastPoint), and all other breakpoints (MK_noEnvError). If the AsympUG's Envelope hasn't been set, MK_noMorePoints is returned.

finishSelf

– (double)**finishSelf**

You never invoke this method; it's invoked automatically when the AsympUG receives the **finish** message. However, its behavior bears description: If the object has yet to see or is waiting at its Envelope's stickpoint, this causes it to head for the first breakpoint after the stickpoint, and then on the end of the Envelope. If the AsympUG's Envelope contains no stickpoint, this method is (virtually) ignored.

preemptEnvelope

– preemptEnvelope

Informs the AsympUG that its Envelope is being preempted. This sets the AsympUG's target to the last breakpoint in the Envelope, and sets its T60 value to the global "preempt duration," as set through the **MKSetPreemptDuration()** function (the default preempt duration is 0.006 seconds). This method is invoked automatically by a SynthInstrument object when it preempts a SynthPatch that contains AsympUG objects.

resetEnvelope:yScale:yOffset:xScale:releaseXScale:funcPtr:transitionTime:

– **resetEnvelope:***envelope*
yScale:(double)*yScaleValue*
yOffset:(double)*yOffsetValue*
xScale:(double)*attackXScaleValue*
releaseXScale:(double)*releaseXScaleValue*
funcPtr:(double(*)())*yScaleFunction*
transitionTime:(double)*transition*

This method is the same as the **setEnvelope:...** method but for this difference: If the AsympUG is running, its target is set to the second breakpoint of the new Envelope and T60 is set to *transition* seconds (the Envelope's first breakpoint is ignored). This affords a more graceful transition into the new Envelope. You would normally call the **MKUpdateAsymp()** function rather than invoke this method directly.

setCurVal:

– **setCurVal:***(double)value*

Sets the current value of the AsympUG to *value*, which is first converted to a long (48-bit) DSP word. The new value replaces the previous output sample, as shown in the computation in the class description above. The object is otherwise undisturbed in executing its appointed task. Returns **self**.

setEnvelope:yScale:yOffset:xScale:releaseXScale:funcPtr:

– **setEnvelope:***envelope*
yScale:(double)*yScaleValue*
yOffset:(double)*yOffsetValue*
xScale:(double)*attackXScaleValue*
releaseXScale:(double)*releaseXScaleValue*
funcPtr:(double(*)())*additionalYFunction*

Associates the AsympUG with the given Envelope. When the AsympUG is run, it automatically schedules the breakpoints from its Envelope to be fed to itself through message requests with the clockConductor. If this method is invoked while the AsympUG is running, the object's current value is immediately set to the (scaled and offset) y value of the first breakpoint in the new Envelope. A kinder interruption is afforded by the **resetEnvelope:...** method.

As breakpoints are delivered to an AsympUG, it's x, y, and smoothing values are used to set the AsympUG's target and rate:

- The breakpoint's y value is scaled by *yScaleValue* and offset by *yOffsetValue*. If you don't specify an *additionalYFunction*, this scaled and offset y value is set as the AsympUG's target. Otherwise, this value is passed as a **double** to *additionalYFunction*, an optional function of your own creation that you can provide to perform additional manipulation of the y value. The function takes two

arguments: the scaled and offset y value as mentioned above and the AsympUG's **id**. The **double** value returned by *additionalYFunction* is set as the AsympUG's target.

- The previous breakpoint's x value is subtracted from this breakpoint's x value and the difference is scaled either by *attackXScaleValue*, if the Envelope's stickpoint has not yet been met, or by *releaseXScaleValue*; it is then further scaled by the breakpoint's smoothing value. This doubly scaled delta value is then set as the AsympUG's T60 time limit.

Since a breakpoint dissolves into a target and a rate, and since these two values imply a passage of time into the future, breakpoints must be fed to the AsympUG one breakpoint in advance.

Normally, you would call the **MKUpdateAsymp()** function rather than invoke this method. The function provides a slightly easier interface to AsympUG management.

If *envelope* isn't an Envelope, **nil** is returned. In addition, if *envelope* is **nil**, the current Envelope, if any, is aborted. Otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **self**, or **nil** if the argument isn't a patchpoint.

setRate:

– **setRate:**(double)*rate*

Sets the rate at which the AsympUG approaches its target, where *rate* is the proportion (on a scale between 0.0 and 1.0, but see below) of the remaining journey that's stepped off at each sample. The T60 value (the amount of time it takes to virtually reach the target) that corresponds to a particular rate depends on the sampling rate. If the AsympUG is running, the new target is approached starting from the object's previous sample. Returns **self**.

Note: While the scale is reckoned between 0.0 and 1.0, the actual maximum value to which the rate can be set, for historical reasons, is 0.125. The maximum rate of 0.125 translates to a T60 of about 0.0014 seconds at the low sampling rate, or 0.0007 seconds at the high sampling rate.

See also: **– setT60:**

setReleaseXScale:

– **setReleaseXScale:**(double)*releaseXScaleValue*

Resets the value by which the release time of the AsympUG's Envelope is scaled. This only has an affect on subsequent breakpoints—you can't, for example, extend the life of an AsympUG by increasing its release scale after the object has read (and is heading for) its last breakpoint. Returns **self**.

setTargetVal:

– **setTargetVal:**(double)*target*

Sets the target to *target*, which should be between 0.0 and 1.0. If the AsympUG is running, the new target is approached starting from the object's previous output sample—in other words, the new target is absorbed into the equation given in the class description above, without affecting the other factors. Returns **self**.

setT60:

– **setT60:**(double)*seconds*

Computes the AsympUG's rate such that the target is perceptually reached in *seconds* seconds. Because of an idiosyncracy in the mechanism that sets the AsympUG's rate, the smallest (fastest) T60 value that you can specify is about 0.0014 if you're running at the low sampling rate and about 0.0007 at the high sampling rate. This restriction is performed automatically; it isn't an error to request smaller T60 values. Returns **self**.

See also: **– setRate:**

setYScale:yOffset:

– **setYScale:**(double)*yScaleValue* **yOffset:**(double)*yOffsetValue*

Resets the values by which the AsympUG scales and offsets its Envelope's y values. If the object is running, its current value is immediately modified. Returns **nil** if the AsympUG has no Envelope (and the current value isn't modified), otherwise returns **self**.

ConstantUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

ConstantUG produces a constant value. Since you can set the value of a patchpoint directly through the SynthData method **setToConstant:**, you rarely need instances of this class. However, a ConstantUG object can be used to initialize, on each tick, a constant-valued patchpoint that may have been written to during the previous tick. For example, you can implement additive synthesis by creating a patch in which each oscillator reads a patchpoint, adds its own signal into the value, and then writes the sum back to the same patchpoint in preparation for the next oscillator. In this case, you would use a ConstantUG to clear the patchpoint before the first oscillator reads it.

MEMORY SPACES

ConstantUG*a*
a output

INSTANCE METHODS

setConstant:

– **setConstant:**(double)*value*

Sets the constant value to a DSPDatum converted from *value*. Returns **self**.

setConstantDSPDatum:

– **setConstantDSPDatum:**(DSPDatum)*value*

Sets the constant value to *value*. Returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

DelayUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

DelayUG delays its input signal by some number of samples. It requires a SynthData object to store the delayed signal. Each DelayUG maintains a single pointer into the delay memory. When the object is run, a tick's worth of samples are read and replaced with an equal number of samples from the input signal. The pointer is then incremented by a tick. When the pointer reaches the end of the delay memory, it automatically jumps back to the beginning, even if it's in the middle of a tick—in other words, the length of the delay memory *needn't* be a multiple of the tick size. The rate at which the pointer is incremented can't be modified, nor can you offset the beginning of the delay memory. However, you can reposition the pointer to any arbitrary sample in the delay memory through the **setPointer:** method.

MEMORY SPACES

DelayUG*abc*

a output
b input
c delay memory

INSTANCE METHODS

adjustLength:

– **adjustLength:**(int)*delayLength*

Sets the number of delayed samples to *delayLength*. The argument must be no greater than the length of the SynthData object that's used as the delay memory. Returns **nil** if *delayLength* is too great or if the delay memory hasn't been set; otherwise returns **self**.

By default, the length of the delay is that of the SynthData that's used as the delay memory. Decreasing the delay length of a running DelayUG doesn't free (nor does it clear) the fallow memory, which is always taken from the end of the SynthData. Keep in mind that decreasing the length may cause the pointer to be considered out of bounds; to avoid this, you should send a **resetPointer** (or **setPointer:**) message to the DelayUG just before you invoke this method.

Before increasing the length of the delay memory, you may want to clear the recommissioned portion by sending a **setToConstant:length:offset:** message to the SynthData.

length

– (int)**length**

Returns the number of samples in the delay memory. Note that this is the length that's currently being used; it isn't necessarily the same as the length of the SynthData that's being used as the delay memory.

resetPointer

– **resetPointer**

Resets the pointer to the beginning of the delay memory. Returns **nil** if the SynthData hasn't been set; otherwise returns **self**.

setDelayMemory:

– **setDelayMemory:***aSynthData*

Sets the SynthData object used as the delay memory to *aSynthData*. The length of the SynthData must be greater than or equal to the amount of delay (in samples) that's desired. If *aSynthData* is **nil**, the delay memory is set to the sink location. Returns **self**.

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setPointer:

– **setPointer:**(int)*n*

Repositions the pointer to point to the *n*'th sample in the delay memory, counting from sample 0. Returns **nil** if *n* is greater than the current length of the delay, or if the delay memory hasn't been set; otherwise returns **self**.

DswitchtUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

DswitchtUG reads a specified number of ticks from its first input signal and then switches to read its second input signal. You can cause a DswitchtUG to switch between its two inputs any number of times while it's running. The input signals can be independently scaled. The input patchpoints must be allocated in the same memory space.

A similar class, DswitchUG, switches on a sample boundary and doesn't allow scaling on the second input.

MEMORY SPACES

DswitchtUG*ab*

a output
b input1 and input2

INSTANCE METHODS

setDelayTicks:

– **setDelayTicks:**(int)*count*

Immediately switches the DswitchtUG to its first input and causes it to switch to its second input after *count* ticks have been read. If *count* is less than or equal to zero, the switch to the second input is performed immediately. If the object is currently reading from its first input because of a previous invocation of this method, the old *count* is superceded by the new one. Returns **self**.

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale1:

– **setScale1:**(double)*scale*

Sets the factor by which the first input signal is scaled. Returns **self**.

setScale2:

– **setScale2:**(double)*scale*

Sets the factor by which the second input signal is scaled. Returns **self**.

DswitchUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

DswitchUG reads a specified number of samples from its first input signal and then switches to read its second input signal. You can cause a DswitchUG to switch between its two inputs any number of times while it's running. A scaler on the first input signal is provided. The input patchpoints must be allocated in the same memory space.

A similar class, DswitchtUG, allows scaling on both signals but restricts the timing of the switch to a tick boundary.

MEMORY SPACES

DswitchUG*ab*

a output
b input1 and input2

INSTANCE METHODS

setDelaySamples:

– **setDelaySamples:**(int)*count*

Immediately switches the DswitchUG to its first input and causes it to switch to its second input after *count* samples have been read. If *count* is less than or equal to zero, the switch to the second input is performed immediately. If the object is currently reading from its first input because of a previous invocation of this method, the old *count* is superceded by the new one. Returns **self**.

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale1:

– **setScale1:**(double)*scale*

Sets the factor by which the first input signal is scaled. Returns **self**.

InterpUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

InterpUG provides dynamic linear interpolation between two input signals, where the interpolation is controlled by a third input signal:

output = *input1* + ((*input2* − *input1*) * *input3*)

When the value of *input3* is 0.0, the output of InterpUG is exactly the signal found at *input1*. When *input3* is 1.0, the output is exactly *input2*. An AsympUG is often used to produce the control signal.

MEMORY SPACES

InterpUG*abcd*

a output
b input1
c input2
d input3 (interpolation control)

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput3:

– **setInput3:***aPatchpoint*

Sets the input 3 patchpoint to *aPatchpoint*. The signal from this input controls the interpolation between the other two input signals. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

Mul1add2UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Mul1add2UG adds one signal to the product of two others:

output = *input1* + (*input2* * *input3*)

MEMORY SPACES

Mul1add2UG*abcd*

a output
b input1
c input2
d input3

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput3:

– **setInput3:***aPatchpoint*

Sets the input 3 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

Mul2UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Mul2UG multiplies two signals:

*output = input1 * input2*

MEMORY SPACES

Mul2UG*abc*

a output
b input1
c input2

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

OnepoleUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

OnepoleUG is a one-pole filter that's implemented by subtracting the previous output sample (initialized as 0.0) from the current input sample:

output = (*b0* * *input*) – (*a1* * *previousOutput*)
previousOutput = *output*

Note that the two samples have their own scalers:

- *b0*, the filter's gain, scales the input sample. Effective gain values are between 0.0 and 1.0 (a negative gain is the same as its absolute value, but with a 180-degree phase shift).

- *a1*, the filter's coefficient, scales the previous output sample. If *a1* is less than 0.0, the OnepoleUG is a low-pass filter; if it's greater than 0.0, the object is a high-pass filter. For stability, the value of *a1* should be between –1.0 and 1.0 (noninclusive).

Similar to the OnepoleUG is the OnezeroUG; it, too, is either a low-pass or a high-pass filter, but the frequency roll-off is gentler than with a OnepoleUG. You should also note that the high-pass/low-pass determination with regard to the sign of the coefficient is switched in the OnezeroUG.

MEMORY SPACES

OnepoleUG*ab*

a output
b input

INSTANCE METHODS

clear

– clear

Clears the filter by setting the delayed sample (the previous output sample) to 0.0. Returns **self**.

setA1:

– **setA1:***(double)value*

Sets the filter's coefficient. If *value* is less than 0.0, the OnepoleUG is a low-pass filter; if it's greater than 0.0, the object is a high-pass filter. For stability, the *value* should be between –1.0 and 1.0. Returns **self**.

setB0:

– **setB0:***(double)value*

Sets the filter's gain. Effective gain values are between 0.0 and 1.0 (a negative gain is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

setBrightness:forFreq:

– **setBrightness:**(double)*brightness* **forFreq:**(double)*frequency*

This is a convenient method that adjusts the filter's gain and coefficient such that a constant *brightness* value produces the same number and relative amplitudes of a tone's harmonics regardless of the value of *frequency*. For example, in a musical phrase during which the brightness of the synthesized notes shouldn't be perceived to change, you would invoke this method once per note passing a constant *brightness* value (the successive *frequency* values would, of course, be determined by the pitches of the notes). Returns **self**.

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

OnezeroUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

OnezeroUG is a one-zero filter that's implemented by adding the previous input sample (initialized as 0.0) to the current input sample:

output = (*b0* * *input*) + (*b1* * *previousInput*)
previousInput = *input*

Note that the two samples have their own scalers:

- *b0* scales the input sample; this is the gain of the filter. Effective gain values are between 0.0 and 1.0 (a negative gain is the same as its absolute value, but with a 180-degree phase shift).

- *b1* scales the previous input sample. This is the filter's coefficient: If *b1* is less than 0.0, the OnezeroUG is a high-pass filter; if it's greater than 0.0, the object is a low-pass filter. For stability, the value of *b1* should be between –1.0 and 1.0 (noninclusive).

Similar to the OnezeroUG is the OnepoleUG; it, too, is either a low-pass or a high-pass filter, but the frequency roll-off is steeper than with a OnezeroUG. You should also note that the high-pass/low-pass determination with regard to the sign of the coefficient is switched in the OnepoleUG.

MEMORY SPACES

OnezeroUG*ab*

a output
b input

INSTANCE METHODS

clear

– clear

Clears the filter by setting the delayed sample (the previous input sample) to 0.0. Returns **self**.

setB0:

– **setB0:**(double)*value*

Sets the filter's gain. Effective gain values are between 0.0 and 1.0 (a negative gain is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

setB1:

– **setB1**:(double)*value*

Sets the filter's coefficient. If *value* is less than 0.0, the OnezeroUG is a high-pass filter; if it's greater than 0.0, the object is a low-pass filter. For stability, the *value* should be between –1.0 and 1.0. Returns **self**.

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

OscgafUG, OscgafiUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

OscgafUG and OscgafiUG are oscillators that allow signal control of amplitude and frequency. OscgafUG and OscgafiUG objects operate much like the simpler OscgUG oscillator: They produce a signal that's created by looking up values in a wavetable. You define the wavetable by associating an Oscgaf(i)UG with a WaveTable or, more directly, with a SynthData object. Alternatively, you can use the DSP's sineROM, a read-only section of Y memory that holds one period of a sine wave. Note well that since the sineROM resides in Y memory, you must allocate an Oscgaf(i)UG that reads Y memory for its wavetable input.

Amplitude control is straightforward: The values that are gotten out of the wavetable are scaled by the signal that arrives through the amplitude input patchpoint. Typically, the patchpoint is written to by an Envelope-handler UnitGenerator (the Music Kit provides the AsympUG class for this task).

Frequency control is a bit more complicated. The signal that arrives at the frequency input patchpoint is taken as the size of the steps, or "phase increment," with which the Oscgaf(i)UG walks through its wavetable: The larger the increment, the higher the frequency. The frequency input patchpoint, like that for amplitude, is often written to by an AsympUG; it can also be written to by another oscillator to create frequency modulation (fm). Since you can't set the frequency of an Oscgaf(i)UG directly, a handy method, **incAtFreq:**, is provided to return the phase increment value that corresponds to a given frequency. In addition, you can provide a ratio by which an Oscgaf(i)UG's phase increment is scaled; this is particularly convenient if you're using, for example, a single AsympUG to control the frequency of more than one oscillator, but you want each oscillator to have its own frequency.

OscgafUG doesn't interpolate between the samples in its wavetable, thus the fidelity of the waveform that it produces is somewhat crude. OscgafiUG provides interpolation, and so produces a higher-fidelity signal, but it requires more DSP resources than does an OscgafUG. Other than in the use of interpolation, the two oscillators are identical.

MEMORY SPACES

OscgafUG*abcd*, OscgafiUG*abcd*

a output
b amplitude input
c phase increment input (phase increment controls frequency)
d wavetable input

incAtFreq:

– (double)**incAtFreq:**(double)*frequency*

Returns the increment that corresponds to *frequency*. The Oscgaf(i)UG's wavetable must be set before you invoke this method; returns 0.0 if it isn't set. You would use the value returned by this method to set the amplitude of the signal produced by the UnitGenerator that writes to the phase increment patchpoint. If you're not interested in frequency control—if you're using an Oscgaf(i)UG for amplitude control only—then you would set the phase increment patchpoint to this value directly.

incRatio

– (double)**incRatio**

Returns the oscillator's phase increment ratio, as set through **setIncRatio:**. The default is 1.0.

setAmpInput:

– **setAmpInput:**(double)*aPatchpoint*

Sets the amplitude input patchpoint to *aPatchpoint*. The values that are read from this patchpoint are used to scale the values that are gotten from the oscillator's wavetable. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setIncInput:

– **setIncInput:**(double)*aPatchpoint*

Sets the phase increment input patchpoint to *aPatchpoint*. The values that are read from this patchpoint are taken as the size of the steps with which the oscillator walks through its wavetable. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setIncRatio:

– **setIncRatio:**(double)*factor*

Sets the factor by which the Oscgaf(i)UG's phase increment is scaled. The default is 1.0. Returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setPhase:

– **setPhase:**(double)*phase*

Sets the Oscgaf(i)UG's initial phase to *phase*, specified as degrees. Returns **self**.

setTable:

– **setTable:***anObject*

Sets the Oscgaf(i)UG's wavetable by invoking **setTable:length:** with a default value as the second argument. Currently the default table length is 256 for a Partials object, and the sample count (length) for a Samples or SynthData object. Returns the value returned by **setTable:length:**.

setTable:defaultToSineROM:

– **setTable:***anObject* **defaultToSineROM:**(BOOL)*useSine*

This is the same as **setTable:** but with an extra Boolean argument, *useSine*, that indicates whether you want to use the DSP sineROM if the resources to accommodate *anObject* aren't available (or if *anObject* is **nil**). Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **self** if a wavetable is acquired (whether it's the sineROM or *anObject*); otherwise returns **nil**.

See also: – **setTable:**, – **setTableToSineROM**

setTable:length:

– **setTable:***anObject* **length:**(int)*sampleCount*

Sets the Oscgaf(i)UG's wavetable to *anObject*—possibly allocating DSP memory to accommodate the table, as described below—and sets the (maximum) length of the wavetable to *sampleCount*, which must be a power of two. Returns **nil** if *anObject* is **nil**, if sufficient DSP memory isn't available to allocate storage for the requested wavetable, or if *sampleCount* isn't a power of two; otherwise returns **self**.

The *anObject* argument can either be a SynthData or a WaveTable object. If you use a WaveTable, DSP memory is allocated for you and made available to all other oscillators through Orchestra's shared object table. Thus, if you send **setTable:length:** to two different oscillators specifying the same WaveTable and length in the two messages, the first message will cause memory to be allocated while the second message will share this memory. Sharing WaveTables between oscillators helps conserve DSP resources, but you should be aware that the sharing mechanism makes it difficult to change the data in a WaveTable and have it affect the oscillator(s) with which it's associated. The most reliable way to change an oscillator's WaveTable is to create a new WaveTable object and set it through this method.

In addition, if you set the wavetable as a WaveTable object, the length that you request as *sampleCount* may not be the length of the wavetable memory that's actually

allocated. If sufficient resources aren't available, the requested length is repeatedly halved until it fits, with a minimum table length of 64 samples. (Note that the 64-sample limit is an imposition only on this halving mechanism—you can pass as *sampleCount* a value that's less than 64 without falling ill of the law.)

See also: – **setTable:**, – **setTable:length:defaultToSineROM:**, – **tableLength**

setTable:length:defaultToSineROM:

– **setTable:***anObject*
length:(int)*sampleCount*
defaultToSineROM:(BOOL)*useSine*

This is the same as **setTable:length:** but with an extra Boolean argument, *useSine*, that indicates whether you want to use the DSP sineROM if the resources to accommodate *anObject* aren't available (or if *anObject* is **nil**). Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **self** if a wavetable is acquired (whether it's the sineROM or *anObject*); otherwise returns **nil**.

See also: – **setTable:length:**, – **setTableToSineROM**

setTableToSineROM

– **setTableToSineROM**

Sets the Oscgaf(i)UG's wavetable to the DSP sineROM. Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **nil** if the Oscgaf(i)UG doesn't fulfill this requirement; otherwise returns **self**.

tableLength

– (unsigned int)**tableLength**

Returns the length of the Oscgaf(i)UG's wavetable, in samples. The value returned by this method may differ from the length that you requested in a previous invocation of **setTable:length:**, as explained in that method's description.

See also: – **setTable:length:**

OscgUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

OscgUG is a simple oscillator. It operates by producing, at its output, the samples that it reads from a wavetable. You can define the wavetable by associating an OscgUG with a WaveTable or, more directly, with a SynthData object. Alternatively, you can use the DSP's sineROM, a read-only section of Y memory that holds one period of a sine wave. Note well that since the sineROM resides in Y memory, you must allocate an OscgUG that reads Y memory for its wavetable input.

The frequency, amplitude, and phase of the signal that an OscgUG produces are set directly and can't be controlled by an Envelope. For Envelope control of frequency or amplitude, you must use an OscgafUG or OscgafiUG object (the Music Kit doesn't provide an oscillator with Envelope-controlled phase).

OscgUG doesn't interpolate between the samples in its wavetable, thus the fidelity of the waveform that it produces is somewhat crude. Because of this, you rarely use OscgUG objects as the primary oscillators in a patch; for example, you wouldn't use such an object as a carrier in frequency modulation. However, an OscgUG can be used to good effect as a controlling oscillator, such as a modulator. Interpolation is provided by the OscgafiUG class of oscillators.

MEMORY SPACES

OscgUG*ab*

a output
b wavetable input

INSTANCE METHODS

setAmp:

– **setAmp:**(double)*amplitude*

Sets the OscgUG's amplitude to *amplitude* and returns **self**. Amplitude values should be between 0.0 and 1.0.

setFreq:

– **setFreq:**(double)*frequency*

Sets the OscgUG's frequency to *frequency* and returns **self**. Frequency values are, ostensibly, in the range 0.0 to half the sampling rate; to avoid foldover, however, you shouldn't request a frequency that would cause the highest partial that's represented in the wavetable to exceed half the sampling rate.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setPhase:

– **setPhase:**(double)*phase*

Sets the OscgUG's phase to *phase*, specified as degrees. Returns **self**.

setTable:

– **setTable:***anObject*

Sets the OscgUG's wavetable by invoking **setTable:length:** with a default value as the second argument. Currently the default table length is 256 for a Partials object, and the sample count (length) for a Samples or SynthData object. Returns the value returned by **setTable:length:**.

setTable:defaultToSineROM:

– **setTable:***anObject* **defaultToSineROM:**(BOOL)*useSine*

This is the same as **setTable:** but with an extra Boolean argument, *useSine*, that indicates whether you want to use the DSP sineROM if the resources to accommodate *anObject* aren't available (or if *anObject* is **nil**). Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **self** if a wavetable is acquired (whether it's the sineROM or *anObject*); otherwise returns **nil**.

See also: – **setTable:**, – **setTableToSineROM**

setTable:length:

– **setTable:***anObject* **length:**(int)*sampleCount*

Sets the OscgUG's wavetable to *anObject*—possibly allocating DSP memory to accommodate the table, as described below—and sets the (maximum) length of the wavetable to *sampleCount*, which must be a power of two. Returns **nil** if *anObject* is **nil**, if sufficient DSP memory isn't available to allocate storage for the requested wavetable, or if *sampleCount* isn't a power of two; otherwise returns **self**.

The *anObject* argument can either be a SynthData or a WaveTable object. If you use a WaveTable, DSP memory is allocated for you and made available to all other oscillators through Orchestra's shared object table. Thus, if you send **setTable:length:** to two different oscillators specifying the same WaveTable and length in the two messages, the first message will cause memory to be allocated while the second message will share this memory. Sharing WaveTables between oscillators helps conserve DSP resources, but you should be aware that the sharing mechanism makes it difficult to change the data in a WaveTable and have it affect the oscillator(s) with which it's associated. The most reliable way to change an oscillator's WaveTable is to create a new WaveTable object and set it through this method.

In addition, if you set the wavetable as a WaveTable object, the length that you request as *sampleCount* may not be the length of the wavetable memory that's actually allocated. If sufficient resources aren't available, the requested length is repeatedly halved until it fits, with a minimum table length of 64 samples. (Note that the 64-sample limit is an imposition only on this halving mechanism—you can pass as *sampleCount* a value that's less than 64 without falling ill of the law.)

See also: – **setTable:**, – **setTable:length:defaultToSineROM:**, – **tableLength**

setTable:length:defaultToSineROM:

– **setTable:***anObject*
length:(int)*sampleCount*
defaultToSineROM:(BOOL)*useSine*

This is the same as **setTable:length:** but with an extra Boolean argument, *useSine*, that indicates whether you want to use the DSP sineROM if the resources to accommodate *anObject* aren't available (or if *anObject* is **nil**). Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **self** if a wavetable is acquired (whether it's the sineROM or *anObject*); otherwise returns **nil**.

See also: – **setTable:length:**, – **setTableToSineROM**

setTableToSineROM

– **setTableToSineROM**

Sets the OscgUG's wavetable to the DSP sineROM. Keep in mind that the sineROM is only accessible to oscillators that read Y memory for their wavetable input. Returns **nil** if the OscgUG doesn't fulfill this requirement; otherwise returns **self**.

tableLength

– (unsigned int)**tableLength**

Returns the length of the OscgUG's wavetable, in samples. The value returned by this method may differ from the length that you requested in a previous invocation of **setTable:length:**, as explained in that method's description.

See also: – **setTable:length:**

Out1aUG, Out1bUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Out1aUG and Out1bUG provide single-channel access to the DSP's stereo output stream; the former writes its input signal to the left channel, and the latter writes to the right. To write a stereo signal, you should use a single Out2sumUG object rather than one of each of these.

Where the samples that are written to the DSP output stream are ultimately sent—whether to sound-out or to a soundfile—depends on the state of the Orchestra from which the Out1aUG or Out1bUG object was allocated. By default, the Orchestra sends the samples to sound-out.

If you're building a SynthPatch subclass, you should note that every SynthPatch object should have its own signal-output UnitGenerator; in other words, you don't allocate just one such object and then share it amongst the various SynthPatches. The output signals produced by all the running Out1aUG's, Out1bUG's, and Out2sumUG's are mixed (added) together into the DSP's output stream.

MEMORY SPACES

Out1aUG*a*
Out1bUG*a*
a input

INSTANCE METHODS

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale:

– **setScale:**(double)*value*

Sets the factor by which the input signal is scaled. By default, the scaler is set to a value that's just a wee bit less than 1.0. Effective values are between 0.0 and 1.0 (negative values are the same as their absolute values, but with a 180-degree phase shift). Returns **self**.

Out2sumUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Out2sumUG adds its input signal to the DSP's stereo output stream. The signal is placed (or "imaged") between the two channels according to the value set through the **setBearing:** or **setBearing:scale:** method. Alternatively, you can set the gain of either channel independently, through the **setRightScale:** and **setLeftScale:** methods.

To write to just the left or to just the right channel of the stereo ouput stream, you should use an Out1aUG or Out1bUG object. Where the samples that are written to the DSP output stream are ultimately sent—whether to sound-out or to a soundfile—depends on the state of the Orchestra from which the Out1aUG or Out1bUG object was allocated. By default, the Orchestra sends the samples to sound-out.

If you're building SynthPatch subclasses, you should note that every SynthPatch object should have its own signal-output UnitGenerator; in other words, you don't allocate just one such object and then share it amongst the various SynthPatches. The output signals produced by all the running Out1aUG's, Out1bUG's, and Out2sumUG's are mixed (added) together into the DSP's output stream.

MEMORY SPACES

Out2sumUG*a*

a input

INSTANCE METHODS

setBearing:

– **setBearing:**(double)*degrees*

Distributes the input signal between the two output channels according to the value of *degrees*: 0.0 degrees is center, –45.0 is hard left, 45.0 is hard right. Bearing is "reflected" as *degrees* exceeds the boundaries; thus, for example, 50.0 degrees is the same as 40.0, 60.0 is 30.0, 90.0 is 0.0, and so on. Returns **self**.

setBearing:scale:

– **setBearing:**(double)*degrees* **scale:**(double)*value*

This is the same as **setBearing:**, but the input signal is scaled by *value* before being distributed between the two output channels. The argument should be between 0.0 and 1.0. Returns **self**.

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setLeftScale:

– **setLeftScale:**(double)*value*

Sets the factor by which the signal that's written to the left output channel is scaled. By default, the scaler is set to a value that's just a tad less than 1.0. Effective values are between 0.0 and 1.0 (a negative *value* is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

setRightScale:

– **setRightScale:**(double)*value*

Sets the factor by which the signal that's written to the right output channel is scaled. By default, the scaler is set to a value that lacks 1.0 by a speck. Effective values are between 0.0 and 1.0 (a negative *value* is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

ScaleUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

ScaleUG multiplies its input by a constant scaler:

*output = input1 * scaler*

MEMORY SPACES

ScaleUG*ab*

a output
b input

INSTANCE METHODS

setInput:

– **setInput:***aPatchpoint*

Sets the input patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale:

– **setScale:**(double)*value*

Sets the constant scaler. Effective values are between 0.0 and 1.0 (a negative scaler is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

Scl1add2UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Scl1add2UG adds two input signals, the first of which is scaled:

output = (*input1* * *scaler*) + *input2*

MEMORY SPACES

Scl1add2UG*abc*

a output
b input 1 (scaled input)
c input 2 (unscaled input)

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale:

– **setScale:**(double)*value*

Sets the constant scaler. Effective values are between 0.0 and 1.0 (a negative scaler is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

Scl2add2UG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

Scl2add2UG adds two input signals, both of which are scaled:

output = (*input1* * *scaler1*) + (*input2* * *scaler2*)

MEMORY SPACES

Scl2add2UG*abc*

a output
b input 1
c input 2

INSTANCE METHODS

setInput1:

– **setInput1:***aPatchpoint*

Sets the input 1 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setInput2:

– **setInput2:***aPatchpoint*

Sets the input 2 patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setScale1:

– **setScale1**:(double)*value*

Sets the scaler on the first input. Effective values are between 0.0 and 1.0 (a negative scaler is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

setScale2:

– **setScale2:**(double)*value*

Sets the scaler on the second input. Effective values are between 0.0 and 1.0 (a negative scaler is the same as its absolute value, but with a 180-degree phase shift). Returns **self**.

SnoiseUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

SnoiseUG produces a series of random values within the range

$0.0 <= f < 1.0$

A new random value is generated once per tick. A similar class, UnoiseUG, produces a new random value every sample.

MEMORY SPACES

SnoiseUG*a*

a output

INSTANCE METHODS

anySeed

– **anySeed**

Sets the random number seed to a value that's guaranteed never to have been used in previous invocations of this method. This is particularly useful if you're using more than one SnoiseUG and you want to ensure that they all produce different signals.

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setSeed:

– **setSeed:**(DSPDatum)*seed*

Sets the seed that's used to prime the random number generator. If you want to create a unique series of random numbers, you should invoke the **anySeed** method instead of this one. Returns **self**.

UnoiseUG

INHERITS FROM UnitGenerator

DECLARED IN musickit/unitgenerators.h

CLASS DESCRIPTION

UnoiseUG produces a series of random values within the range

$0.0 <= f < 1.0$

A new random value is generated every sample. A similar class, SnoiseUG, produces a new random value every tick.

MEMORY SPACES

UnoiseUG*a*

a output

INSTANCE METHODS

setOutput:

– **setOutput:***aPatchpoint*

Sets the output patchpoint to *aPatchpoint*. Returns **nil** if the argument isn't a patchpoint; otherwise returns **self**.

setSeed:

– **setSeed:**(DSPDatum)*seed*

Sets the seed that's used to prime the random number generator. To create a unique series of random numbers, you should set the seed itself to a randomly generated number. Returns **self**.

Chapter 3
C Functions

Chapter 3
C Functions

This chapter gives detailed descriptions of the C functions defined by the Music Kit, the sound library, the sound/DSP driver, and the array processing library. Functions are listed alphabetically within these topical categories.

For quick reference, the function protocols are given in *NeXT Technical Summaries*.

MKAdjustFreqWithPitchBend() → See MKKeyNumToFreq()

MKAmpToMidi(), MKAmpAttenuationToMidi(), MKMidiToAmp(), MKMidiToAmpAttenuation(), MKMidiToAmpWithSensitivity(), MKMidiToAmpAttenuationWithSensitivity()

SUMMARY Translate loudness from the Music Kit to MIDI

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Note.h>

int **MKAmpToMidi**(double *amp*)
int **MKAmpAttenuationToMidi**(double *amp*)
double **MKMidiToAmp**(int *midiValue*)
double **MKMidiToAmpAttenuation**(int *midiValue*)
double **MKMidiToAmpWithSensitivity**(int *midiValue*, double *sensitivity*)
double **MKMidiToAmpAttenuationWithSensitivity**(int *midiValue*, double *sensitivity*)

DESCRIPTION

These functions help you convert Music Kit amplitude values to MIDI values and vice versa.

MKAmpToMidi() and **MKMidiToAmp**() are complementary functions that provide a non-linear mapping of amplitude to MIDI values, as described below:

MKAmpToMidi(double *amp*)	returns	$64 + (64 * \log_{10} amp)$
MKMidiToAmp(int *midiValue*)	returns	$10.0^{(midiValue-64)/64}$

This provides a scale in which an amp of 0.0 yields a MIDI value of 0, 1.0 produces 64, and 10.0 gives 127.

MKAmpAttenuationToMidi() and **MKMidiToAmpAttenuation**() are similarly complementary, and the curve of the mapping is the same as in the foregoing, but the scale is attenuated by a factor of ten: 0.0 maps to 0, 0.1 to 64, and 1.0 to 127.

MKMidiToAmpWithSensitivity() and **MKMidiToAmpAttenuationWithSensitivity**() are modifications of the similarly named MIDI-to-amp and MIDI-to-ampAttenuation functions in which an additional sensitivity value, nominally in the range 0.0 to 1.0, is used to scale the product of the conversion.

The multiplicity of conversion functions is provided in deference to the nature of MIDI volume computation: Unlike DSP-bound amplitude values (specifically, the value of the MK_amp parameter), effective MIDI volume is a combination of a number of parameters, the primary ones being velocity, main volume control, and foot pedal control. While the velocity value generated by a MIDI instrument is almost never at the maximum, the other values often are. In general, you use **MKAmpToMidi**() and **MKMidiToAmp**() (or **MKMidiToAmpWithSensitivy**()) to convert between amplitude and velocity. The amp attenuation functions are used to generate a value from, or apply a value to, one of the MIDI controller parameters.

MKAmpAttenuationToMidi() → See MKAmpToMidi()

MKCancelMsgRequest() → See MKNewMsgRequest()

MKClearTrace() → See MKSetTrace()

MKdB()

SUMMARY Convert decibels to amplitude

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>

double **MKdB**(double *dB*)

DESCRIPTION

MKdB() returns an amplitude value (within the range [0.0, 1.0]) converted from its argument specified as decibels. The returned value can be used to set a UnitGenerator's amplitude, for example. The value is converted using the following formula:

$$amplitude = 10.0^{\,dB/20.0}$$

MKError(), MKSetErrorProc(), MKSetErrorStream(), MKErrorStream()

SUMMARY Handle Music Kit errors

LIBRARY libmusickit

SYNOPSIS

#import <musickit/errors.h>

id **MKError**(char **msg*)
void **MKSetErrorProc**(void *(*errProc)(char *msg)*)
void **MKSetErrorStream**(NXStream **aStream*)
NXStream ***MKErrorStream**()

DESCRIPTION

These functions define the Music Kit's error handling mechanism. **MKError()** is used to signal an error. It calls the current Music Kit error function, set through **MKSetErrorProc()**, to which it passes the single argument *msg*. If the user hasn't declared an error function, then *msg* is written to the Music Kit error stream, as set through **MKSetErrorStream()**. The default error stream is open to **stderr**. **MKErrorStream()** returns a pointer to the current Music Kit error stream. Note that you *shouldn't* use **stderr** as the error stream if you're running a separate-threaded performance.

A number of error codes represented by integer constants are provided by the Music Kit and listed in **/usr/include/musickit/errors.h**. If the Music Kit itself generates an error, the global system variable **errno** is set to one of these error codes. If you call **MKError()** from your application, **errno** isn't set.

MKErrorStream() → See MKError()

MKFinishPerformance() → See MKGetTime()

MKFreqToKeyNum() → See MKKeyNumToFreq()

MKGetDeltaT() → See MKGetTime()

MKGetDeltaTTime() → See MKGetTime()

MKGetEnvelopeClass() → See MKSetNoteClass()

MKGetNamedObject() → See MKNameObject()

MKGetNoDVal() → See MKIsNoDVal()

MKGetNoteParAsDouble() → See MKSetNoteParToDouble()

MKGetNoteParAsInt() → See MKSetNoteParToDouble()

MKGetNoteParAsObject() → See MKSetNoteParToDouble()

MKGetNoteParAsString() → See MKSetNoteParToDouble()

MKGetNoteParAsStringNoCopy() → See MKSetNoteParToDouble()

MKGetNoteParAsWaveTable() → See MKSetNoteParToDouble()

MKGetObjectName() → See MKNameObject()

MKGetPartClass() → See MKSetNoteClass()

MKGetPartialsClass() → See MKSetNoteClass()

MKGetPreemptDuration() → See MKSetPreemptDuration()

MKGetSamplesClass() → See MKSetNoteClass()

MKGetTime(), MKGetDeltaT(), MKSetDeltaT(), MKGetDeltaTTime(), MKFinishPerformance(), MKSetTime()

SUMMARY Set and get Music Kit time values

LIBRARY libmusickit

SYNOPSIS

```
#import <musickit/musickit.h>

double MKGetTime()
double MKGetDeltaT()
void MKSetDeltaT(double val)
double MKGetDeltaTTime()
double MKSetTime(double newTime)
void MKFinishPerformance()
```

DESCRIPTION

MKGetTime() returns the current time, in seconds, during a Music Kit performance.

MKSetDeltaT() sets a performance's delta time in seconds. The delta time value is added into the timestamps of DSP and MIDI messages, thus imposing a time lag between the Music Kit and these devices. This lag is sometimes necessary to allow the

Music Kit sufficient compute time while maintaining rhythmic integrity. For an application that requires real-time response, a delta time of as much as 10 milliseconds (0.01 seconds) is tolerable. Delta time only affects devices that are timed. In addition, in order for the delta time value to be valid, the performance and the devices must be started at (virtually) the same time.

MKGetDeltaT() returns the delta time value.

MKGetDeltaTTime() returns the sum of the values returned by **MKGetTime()** and **MKGetDeltaT()**.

MKSetTime() and **MKFinishPerformance()** are provided to set the performance time and to end a performance, respectively. You only call these functions if you're running a performance without the Conductor class. During a conducted performance, **MKSetTime()** has no effect and **MKFinishPerformance()** is the same as sending **finishPerformance** to the Conductor class.

MKInitParameterIteration() → See MKIsNoteParPresent()

MKIsNoDVal(), MKGetNoDVal()

SUMMARY Test for no **double** value

LIBRARY libmusickit

SYNOPSIS

#import <musickit/noDVal.h>

int **MKIsNoDVal**(double *value*)
double **MKGetNoDVal**()

DESCRIPTION

A number of Music Kit functions and methods query for and return **double**-valued quantities, such as the values of parameters and time tags. By convention, the value MK_NODVAL is returned if the queried-for value hasn't been set; however, you can't test for this value directly. You must use the function **MKIsNoDVal()** instead, passing as the argument the value that you wish to test. The function returns nonzero if *value* is equal to MK_NODVAL and 0 if it isn't.

MKGetNoDVal() returns the no-**double**-value indicator. You use this function as the return value for functions and methods of your own design in which you wish to indicate that a **double**-valued quantity hasn't been set. For convenience, MK_NODVAL is defined as this function.

MKIsNoteParPresent(), MKInitParameterIteration(), MKNextParameter()

SUMMARY Query for a Note's parameters

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Note.h>

BOOL **MKIsNoteParPresent**(Note **aNote*, int *par*)
void ***MKInitParameterIteration**(Note **aNote*)
int **MKNextParameter**(Note **aNote*, void **iterationState*)

DESCRIPTION

MKIsNoteParPresent() returns YES or NO as the parameter *par* within the Note *aNote* is or isn't present; a parameter is considered present only if it's been given a value. The function is equivalent to Note's **isParPresent:** method. Unless the mere existence of the parameter is significant, you would follow a call to **MKIsNoteParPresent()** with a parameter value retrieval function, such as **MKGetNoteParAsDouble()**:

```
double freq;

/* Get the value of MK_freq only if the parameter has been set. */
if (MKIsNoteParPresent(aNote, MK_freq))
{
    freq = MKGetParAsDouble(aNote, MK_freq);
    . . . /* do something with freq */
}
```

MKInitParameterIteration() and **MKNextParameter()** work together to return, one by one, the full complement of a Note's parameter identifiers. **MKInitParameterIteration()** primes its Note argument for successive calls to **MKNextParameter()**, each of which retrieves the next parameter in the Note. When all the parameters have been visited, **MKNextParameter()** returns the value MK_noPar. The pointer returned by **MKInitParameterIteration()** must be passed as the *iterationState* argument to **MKNextParameter()**. Keep in mind that **MKNextParameter()** returns parameter identifiers; you still must retrieve the value of the parameter. An example for your delight:

```
/* Initialize the iteration state for the desired Note. */
void *aState = MKInitParameterIteration(aNote);
int par;

/* Get the parameters until the Note is exhausted. */
while ((par = MKNextParameter(aNote, aState)) != MK_noPar)
{
    /* Operate on the parameters of interest. */
    switch (par)
    {
        case MK_freq:
            /* Get the value of MK_freq and apply it. */
            . . .
            break;
        case MK_amp:
            /* Get the value of MK_amp and apply it. */
            . . .
            break;
        default:
            /* Ignore all other parameters. */
            break;
    }
}
```

In essence, the two examples do the same thing: They find and operate on parameters of interest. Which methodology to adopt—whether to test for the existence of particular parameters as in the first example, or to retrieve the identifiers of all present parameters as in the second—depends on how "saturated" the Note is with interesting parameters. If you only want a couple of parameters then it's generally more efficient to call **MKIsNoteParPresent()** for each of them. However, if you're interested in most—or what you assume to be most—of a Note's parameters (as is usually the case for a reasonably sophisticated SynthPatch, for example), then it's probably faster to iterate over all the parameters through **MKNextParameter()**.

SEE ALSO

MKGetNoteParAsDouble(), **MKGetNoteParAsInt()**, etc., **MKIsNoDVal()**

MKIsTraced() → See MKSetTrace()

MKKeyNumToFreq(), MKFreqToKeyNum(), MKTranspose(), MKAdjustFreqWithPitchBend()

SUMMARY Convert and adjust frequencies

LIBRARY libmusickit

SYNOPSIS

#import <musickit/TuningSystem.h>

double **MKKeyNumToFreq**(MKKeyNum *keyNum*)
double **MKTranspose**(double *freq*, double *semitones*)
MKKeyNum **MKFreqToKeyNum**(double *freq*, int **bendPtr*, double *sensitivity*)
double **MKAdjustFreqWithPitchBend**(double *freq*, int *pitchBend*,
double *sensitivity*)

DESCRIPTION

MKKeyNumToFreq() returns the frequency that corresponds to the given key number, based upon the mapping of key numbers to frequencies in the *installed tuning system* (see the TuningSystem class description in Chapter 2 for more information on the installed tuning system).

MKTranspose() returns the frequency that results from transposing *freq* by the specified number of semitones. A negative *semitones* value transposes down; a fractional value can be used to transpose by less than a semitone. The transposition afforded by this function is always in twelve-tone equal-temperament, regardless of the installed tuning system, as computed by the formula

$$\text{result} = freq * 2^{semitones/12.0}$$

MKFreqToKeyNum() returns the key number that most closely corresponds to the given frequency. The amount of pitch bend needed to temper the pitch of the key number in order to match the actual frequency is returned by reference in *bendPtr*. This value is computed using the *sensitivity* argument as the number of semitones by which the key number is tempered given a maximum pitch bend; in other words, you supply the maximum pitch bend by passing in a *sensitivity* value, and the function returns, in *bendPtr*, the amount of the bend that's needed. The value of *bendPtr* is a 14-bit MIDI pitch bend number; you would use it to set the value of a Note's MK_pitchBend parameter (assuming that you use *sensitivity* as value of the Note's MK_pitchBendSensitivity parameter).

MKAdjustFreqWithPitchBend() returns the frequency that results when *freq* is tempered by *pitchBend* worth of *sensitivity* semitones, where *pitchBend* is, again, a 14-bit MIDI pitch bend number.

RETURN

MKKeyNumToFreq() returns MK_NODVAL if *keyNum* is out of bounds (less than 0 or greater than 127). Use **MKIsNoDVal()** to check for MK_NODVAL.

MKMidiToAmp() → See MKAmpToMidi()

MKMidiToAmpAttenuation() → See MKAmpToMidi()

MKMidiToAmpWithSensitivity() → See MKAmpToMidi()

MKMidiToAmpAttenuationWithSensitivity() → See MKAmpToMidi()

MKNameObject(),MKGetObjectName(), MKRemoveObjectName(), MKGetNamedObject()

SUMMARY Identify and return objects by name

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>

BOOL **MKNameObject**(char ***name*, id *object*)
const char ***MKGetObjectName**(id *object*)
id **MKRemoveObjectName**(id *object*)
id **MKGetNamedObject**(char **name*)

DESCRIPTION

The Music Kit provides a global naming mechanism that lets you identify and locate objects by name. While names are primarily used in reading and writing scorefiles, any object—even a non-Music Kit object—can be named. Names needn't be unique; more than one object can be given the same name. However, a single object can have but one name at a time.

MKNameObject() sets *object*'s name to a copy of *name* and returns YES. If the object already has a name, then this function does nothing and returns NO.

MKGetObjectName() returns its argument's name, or NULL if it isn't named. The returned value is read-only and shouldn't be freed by the caller.

MKRemoveObjectName() removes its argument's name (if any) and returns **nil**.

MKGetNamedObject() returns the first object in the name table that has the name *name*.

MKNewMsgRequest(), MKScheduleMsgRequest(), MKRepositionMsgRequest(), MKCancelMsgRequest(), MKRescheduleMsgRequest()

SUMMARY Create and manipulate Conductor message requests

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Conductor.h>

MKMsgStruct ***MKNewMsgRequest**(double *timeOfMsg*, SEL *whichSelector*, id *destinationObject*, int *argCount*, ...)
void **MKScheduleMsgRequest**(MKMsgStruct **aMsgStructPtr*, id *conductor*)
MKMsgStruct ***MKRepositionMsgRequest**(MKMsgStruct **aMsgStructPtr*, double *newTimeOfMsg*)
MKMsgStruct ***MKCancelMsgRequest**(MKMsgStruct **aMsgStructPtr*)
MKMsgStruct ***MKRescheduleMsgRequest**(MKMsgStruct **aMsgStructPtr*, id *conductor*, double *newTimeOfMsg*, SEL *whichSelector*, id *destinationObject*, int *argCount*, ...)

DESCRIPTION

These functions let you enqueue message requests with a Conductor object. The MKMsgStruct structure encapsulates a message request; it consists of a method selector and its arguments, the recipient of the message, and the time that the message should be sent. A selector can take a maximum of two 4-byte arguments. You should never modify the fields of a MKMsgStruct structure directly.

MKNewMsgRequest() creates and returns a new MKMsgStruct. *timeOfMsg* is the time in beats from the beginning of the performance that the message will be sent, *whichSelector* is the selector, *destinationObject* is the recipient of the message, and *argCount* is the number of arguments to the selector followed by the arguments themselves separated by commas.

After you've created a message request structure, you schedule it with a Conductor by calling **MKScheduleMsgRequest()**. The message is enqueued to be sent at the time specified in the call to **MKNewMsgRequest()**, interpreted as beats in the Conductor's tempo.

If you want to move a message request within a Conductor's queue you call the **MKRepositionMsgRequest()** function. The specified MKMsgStruct is moved to the time given by *newTimeOfMsg*. You should note well that the MKMsgStruct you pass as the *aMsgStructPtr* argument is replaced with a new structure that's returned by the

function. So if you're in the mood to repeatedly reposition a MKMsgStruct—or if you're planning on referencing the structure for any other reason—you should reset the structure to the function's return value; for example:

```
/* Reposition and prime aMsgReq for additional functions calls. */
aMsgReq = MKRepositionMsgRequest(aMsgReq, 3.0);
```

MKCancelMsgRequest() cancels the given message request and frees the structure pointed to by *aMsgStructPtr.*

MKRescheduleMsgRequest() is a convenience function that cancels the structure pointed to by *aMsgStructPtr*, and then creates and schedules a new request according to the arguments. The new MKMsgStruct is returned.

RETURN

MKNewMsgRequest() and **MKRescheduleMsgRequest()** return NULL if *argCount* is greater than 2. **MKCancelMsgRequest()** always returns NULL.

MKNextParameter() → See MKIsNoteParPresent()

MKNoteTag(), MKNoteTags()

SUMMARY Create note tags

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Note.h>

unsigned int **MKNoteTag()**
unsigned int **MKNoteTags**(unsigned int *n*)

DESCRIPTION

Note tags are positive integers used to identify a series of Note objects as part of the same musical event, gesture, or phrase. A common use of note tags is to create a noteOn/noteOff pair by giving the two Notes the same note tag value.

MKNoteTag() returns a note tag value that's guaranteed to be unique across your entire application. **MKNoteTags()** returns the first of a block of *n* unique, contiguous note tags.

You should never create note tag values except through these functions.

RETURN

Returns MAXINT (the maximum note tag value) if a sufficient number of note tags aren't available, an unlikely occurrence.

MKNoteTags() → See MKNoteTag()

MKRemoveObjectName() → See MKNameObject()

MKRepositionMsgRequest() → See MKNewMsgRequest()

MKRescheduleMsgRequest() → See MKNewMsgRequest()

MKScheduleMsgRequest() → See MKNewMsgRequest()

MKSetDeltaT() → See MKGetTime()

MKSetEnvelopeClass() → See MKSetNoteClass()

MKSetErrorStream() → See MKError()

MKSetNoteParToEnvelope() → See MKSetNoteParToDouble()

MKSetNoteParToInt() → See MKSetNoteParToDouble()

MKSetNoteParToObject() → See MKSetNoteParToDouble()

MKSetNoteParToString() → See MKSetNoteParToDouble()

MKSetNoteParToWaveTable() → See MKSetNoteParToDouble()

MKSetNotePar ToDouble(), MKSetNotePar ToInt(), MKSetNotePar ToString(), MKSetNotePar ToEnvelope(), MKSetNotePar ToWaveTable(), MKSetNotePar ToObject(), MKGetNotePar AsDouble(), MKGetNotePar AsInt(), MKGetNotePar AsString(), MKGetNotePar AsStringNoCopy(), MKGetNotePar AsEnvelope(), MKGetNotePar AsWaveTable(), MKGetNotePar AsObject()

SUMMARY Set and retrieve a Note's parameters

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Note.h>

Note ***MKSetNotePar ToDouble**(Note **aNote*, int *par*, double *value*)
Note ***MKSetNotePar ToInt**(Note **aNote*, int *par*, int *value*)
Note ***MKSetNotePar ToString**(Note **aNote*, int *par*, char **value*)
Note ***MKSetNotePar ToEnvelope**(Note **aNote*, int *par*, Envelope **value*)
Note ***MKSetNotePar ToWaveTable**(Note **aNote*, int *par*,WaveTable **value*)
Note ***MKSetNotePar ToObject**(Note **aNote*, int *par*, Object **value*)

double **MKGetNotePar AsDouble**(Note **aNote*, int *par*)
int **MKGetNotePar AsInt**(Note **aNote*, int *par*)
char ***MKGetNotePar AsString**(Note **aNote*, int *par*)
char ***MKGetNotePar AsStringNoCopy**(Note **aNote*, int *par*)
Envelope ***MKGetNotePar AsEnvelope**(Note **aNote*, int *par*)
WaveTable ***MKGetNotePar AsWaveTable**(Note **aNote*, int *par*)
Object ***MKGetNotePar AsObject**(Note **aNote*, int *par*)

DESCRIPTION

These functions set and retrieve the values of a Note's parameters, one parameter at a time. They're equivalent to the similarly named Note methods; for example, the function call

```
MKSetNoteParToDouble(aNote, MK_freq, 440.0)
```

is the same as the message:

```
[aNote setPar:MK_freq toDouble:440.0]
```

As ever, calling a function is somewhat faster than sending a message, thus you may want to use these functions, rather than the corresponding methods, if you're examining and manipulating barrels of parameters, or in situations where speed is crucial. See the method descriptions in the Note class for more information (by implication) regarding the operations of these functions.

RETURN

The **MKSetParTo...()** functions return *aNote*, or **nil** if either *aNote* is **nil** or *par* isn't a valid parameter identifier.

The **MKGetParAs...()** functions return the requested value, or **nil** if either *aNote* is **nil** of *par* isn't a valid parameter identifier. If the parameter value hasn't been set, an indicative value is returned:

Function	**No-set return value**
MKGetNoteParAsInt()	MAXINT
MKGetNoteParAsDouble()	MK_NODVAL (check with MKIsNoDVal())
MKGetNoteParAsString()	""
MKGetNoteParAsStringNoCopy()	""
MKGetNoteParAsEnvelope()	nil
MKGetNoteParAsWaveTable()	nil
MKGetNoteParAsObject()	nil

SEE ALSO

MKIsNoteParPresent(), **MKInitParameterIteration()**, **MKNextParameter()**, **MKIsNoDVal()**

MKSetNoteClass(), MKSetPartClass(), MKSetEnvelopeClass(), MKSetPartialsClass(), MKSetSamplesClass(), MKGetNoteClass(), MKGetPartClass(), MKGetEnvelopeClass(), MKGetPartialsClass(), MKGetSamplesClass()

SUMMARY Set and retrieve scorefile creation classes

LIBRARY libmusickit

SYNOPSIS

#import <musickit/Note.h>

BOOL **MKSetNoteClass**(Note **noteSubclass*)
BOOL **MKSetPartClass**(Part **partSubclass*)
BOOL **MKSetEnvelopeClass**(Envelope **envelopeSubclass*)
BOOL **MKSetPartialsClass**(Partials **partialsSubclass*)
BOOL **MKSetSamplesClass**(Samples **samplesSubclass*)

Note ***MKGetNoteClass**()
Part ***MKGetPartClass**()
Envelope ***MKGetEnvelopeClass**()
Partials ***MKGetPartialsClass**()
Samples ***MKGetSamplesClass**()

DESCRIPTION

When you read a scorefile into your application, some number of objects are automatically created. Specifically, these objects are instances of Note, Part, Envelope, Partials, and Samples. You can supply your own classes from which these instances are created through these functions. The one restriction is that the class you set must be a subclass of the original class; for example, the class you pass the argument to **MKSetNoteClass()** must be a subclass of Note.

The **MKGet*Class*Class()** functions return the requested classes as set through the functions above.

RETURN

MKSet*Class*Class() returns NO if the argument isn't a subclass of ***Class***; otherwise it returns YES.

MKSetPartClass() → See MKSetNoteClass()

MKSetPartialsClass() → See MKSetNoteClass()

MKSetPreemptDuration(), MKGetPreemptDuration()

SUMMARY Set the SynthPatch preemption time

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>

void **MKSetPreemptDuration**(double *seconds*)
double **MKGetPreemptDuration**()

DESCRIPTION

During a performance, DSP resources can become scarce; it's sometimes necessary to preempt active SynthPatches in order to synthesize new Notes. This preemption is handled by SynthInstrument objects. But rather than simply yank the rug from under an active SynthPatch, a certain amount of time is given to allow the patch to "wind down" before it's killed. By default, this grace period, or "preempt duration", is 0.006 seconds—not a lot of time but enough to avoid snapping the SynthPatch's envelopes. You can set the preempt duration yourself through **MKSetPreemptDuration()**. Preempt duration is global to an application; its current value is retrieved through **MKGetPreemptDuration()**.

MKSetSamplesClass() → See MKSetNoteClass()

MKSetScorefileParseErrorAbort()

SUMMARY Set the scorefile error threshold

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>

void **MKSetScorefileParseErrorAbort**(int *thresholdCount*)

DESCRIPTION

As a scorefile is read into an application, errors sometimes occur: Time tags may be out of order; undeclared or mistyped names may pop up in the middle of the file. The Music Kit keeps a count of these errors for each file it reads. If the error count for a particular file exceeds the threshold set as the *thresholdCount* argument to this function, the scorefile parsing is aborted and the file is closed (if the Music Kit opened it itself). The default limit is ten errors.

MKSetTime() → See MKGetTime()

MKSetTrace(), MKClearTrace(), MKIsTraced()

SUMMARY Trouble-shoot the Music Kit

LIBRARY libmusickit

SYNOPSIS

#import <musickit/errors.h>

unsigned int **MKSetTrace**(int *traceCode*)
unsigned int **MKClearTrace**(int *traceCode*)
BOOL **MKIsTraced**(int *traceCode*)

DESCRIPTION

To aid in debugging, the Music Kit is peppered with activity-tracing messages that print to **stderr** if but asked. The trace messages are divided into eight categories, represented by the following codes:

Code	Value	Meaning
MK_TRACEORCHALLOC	1	DSP resource allocation
MK_TRACEPARS	2	Application-defined parameters
MK_TRACEDSP	4	DSP manipulation
MK_TRACEMIDI	8	MIDI manipulation
MK_TRACEPREEMPT	16	SynthPatch preemption
MK_SYNTHINS	32	SynthInstrument machinations
MK_SYNTHPATCH	64	SynthPatch library messages
MK_UNITGENERATOR	128	UnitGenerator library messages

To enable a set of messages, you pass a trace code to the **MKSetTrace()** function. You can enable more than one set with a single function call by bitwise-or'ing the codes. Clearing a trace is done similarly by passing codes to **MKClearTrace()**. The **MKIsTraced()** function returns YES or NO as the argument code is or isn't currently traced. These functions should only be used while you're debugging and fine-tuning your application.

You should note that the codes given above are **#define**'d as their corresponding values and so can be used only when you call one of these functions within an application—they can't be used in a symbolic debugger such as **gdb**. For this reason, the integer values themselves are also given; you must use the integer values to enable and disable a set of trace messages from within a debugger.

MK_TRACEORCHALLOC

The Orchestra allocation messages inform you of DSP resource allocation. The most important of these have to do with SynthPatch, UnitGenerator, and SynthData allocation. When a SynthPatch is allocated, one of the following messages is printed:

```
"allocSynthPatch returns SynthPatchClass_SynthPatchId"
"allocSynthPatch building SynthPatchClass_SynthPatchId..."
"allocSynthPatch can't allocate SynthPatchClass"
```

The first of these signifies that an appropriate SynthPatch object was found. The second means that a new object was created. The third denotes an inability to construct the requested object because of insufficient DSP resources. As a SynthPatch's UnitGenerators are connected, the following message is printed:

```
"allocSynthPatch connectsContents of SynthPatchClass_SynthPatchId"
```

When a SynthPatch is deallocated and when it's freed, respectively, the following are printed:

```
"Returning SynthPatchClass_SynthPatchId to avail pool."
"Freeing SynthPatchClass_SynthPatchId"
```

A UnitGenerator can be allocated without reference to other UnitGenerators, or it can be positioned before, after, or between other objects. First, an available object is searched for:

```
"allocUnitGenerator looking for a UGClass."
"allocUnitGenerator looking for a UGClass before UGClass_UGid"
"allocUnitGenerator looking for a UGClass after UGClass_UGid"
"allocUnitGenerator looking for a UGClass after UGClass_UGid
    and before UGClass_UGid"
```

If a new UnitGenerator is built, the addresses ("Reloc") and sizes ("Reso") of the allocated DSP resources are given:

```
"Reloc: pLoop address, xArg address, yArg address, lArg address,
    xData address, yData address, pSubr address"
"Reso: pLoop size, xArg size, yArg size, lArg size, xData size,
    yData size, pSubr size, time orchestraLoopDuration"
```

As the UnitGenerator search (or allocation) succeeds or fails, one of the following is printed:

```
"allocUnitGenerator returns UGClass_UGid"
"Allocation failure: Can't allocate before specified ug."
"Allocation failure. DSP error."
"Allocation failure. Not enough computeTime."
"Allocation failure. Not enough memorySegment memory."
```

Allocating a SynthData generates the first and then either the second or third of these messages:

```
"allocSynthData: looking in segment memorySegment for size size."
"allocSynthData returns memorySegment address of length size."
"Allocation failure: No more offchip data memory."
```

When you install shared data, the following is printed:

```
"Installing shared data keyObjectName in segment memorySegment."
```

During allocation of UnitGenerators and SynthDatas, existing resources might be compacted. Compaction can cause free UnitGenerators and unreferenced shared data to be garbage collected, and active UnitGenerators to be relocated:

```
"Compacting stack."
"Copying arguments."
"Copying p memory."
"Garbage collecting freed unit generator UGClass_UGid"
"Moving UGClass_UGid."
"NewReloc: pLoop address, xArg address, yArg address, lArg
address."
"Garbage collecting unreferenced shared data."
"No unreferenced shared data found."
```

MK_TRACEDSP

The DSP-trace messages give you details of how the DSP is being used. For example, when a UnitGenerator is allocated, the following message is printed among the search-build-return messages given above:

```
"Loading UGClass_UGid."
```

The most important of the DSP-trace messages reflect the setting of a UnitGenerator's memory arguments. A memory argument takes either an address value or a data value. When you set an address-valued argument, the following is printed:

```
"Setting argNum of UGClass_UGid to address 0xaddress."
```

A data-valued arguments is either a 24-bit or 48-bit word; separate functions (and cover methods) are defined for setting the two sizes of arguments. The following messages are printed as the "correct" function is used to set an argument's value:

```
"Setting argNum of UGClass_UGid to datum value."
"Setting argNum of UGClass_UGid to long:
    hi wd value and low wd value."
```

A 24-bit argument that's set with the long-setting function and vice versa produce these messages, respectively:

```
"Setting (L-just, 0-filled) argNum of UGClass_UGid to datum
value."
"Setting argNum of UGClass_UGid to: value"
```

If an argument is declared as optimizable, the following is printed when the optimization obtains:

```
"Optimizing away poke of argNum of UGClass_UGid."
```

SynthData allocation doesn't actually involve the DSP; the address of the memory that will be allocated on the DSP is computed, but the state of the DSP itself doesn't change until data is loaded into the SynthData:

```
"Loading array into memory block SynthDataClass_SynthDataId."
"Loading constant value into memory block
    SynthDataClass_SynthDataId."
```

Clearing a SynthData's memory produces the following:

```
"Clearing memory block SynthDataClass_SynthDataId."
```

DSP manipulations that are performed as an atomic unit are bracketed by the messages:

```
"<<< Begin orchestra atomic unit "
"end orchestra atomic unit.>>> "
```

MK_TRACESYNTHINS

The SynthInstrument messages are printed when a SynthInstrument object receives Notes, and as it finds or creates SynthPatches to realize these Notes.

If a received Note's note tag is active or inactive, or if its note type is mute, the following are printed, respectively:

```
"SynthInstrument receives note for active notetag stream noteTag
    at time time."
"SynthInstrument receives note for new notetag stream noteTag
    at time time."
"SynthInstrument receives mute Note at time time."
```

SynthPatch allocation is noted *only* if the SynthInstrument is in auto-allocation mode:

```
"SynthInstrument creates patch synthPatchId at time time
    for tag noteTag."
```

However, SynthPatch reuse and preemption produce the following messages, respectively, regardless of the SynthInstrument's allocation mode:

```
"SynthInstrument uses patch synthPatchId at time time
    for tag noteTag."
"SynthInstrument preempts patch synthPatchId at time time
    for tag noteTag."
```

If a SynthPatch of the correct PatchTemplate isn't found and can't be allocated, an alternative is used; barring that, the SynthInstrument omits the Note:

```
"No patch of requested template was available.
    Using alternative template."
"SynthInstrument omits note at time time for tag noteTag."
```

MK_TRACEPREEMPT

These are a subset of the SynthInstrument messages that deal with SynthPatch preemption and Note omission:

```
"SynthInstrument preempts patch synthPatchId at time time
    for tag noteTag."
"SynthInstrument omits note at time time for tag noteTag.
```

MK_TRACEMIDI

When MIDI messages are converted to Music Kit Notes (and the parameters therein), the following messages appear if there's an error in the message stream:

```
"Two noteOns on same keyNum without intervening noteOff."
"NoteOff for multiply on keyNum." [sic]
```

The first of these indicates that two noteOns on the same channel and key number were found without an intervening noteOff; the second is printed as the "missing" noteOffs arrive.

The following are printed as ill-defined Note objects are converted to MIDI messages:

```
"NoteOn missing a noteTag at time time"
"NoteOff missing a note tag at time time"
"NoteOff for noteTag which is already off at time time"
"PolyKeyPressure with invalid noteTag
    or missing keyNum: time time;"
```

MK_TRACESYNTHPATCH and MK_TRACEUNITGENERATOR

Currently, the SynthPatch library and UnitGenerator library messages refer only to WaveTable allocation. You should always trace these two together as the messages are virtually indistinguishable. If insufficient DSP memory is available to load a WaveTable of the requested length, the following is printed:

```
"Insufficient wavetable memory at time time.
    Using smaller table length newLength."
```

If the sine ROM, which resides in Y memory, is requested by a UnitGenerator's X-space memory argument, the following appears:

```
"X-space oscgaf cannot use sine ROM at time time."
```

MK_TRACEPARS

By tracing MK_TRACEPARS, you're informed when an application-defined parameter is created:

```
"Adding new parameter parameterName"
```

RETURN

MKSetTrace() and **MKClearTrace()** return the value of the new (cumulative) trace code.

MKTranspose() → See MKKeyNumToFreq()

MKSetUGAddressArg(), MKSetUGAddressArgToInt(), MKSetUGDatumArg, MKSetUGDatumArgLong()

SUMMARY Set DSP unit generator arguments

LIBRARY libmusickit

SYNOPSIS

#import <musickit/UnitGenerator.h>

id **MKSetUGAddressArg**(UnitGenerator *_ug_, unsigned int _argNum_, SynthData *_obj_)
id **MKSetUGAddressArgToInt**(UnitGenerator *_ug_, unsigned int _argNum_, DSPAddress _address_)
id **MKSetUGDatumArg**(UnitGenerator *_ug_, unsigned int _argNum_, DSPDatum _value_)
id **MKSetUGDatumArgLong**(UnitGenerator *_ug_, unsigned int _argNum_, DSPLongDatum *_value_)

DESCRIPTION

These functions let you set the value of a DSP unit generator argument; they can only be called as part of the implementation of a UnitGenerator subclass. The arguments to all four functions are similar:

- *ug* is the UnitGenerator object that represents the DSP unit generator; because of the nature of these functions, *ug* can only be **self**.
- *argNum* is the integer that identifies the unit generator argument you want to affect.
- The final argument is (or gives the address of) the value you want to set the unit generator argument to.

MKSetUGAddressArg() and **MKSetUGAddressArgToInt()** are used to set address-valued unit generator arguments. The former sets the argument to the DSP address of *obj*, which must be a SynthData object. The latter function sets it directly as the value of its *address* argument. The DSPAddress data type is defined as an **int**.

MKSetUGDatumArg() and **MKSetUGDatumArgLong()** set data-valued unit generator arguments. The former takes a DSPDatum (**int**) directly and sets the unit generator argument to the rightmost 24 bits of this value. The latter is used to set 48-bit

DSP values; it takes, as the *value* argument, a pointer to a DSPLongDatum value. DSPLongDatum is defined as a DSPFix48 structure:

```
typedef struct _DSPFix48 {
    int high24;     /* High order 24 bits, right justified */
    int low24;      /* Low order 24 bits, right justified */
    DSPFix48;
```

If the argument identified by *argNum* isn't allocated in the DSP's long memory, then only the **high24** field of the structure is taken as the value.

RETURN

If *argNum* is out of bounds, or if an address-setting function is used to set a data argument (or vice versa), an error is generated and **nil** is returned; otherwise the *ug* argument, which is always **self**, is returned.

MKUpdateAsymp()

SUMMARY Apply an Envelope on the DSP

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>

void **MKUpdateAsymp**(AsympUG **asymp*, Envelope **envelope*, double *valueAt0*, double *valueAt1*, double *attackDur*, double *releaseDur*, double *portamentoTime*, MKPhraseStatus *status*)

DESCRIPTION

This is a fairly complicated function that, simply put, does the "right thing" in applying an Envelope object to a DSP-synthesized musical attribute during a Music Kit performance. It's typically used as part of the implementation of a SynthPatch subclass.

The *asymp* argument is an AsympUG object that will handle the Envelope on the DSP; *envelope* is the Envelope object itself. The arguments *valueAt0*, *valueAt1*, *attackDur*, and *releaseDur* scale and stretch the Envelope; their values are expected to be taken from an associated group of Note parameters. For example, to apply an Envelope to the frequency of a synthesized Note, the values of these arguments would be retrieved as follows:

```
Envelope *envelope = [aNote parAsEnvelope:MK_freqEnv];
double valueAt0 = [aNote parAsDouble:MK_freq0];
double valueAt1 = [aNote parAsDouble:MK_freq1];
double attackDur = [aNote parAsDouble:MK_freqAtt];
double releaseDur = [aNote parAsDouble:MK_freqRel];
```

The *portamentoTime* argument is taken as the Note's MK_portamentoTime value. As the name implies, it sets the portamento or "slur" between Notes and is only applied if the Note to which the Envelope belongs is a noteOn that's interrupting an existing Note.

The final argument, *status*, is used to distinguish the phrase state of the SynthPatch at the time that the Envelope is applied. You retrieve phrase status through SynthPatch's **phraseStatus** method. The use of portamento, for example, is determined by the value of this argument.

The *asymp* and *status* arguments are essential; the parameter-valued arguments aren't. The function tries to be intelligent with regard to missing parameter-valued arguments—you can even exclude the Envelope argument: If *envelope* is **nil**, the value of *valueAt1* is applied such that the AsympUG will produce this value as a constant.

MKUpdateAsymp() handles all the Envelope breakpoint scheduling for you. An Envelope object isn't downloaded to the DSP as a whole but, instead, its breakpoints are fed one-by-one to the DSP through message requests scheduled with a Conductor. This function always uses the clockConductor for this task.

MKWritePitchNames()

SUMMARY Write pitches to a scorefile

LIBRARY libmusickit

SYNOPSIS

#import <musickit/musickit.h>
#import <musickit/pitches.h>

void **MKWritePitchNames**(BOOL *yesOrNo*)

DESCRIPTION

This function sets the format by which frequency and key number parameter values are written to a scorefile. If the argument is YES, the parameter values are written as pitch name and key number constants such as "a4" and "a4k". If it's NO, frequencies are written as fractional numbers and key numbers as integers.

Sound Functions

SNDAcquire(), SNDReset(), SNDRelease(), SNDBootDSP(), SNDRunDSP()

SUMMARY Access sound resources

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDAcquire**(int *soundResource*, int *priority*, int *preempt*, int *timeout*, SNDNegotiationFun *negotiationFunction*, void **arg*, port_t **devicePort*, port_t **ownerPort*)
int **SNDReset**(int *soundResource*, port_t **devicePort*, port_t **ownerPort*)
int **SNDRelease**(int *soundResource*, port_t **devicePort*, port_t **ownerPort*)
int **SNDBootDSP**(port_t **devicePort*, port_t **ownerPort*, SNDSoundStruct **dspCore*)
int **SNDRunDSP**(SNDSoundStruct **dspCore*, char **toDSP*, int *toCount*, int *toWidth*, int *toBufferSize*, char ***fromDSP*, int **fromCount*, int *fromWidth*, int *negotiationTimeout*, int *flushTimeout*, int *conversionTimeout*)

DESCRIPTION

SNDAcquire() attempts to gain ownership of the sound resources specified in *soundResource*, a value that's created by (bitwise) or'ing a combination of the following resource codes:

Code	Resource
SND_ACCESS_OUT	sound-out
SND_ACCESS_IN	sound-in
SND_ACCESS_DSP	the DSP

Device and ownership ports to a successfully acquired device are returned in *devicePort* and *ownerPort*, respectively. Acquiring a resource makes it active, such that other acquisition requests may fail, even if the requests are in the same process. You can grant a priority to the acquisition by setting the value of the *priority* argument to an integer between 0 and 10. In a subsequent call to **SNDAcquire()**, the acquisition with the higher priority wins. The function's *preempt*, *timeout*, *negotiationFunction*, and *arg* arguments are currently unused.

SNDReset() and **SNDRelease()** reset to a virgin state and release, respectively, the specified resources. The resources must have been previously acquired through **SNDAcquire()**; the device and owner port arguments are values returned by that function.

SNDBootDSP() boots the DSP using the DSP bootstrap image specified in *dspCore*. This allows you to load all internal RAM and all but the top six words of external RAM on the DSP. The owner and device ports must have been previously acquired through **SNDAcquire()**. The format of *dspCore* must SND_FORMAT_DSP core image should be in loadable (".lod") form, such as is created through the **SNDReadDSPfile()** function.

SNDRunDSP() is similar to **DSPBootDSP()** in that it loads and runs a program you provide. However, **SNDRunDSP()** is designed to be used with DSP programs that process sound data—you typically use this function to provide your own sound conversion algorithms. The arguments are as follows:

- The DSP program is represented by *dspCore*; it should implement complex DMA mode for its output, and it should be in loadable form.
- *toDSP* is a pointer to the data that you wish to feed to the DSP.
- *toCount* is the number of samples to process.
- *toWidth* is the size of a single unprocessed sample.
- *toBufferSize* is the total size, in bytes, of the *toDSP* data.
- *fromDSP* is a pointer to the address of the processed data. The memory to store the data is allocated for you.
- *fromCount* is returned by the function to give the number of samples that it actually processed.
- *fromWidth* is the size, in bytes, of a single processed sample.
- The timeout arguments, *negotiationTimeout*, *flushTimeout*, and *conversionTimeout*, are ignored.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

ERRORS

If **SNDAcquire()** is unable to acquire any one of the resources specified in *soundResource*, none of the resources are acquired.

SNDAlloc(), SNDFree()

SUMMARY Create and free a sound structure

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDAlloc**(SNDSoundStruct ***sound*, int *dataSize*, int *dataFormat*,
int *samplingRate*, int *channelCount*, int *infoSize*)
int **SNDFree**(SNDSoundStruct **sound*)

DESCRIPTION

The SNDSoundStruct structure is the data format used by the sound software to encapsulate a sound. It defines the soundfile format and the NeXT sound pasteboard type, and it lies at the heart of every Sound object. **SNDAlloc()** creates and returns, in *sound*, a new SNDSoundStruct. The arguments to **SNDAlloc()** correspond to the SNDSoundStruct fields described below. **SNDFree()** frees the SNDSoundStruct pointed to by *sound*. You should always use **SNDFree()** to free a sound structure.

The fields of the SNDSoundStruct structure list the attributes of the sound that the structure represents. The sound data itself isn't contained in the structure, but is located by a structure field. Nonetheless, it's often convenient to think of a SNDSoundStruct as containing the sound data that it represents. By convention, the structure is referred to as the sound's "header." It's defined as:

```
typedef struct  {
    int  magic;          /* SND_MAGIC ((int)0x2e736e64) */
    int  dataLocation;   /* Offset or pointer to the raw data */
    int  dataSize;       /* Raw data size in bytes */
    int  dataFormat;     /* The data format code */
    int  samplingRate;   /* The sampling rate */
    int  channelCount;   /* The number of channels */
    char info[4];        /* Textual information about the sound */
SNDSoundStruct;
```

The **magic** field is a magic number that identifies a SNDSoundStruct. It's automatically set when you allocate the structure.

The **dataLocation** field indicates the location of the actual sound data. Usually, the data immediately follows the header. In this case, **dataLocation** is the offset from the beginning of the structure to the first byte of the sound data—in other words, it's the size of the sound's header. However, if you edit the sound through functions such as **SNDDeleteSamples()** or **SNDInsertSamples()**, the sound can become fragmented such that the data no longer follows the header. In this case, **dataLocation** is a pointer to a NULL-terminated block of addresses, each of which points to a separate SNDSoundStruct. The collection of these SNDSoundStructs make up the fragmented data.

dataSize is the size, in bytes, of the memory allocated for the sound data. The data is uninitialized.

dataFormat describes the sound data as one of the following codes:

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 due to editing. Only sampled data can become fragmented. You never allocate a sound with this format.

- SND_FORMAT_UNSPECIFIED is used for unrecognized formats.

samplingRate is also given as a code and should be cast into an **int**. The NeXT sound hardware supports the following sampling rates for recording and playback:

Code	Sampling Rate (Hz)
SND_RATE_CODEC	8012.8210513
SND_RATE_LOW	22050
SND_RATE_HIGH	44100

channelCount is the number of channels of sound. Playback of one- and two-channel sounds is supported; a sound with more than two channels is unplayable.

infoSize is the size of a variable-length string that can be used to textually describe the sound. The size is extended to the next 4-byte boundary (the minimum size is 4 bytes). You can't increase the length of the info string once its size has been set.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError**(), is returned.

SNDBootDSP() → See SNDAcquire()

SNDBytesToSamples() → See SNDSampleCount()

SNDCompactSamples() → See SNDInsertSamples()

SNDCompressSound(), SNDSetCompressionOptions(), SNDGetCompressionOptions()

SUMMARY Compress or decompress a sound

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDCompressSound**(SNDSoundStruct **fromSound*,
SNDSoundStruct ***toSound*, BOOL *bitFaithful*, int *compressionAmount*)
int **SNDSetCompressionOptions**(SNDSoundStruct **sound*, int *bitFaithful*,
int *compressionAmount*)
int **SNDGetCompressionOptions**(SNDSoundStruct *sound, int **bitFaithful*,
int **compressionAmount*)

DESCRIPTION

SNDCompressSound() creates and returns, in *toSound*, a new SNDSoundStruct that contains a compressed or decompressed version of the sound in *fromSound*:

- If *fromSound's* format is SND_FORMAT_LINEAR_16 or SND_FORMAT_EMPHASIZED, the sound returned in *toSound* is compressed.

- If its format is SND_FORMAT_COMPRESSED or SND_FORMAT_COMPRESSED_EMPHASIZED, the sound is decompressed.

No other formats are allowed; in addition, the sound can't be more than two channels.

The function's *bitFaithful* and *compressionAmount* arguments are used only when compressing:

- *bitFaithful* determines the fidelity with which a compressed sound can be restored to its original state. If *bitFaithful* is TRUE, the sound returned in *toSound* can be decompressed to exactly match the original sound data; if it's FALSE, some degradation can be expected.

- The *compressionAmount* argument controls the amount of compression. Its value ranges from 4 to 8 with higher numbers giving more compression but less fidelity. Depending on the signal, a *compressionAmount* of 4 will compress the sound to about half its original size; a value of 8 compresses to about one-sixth the size. For bit-faithful compression, you should set *compressionAmount* to 4.

SNDSetCompressionOptions() sets the bit-faithfulness and the compression amount that **SNDStartRecording()** uses during subsequent recordings into the sound *sound*. These values are effective only if *sound*'s format specifies compression. By default, such a recording is bit-faithful with a compression amount of 4.

SNDGetCompressionOptions() returns, in its arguments, pointers to the currently established compression options.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise an error code, as described in **SNDSoundError()**, is returned.

SEE ALSO

SNDStartRecording()

SNDConvertSound(), SNDMulaw(), SNDiMulaw()

SUMMARY Convert a sound's attributes

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDConvertSound**(SNDSoundStruct **fromSound*, SNDSoundStruct ***toSound*)
unsigned char **SNDMulaw**(short *linearValue*)
short **SNDiMulaw**(unsigned char *mulawValue*)

DESCRIPTION

SNDConvertSound() copies the sampled data from *fromSound* into *toSound*, converting the copied data to the format, channel count, and sampling rate specified by *toSound*. Memory for the converted data is automatically allocated. Only the conversions listed below are allowed:

- CODEC MuLaw to low sampling rate 16-bit linear; mono to stereo or the two sounds must have identical channel counts.
- MuLaw to 16-bit linear (and vice versa); *toSound*'s sampling rate and channel count are taken from *fromSound*.
- Mono to stereo; *fromSound* and *toSound* must have identical formats and sampling rates.
- High sampling rate to low sampling rate; *fromSound* and *toSound* must both be 16-bit linear and have the same channel count.

SNDMulaw() converts a value from 16-bit linear to mu-law: It takes a single linear 16-bit argument and returns the corresponding mu-law value. **SNDiMulaw()** performs the inverse operation: It takes a mu-law argument and returns the 16-bit linear value.

RETURN

If no error occurs, **SNDConvertSound()** returns SND_ERR_NONE. Otherwise an error code, as described in **SNDSoundError()**, is returned.

SEE ALSO

SNDAlloc(), **SNDSamplesToBytes()**

SNDCopySamples() → See SNDCopySound()

SNDCopySound(), SNDCopySamples()

SUMMARY Copy all or part of a sound

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDCopySound**(SNDSoundStruct ***toSound*, SNDSoundStruct **fromSound*)
int **SNDCopySamples**(SNDSoundStruct ***toSound*, SNDSoundStruct **fromSound*, int *startSample*, int *sampleCount*)

DESCRIPTION

SNDCopySound() creates and returns, in *toSound*, a new SNDSoundStruct that contains a copy of the sound in *fromSound*. This works for any type of sound, including DSP sounds.

SNDCopySamples() also creates a new SNDSoundStruct pointed to by *toSound*, but copies only the specified of *fromSound*, starting with the *startSample* sample (counting from sample 0) and copying *sampleCount* samples. This function works only for sampled sounds.

toSound should eventually be freed with **SNDFree**().

RETURN

Both functions return an error code as described in **SNDSoundError**().

ERRORS

If an error occurs, the SNDSoundStruct isn't created.

SNDDeleteSamples() → See SNDInsertSamples()

SNDFree() → See SNDAlloc()

SNDGetCompressionOptions() → See SNDCompressSound()

SNDGetDataPointer()

SUMMARY Gain access to sampled sound data

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDGetDataPointer**(SNDSoundStruct **sound*, char ***ptr*, int **size*, int **width*)

DESCRIPTION

The **SNDGetDataPointer()** provides access to *sound*'s sound data. A pointer to the sound data is returned by reference in *sound*, the size of the data is returned in *samples*, and the width (in bytes) of a single sample is returned in *width*. Note that *size* is the total sample count—it isn't a count of the sample frames. The data itself should be unfragmented, sampled data.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

SNDGetFilter() → See SNDSetVolume()

SNDGetMute() → See SNDSetVolume()

SNDGetVolume() → See SNDSetVolume()

SNDiMulaw() → See SNDConvertSound()

SNDInsertSamples(), SNDDeleteSamples(), SNDCompactSamples()

SUMMARY Edit a sampled sound

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDInsertSamples**(SNDSoundStruct **toSound*, SNDSoundStruct **fromSound*, int *startSample*)
int **SNDDeleteSamples**(SNDSoundStruct **sound*, int *startSample*, int *sampleCount*)
int **SNDCompactSamples**(SNDSoundStruct ***toSound*, SNDSoundStruct **fromSound*)

DESCRIPTION

SNDInsertSamples() inserts a copy of *fromSound* into *toSound* at position *startSample* of *toSound* (counting from sample 0). This operation may fragment *toSound*.

SNDDeleteSamples() deletes *sampleCount* samples from *sound*, starting at sample *startSample*. The memory occupied by the deleted segment is freed. The sound may become fragmented.

SNDCompactSamples() creates and returns, in *toSound*, a new SNDSoundStruct that contains a compacted version of *fromSound*. Compaction eliminates the fragmentation that can be caused by inserting and deleting samples.

These functions work only on sounds that contain sampled data.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

SNDModifyPriority() → See SNDStartPlaying()

SNDMulaw() → See SNDConvertSound()

SNDPlaySoundfile() → See SNDStartPlaying()

SNDRead() → See SNDReadSoundfile()

SNDReadDSPfile() → See SNDReadSoundfile()

SNDReadHeader() → See SNDReadSoundfile()

SNDReadSoundfile(), SNDRead(), SNDReadHeader(), SNDReadDSPfile()

SUMMARY Read a sound from a file

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDReadSoundfile**(char **path*, SNDSoundStruct ***sound*)
int **SNDRead**(int *fd*, SNDSoundStruct ***sound*)
int **SNDReadHeader**(int *fd*, SNDSoundStruct ***sound*)
int **SNDReadDSPfile**(char **path*, SNDSoundStruct ***sound*, char **info*)

DESCRIPTION

Each of these functions creates and returns, by reference in the *sound* argument, a SNDSoundStruct that contains the sound represented in a specified file.

SNDReadSoundfile() and **SNDRead**() read the entire contents of a soundfile. The *path* argument to **SNDReadSoundfile**() is a pathname; the function opens and closes the file automatically. **SNDRead**() takes a file descriptor *fd* that must be open for reading.

SNDReadHeader() reads only the header portion of the file descriptor *fd*. Storage for the actual sound data isn't allocated. The **dataLocation** field of the new SNDSoundStruct can be interpreted as the size of the header.

SNDReadDSPfile() creates a SNDSoundStruct for the given loadable DSP core file. The file, which is opened and closed by the function, is specified as a pathname and must have a ".lod" extension. The *info* argument is provided as a convenience, allowing you to specify an information string that's written in *sound*'s header. The DSP program is executed by calling **SNDBootDSP**() or **SNDRunDSP**().

For all three functions, *sound* should eventually be deallocated with **SNDFree**().

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError**(), is returned.

ERRORS

If an error occurs, the SNDSoundStruct isn't created.

SEE ALSO

SNDFree()

SNDRelease() → See SNDAcquire()

SNDReserve(), SNDUnreserve()

SUMMARY Reserve sound resources for recording or playback

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDReserve**(int *soundResource*, int *priority*)
int **SNDUnreserve**(int *soundResource*)

DESCRIPTION

SNDReserve() attempts to establish the exclusive use of the sound resources specified in **soundResource**, a value that's created by (bitwise) or'ing a combination of the following resource codes:

Code	Resource
SND_ACCESS_OUT	sound out
SND_ACCESS_IN	sound in
SND_ACCESS_DSP	the DSP

The *priority* argument sets the priority of the reservation (0 is the lowest priority). In general, a process has exclusive access to the resources that it reserves. However, another process can overrule a reservation by specifying a higher priority. Use of **SNDReserve()** is optional; prioritized access to the appropriate resource is established when either the **SNDStartPlaying()** or the **SNDStartRecording()** function is called. The process should eventually free its reserved resources by calling **SNDUnreserve()**. Sound resources are automatically freed when the process terminates.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

ERRORS

If **SNDReserve()** is unable to reserve any one of the resources specified in *soundResource*, it won't reserve any of them.

SNDReset() → See SNDAcquire()

SNDRunDSP() → See SNDAcquire()

SNDSampleCount(), SNDBytesToSamples(), SNDSamplesToBytes()

SUMMARY Measure samples in a sound

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDSampleCount**(SNDSoundStruct **sound*)
int **SNDBytesToSamples**(int *byteCount*, int *channelCount*, int *dataFormat*)
int **SNDSamplesToBytes**(int *sampleCount*, int *channelCount*, int *dataFormat*)

DESCRIPTION

SNDSampleCount() returns the number of sample frames, or channel-independent samples, in *sound*. The sound must contain sampled data.

SNDBytesToSamples() returns the number of samples contained in *byteCount* bytes of sound data with the given channel count and data format. **SNDSamplesToBytes**() performs the inverse operation, returning the number of bytes needed to store *sampleCount* samples. The value returned by **SNDSamplesToBytes**() is useful for computing the *dataSize* argument to **SNDAlloc**().

RETURN

If *sound* doesn't contain sampled data (or if for any other reason the sample count can't be determined), **SNDSampleCount**() returns –1. **SNDBytesToSamples**() and **SNDSamplesToBytes**() return 0 if *dataFormat* isn't a sampled sound format and –1 if *dataFormat* isn't recognized.

SNDSamplesProcessed() → See SNDStartPlaying()

SNDSamplesToBytes() → See SNDSampleCount()

SNDSetCompressionOptions() → See SNDCompressSound()

SNDSetFilter() → See SNDSetVolume()

SNDSetHost() → See SNDSetVolume()

SNDSetMute() → See SNDSetVolume()

SNDSetVolume(), SNDGetVolume(), SNDSetMute(), SNDGetMute(), SNDSetFilter(), SNDGetFilter(), SNDSetHost()

SUMMARY Sound playback utilities

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDSetVolume**(int *left*, int *right*)
int **SNDGetVolume**(int **left*, int **right*)
int **SNDSetMute**(int *speakerOn*)
int **SNDGetMute**(int **speakerOn*)
int **SNDSetFilter**(int *filterOn*)
int **SNDSetFilter**(int **filterOn*)
int **SNDSetHost**(char **newHostname*)

DESCRIPTION

SNDSetVolume() sets the sound playback level for the left and right channels, specified as an integer between 1 and 43 (inclusive). This only affects the signal to the internal speaker and the stereo headphone jack; the line-out level is undisturbed. **SNDGetVolume()** returns, in its arguments, pointers to the playback levels of either channel.

SNDSetMute() mutes and unmutes the internal speaker and headphone level as *speakerOn* is 0 and nonzero, respectively. **SNDGetMute()** returns, in its argument, a pointer to the mute status.

SNDSetFilter() turns the low-pass filter off or on as *filterOn* is 0 or nonzero, respectively. **SNDGetFilter()** returns, in its argument, a pointer to the state of the filter. The filter is automatically turned on while sounds whose format is SND_FORMAT_EMPHASIZED or SND_FORMAT_COMPRESSED_EMPHASIZED are being played.

SNDSetHost() gives you access to the named host for subsequent playbacks or recordings. If *newHostname* is NULL or a zero-length string, the default (the local host) is restored.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

SNDSoundError()

SUMMARY Describe a sound error

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

char ***SNDSoundError**(int *err*)

DESCRIPTION

SNDSoundError() returns a pointer to a string that describes the given error code. The following are defined as error codes:

Code	String
SND_ERR_NONE	""
SND_ERR_NOT_SOUND	"Not a sound"
SND_ERR_BAD_FORMAT	"Bad data format"
SND_ERR_BAD_RATE	"Bad sampling rate"
SND_ERR_BAD_CHANNEL	"bad channel count"
SND_ERR_BAD_SIZE	"bad size"
SND_ERR_BAD_FILENAME	"Bad file name"
SND_ERR_CANNOT_OPEN	"Cannot open file"
SND_ERR_CANNOT_WRITE	"Cannot write file"
SND_ERR_CANNOT_READ	"Cannot read file"
SND_ERR_CANNOT_ALLOC	"Cannot allocate memory"
SND_ERR_CANNOT_FREE	"Cannot free memory"
SND_ERR_CANNOT_COPY	"Cannot copy"
SND_ERR_CANNOT_RESERVE	"Cannot reserve access"
SND_ERR_NOT_RESERVED	"Access not reserved"
SND_ERR_CANNOT_RECORD	"Cannot record sound"
SND_ERR_ALREADY_RECORDING	"Already recording sound"
SND_ERR_NOT_RECORDING	"Not recording sound"
SND_ERR_CANNOT_PLAY	"Cannot play sound"
SND_ERR_ALREADY_PLAYING	"Already playing sound"
SND_ERR_NOT_IMPLEMENTED	"Not implemented"
SND_ERR_NOT_PLAYING	"Not playing sound"
SND_ERR_CANNOT_FIND	"Cannot find sound"
SND_ERR_CANNOT_EDIT	"Cannot edit sound"
SND_ERR_BAD_SPACE	"Bad memory space in DSP load image"
SND_ERR_KERNEL	"Mach kernel error"
SND_ERR_BAD_CONFIGURATION	"Bad configuration"
SND_ERR_CANNOT_CONFIGURE	"Cannot configure"
SND_ERR_UNDERRUN	"Data underrun"
SND_ERR_ABORTED	"Aborted"

(continued)

Code	String
SND_ERR_BAD_TAG	"Bad tag"
SND_ERR_CANNOT_ACCESS	"Cannot access hardware resources"
SND_ERR_TIMEOUT	"Timeout"
SND_ERR_BUSY	"Hardware resources already in use"
SND_ERR_CANNOT_ABORT	"Cannot abort operation"
SND_ERR_INFO_TOO_BIG	"Information string too large"
SND_ERR_UNKNOWN	"Unknown error"

SNDStartPlaying(), SNDPlaySoundfile(), SNDStartRecording(), SNDStartRecordingFile(), SNDWait(), SNDStop(), SNDSamplesProcessed(), SNDModifyPriority()

SUMMARY Recording and playing a sound

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDStartPlaying**(SNDSoundStruct ***sound*, int *tag*, int *priority*, int *preempt*, SNDNotificationFun *beginFun*, SNDNotificationFun *endFun*)
int **SNDPlaySoundfile**(char ***path*, int *priority*)
int **SNDStartRecording**(SNDSoundStruct ***sound*, int *tag*, int *priority*, int *preempt*, SNDNotificationFun *beginFun*, SNDNotificationFun *endFun*)
int **SNDStartRecordingFile**(char ***fileName*, SNDSoundStruct ***sound*, int *tag*, int *priority*, int *preempt*, SNDNotificationFun *beginFun*, SNDNotificationFun *endFun*)
int **SNDStop**(int *tag*)
int **SNDWait**(int *tag*)
int **SNDSamplesProcessed**(int *tag*)
int **SNDModifyPriority**(int *tag*, int *newPriority*)

DESCRIPTION

SNDStartPlaying() initiates the playback of *sound*. The function returns immediately while the playback continues in a background thread. During playback, the sound is played on the internal speaker and sent to the stereo line-out jacks.

The *tag* argument is an arbitrary positive integer that the caller supplies to identify the playback session in subsequent calls to **SNDWait()**, **SNDStop()**, **SNDSamplesProcessed()**, and **SNDModifyPriority()**. You should never set a sound's tag to 0.

The value of *priority* establishes the sound's right to use the playback resources. The lowest priority is 0, larger numbers signify higher priorities. Negative priorities are reserved. A call to **SNDStartPlaying()** will interrupt a currently playing sound if the

new sound has a higher priority. If the new sound has a lower priority, the old sound continues and the new sound is put in a sound playback queue. Sounds in the queue are sorted by priority.

A nonzero *preempt* flag is used for urgent sounds, such as system beeps. Preemption allows a new sound to interrupt a sound that has the same (or lower) priority. However, if the new sound can't interrupt, it isn't put in the queue.

beginFun and *endFun* are user-defined notification functions that are automatically called when the sound begins playing and when it ends, respectively. A notification function is defined as an integer function with three arguments:

typedef int (*SNDNotificationFun)(SNDSoundStruct **sound*, int *tag*, int *err*);

The *sound* and *tag* arguments are taken directly from the **SNDStartPlaying()** call. The *err* argument is one of the error codes listed in **SNDSoundError()** and is generated automatically to inform the notification function of the state of the playback. The return value is ignored. The value SND_NULL_FUN should be used to specify no function as either *beginFun* or *endFun*.

SNDPlaySoundfile() plays the soundfile named *path*. As with **SNDStartPlaying()**, the function returns immediately while playback continues in a background thread. Playback interrupts a currently playing sound of the same or lower priority.

The arguments to **SNDStartRecording()** are like those to **SNDStartPlaying()**. The sound resource used for recording is implied by information in *sound*'s header; currently, two configurations are allowed:

- If the sound is one channel of mulaw format (SND_FORMAT_MULAW) at the CODEC sampling rate (SND_RATE_CODEC), then the recording is made from the CODEC input (the microphone jack at the back of the monitor).

- If the format is one of the DSP data or compressed formats, the recording is made from the DSP port. In the case of a compressed format, the sound is compressed according to options set by **SNDSetCompressionOptions()**.

Like playback, recording is performed in a background thread. The recording completes when the storage allocated for the sound is filled with data.

SNDStartRecordingFile() is similar to **SNDStartRecording()**, but the sound is written directly to the file *fileName*. The *sound* argument is used for its size and format information.

SNDStop() terminates the playback or recording session that has a tag of *tag*.

SNDWait() returns only when the playback or recording with a tag of *tag* has completed. Note that if you call this function from the main thread of an application that has an asynchronous event-driven user interface, the interface will be effectively frozen until this function returns.

SNDSamplesProcessed() returns the number of samples that have been played or recorded so far in the playback or recording with a tag of *tag*.

SNDModifyPriority() resets the priority, as *newPriority*, of the playback or recording that has a tag of *tag*.

For these last four functions, a tag of 0 acts as a wildcard, matching the tag of any recording or playback that's currently in progress.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

SNDStartRecording() → See SNDStartPlaying()

SNDStartRecordingFile() → See SNDStartPlaying()

SNDStop() → See SNDStartPlaying()

SNDUnreserve() → See SNDReserve()

SNDWait() → See SNDStartPlaying()

SNDWrite() → See SNDWriteSoundfile()

SNDWriteHeader() → See SNDWriteSoundfile()

SNDWriteSoundfile(), SNDWrite(), SNDWriteHeader()

SUMMARY Write a sound to a file

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sound.h>

int **SNDWriteSoundfile**(char **path*, SNDSoundStruct **sound*)
int **SNDWrite**(int *fd*, SNDSoundStruct **sound*)
int **SNDWriteHeader**(int *fd*, SNDSoundStruct **sound*)

DESCRIPTION

SNDWriteSoundfile() writes the specified sound structure as the soundfile. *path* is a full pathname that should include the ".snd" extension (the convention for soundfiles). The function automatically opens and closes the file.

SNDWrite() also writes a complete soundfile, but its argument is a file descriptor rather than a pathname. The file must be open for writing.

With both **SNDWriteSoundfile()** and **SNDWrite()**, the actual sound data is written as a contiguous block, even if *sound* is fragmented. However, *sound* itself isn't affected—if it's fragmented, it remains fragmented.

SNDWriteHeader() is similar to **SNDWrite()**, but it only writes *sound*'s header to the file.

RETURN

If no error occurs, SND_ERR_NONE is returned. Otherwise, an error code, as described in **SNDSoundError()**, is returned.

Sound/DSP Driver Functions

These functions access the sound/DSP driver. For brevity, this driver is referred to as "the sound driver" throughout the following.

snddriver_dsp_boot(), snddriver_dsp_reset()

SUMMARY Start the DSP

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_boot**(port_t *commandPort*, int **bootImage*, int *imageSize*, int *priority*)
kern_return_t **snddriver_dsp_reset**(port_t *commandPort*, int *priority*)

DESCRIPTION

snddriver_dsp_boot() enqueues a command to boot the DSP. The arguments are as follows:

- *commandPort* is the DSP command port, as retrieved by **snddriver_get_dsp_cmd_port()**.
- *bootImage* is a pointer to a DSP program image that's downloaded to the DSP (program memory location 0x0) and immediately executed. The image is created by reading a ".lod" file that's assembled from DSP56001 assembly code.
- *imageSize* is the size of the DSP boot image, in bytes. The image must not exceed 512 words (24-bit DSP words right-justified within 32-bit integers).
- *priority* is one of the three priority constants SNDDRIVER_LOW_PRIORITY, SNDDRIVER_MED_PRIORITY, or SNDDRIVER_HIGH_PRIORITY. The sound driver sorts the commands in its DSP command queue according to priority.

Booting the DSP clears neither external memory nor on-chip data memory.

snddriver_dsp_reset() puts the DSP in its reset state. By this it's meant that the DSP's execution is immediately halted and a bootstrap program is awaited. Booting the DSP automatically resets it, thus you don't need to call this function before calling **snddriver_boot_dsp()**.

RETURN

Returns an error code: 0 on success, nonzero on failure.

snddriver_dsp_dma_read() → See snddriver_dsp_dma_write()

snddriver_dsp_dma_write(), snddriver_dsp_dma_read()

SUMMARY Transfer data to and from the DSP via DMA

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_dma_write**(port_t *commandPort*, int *elementCount*, int *dataFormat*, pointer_t *data*)
kern_return_t **snddriver_dsp_dma_read**(port_t *commandPort*, int *elementCount*, int *dataFormat*, pointer_t *data*)

DESCRIPTION

These functions enqueue commands that perform application-initiated DMA transfers to and from the DSP. You must include complex DMA protocol to use these functions. The arguments to the two functions are similar:

- *commandPort* is the DSP command port, as retrieved by **snddriver_get_dsp_cmd_port()**.
- *elementCount* is the number of data elements to send during each transfer.
- *dataFormat* is an integer constant that describes the size and packing of an individual data element. These are

DSP_MODE8	1 byte per element
DSP_MODE16	2 bytes per element
DSP_MODE24	3 bytes per element
DSP_MODE32	3 bytes per element, right-justified in 4
DSP_MODE2416	2 bytes per element, packed and right-justified in 4

- *data* is a pointer to the data that you're transferring.

There are three rules regarding the size and alignment of a DMA transfer buffer:

- The size in bytes of a single DMA transfer buffer, reckoned as *elementCount* * bytes-per-element, must be a multiple of 16. Note that bytes-per-element isn't given directly as an argument.
- The data must be "quad-aligned"; in other words, the starting address (*data*) must be a multiple of 16.
- All the data in a transfer buffer must lie on the same page of virtual memory.

If you're writing data, the **snddriver_dsp_dma_write()** function enqueues a command to send the data to the DSP and then immediately returns.
snddriver_dsp_dma_read(), on the other hand, waits until it has read the prescribed amount of data and returns with *data* filled. DMA-transfer commands are always enqueued with high priority.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_dsp_read(), **snddriver_dsp_write()**, **snddriver_dsp_protocol()**

snddriver_dsp_host_cmd()

SUMMARY Enqueue a DSP command

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_host_cmd**(port_t *commandPort*, u_int *hostCommand*, u_int *priority*)

DESCRIPTION

snddriver_dsp_host_cmd() enqueues a command on the sound driver's DSP command queue that interrupts the DSP and causes it to execute one of 32 interrupt routines (or *host commands*). Its arguments are as follows:

- *commandPort* is the DSP command port, as retrieved by **snddriver_get_dsp_cmd_port()**.
- *hostCommand* is an integer that represents the host command you want to execute. The first 22 host commands are already defined (or reserved). The host commands provided by NeXT are represented by constants (prefix "DSP_hc_") that are defined in **/usr/include/nextdev/snd_dsp.h**. Creating your own host command requires a familiarity with DSP programming that lies beyond the scope of this description.
- *priority* is one of the three priority constants SNDDRIVER_LOW_PRIORITY, SNDDRIVER_MED_PRIORITY, or SNDDRIVER_HIGH_PRIORITY. The sound driver sorts the commands in its DSP command queue according to priority.

When the DSP receives a host command, it sets the HC flag in the Command Vector Register. After executing the command, the DSP clears the flag. You should always

precede a call to **snddriver_dsp_host_cmd()** with a call to **snddriver_dspcmd_req_condition()** that waits for HC to clear in order to avoid overwriting a previously requested, but as yet unexecuted, host command:

```
/* CVR_HC is defined in <nextdev/snd_dspreg.h> */
err = snddriver_dspcmd_req_condition(commandPort, CVR_HC, 0, ...);
if (err != 0)
    . . .

/* Now enqueue the host command request. */
err = snddriver_dsp_host_cmd(...);
if (err != 0)
    . . .
```

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_dspcmd_req_condition()

snddriver_dsp_protocol()

SUMMARY Set the sound driver's protocol vis-a-vis the DSP

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_protocol**(port_t *devicePort*, port_t *ownerPort*, int *protocol*)

DESCRIPTION

snddriver_dsp_protocol() lets you establish the manner in which the sound driver communicates with the DSP; specifically, it determines whether to create 0, 1, or 2 DSP-reply buffers and whether DSP interrupts are enabled. The existence of the DSP-reply buffers determines whether you can use streams to transfer data.

The function's first two arguments are the sound driver device port and the DSP owner port, as acquired through **SNDAcquire()**.

protocol is the heart of the matter: It's a code that represents the protocol that you wish to establish. There are two ways to create the appropriate protocol: If you're using streams to access the DSP, then you should pass the protocol variable that's modified by calls to **snddriver_stream_setup()**, as explained (with an example) in the

description of that function. Alternatively—or in addition to the foregoing—you can create a protocol code by or'ing the following DSP protocol constants:

- SNDDRIVER_DSP_PROTO_RAW represents the barest protocol. The sound driver makes no assumptions about how the DSP is being used: No DSP-reply buffers are created and the DSP can't interrupt the host. You can't use streams in raw protocol; to transfer data, you use the **snddriver_dsp_write()** and **snddriver_dsp_read()** functions.

All the other protocols create at least one DSP-reply buffer and allow DSP interrupts, thus allowing you to transfer data through a stream:

- SNDDRIVER_DSP_PROTO_DSPMSG ("DSP-message") creates a buffer that can hold 512 DSP-reply messages. A message from the DSP (as it lies in the reply buffer) is a 24-bit word right-justified in 32 bits. To receive the contents of this buffer, you enqueue a request through **snddriver_dsp_req_msg()**.

- SNDDRIVER_DSP_PROTO_DSPERR ("DSP-error") creates an additional 512-message DSP-reply buffer that collects error messages sent from the DSP. An error message is identified as having its MSB (bit 23) set. You can request the contents of the error buffer through **snddriver_dsp_req_err()**.

- SNDDRIVER_DSP_PROTO_C_DMA ("complex DMA") implies DSP message mode (a single DSP-reply buffer is created) and allows DSP-initiated DMA transfers.

- SNDDRIVER_DSP_PROTO_HFABORT ("host flag abort") causes the driver to take note if the DSP aborts. (The DSP indicates that it has aborted by setting HF2 and HF3.)

To get the documented behavior from these protocols, you *must* include SNDDRIVER_DSP_PROTO_RAW.

Note: A protocol of 0 produces Release 1.0 behavior; this is roughly equivalent to a combination of DSP message, DSP error, and host flag abort modes.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_setup()

snddriver_dsp_read() → See snddriver_dsp_write()

snddriver_dsp_read_data() → See snddriver_dsp_write()

snddriver_dsp_read_messages() → See snddriver_dsp_write()

snddriver_dsp_reset() → See snddriver_dsp_boot()

snddriver_dsp_set_flags()

SUMMARY Set the DSP host flags

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_set_flags**(port_t *commandPort*, u_int *flagMask*, u_int *flagValue*, u_int *priority*)

DESCRIPTION

snddriver_dsp_set_flags() enqueues a command to modify one or both of the DSP host interface flags HF0 (host flag 0) and HF1 (host flag 1).

The *flagMask* argument defines which of the host flags you want to affect. The flags are represented by the constants SNDDRIVER_ICR_HF0 and SNDDRIVER_ICR_HF1. You can set both flags at the same time by or'ing these two constants. (ICR stands for "Interrupt Control Register"; this is the register to which the host flags belong.)

flagValue is the value to which you're setting the flag(s). A host flag can be either on or off, states that are also referred to as "set" and "cleared". To set a flag, you pass its constant identifier; to clear it, you pass 0. The following examples illustrate this concept:

```
/* Set HF0 (turn it on). */
snddriver_dsp_set_flags(..., SNDDRIVER_ICR_HF0,
           SNDDRIVER_ICR_HF0,...)

/* Clear HF1. */
snddriver_dsp_set_flags(..., SNDDRIVER_ICR_HF1,0,...)

/* Set both flags. */
snddriver_dsp_set_flags(...,
        SNDDRIVER_ICR_HF0 | SNDDRIVER_ICR_HF1,
           SNDDRIVER_ICR_HF0 | SNDDRIVER_ICR_HF1, ...)

/* Set HF0 and clear HF1. */
snddriver_dsp_set_flags(...,
        SNDDRIVER_ICR_HF0 | SNDDRIVER_ICR_HF1,
           SNDDRIVER_ICR_HF0, ...)
```

```
/* Clear both flags. */
snddriver_dsp_set_flags(...,
        SNDDRIVER_ICR_HF0 | SNDDRIVER_ICR_HF1, 0,...)
```

The other two arguments, *commandPort* and *priority*, are the DSP command port and command-queue priority, respectively. The DSP command port is retrieved through **snddriver_dsp_cmd_port()**; you set the priority to one of SNDDRIVER_HIGH_PRIORITY, SNDDRIVER_MED_PRIORITY, or SNDDRIVER_LOW_PRIORITY.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_dspcmd_req_condition()

snddriver_dsp_write(), snddriver_dsp_read(), snddriver_dsp_read_data(), snddriver_dsp_read_messages()

SUMMARY Transfer data to and from the DSP

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dsp_write**(port_t *commandPort*, void **buffer*,
int *elementCount*, int *elementSize*, int *priority*)
kern_return_t **snddriver_dsp_read**(port_t *commandPort*, void **buffer*,
int *elementCount*, int *elementSize*, int *priority*)
kern_return_t **snddriver_dsp_read_messages**(port_t *commandPort*, void **buffer*,
int *elementCount*, int *elementSize*, int *priority*)
kern_return_t **snddriver_dsp_read_data**(port_t *commandPort*, void ***buffer*,
int *elementCount*, int *elementSize*, int *priority*)

DESCRIPTION

snddriver_dsp_write() enqueues a command to perform a one-shot, application-initiated data transfer to the DSP; **snddriver_dsp_read()** brings data back from the DSP in a like manner. You generally use these functions if you have a small amount of data to transfer or if the transfers are infrequent enough that the overhead of the obvious alternative—setting up a DMA stream—would be exorbitant.

The other two functions, **snddriver_dsp_read_messages()** and **snddriver_dsp_read_data()** are auxiliary to **snddriver_dsp_read()**. When you call **snddriver_dsp_read()**, it, in turn, calls one of the auxiliary functions; which of the two

functions it calls depends on the current DSP protocol, as described below. You can call these functions yourself by-passing **snddriver_dsp_read()**, although you should adhere to the same protocol rules that **snddriver_dsp_read()** obeys.

The arguments to all four functions are similar:

- *commandPort* is the DSP command port, as retrieved through **snddriver_get_dsp_cmd_port().**

- *buffer*, as used by **snddriver_dsp_write()**, is a pointer to the data you want to send to the DSP. For the **snddriver_dsp_read...()** functions, it's a pointer to the location where you want the retrieved data to be stored. Note that for **snddriver_dsp_read_data()**, *buffer* is the address of a pointer; this allows the function to allocate memory for the data if you haven't allocated it yourself.

- *elementCount* and *elementSize* are the number of data elements to transfer and the size, in bytes, of a single element, respectively.

- *priority* is an integer used to sort the command on the DSP command queue. The sound driver defines three priorities represented by the constants SNDDRIVER_LOW_PRIORITY, SNDDRIVER_MED_PRIORITY, and SNDDRIVER_HIGH_PRIORITY. You normally set all application-initiated data transfers to low priority, thus reserving medium and high priority for operations that need to jump to the head of the DSP command queue.

Of these functions, **snddriver_dsp_write()** is most straightforward: When it's called, a transfer-data-to-the-DSP command is sorted (by priority) into the DSP command queue. If, when its turn comes, the command can't be executed, the driver simply pushes it back on the queue and tries again. No other commands of equal or lower priority can be executed while a frustrated write command is sitting on top of the queue. Note, however, that higher priority commands *will* get through.

As mentioned earlier, **snddriver_dsp_read()** calls one of its two auxiliary functions as determined by the current DSP protocol:

- If your application is in raw protocol, then **snddriver_dsp_read_data()** is used to read data from the DSP transmit registers.

- If DSP message protocol is included, **snddriver_dsp_read_messages()** is used to read data from the DSP-reply buffer.

The difference between the two mechanisms is generally transparent such that you can call **snddriver_read_data()** without regard for the current protocol. However, the manner in which either of the underlying functions handles incomplete reads can influence the design of your application: If the read can't be completed (typically because the DSP hasn't generated enough data), **snddriver_dsp_read_data()** blocks the DSP command queue in the fashion of **snddriver_dsp_write()**. In the same situation, **snddriver_dsp_read_messages()** waits for more data without blocking the

command queue. Thus **snddriver_dsp_read_messages()** can safely be called from a separate thread at any time. This isn't true of **snddriver_dsp_read_data()**; you should be scrupulous about ensuring that sufficient data has been processed by the DSP before you attempt to read it through this function (or through **snddriver_dsp_read()** while in raw protocol).

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_dsp_dma_read(), **snddriver_dsp_dma_write()**

snddriver_dspcmd_req_condition

SUMMARY Request a DSP host interface register condition

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dspcmd_req_condition**(port_t *commandPort*,
u_int *registerMask*, u_int *conditionFlags*, int *priority*, port_t *replyPort*)

DESCRIPTION

snddriver_dspcmd_req_condition() does two things: It causes the DSP command queue to block until the specified host interface register condition is true, and it registers a request for an asynchronous message to be sent to *replyPort* when the condition is fulfilled. The function returns immediately.

You specify a condition through a combination of the *registerMask* and *conditionFlags* arguments:

- *registerMask* specifies the host interface registers (actually, the bits therein) that you're interested in. It's created by or'ing the register-bit constants defined in **<nextdev/snd_dspregs.h>**. A subset of these are also defined as sound driver constants in **<sound/sounddriver.h>**.

- *conditionFlags* encodes the states of the register bits that define a satisfied condition. To specify that you want a register bit set, you **or** the register-bit constant that represents it; if you want it clear, you exclude the constant. If you want all the specified bits to be clear, set *conditionFlags* to 0.

In the following example, the command queue is blocked until HF0 is set and HF1 is clear (both flags are in the Interrupt Control Register):

```
/* Block until HF0 is set and HF1 is clear. */
snddriver_dspcmd_req_condition(...,
        SNDDRIVER_ICR_HF0 | SNDDRIVER_ICR_HF1,
        SNDDRIVER_ICR_HF2, ...)
```

The condition request is sorted into the DSP command queue according to *priority*, which must be one of SNDDRIVER_LOW_PRIORITY, SNDDRIVER_MED_PRIORITY, or SNDDRIVER_HIGH_PRIORITY.

The message that's sent to the reply port when the condition is fulfilled contains the value of the host interface register. By setting the *registerMask* argument to 0, you can use the **snddriver_dspcmd_req_condition()** function to simply poll for this value.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_dsp_set_flags()

snddriver_dspcmd_req_err() → See snddriver_dspcmd_req_msg()

snddriver_dspcmd_req_msg(), snddriver_dspcmd_req_err()

SUMMARY Request the contents of the DSP-reply buffers

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_dspcmd_req_msg**(port_t *commandPort*, port_t *replyPort*)
kern_return_t **snddriver_dspcmd_req_err**(port_t *commandPort*, port_t *replyPort*)

DESCRIPTION

The **snddriver_dspcmd_req_msg()** and **snddriver_dspcmd_req_err()** functions are part of the mechanism by which your application retrieves messages from the sound driver's DSP-reply buffers. They request that the contents of the appropriate buffer (as described below) be sent in a Mach message to *replyPort*, a valid port that must already be allocated. Simply requesting a message is only half of the story: You then have to

receive the message that's been sent, usually by sitting in a **msg_receive()** loop. You typically process the Mach messages that these functions induce by passing the messages to the **snddriver_reply_handler()** function.

The utility of these functions depends on your application's DSP protocol:

- You should never use these functions in raw protocol since the sound driver doesn't create any DSP-reply buffers.
- By including DSP message protocol, a single DSP-reply buffer is created in which both error and non-error messages are stored; thus **...req_msg()** is of use, but **...req_err()** isn't.
- DSP error protocol deems that two buffers be created, one for error messages and the other for non-error messages. Both functions are useful in this protocol.

DSP protocol and how to set it is explained in the description of the **snddriver_set_dsp_protocol()** function. For both functions, the *commandPort* argument is the DSP command port as retrieved by **snddriver_get_dsp_cmd_port()**.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_set_dsp_protocol(), **snddriver_reply_handler()**

snddriver_get_device_parms() → See snddriver_set_device_parms()

snddriver_get_dsp_cmd_port()

SUMMARY Get the DSP command port

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_get_dsp_cmd_port**(port_t *devicePort*, port_t *ownerPort*, port_t **commandPort*)

DESCRIPTION

snddriver_get_dsp_cmd_port() attempts to get the *DSP command port*, the port through which the sound driver issues commands to the DSP. If it's successful, the port is returned in the *commandPort* argument, which needn't have been previously allocated.

The first two arguments, *devicePort* and *ownerPort*, are the sound driver device port and the DSP owner port, as acquired through **SNDAcquire()**.

The DSP command port is required as an argument by almost all sound driver functions that communicate with the DSP. The one notable exception, for which you don't have to get the command port as it's gotten implicitly when needed, is if you send and retrieve DSP data via streams after having booted the DSP through the **SNDBootDSP()** sound library function. But even in this case getting the command port as a reflex to getting the DSP owner port won't serve you ill.

RETURN

Returns an error code: 0 on success, nonzero on failure.

snddriver_get_volume() → See snddriver_set_device_parms()

snddriver_new_device_port()

SUMMARY Reallocate the sound driver device port

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_new_device_port**(port_t *devicePort*, port_t *superuserPort*, port_t **newDevicePort*)

DESCRIPTION

This function deallocates the sound driver device port *devicePort*, as previously acquired through **SNDAcquire()**, then allocates a new port to the device which it returns as *newDevicePort*. When the old device port is deallocated, so, too, are all its resource owner ports and sound streams; thus any currently operating sound driver tasks, such as recording and playing sounds, are aborted. Because of the ruthlessness of this act, you must be the UNIX® superuser to call this function, as verified by the *superuserPort* argument, for which you should pass the return value of

host_priv_self(). The new device port's registration with regard to the Network Name Server is the same as that of the old; in other words, if the old port had been registered (through **netname_check_in()**), the new one will be registered automatically.

RETURN

Returns an error code: 0 on success, nonzero on failure.

snddriver_reply_handler()

SUMMARY Respond to asynchronous sound driver messages

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_reply_handler**(msg_header_t **reply*,
snddriver_handlers_t **handlers*)

DESCRIPTION

snddriver_reply_handler() helps your application respond to asynchronous sound driver messages. The function is designed around the **snddriver_handlers** structure, which provides a correspondence between the sound driver messages and a list of C functions that you provide. When you receive a message from the sound driver, you pass the message and a **snddriver_handlers** structure to **snddriver_reply_handler()** which then executes the handler function that corresponds to the message.

The definition of the **snddriver_handlers** structure (**typedef**'d, for convenience, as **snddriver_handlers_t**) reveals the nature of the functions that you can register as reply handlers:

```
typedef struct snddriver_handlers {
    void                        *arg;
    int                          timeout;
    sndreply_tagged_t            started;
    sndreply_tagged_t            completed;
    sndreply_tagged_t            aborted;
    sndreply_tagged_t            paused;
    sndreply_tagged_t            resumed;
    sndreply_tagged_t            overflow;
    sndreply_recorded_data_t     recorded_data;
    sndreply_dsp_cond_true_t     condition_true;
    sndreply_dsp_msg_t           dsp_message;
    sndreply_dsp_msg_t           dsp_error;
} snddriver_handlers_t;
```

The structure's **arg** field is a value that's passed to the reply handlers when they're called by **snddriver_reply_handler()**; you can set it to whatever value best suits your application, but keep in mind that the value must fit within the size of a pointer (four bytes). The **timeout** field is currently unused.

The final ten fields are the heart of the structure: Each corresponds to a particular sound driver message. The first six of these correspond to messages that indicate a change in the state of a stream ("stream-state" messages); in other words, the sound driver sends a specific message when a stream starts processing data, when it completes its processing, when it aborts, and so on. By setting a field to a particular function, you register that function as the handler for the message to which the field corresponds. For example, to establish a function named **handleStreamStart()** as the function that's executed when your application receives a stream-started message from the sound driver, you would do the following:

```
/* Create a snddriver_handlers_t and register the
 * function handleStreamStart() (which we'll assume already
 * exists) to process stream-started messages.
 */
snddriver_handlers_t replyHandlers;
replyHandlers.started = handleStreamStart;
```

While this registers **handleStreamStart()** as the handler for stream-started messages, you must also tell the sound driver that you actually want such messages sent to your application. To do this, you set the *msgStarted* boolean argument to true when you call **snddriver_stream_start_reading()** or **snddriver_stream_start_writing()**. Analogous *msg...* message flags exist for the other five stream-state messages.

When the sound driver sends a stream-state message to your application, it sends it to the port that you specify as the last argument (*replyPort*) to **snddriver_stream_start_reading()** or **snddriver_stream_start_writing()**. To receive the message, you create a **msg_header_t** structure, set its **local_port** field to the stream's reply port, and then wait for the message to arrive by sitting in a message receive (**msg_receive()**) loop. After so capturing the message, you then pass it, along with your handler structure, to **snddriver_reply_handler()**. This is demonstrated by the example below.

Notice, from the definition of **snddriver_handlers**, that the six stream-state handlers are all of type **sndreply_tagged_t**. This type represents a two-argument function protocol that's defined as

```
typedef void (*sndreply_tagged_t) (void *arg, int tag);
```

The functions that you register to handle the stream-state messages must adhere to this protocol. The values of the arguments are set by **snddriver_reply_handler()**:

- *arg* is given the value of the *arg* field of the **snddriver_handlers** structure in which the function is registered. As mentioned earlier, you can set the structure's *arg* field to a (four-byte) value that suits the needs of your application.

- *tag* is the region-identifying tag that you provide as an argument to **snddriver_stream_start_writing()** or **snddriver_stream_start_reading()**.

The seventh of the ten **snddriver_handlers** handler fields—**recorded_data**—also applies to streams. However, unlike the fist six, which are optional, **recorded_data** is essential when you're reading data from a stream. Its importance arises from the way that the sound driver handles read data: It keeps the data in the kernel's virtual memory until you ask to bring it into your application. The only way to bring this data back is to supply a **recorded_data** handler that does so. The following program excerpt, a modified and distilled version of the example given in **/NextDeveloper/Examples/DSP/SoundDSPDriver/dsp_example_3/**, demonstrates a typical way to achieve this effect. In the example, details such as acquiring the sound driver and sound resource owner ports are omitted. The read stream shown here is anonymous—the code can be used equally well for a stream that reads from sound-in or from the DSP:

```
/* The code shown in the example requires the following header
   files  */
#import <sound/sounddriver.h>
#import <mach.h>

/* Define a read stream tag, a read pointer, and a byte count
   variable. */
#define READ_TAG 1
static short *readData;
static int readCount;

/* Create a recorded_data handler; the function's protocol is
 * explained following the example.
 */
static void read_completed(void *arg, int tag, void *kernelData,
                           int size)
{
    /* Make sure this is the read stream. */
    if (tag == READ_TAG) {
        readData = (short *)kernelData;
        readCount = size;
    }
}

main()
{
    /* Define a read port, a reply port, and a reply structure. */
    port_t readPort, replyPort;
    snddriver_handlers_t replyHandlers;

    /* Allocate a Mach message header. msg_header_t and MSG_SIZE_MAX
     * (and msg_receive, below) are defined in mach.h.
     */
    msg_header_t *reply_msg = (msg_header_t *)malloc(MSG_SIZE_MAX);
```

```
    /* Create an error-check variable. */
    int err;

    /* Allocate the reply port. */
    err = port_allocate(task_self(), &replyPort);
    if (err != 0)
        . . .

    /* Set the recorded_data handler. `*/
    replyHandlers.recorded_data = read_completed;

    /* Set the amount of data you want to read; for the purposes of
     * this example, an arbitrary amount is specified.
     */
    readCount = 1024;

    /* Here, a number of activities -- such as acquiring the sound
     * driver port and sound resource owner port, setting up a read
     * stream through snddriver_stream_setup(), and (possibly)
     * booting the DSP and sending it data -- are omitted.
     */
    . . .

    /* Enqueue a read request.  The six 0 arguments are the message
     * request flags.
     */
    err = snddriver_stream_start_reading(readPort, 0, readCount,
                        READ_TAG, 0,0,0,0,0,0, replyPort);
    if (err != 0)
        .  .  .

    /* Sit in a message-receive loop. */
    while(1) {
        /* Set up the reply message.  This must be done inside the
         * loop since msg_receive() may change the message header.
         */
        replyMsg->msg_size = MSG_SIZE_MAX;
        replyMsg->msg_local_port = replyPort;
        err = msg_receive(replyMsg, MSG_OPTION_NONE, 0);
        if (err != 0)
            . . .
    }

    /* Dispatch the message to the reply handlers.*/
    err = snddriver_reply_handler(replyMsg, &replyHandlers);
    if (err != 0)
        . . .
    /* Provide a means to break out of the loop. */
    . . .
    }
}
```

As implied by the example, you don't need to tell the sound driver that you want a data-recorded message to be sent to your application; the message is always sent automatically. The example also illustrates the rule that the reply port used to receive messages while in the **msg_receive()** loop is that which is specified as the final argument to the **snddriver_stream_start_reading()** function.

The data type of the **recorded_data** field dictates the protocol of the function that you design to bring data back to the application. The type is **sndreply_recorded_data_t**:

```
typedef void (*sndreply_recorded_data_t)(void *arg, int tag,
                void *kernelData, int size);
```

The first two arguments, *arg* and *tag*, are the same as in the **snddreply_tagged_t** type. *kernelData* is a pointer to the recorded data as it resides in the kernel; *size* is the size of the recorded data in bytes.

The final three **snddriver_handlers** fields correspond to messages that are inspired by the DSP:

- The **condition_true** handler is called when a requested DSP host interface register condition comes true. (More accurately, the handler is called when the message that indicates that the condition is true is passed to **snddriver_reply_handler()**.)

- **dsp_message** handles general messages that the sound driver receives from the DSP.

- **dsp_error** does the same for DSP error messages.

For each of these three handlers, there is a corresponding sound driver function that enqueues a request for a condition, a DSP message, or a DSP error message, respectively:

- **snddriver_dspcmd_req_condition()** blocks the DSP command queue until the state of the DSP host interface registers satisfies a requested condition.

- **snddriver_dspcmd_req_msg()** requests that the messages in the DSP-reply buffer be sent to your application. You must include DSP-message protocol for this to have an effect.

- **snddriver_dspcmd_req_err()** requests that the 512-byte DSP-reply error buffer be sent in a message. You must include DSP-error protocol for this to have an effect.

As with the **snddriver_stream_start...()** functions, the three DSP request functions require that you provide a reply port as an argument. It's to this reply port that the sound driver sends the requested DSP-inspired messages. A single call to one of these functions causes a single reply message to be sent to your application. Thus, for each call to **snddriver_dspcmd_req_msg()**, for example, your application will receive one message from the sound driver.

The **condition_true** handler is of type **sndreply_dsp_cond_true_t**:

```
typedef void (*sndreply_dsp_cond_true_t)(void *arg, u_int mask,
                u_int flags, u_int registers);
```

arg is the value of the **arg** field. The next two arguments, *mask* and *flags*, are given the values that were passed to **snddriver_dspcmd_req_condition()** (which also has *mask* and *flags* arguments). *registers* encodes the current status of the four DSP host interface registers in a single 32-bit vector. See the description of **snddriver_dspcmd_req_condition()** for more information on how this works.

The **dsp_message** and **dsp_error** are of type **sndreply_dsp_msg_t**:

```
typedef void (*sndreply_dsp_msg_t)(void *arg, int *data,
                int size);
```

arg is the value of the **arg** field. *data* is a pointer to the contents of the appropriate DSP-message buffer (regular or error, as the handler is **dsp_message** or **dsp_error**). *size* is the size of the buffer contents, in bytes.

snddriver_reply_handler() ignores messages for which you haven't created and registered a handler function.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_start_reading(), **snddriver_stream_start_writing()**, **snddriver_dspcmd_req_condition()**, **snddriver_dspcmd_req_msg()**, **snddriver_dspcmd_req_err()**

snddriver_set_device_parms(), snddriver_get_device_parms(), snddriver_set_volume(), snddriver_get_volume(), snddriver_set_ramp()

SUMMARY Set and get sound playback attributes

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_set_device_parms**(port_t *devicePort*, boolean_t *speakerOn*, boolean_t *filterOn*, boolean_t *zerofill*)
kern_return_t **snddriver_get_device_parms**(port_t *devicePort*, boolean_t **speakerOn*, boolean_t **filterOn*, boolean_t **zerofill*)
kern_return_t **snddriver_set_volume**(port_t *devicePort*, int *leftVolume*, int *rightVolume*)
kern_return_t **snddriver_get_volume**(port_t *devicePort*, int **leftVolume*, int **rightVolume*)
kern_return_t **snddriver_set_ramp**(port_t *devicePort*, int *rampOn*)

DESCRIPTION

These functions set and get attributes of the sound playback system. Each takes, as its first argument, the sound driver device port as acquired through **SNDAcquire**(). You needn't acquire ownership of sound-out to set the playback attributes.

snddriver_set_device_parms() sets three attributes as specified by the values of its boolean arguments:

- The internal speaker is turned on or off as *speakerOn* is true or false. Calling the function with alternating true and false *speakerOn* values is equivalent to toggling the Mute key (Command Mute) on the keyboard.

- Similarly, the value of *filterOn* turns the de-emphasis filter on or off. The filter can be controlled from the keyboard by toggling the louder key while holding down the Command key (this isn't marked on the keyboard). In addition, the de-emphasis filter is automatically turned on when a de-emphasis format sound is played and returned to its previous state when the sound is done playing.

- During playback, low sampling rate (22.05 kHz) sounds are converted to the high sampling rate (44.1 kHz) as they are sent to the DAC (which converts data at 44.1 kHz only). To do this, the sound driver emits an extra sample for every existing sample in the sound data. The value of *zerofill* determines whether these extra samples are set to 0 (true) or if they're copies of the existing samples (false). In almost all cases, copying the samples is preferable, since zerofilling results in a decrease in power. Note that you can't toggle this attribute from the keyboard. Also, keep in mind that CODEC rate sounds are converted to 22.05 kHz before being sent to the DAC and so are also affected by the state of *zerofill*.

snddriver_get_device_parms() returns, by reference in its final three arguments, the values of the attributes described above.

snddriver_set_volume() sets the volume of the internal speaker and similarly adjusts the signal that's sent to the stereo headphone jack (the signal to the line-out jacks is unaffected). The two channels of the stereo signal are set independent of each other, specified as the values of *leftSpeaker* and *rightSpeaker*. The volume of the internal speaker is the sum of these two values. Volume values are integers in the range 0 to 43, inclusive, where 0 is inaudible and 43 is full blast. You can also adjust playback volume by pressing the speaker-louder and speaker-softer keys on the keyboard. Each discrete tap on a volume key increments or decrements both the left and the right volume settings by 1.

snddriver_get_volume() returns the left and right playback volumes by reference in *leftVolume* and *rightVolume*, respectively.

By default, sounds are ramped during playback: The first few samples are ramped up from zero and the last samples are ramped down. This helps prevent clicks at the beginnings and ends of sounds. **snddriver_set_ramp()** enables or disables this feature as its *rampOn* argument is nonzero or zero. You almost always want ramping enabled; the one obvious case in which it's undesirable is if you're chaining a series of separate sounds that are meant to be played seamlessly, one immediately after the other. In this case, ramping will cause annoying amplitude dips at each seam.

RETURN

Returns an error code: 0 on success, nonzero on failure.

snddriver_set_dsp_owner_port(), snddriver_set_sndin_owner_port(), snddriver_set_sndout_owner_port()

SUMMARY Acquire ownership of sound resources

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_set_dsp_owner_port**(port_t *devicePort*, port_t *ownerPort*, port_t **negotiationPort*)
kern_return_t **snddriver_set_sndin_owner_port**(port_t *devicePort*, port_t *ownerPort*, port_t **negotiationPort*)
kern_return_t **snddriver_set_sndout_owner_port**(port_t *devicePort*, port_t *ownerPort*, port_t **negotiationPort*)

DESCRIPTION

These functions try to acquire ownership of the DSP, sound-in, or sound-out by setting the resource's owner port to a port that you supply. They duplicate part of the functionality provided by **SNDAcquire()**; the latter should, in most cases, be used to the exclusion of these.

The arguments are the same for all three functions:

- *devicePort* is a valid port to the sound driver device, as acquired through **SNDAcquire()**.

- *ownerPort* is the port that will become the owner port for the requested resource if the function is successful. You must have already allocated *ownerPort* through the function **port_allocate()**.

- If the function successfully acquires ownership of the resource, then the port pointed to by *negotiationPort* is registered as the negotiation port for the resource. However, if the function isn't successful—most likely because ownership of the resource has already been claimed—then the currently registered negotiation port is returned in the *negotiationPort* argument. By convention you point *negotiationPort* to *ownerPort* before calling these functions, thereby making the owner port accessible to other tasks. Similarly, if your bid for ownership fails and the current owner has followed this convention, then you can use the port returned in *negotiationPort* as the owner port for the resource. Note, however, that if the function call fails, there's no way to determine if the port pointed to by *negotiationPort* is actually the owner port. If you want to acquire sole ownership of a resource, set *negotiationPort* to something other than the *ownerPort* before calling these functions. This will ensure that only the caller will have access to the resource (assuming that the function is successful).

A single port can be used to claim ownership of more than one device. This is sometimes necessary when setting up a multiple-device stream (as explained in **snddriver_stream_setup()**). In the following example, the same port attempts to own both the DSP and sound-out:

```
err = port_allocate(task_self(), &ownerPort)
. . .

/* Acquire ownership of the DSP. */
err=snddriver_set_dsp_owner_port(devPort, ownerPort, &negPort);
. . .

/* Acquire ownership of sound-out. */
err=snddriver_set_sndout_owner_port(devPort,ownerPort,&negPort);
```

After you've claimed ownership of a resource, you should do something with it. With sound-in you set up a stream port through which you read (record) data. This is done by calling the **snddriver_stream_setup()** and **snddriver_stream_start_reading()** functions. Analogously, with sound-out you set up a stream through which you write (playback) data through the **snddriver_stream_start_writing()** function.

If you claim ownership of the DSP you should also acquire the DSP command port by calling **snddriver_get_dsp_cmd_port()**. Most of the functions that access the DSP require the command port as an argument. You can also set up streams to the DSP as you would to sound-in or sound-out. Successfully setting the DSP's owner port puts the DSP in its reset state.

To relinquish ownership of a resource, you deallocate the owner port by calling **port_deallocate()**:

```
err = port_deallocate(task_self(), ownerPort);
```

Deallocating a resource's owner unregisters the resource's negotiation port. All ports are automatically deallocated when your application exits.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_setup(), **snddriver_get_dsp_cmd_port()**

snddriver_set_ramp() → See snddriver_set_device_parms()

snddriver_set_sndin_owner_port() → See snddriver_set_dsp_owner_port()

snddriver_set_sndout_bufcount(), snddriver_set_sndout_bufsize(), snddriver_stream_ndma()

SUMMARY Configure stream transfer buffers

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_set_sndout_bufcount**(port_t *devicePort*, port_t *sndoutPort*, int *count*)
kern_return_t **snddriver_set_sndout_bufsize**(port_t *devicePort*, port_t *sndoutPort*, int *size*)
kern_return_t **snddriver_stream_ndma**(port_t *streamPort*, int *tag*, int *count*)

DESCRIPTION

These functions let you control the number and size of the buffers that are used to transfer data in a stream.

snddriver_set_sndout_bufcount() sets the number of buffers that are used when playing back sounds; the *count* argument, which must be greater than 0, establishes the buffer count. Four buffers are used in the default configuration.

snddriver_set_sndout_bufsize() sets the size of the sound-out buffers (in bytes) to the value of the *size* argument. This function is needed only if you're using a linked stream to sound-out (see the **snddriver_stream_setup()** function for more on linked streams). The value of *size* must be less than or equal to **vm_page_size**, the size of a page of virtual memory; the default is **vm_page_size**. If you're writing directly to the sound-out stream—in other words if the stream to sound-out is configured as SNDDRIVER_STREAM_TO_SNDOUT_22 or SNDOUT_44—then the size of the sound-out buffers is computed from the *sampleCount* argument to **snddriver_stream_setup()** and the size set here is ignored.

For both of these functions, the *devicePort* and *sndoutPort* arguments are ports to the sound driver device and to sound-out, respectively, as acquired through **SNDAcquire()**.

snddriver_stream_ndma() sets the number of DMA transfer buffers that are used to transmit and receive data that's sent to the DSP. The DMA buffer count can be set on a region-by-region basis; the stream and region therein to which a particular setting applies are identified by the *streamPort* and *tag* arguments, respectively. This function applies only to simple streams to or from the DSP; if you set up a linked stream to sound-out, then the DMA buffer count to and from the DSP is the same as the sound-out buffer count.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_setup(), **snddriver_stream_ndma()**

snddriver_set_sndout_bufsize() → See snddriver_set_sndout_bufcount()

snddriver_set_sndout_owner_port() → See snddriver_set_dsp_owner_port()

snddriver_set_volume() → See snddriver_set_device_parms()

snddriver_stream_control(), snddriver_stream_nsamples()

SUMMARY Control and query a stream

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_stream_control**(port_t *streamPort*, int *tag*, int *control*)
kern_return_t **snddriver_stream_nsamples**(port_t *streamPort*, int **byteCount*)

DESCRIPTION

snddriver_stream_control() provides control over an active stream by allowing you to apply a controlling operation to one or more of the stream's enqueued regions. The stream and the regions therein are identified by the function's first two arguments: *streamPort* is the stream's port, as created by **snddriver_stream_setup()**; *tag* is the integer identifier that you gave the region (or regions) in a previous call to **snddriver_stream_start_writing()** or **snddriver_stream_start_reading()**. A tag value of 0 causes the controlling operation to be applied to all regions enqueued on the stream. *control* specifies the controlling operation by the following constants:

- SNDDRIVER_PAUSE_STREAM causes the stream to pause. If data is currently being read from or written to the specified region, the read or write is immediately suspended. If the region isn't yet active, the pause takes effect when the region comes to the top of the stream's queue (it's paused just before the first sample is read or written).

- SNDDRIVER_RESUME_STREAM resumes a previously paused stream.

- SNDDRIVER_ABORT_STREAM terminates the stream's activity when the specified region comes to the top of the queue; the queue is then cleared. If the region is currently being acted upon, the stream is terminated immediately.

- SNDDRIVER_AWAIT_STREAM is used to retrieve a partially recorded region from a stream that's reading data—normally, you can't retrieve such data until the entire region has been filled. If the specified region is currently active, a data-recorded message is sent to the reply port that you registered in **snddriver_stream_start_reading()**. You then pass the message to **snddriver_reply_handler()** which calls the **recorded_data** reply handler. The unrecorded portion of the region continues. If the specified region isn't currently active, this has no effect.

While you can use any of these four at the same time by or'ing them in *control*, the only combination that's of use is SNDDRIVER_AWAIT_STREAM or'd with one of the other three. For example, by setting *tag* to 0 and *control* to

```
SNDDRIVER_PAUSE_STREAM | SNDDRIVER_AWAIT_STREAM
```

you immediately pause the stream and can then bring back data from the current region.

You can request that a stream-paused, stream-resumed, or stream-aborted message be sent to the reply port when you pause, resume, or abort a stream, respectively, by setting the appropriate *msg...* flag to true in your call to **snddriver_stream_start_...()**.

snddriver_stream_nsamples() returns the number of bytes (*not* samples, despite the name of the function) that have been read from or written to a particular stream. The steam is specified by *streamPort*. The byte count is returned by reference in the *byteCount* argument.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_setup(), **snddriver_stream_start_writing()**, **snddriver_stream_start_writing()**, **snddriver_reply_handler()**

snddriver_stream_nsamples() → See snddriver_stream_control()

snddriver_stream_setup()

SUMMARY Configure a sound stream

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_stream_setup**(port_t *devicePort*, port_t *ownerPort*, int *dataPath*, int *sampleCount*, int *sampleSize*, int *lowWater*, int *highWater*, int **protocol*, port_t **streamPort*)

DESCRIPTION

A stream, as it applies to the sound driver, is a path through which an indefinitely long sequence of data passes. One end of a sound driver stream typically lies in your application's memory, while at the other end is a sound device. For example, to record a sound from the microphone you create a stream from sound-in to your application. Analogously, a stream from your application to sound-out is required to play back sound data. A single stream of data can pass through more than one sound device; for example, you can send data from your application to the DSP from whence it issues directly to sound-out. Thus you can DSP-process and play your sound data in one motion, without incurring the overhead of bringing the processed data back into your application.

The **snddriver_stream_setup()** function creates a port to a sound stream. The port, returned in the *streamPort* argument, is used as an identifier in subsequent calls to functions that write to, read from, and otherwise control the stream (as listed at the end of this description).

The function's first two arguments are the usual capability ports: *devicePort* is a port to the sound driver device, and *ownerPort* is the owner port for *all* resources that are touched by the stream, as acquired through **SNDAcquire()**.

You establish the stream's course—the source and destination of its data—by setting *dataPath* to one of constants listed below. There are two types of data paths: "simple" and "linked." The simple data paths (listed below) connect your application to a sound resource:

- SNDDRIVER_STREAM_FROM_SNDIN; read samples from the CODEC microphone.
- SNDDRIVER_STREAM_TO_SNDOUT_44; write samples to the stereo DAC at the high sampling rate (44.1 kHz).
- SNDDRIVER_STREAM_TO_SNDOUT_22; write samples to the stereo DAC at the low sampling rate (22.05 kHz).
- SNDDRIVER_DMA_STREAM_TO_DSP; write data via DMA to the DSP.
- SNDDRIVER_DMA_STREAM_FROM_DSP; read data via DMA from the DSP.

Four linked data paths connect the DSP directly to sound-out:

- SNDDRIVER_STREAM_DSP_TO_SNDOUT_44 and ...SNDOUT_22; DSP-processed samples are sent directly to sound-out at the low or high sampling rate.
- SNDDRIVER_DMA_STREAM_THROUGH_DSP_TO_SNDOUT_44 and ...SNDOUT_22; data flows from your application to the DSP and thence directly to sound-out at the high or low sampling rate.

Data is transferred through a stream in buffers. The *sampleCount* argument establishes the length of a single transfer buffer in samples (or data elements); the size of a single sample is set by the *sampleSize* argument. The maximum size for a transfer buffer (in bytes) is that of a page of virtual memory, as given by the global read-only variable **vm_page_size**. Typically, the transfer buffer size is set to this limit: If, for example, the samples that you're sending through the stream are two bytes wide, then, to follow this convention, you would set *sampleCount* to **vm_page_size**/2. If the stream uses DMA, then the size of a transfer buffer (in bytes) must be a power of 2 greater than or equal to 16.

For some applications—particularly those in which latency is an issue—setting the number of transfer buffers that are used can be as important as setting the size of the buffers. This is done through the **snddriver_set_sndout_bufcount()** and **snddriver_stream_ndma()** functions.

The range of acceptable values for the *sampleSize* argument depends on the stream's data path:

- If you're reading from sound-in into your application (SNDDRIVER_STREAM_FROM_SNDIN), then *sampleSize* must be set to 1 to accommodate the 8-bit mu-law samples generated by the CODEC microphone input.

- If you're writing from your application to sound-out (SNDDRIVER_STREAM_TO_SNDOUT_44/22) or from the DSP to sound-out (SNDDRIVER_STREAM_DSP_TO_SNDOUT_44/22), then *sampleSize* must be 2 since the DAC expects 16-bit interleaved-stereo samples. Note that while the DAC processes data only at the high sampling rate, the sound driver performs the conversion from low to high for you. This isn't true for playback of CODEC-rate sounds for which you typically download a sampling-rate conversion program to the DSP, and then create a stream that goes through the DSP and then directly to sound-out. This is what the **SNDStartPlaying()** function does, for example.

- In all the other paths, your application writes to or reads from the DSP. Here, *sampleSize* can be 1, 2, or 4, according to the sample size expected by or produced by your DSP program.

The *lowWater* and *highWater* arguments are memory threshold values, measured in bytes, that are inspected by the sound driver. During an operation such as recording or playback, successive pages of sound data are locked into physical memory (or "wired down") during which time they're read from or written to. As a page is completed, it's unwired. The driver tries to maintain at least *lowWater* bytes of wired-down memory; if the amount drops below this threshold, the driver wires down pages until it reaches the *highWater* mark.

If your stream touches the DSP, then you need to set the DSP protocol by passing the appropriate value to **snddriver_dsp_protocol()**. The *protocol* argument found here helps you create this value: The function or's the appropriate protocol constants, as determined by the characteristics of the stream that you're setting up, into *protocol* and returns the new value by reference. You then pass the variable to

snddriver_dsp_protocol(). You should initialize your protocol variable to SNDDRIVER_DSP_PROTO_RAW before calling **snddriver_stream_setup()**, as shown in the following example:

```
/* Initialize the protocol variable. */
    int protocol = SNDDRIVER_DSP_PROTO_RAW;
    int err;

    /* Set up a stream to the DSP. */
    err = snddriver_stream_setup(..., SNDDRIVER_STREAM_TO_DSP,
                                 .., &protocol, ...);
    if (err != 0)
    . . .

    /* Set up a stream from the DSP. */
    err = snddriver_stream_setup(..., SNDDRIVER_STREAM_FROM_DSP,
                                 ..., &protocol, ...);
    if (err != 0)
    . . .

    /* Pass the protocol to the sound driver. */
    err = snddriver_dsp_protocol(..., protocol);
    if (err != 0)
    . . .
```

The protocol constants are described as part of the **snddriver_dsp_protocol()** function.

Having created a stream, you can read from it, write to it, and control it by passing the port returned in *streamPort* to the following functions:

- **snddriver_stream_start_reading()** and **snddriver_stream_start_writing()** read from and write to a stream, respectively. Streams from sound-in or from the DSP can only be read; similarly, streams to sound-out or to the DSP can only be written.
- **snddriver_stream_control()** pauses, resumes, and aborts an active stream.
- **snddriver_stream_nsamples()** measures the amount of data that has passed through the stream.

For sound-in and sound-out, streams are the only way to travel. This isn't true of the DSP; the sound driver provides a one-shot, non-stream DSP read and write mechanism, embodied in **snddriver_dsp_read()**, **snddriver_dsp_dma_read()**, and analogous **...write()** functions, that can be more efficient for short data transfers.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_stream_start_reading(), snddriver_stream_start_writing(), snddriver_set_sndout_bufcount(), snddriver_stream_ndma()

snddriver_stream_start_reading() → See snddriver_stream_start_writing()

snddriver_stream_start_writing(), snddriver_stream_start_reading()

SUMMARY Send data to and retrieve data fom a stream

LIBRARY libsys_s.a

SYNOPSIS

#import <sound/sounddriver.h>

kern_return_t **snddriver_stream_start_writing**(port_t *streamPort*, void **data*, int *sampleCount*, int *tag*, boolean_t *preempt*, boolean_t *deallocateWhenDone*, boolean_t *msgStarted*, boolean_t *msgCompleted*, boolean_t *msgAborted*, boolean_t *msgPaused*, boolean_t *msgResumed*, boolean_t *msgUnderrun*, port_t *replyPort*)

kern_return_t **snddriver_stream_start_reading**(port_t *streamPort*, char **filename*, int *sampleCount*, int *tag*, boolean_t *msgStarted*, boolean_t *msgCompleted*, boolean_t *msgAborted*, boolean_t *msgPaused*, boolean_t *msgResumed*, boolean_t *msgOverrun*, port_t *replyPort*)

DESCRIPTION

These two functions cause data to be written to or read from a sound stream identified by *streamPort*, which must have been created by a previous call to **snddriver_stream_setup()**. The two functions operate in much the same manner: Each invocation enqueues a single region of data that's operated on (either read from or written to) asynchronously by the sound driver. However, there's a fundamental difference between the two functions in that **...writing()** enqueues a region that you pass as the *data* argument, while **...reading()** stores the data it reads in a region that it allocates itself. To bring the read data back into your application, you must create and register a reply-handler function that transfers the data when the read is complete. The mechanism for doing this is explained (and an example given) in the **snddriver_reply_handler()** function description. Note that the **...reading()** argument *filename*—which would imply that the read data is written to a file—is currently unused. Also note that *data* is copy-on-write protected: Any changes that you make to the data after the region has been passed to **...writing()** won't be seen by the driver.

sampleCount is the number of samples in the region that's being written or read. If you're writing to the DSP, *sampleCount* must be a multiple of the *sampleCount* argument to **snddriver_stream_setup()**. In all other cases, *sampleCount* can be any value.

tag is an integer used to identify the region. While you can give each region a distinct tag, you usually create a single tag value for each stream that you set up. For example, if you have a stream that reads data from sound-in and another that writes to sound-out, you would create two tag values, one for either stream, and then tag each region with the value associated with its stream.

If the *preempt* flag (**...writing()** only) is true, the sound driver starts writing *data* immediately after the current transfer buffer has been completely processed. When it's finished with the preempting region, the driver returns to its region queue, disregarding the rest of the partially-processed preempted region.

If *deallocateWhenDone* (**...writing()** only) is true, the region's data is deallocated after it's written.

The six *msg...* flags register requests for stream-state messages to be sent asynchronously to the port *replyPort*. The first flags, *msgStarted* and *msgCompleted*, if true, cause messages to be sent just as the driver begins its first and just after it finishes its last transfer of data from the region, respectively. The conditions referred to by the next three arguments, *msgAborted*, *msgPaused*, and *msgResumed*, occur as a result of calls to **snddriver_stream_control()**. The *msgUnderrun* (for **...writing()**) or *msgOverrun* (for **...reading()**) argument, if true, causes a message to be sent if the driver can't transfer data quickly enough to keep up with real time. In general this is only signficant if data is being read from sound-in or written to sound-out: Underrun results in brief pauses in playback; overrun causes incoming samples to be lost. You normally process the asynchronous messages that you receive by passing them to the **snddriver_reply_handler()** function.

A working example that shows a typical set up for reading and writing streams is given in **/NextDeveloper/Examples/DSP/SoundDSPDriver/dsp_example_3/**.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

snddriver_reply_handler, **snddriver_stream_setup**

Array Processing Functions

The array processing functions let you manipulate arrays of data that have been downloaded to the DSP. The functions were generated from DSP assembly language macros by the **dspwrap** program. The macro source code files are provided in the directory **/usr/lib/dsp/apsrc**; however, you don't need to be familiar with these files, nor with DSP assembly language in order to use these functions.

Function Protocol

The array processing function protocol follows this format:

int **DSPAP***name*(int *inVector*, int *inSkip*, int *outVector*, int *outSkip*, int *count*)

Every array processing function name begins with the "DSPAP" prefix; *name* is the filename of the array processing macro from which the function was generated (without the ".asm" extension).

All arguments are type **int**. There are four categories of arguments:

- DSP memory locations
- Vector-index increments
- Operation counts
- Immediate values used in computation

For example, the **DSPAPvpv()** function, which adds the elements of two input vectors and writes the sums to an output vector ("vpv" stands for "vector plus vector"), presents a typical use of arguments from the first three categories (because all arguments are **int**s, the type isn't given in the example here, nor are they present in the function synopses below):

int **DSPAPvpv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorOut*, *outSkip*, *count*)

- *vectorInA*, *vectorInB*, and *vectorOut* are DSP memory locations. The values of these arguments are taken as the addresses of the two input vectors and the output vector, respectively. The data must have been previously downloaded through an array processing system function such as **DSPAPWriteFloatArray()** or **DSPAPWriteIntArray().**

- *inSkipA*, *inSkipB*, and *outSkip* are index increments. They determine the number of words by which the indices into the vectors are incremented before performing the next addition; for operations on non-complex data, an index increment of 1 means that contiguous elements are accessed (complex data operations are described below). Usually, as exemplified here, each vector involved in an array processing operation has its own index increment.

- *count* is the number of operations that are performed; typically, this is the number of elements in each vector.

Arguments of the fourth type—immediate values—are used directly in a computation; in other words, an immediate argument is considered as a numeric value and not as an address. Only a handful of functions expect an immediate value as an argument. **DSPAPvfilli()** (vector fill immediate), which fills an output vector with a constant value, is one of these:

int **DSPAPvfilli**(*immediateValue*, *vectorOut*, *outSkip*, *count*)

- *immediateValue* is the constant that's placed in each element of the output vector.
- *vectorOut*, *outSkip*, and *count* are the expected and previously described DSP memory location, index increment, and operation count arguments.

Keep in mind that *immediateValue*, like all array processing function arguments, must be (or appear to be) an **int**. Using this function to fill a vector with an integer is straightforward—you simply pass the integer value directly as the *immediateValue* argument. However, to fill the vector with a fraction, you have to convert the value using one of these convenient type-conversion macros:

- **DSP_FLOAT_TO_INT()** converts single-precision floating-point values.
- **DSP_DOUBLE_TO_INT()** converts double-precision floating-point values.

For example:

```
/* Fill a vector with the constant 0.123. */
float  myConst = 0.123;
DSPAPvfilli(DSP_FLOAT_TO_INT(myConst), 0, 1, 100);
```

The type-conversion macros convert fractional floating-point data into the fractional fixed-point numbers recognized by the DSP.

For every function that takes an immediate argument, there's also a version of the function that lets you pass a constant by reference to a DSP address. For example, for **DSPAPvfilli()**, there's **DSPAPvfill()**; in the latter, the constant value that's used to fill the vector is read from the location indicated by the first argument. You would use the immediate version of the function (**DSPAPvfilli()**) if the constant value is defined or generated on the host (as shown in the example above), while you would use the address version (**DSPAPvfill()**) if the constant is generated on the DSP, presumably as the result of a previous array processing computation.

Data Format and Range

Most of the array processing functions operate equally well on vectors that contain fractional numbers and those that contain integers. However, whenever multiplication is involved in a function's operation, the participating vectors are assumed to contain fractional numbers that lie within the range

$$-1.0 <= f < 1.0$$

Functions that operate on fractional data only are so noted in their descriptions.

Values on the DSP must be representable in 24 bits, as explained under "DSP System Functions," earlier in this chapter. The intermediate results of an array processing function are stored in a 56-bit accumulator, but the final result is reduced to 24 bits before it's written to an output vector. Usually, overflow values are limited to the greatest positive or negative value; however, some functions suppress this limiting and allow an overflow value to *wrap around*: The appropriate 24-bits are plucked from the 56-bit accumulator as is with no indication of whether the actual result exceeded 24 bits.

Complex Vector Functions

Some of the array processing functions operate on vectors of complex numbers, or numbers that have a real and an imaginary part. With the exception of **DSPAPfft2ra()** (radix 2 FFT), all the complex number functions expect the real and imaginary parts to be interleaved: Each pair of contiguous words in a complex vector represents a single complex number where the first word is the real part and the second word is the imaginary part. Functions are provided that help you create complex vectors in this format. (The complex number format expected by **DSPAPfft2ra()** is fully described in the function's description, below.)

Complex vector functions follow the same protocol as simple vector functions, as presented at the beginning of this introduction. However, the index increment for a complex vector deserves special attention: A complex vector element must be considered to be two words, rather than one element, when computing the index increment. Thus to read or write consecutive elements in a complex vector, for example, the increment must be set to 2.

Return Values

Every array processing function returns an **int** error code, where 0 indicates success and nonzero means that the function failed. The error codes are listed with the DSP system functions, earlier in this chapter, and in **/usr/include/dsp/dsp_errno.h**.

DSPAPcvcombine(), DSPAPcvreal(), DSPAPcvfill(), DSPAPcvfilli()

SUMMARY Create a vector of complex numbers

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPcvcombine**(*realIn*, *realSkip*, *imagIn*, *imagSkip*, *complexOut*, *outSkip*, *count*)
int **DSPAPcvreal**(*realIn*, *realSkip*, *complexOut*, *outSkip*, *count*)
int **DSPAPcvfill**(*complexIn*, *complexOut*, *outSkip*, *count*)
int **DSPAPcvfilli**(*realImmediate*, *imagImmediate*, *complexOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions help you create vectors of complex numbers in the format expected by the complex vector functions. The complex number format is described in the section "Complex Vector Functions," above.

DSPAPcvcombine() combines two simple vectors, *realIn* and *imagIn*, to create one complex vector, *complexOut*. The *realIn* vector supplies the real parts of the complex numbers and *imagIn* supplies the imaginary parts.

DSPAPcvreal() takes a single input vector, *realIn*, that supplies the real parts of the complex numbers written to *complexOut*. The imaginary part of each element in *complexOut* is set to 0.0.

DSPAPcvfill() fills *complexOut* by iterating over the single (two word) complex number stored in *complexIn*. **DSPAPcvfilli**() also fills *complexOut* through iterating over a single complex number; however, the value is passed directly as *realImmediate* and *imagImmediate*, the real and imaginary parts of the number, respectively.

For all these functions, *complexOut* must accommodate *count* * 2 words of data.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvreal(), **DSPAPvimag**()

DSPAPcvconjugate() → See DSPAPcvmove()

DSPAPcvfill() → See DSPAPcvcombine()

DSPAPcvfilli() → See DSPAPcvcombine()

DSPAPcvmandelbrot()

SUMMARY Generate a Mandelbrot set

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPcvmandelbrot**(*complexIn*, *inSkip*, *vectorOut*, *outSkip*, *count*, *limit*)

All arguments are of type **int**.

DESCRIPTION

DSPAPcvmandelbrot() generates a Mandelbrot set over the complex data in *complexIn*. The generated values, which are non-complex integers, are written to *vectorOut*.

A Mandelbrot value is the number of iterations of the formula

$$z = z * z + c$$

required to reach

$$|z| > 2$$

where **z** is a complex variable and **c** is a complex constant that's set to the initial value of **z**.

To make this easier in fixed-point format, **DSPAPcvmandelbrot()** computes the iterations of

$$w = 2 * w * w + d$$

required to reach

$$|w| > 2$$

where **w** is **z**/2 and **d** is **c**/2. Note, however, that the function assumes it's being given a **z**/2 input vector; in other words, to get a true Mandelbrot set for a given complex vector, you should divide the elements of the vector by 2 before calling this function.

The *limit* argument is the maximum number of iterations allowed. Keep in mind that since the input vector is complex, *inSkip* must be 2 to read contiguous elements.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPcvmcv() → See DSPAPcvpcv()

DSPAPcvmove(), DSPAPcvnegate(), DSPAPcvconjugate()

SUMMARY Create copies of complex-number vectors

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPcvmove**(*complexIn*, *inSkip*, *complexOut*, *outSkip*, *count*)
int **DSPAPcvnegate**(*complexIn*, *inSkip*, *complexOut*, *outSkip*, *count*)
int **DSPAPcvconjugate**(*complexIn*, *inSkip*, *complexOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

Each of these functions places, in *complexOut*, a copy of the complex number vector *complexIn*.

DSPAPcvmove() creates a literal copy.

DSPAPcvnegate() creates a value-negated copy (both the real and the imaginary parts are negated).

DSPAPcvconjugate() creates a conjugated copy. The conjugate of a complex number is its reflection about the x-axis; in other words, the real part stays the same and the imaginary part is negated.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPcvnegate() → See DSPAPcvmove()

DSPAPcvpcv(), DSPAPcvmcv(), DSPAPcvtcv()

SUMMARY Perform arithmetic operations on complex-number vectors

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPcvpcv**(*complexInA*, *inSkipA*, *complexInB*, *inSkipB*, *complexSumOut*, *outSkip*, *count*)
int **DSPAPcvmcv**(*complexMinuend*, *minuendSkip*, *complexSubtrahend*, *subtrahendSkip*, *complexDifferenceOut*, *outSkip*, *count*)
int **DSPAPcvtcv**(*complexInA*, *inSkipA*, *complexInB*, *inSkipB*, *complexProductOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions perform arithmetic operations on two complex input vectors and write the results to an output vector.

DSPAPcvpcv() (complex vector plus complex vector) adds the values in *complexInA* to the values in *complexInB* to produce the complex vector *complexSumOut*.

DSPAPcvmcv() (complex vector minus complex vector) subtracts the values in *complexSubtrahend* from the values in *complexMinuend* to produce the complex vector *complexDifferenceOut*.

DSPAPcvtcv() (complex vector times complex vector) multiplies the values in *complexInA* by the values in *complexInB* to produce the complex vector *complexProductOut*.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPcvreal() → See DSPAPcvcombine()

DSPAPcvtcv() → See DSPAPcvpcv()

DSPAPfftr2a(), DSPAPvmovebr()

SUMMARY Perform a radix 2 FFT

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPfftr2a**(*count*, *complexVector*, *lookupTable*)
int **DSPAPvmovebr**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPfftr2a() performs a radix 2 Fast Fourier Transform on the complex data in *complexVector*. The transformed data is written in-place in *complexVector* and can be unscrambled using **DSPAPvmovebr()**.

Unlike the other functions that operate on complex data, **DSPAPfftr2a()** expects the data to reside in separate DSP memory partitions: The real part is in x memory and the imaginary part is in y memory. The real and imaginary vectors must be the same length and must have corresponding addresses within their respective partitions. The following example demonstrates how to establish the addresses for memory-partitioned complex data:

```
/* DSPAPGetLowestAddressXY() returns the lowest address that's
 * unused in both x and y memory.
 */
#define  DATA_ADR    DSPAPGetLowestAddressXY()

/* DSPMapPMemX() returns the location in x memory that
 * corresponds to its argument.  DSPMapPMemY() does the same for
 * y memory.
 */
#define  REAL_DATA   DSPMapPMemX(DATA_ADR)
#define  IMAG_DATA   DSPMapPMemY(DATA_ADR)
```

The *lookupTable* argument is the address of memory-partitioned sine and cosine tables that are used in computing the FFT. The array processing system functions **DSPAPSinTable()** and **DSPAPCosTable()** are provided to create the (fractional) data for these tables, as shown in the following example. As with the two parts of complex data, the two lookup tables are stored in parallel memory locations; the cosine table is in x memory and the sine table is in y:

```
/* Define the locations for the sine and cosine lookup tables.
 * COUNT is assumed to have been defined as the number of
 * elements in the complex data (in other words, the number of
 * points in the FFT).
 */
#define  LOOKUP_ADR DATA_ADR + COUNT
#define  SIN_TABLE DSPMapPMemY(LOOKUP_ADR)
#define  COS_TABLE DSPMapPMemX(LOOKUP_ADR)

/* Create the lookup table data. */
float  *sinTab = DSPAPSinTable(COUNT);
float  *cosTab = DSPAPCosTable(COUNT);
```

The sine and cosine tables need only be half as long as the complex data that you're transforming. In deference to their use in the FFT, **DSPAPSinTable()** and **DSPAPCosTable()** return pointers to arrays of data that are half as long as the length specified by the argument.

Writing the complex data and the lookup tables to the DSP is done in the normal fashion through calls to the DSP array processing system functions. The two vectors that make up the complex data must contain fractional numbers in the range $-1.0 <= f < 1.0$:

```
/* Write the complex data.  realPart and imagPart are assumed to
 * be pointers to real and imaginary data on the host.
 */
DSPAPWriteFloatArray(realPart, REAL_DATA, 1, COUNT);
DSPAPWriteFloatArray(imagPart, IMAG_DATA, 1, COUNT);

/* Write the cosine and sine tables.  Note that the lengths are
 * half that of the complex data.
 */
DSPAPWriteFloatArray(cosTab, COS_TABLE, 1, COUNT/2);
DSPAPWriteFloatArray(sinTab, SIN_TABLE, 1, COUNT/2);

/* Perform the FFT. */
DSPAPfftr2a(COUNT, DATA_ADR, LOOKUP_ADR);
```

The data that's written by **DSPAPfftr2a()** is scrambled according to bit-reversed indexing. **DSPAPvmovebr()** unscrambles FFT data as it writes it to *vectorOut*. When used to unscramble transformed data, *inSkip* should be set to *count*/2.

You can also unscramble transformed data and read it back to the host at the same time through this sequence of function calls:

```
/* Tell the monitor to unscramble the data as it's read. */
DSPSetDMAReadMReg(0);

/* Read the data; the index increment must be COUNT/2. */
DSPAPReadFloatArray(realData, REAL_DATA, COUNT/2, COUNT);
DSPAPReadFloatArray(imagData, IMAG_DATA, COUNT/2, COUNT);

/* Reset the monitor's indexing mode. */
DSPSetDMAReadMReg(-1);
```

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPmaxmagv() → See DSPAPmaxv()

DSPAPmaxv(), DSPAPminv(), DSPAPmaxmagv(), DSPAPminmagv()

SUMMARY Find minimum and maximum values in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPminv**(*vectorIn*, *inSkip*, *minOut*, *count*)
int **DSPAPmaxv**(*vectorIn*, *inSkip*, *maxOut*, *count*)
int **DSPAPminmagv**(*vectorIn*, *inSkip*, *minMagOut*, *count*)
int **DSPAPmaxmagv**(*vectorIn*, *inSkip*, *maxMagOut*, *count*)

All arguments are of type **int**.

DESCRIPTION

Each of these functions finds an extreme value among the elements of *vectorIn* and writes the value as a one-element vector:

- **DSPAPminv()** writes the least value to *minOut*.
- **DSPAPmaxv()** writes the greatest value to *maxOut*.
- **DSPAPminvmag()** writes the least magnitude (absolute value) to *minMagOut*.
- **DSPAPmaxmagv()** writes the greatest magnitude to *maxMagOut*.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPminv() → See DSPAPmaxv()

DSPAPminmagv() → See DSPAPmaxv()

DSPAPmtm()

SUMMARY Perform matrix multiplication

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPmtm**(*matrixInA*, *matrixInB*, *matrixOut*, *a2b1*, *b2*, *a1*)

All arguments are of type **int**.

DESCRIPTION

DSPAPmtm() (matrix times matrix) multiplies *matrixInA* by *matrixInB* and writes the results to *matrixOut*. The input matrices are two-dimensional and are assumed to contain fractional numbers in the range $-1.0 <= f < 1.0$. The sizes of their dimensions are given in the final three arguments:

- *a2b1* is the number of columns in *matrixInA* and the number of rows in *matrixInB*. To perform matrix multiplication, these two dimensions must be the same size.
- *b2* is the number of columns in *matrixInB*.
- *a1* is the number of rows in *matrixInA*.

The product of a matrix multiply is an *a1* by *b2* matrix of (fractional) values. The value of an element at x, y in the output matrix is computed as:

$$matrixOut[x][y] = \sum_{k=1}^{a2b1} matrixInA[x][k] * matrixInB[k][y]$$

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPsumv(), DSPAPsumvnolim(), DSPAPsumvmag(), DSPAPsumvsq(), DSPAPsumvsquares()

SUMMARY Add the elements in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPsumv**(*vectorIn*, *inSkip*, *sumOut*, *count*)
int **DSPAPsumvnolim**(*vectorIn*, *inSkip*, *sumOut*, *count*)
int **DSPAPsumvmag**(*vectorIn*, *inSkip*, *sumOut*, *count*)
int **DSPAPsumvsq**(*vectorIn*, *inSkip*, *sumOut*, *count*)
int **DSPAPsumvsquares**(*vectorIn*, *inSkip*, *sumOut*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions add up the values in *vectorIn* and write the resulting sums to *sumOut*.

DSPAPsumv() adds the vector elements as they are given. **DSPAPsumvnolim**() is similar but doesn't limit the sum; overflow sums are allowed to wrap around.

DSPAPsumvmag() adds the magnitudes (absolute values) of the elements.

DSPAPsumvsq() and **DSPAPsumvsquares**() add the squares and the signed squares, respectively, of the elements. For both of these functions, the values in *vectorIn* are assumed to be fractional numbers in the range $-1.0 <= f < 1.0$.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPsumvmag() → See DSPAPsumv()

DSPAPsumvnolim() → See DSPAPsumv()

DSPAPsumvsq() → See DSPAPsumv()

DSPAPsumvsquares() → See DSPAPsumv()

DSPAPvabs()

SUMMARY Compute the magnitude of a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvabs**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvabs() computes the magnitude (absolute value) of each element in *vectorIn* and writes the results to *vectorOut*.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPvasl(), DSPAPvasr(), DSPAPvlsl(), DSPAPvlsr()

SUMMARY Bit-shift the elements in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvasl**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvasr**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvlsl**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvlsr**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions shift the bits of the elements in *vectorIn* and write the shifted values to *vectorOut*.

DSPAPvasl() and **DSPAPvasr**() perform arithmetic left- and right-shifts, respectively. In essence, this multiplies (left-shift) or divides (right-shift) the original value by 2, with overflow values limited to the maximum or minimum value.

DSPAPvlsl() and **DSPAPvlsr()** perform logical shifts. All 24 bits of the original value—including the sign extension—are shifted, and the vacated bit is filled with 0. Overflow values aren't limited.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvts()

DSPAPvasr() → See DSPAPvasl()

DSPAPvclear() → See DSPAPvfill()

DSPAPvfill(), DSPAPvfilli(), DSPAPvramp(), DSPAPvrampi(), DSPAPvrand(), DSPAPvclear()

SUMMARY Fill a vector with values

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvfill**(*constantAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvfilli**(*constantImmediate*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvramp**(*offsetAddr*, *scaleAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvrampi**(*offsetImmediate*, *scaleImmediate*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvrand**(*seedAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvclear**(*vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions fill the vector *vectorOut* with constant or DSP-generated values.

DSPAPvfill() and **DSPAPvfilli()** fill the vector with a constant value. The former reads the value from the location given as *constantAddr*; the latter takes the value directly as *constantImmediate*.

DSPAPvramp() fills the vector with a ramp: Successive elements are given an incrementally increasing (or decreasing) value starting from an initial offset value

vectorOut[*k*] = *offsetAddr*[0] + (*k* * *scaleAddr*[0])

DSPAPvrampi() also fills the vector with a ramp, but it takes the offset and scale values directly as given in *offsetImmediate* and *scaleImmediate*.

DSPAPvrand() fills the vector with uniform pseudo-random numbers using the linear congruential method for random number generation (from Volume II of *The Art of Computer Programming* by Donald Knuth; the multiplier used is 5609937 and the offset is 1).

DSPAPvclear() clears the vector by setting each element to 0.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPvfilli() → See DSPAPvfill()

DSPAPvimag() → See DSPAPvreal()

DSPAPvlsl() → See DSPAPvasl()

DSPAPvlsr() → See DSPAPvasl()

DSPAPvmove(), DSPAPvmoveb()

SUMMARY Copy a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvmove**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvmoveb**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions copy the elements of *vectorIn* into *vectorOut*; **DSPAPvmove**() starts with the first element and works its way towards the last, while **DSPAPvmoveb**() starts with the last element (of both vectors) and works towards the first. If the input and

output vectors don't overlap (and the index increments are equal), then the two functions are, in essence, the same. For overlapping vectors, you should use **DSPAPvmove()** if *vectorIn* is greater than *vectorOut* and **DSPAPvmoveb()** otherwise. In general, this ensures that the data in *vectorIn* isn't overwritten before it's copied, although the copied data may still overwrite the original if *skipOut* is greater than *skipIn*.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvmovebr() (used for unscrambling FFT output)

DSPAPvmoveb() → See DSPAPvmove()

DSPAPvmovebr() → See DSPAPfftr2a()

DSPAPvmv() → See DSPAPvpv()

DSPAPvnegate()

SUMMARY Negate the elements in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvnegate**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvnegate() negates the values in *vectorIn*—positive values become negative and negative values become positive—and writes the results to *vectorOut*.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvts()

DSPAPvps(), DSPAPvpsi(), DSPAPvts(), DSPAPvtsi()

SUMMARY Offset and scale a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvps**(*vectorIn*, *inSkip*, *offsetAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvpsi**(*vectorIn*, *inSkip*, *offsetImmediate*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvts**(*vectorIn*, *inSkip*, *scaleAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtsi**(*vectorIn*, *inSkip*, *scaleImmediate*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions offset (add a constant value to) or scale (multiply by a constant value) the values in *vectorIn*, and they write the result to *vectorOut*.

DSPAPvps() (vector plus scaler) and **DSPAPvpsi()** (vector plus scaler immediate) offset the vector. The former takes as its offset the value at *offsetAddr*; the latter takes its offset directly as *offsetImmediate*.

DSPAPvts() (vector times scaler) and **DSPAPvtsi()** (vector times scaler immediate) scale the vector, using the value stored at *scaleAddr* and the value *scaleImmediate*, respectively. For both of these functions, the scaling value and the values in *vectorIn* are assumed to be fractional numbers in the range $-1.0 <= f < 1.0$.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvasl(), **DSPAPvasr()**, **DSPAPvnegate()**

DSPAPvpsi() → See DSPAPvps()

DSPAPvpv(), DSPAPvpvnolim(), DSPAPvmv(), DSPAPvtv()

SUMMARY Add, subtract, and multiply two vectors

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvpv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *sumOut*, *outSkip*, *count*)
int **DSPAPvpvnolim**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvmv**(*vectorMinuend*, *minuendSkip*, *vectorSubtrahend*, *subtrahendSkip*, *differenceOut*, *outSkip*, *count*)
int **DSPAPvtv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *productOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions add, subtract, and multiply two input vectors and write the results to an output vector. In vector arithmetic, the value at index *k* in the output vector is created by performing the specified operation on the values at index *k* in the input vectors (with the prescribed index-incrementing of *k* for each vector).

DSPAPvpv() (vector plus vector) and **DSPAPvpvnolim**() (vector plus vector no limiting) add *vectorInA* to *vectorInB* and write the sum to *sumOut*:

$$sumOut[k] = vectorInA[k] + vectorInB[k]$$

The difference between the two functions is their treatment of overflow sums: **DSPAPvpv**() limits an overflow sum to the minimum or maximum possible value, while **DSPAPvpvnolim**() allows overflow sums to wrap around.

DSPAPvmv() (vector minus vector) subtracts *vectorSubtrahend* from *vectorMinuend* and writes the difference to *differenceOut*:

$$differenceOut[k] = vectorMinuend[k] - vectorSubtrahend[k]$$

DSPAPvtv() (vector times vector) multiplies *vectorInA* by *vectorInB* and writes the product to *productOut*:

$$productOut[k] = vectorInA[k] * vectorInB[k]$$

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvsquare(), DSPAPvasl(), DSPAPvasr(), DSPAPvnegate(), DSPAPvps(), DSPAPvts()

DSPAPvpvnolim() → See DSPAPvpv()

DSPAPvramp() → See DSPAPvfill()

DSPAPvrampi() → See DSPAPvfill()

DSPAPvrand() → See DSPAPvfill()

DSPAPvreal(), DSPAPvimag()

SUMMARY Retrieve data from a vector of complex numbers

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvreal**(*complexIn*, *inSkip*, *realOut*, *outSkip*, *count*)
int **DSPAPvimag**(*complexIn*, *inSkip*, *imagOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvreal() and **DSPAPvimag()** retrieve the real and imaginary parts, respectively, of the values in a vector of complex numbers. For both functions, *complexIn* should point to the first word (the real part) of a two-word complex value. The retrieved data is written as a non-complex vector, *realOut* or *imagOut*.

count is the number of complex values (two words each) that are read. To read contiguous complex values, *inSkip* should be 2. Since *realOut* and *imagOut* are vectors of simple (single-word) values, an *outSkip* of 1 writes contiguously.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPvreverse()

SUMMARY Reverse the position of the elements in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvreverse**(*vectorIn*, *vectorOut*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvreverse() reads the elements in *vectorIn* and writes them, in reverse order, to *vectorOut*. You can reverse a vector's elements in-place by passing the same location for the two vector arguments.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPvsquare(), DSPAPvssq()

SUMMARY Square the elements in a vector

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvsquare**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvssq**(*vectorIn*, *inSkip*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvsquare() squares each element in *vectorIn* and writes the resulting products to *vectorOut*. **DSPAPvssq()** does the same but maintains the signs of the original elements. For both functions, the values in *vectorIn* are assumed to be fractional numbers in the range $-1.0 <= f < 1.0$.

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvtv()

DSPAPvssq() → See DSPAPvsquare()

DSPAPvswap()

SUMMARY Swap the elements in two vectors

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvswap**(*vectorA*, *aSkip*, *vectorB*, *bSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

DSPAPvswap() swaps the elements in *vectorA* and *vectorB*. Data may be lost if the vectors overlap.

RETURN

Returns an error code: 0 on success, nonzero on failure.

DSPAPvts() → See DSPAPvps()

DSPAPvtsi() → See DSPAPvps()

DSPAPvtsmv() → See DSPAPvtvps()

DSPAPvtspv() → See DSPAPvtvps()

DSPAPvtv() → See DSPAPvpv()

DSPAPvtvmv() → See DSPAPvtvpv()

DSPAPvtvmvtv() → See DSPAPvtvpv()

DSPAPvtvps(), DSPAPvtspv(), DSPAPvtsmv()

SUMMARY Perform arithmetic operations with two vectors and a constant

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvtvps**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *offsetAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtspv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *scaleAddr*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtsmv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *scaleAddr*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions perform compound arithmetic operations on two input vectors and a constant value and write their results to *vectorOut*. The values in all the input vectors are assumed to be fractional numbers in the range $-1.0 <= f < 1.0$. The constant value is given by reference; in other words, the constant value argument (either *offsetAddr* or *scaleAddr*) is an address, not an immediate value.

DSPAPvtvps() (vector times vector plus scaler) multiplies *vectorInA* by *vectorInB* and adds the value at *offsetAddr* to the product:

$$vectorOut[k] = (vectorInA[k] * vectorInB[k]) + offsetAddr[0]$$

DSPAPvtspv() (vector times scaler plus vector) scales *vectorInA* by the value at *scaleAddr* and adds *vectorInB* to the product:

$$vectorOut[k] = (vectorInA[k] * scaleAddr[0]) + vectorInB[k]$$

DSPAPvtsmv() (vector times scaler minus vector) scales *vectorInA* by the value at *scaleAddr* and subtracts *vectorInB* from the product:

$$vectorOut[k] = (vectorIn[k] * scaleAddr[0]) - vectorInB[k]$$

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvtvpv(), **DSPAPvtvmv()**, **DSPAPvts()**, **DSPAPvps()**

DSPAPvtvpv(), DSPAPvtvmv(), DSPAPvtvpvtv(), DSPAPvtvmvtv()

SUMMARY Perform arithmetic operations on three and four vectors

LIBRARY libarrayproc.a

SYNOPSIS

#import <dsp/arrayproc.h>

int **DSPAPvtvpv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorInC*, *inSkipC*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtvmv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorInC*, *inSkipC*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtvpvtv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorInC*, *inSkipC*, *vectorInD*, *inSkipD*, *vectorOut*, *outSkip*, *count*)
int **DSPAPvtvmvtv**(*vectorInA*, *inSkipA*, *vectorInB*, *inSkipB*, *vectorInC*, *inSkipC*, *vectorInD*, *inSkipD*, *vectorOut*, *outSkip*, *count*)

All arguments are of type **int**.

DESCRIPTION

These functions perform a variety of arithmetic operations on two or more input vectors and write their results to *vectorOut*. The values in all the input vectors are assumed to be fractional numbers in the range $-1.0 <= f < 1.0$.

DSPAPvtvpv() (vector times vector plus vector) adds a vector to the product of a vector multiply:

$$vectorOut[k] = (vectorInA[k] * vectorInB[k]) + vectorInC[k]$$

DSPAPvtvmv() (vector times vector minus vector) subtracts a vector from the product of a vector multiply:

$$vectorOut[k] = (vectorInA[k] * vectorInB[k]) - vectorInC[k]$$

DSPAPvtvpvtv() (vector times vector plus vector times vector) adds the products of two vector multiplies:

$$vectorOut[k] = (vectorInA[k] * vectorInB[k]) + (vectorInC[k] * vectorInD[k])$$

DSPAPvtvmvtv() (vector times vector minus vector times vector) subtracts the product of a vector multiply from the product of another vector multiply:

$$vectorOut[k] = (vectorInA[k] * vectorInB[k]) - (vectorInC[k] * vectorInD[k])$$

RETURN

Returns an error code: 0 on success, nonzero on failure.

SEE ALSO

DSPAPvtv(), DSPAPvtvps(), DSPAPvtspv(), DSPAPvtsmv()

DSPAPvtvpvtv() → See DSPAPvtvpv()

Chapter 4
ScoreFile Language Reference

Chapter 4
ScoreFile Language Reference

ScoreFile is a language designed to represent, create, and manipulate music data. The code for a ScoreFile program is maintained in a file, called a *scorefile*, on the disk. A scorefile represents a Music Kit Score object and its contents in ASCII form. Scorefiles can be created from a text editor or generated automatically by a Score or ScorefileWriter object. A scorefile is interpreted when it's read by a Score object or performed by a ScorefilePerformer object.

This chapter describes the syntax and conventions of the ScoreFile language. The presentation in this chapter assumes a familiarity with Chapter 3 in the *Concepts* manual. A concise outline of ScoreFile syntax can be found in Appendix C, "Summary of ScoreFile Language Syntax."

Program Structure

A ScoreFile program is divided into two sections: the header and the body. The header always precedes the body; the two sections are separated by a **BEGIN** statement. The end of the scorefile can be marked by an optional **END** statement:

header
BEGIN;
body
[**END;**]

Either section can be empty. If the body is empty, the **BEGIN** statement can be omitted.

Both the header and the body are made up of ScoreFile statements. The header contains statements that establish the context in which the body is interpreted. The following statements can appear only in the header:

- Score info statements
- **part** statements
- Part info statements
- **tagRange** statements

The body consists of a time-ordered series of statements that represent Note objects. This information is found only in the body:

- Time statements
- Note statements

A number of other statements can appear in either the header or the body:

- Variable declarations
- Assignment statements
- **envelope** statements
- **waveTable** statements
- **object** statements
- **include** statements
- **print** statements
- **tune** statements
- **comment** and **endComment** statements

Header Statements

Score Info Statements

A scorefile can have a Score info statement that consists of the keyword **info** followed by one or more parameters:

info *parameter* [**,** *parameter*] ... **;**

The Score info statement represents a Score object's info Note; it can contain any amount and type of information. Typically, the Score info statement contains one or more of the following parameters:

Parameter	Meaning
tempo	The tempo that should be used when performing the Score
samplingRate	The performance sampling rate
headroom	The Orchestra's headroom setting; a value between −1.0 and 1.0

A scorefile can have more than one Score info statement; if a parameter conflicts with a parameter set in a previous **info** statement, the subsequent setting takes precedence. Parameters are similarly merged if a scorefile is read into a Score object that already has an info Note (a Score object can have only one info Note). Parameter syntax is described in the section "Note Statements," below.

The parameters in the **info** statement aren't explicitly used when the scorefile is read by a Score or ScorefilePerformer. It's left to the application designer to provide an implementation that acts on the **info** statement's parameters.

part Statements

The names of all the Part objects that are represented in a scorefile must be declared in a **part** statement in the header:

part *partName* [, *partName*] ... ;

partName is an identifier that must not have been previously declared. A scorefile can contain more than one **part** statement. When the scorefile is read by an application, a Part object is created and named for each partName in the file's **part** statements. If a name conflict results from reading a scorefile into a Score, the Part represented in the scorefile is merged into the similarly named Part in the Score.

Part Info Statements

Each Part represented in the scorefile can have a Part info statement that consists of the Part's *partName* as it appears in the **part** statement followed by one or more parameters:

partName parameter [, *parameter*] ... ;

The Part info statement represents a Part object's info Note; it can contain any amount and type of information. The following parameters are typically used in a part info statement:

Parameter	Meaning
synthPatch	The name of the SynthPatch class used to realize the Part
synthPatchCount	The number of manually allocated SynthPatch objects
midiChan	The MIDI channel on which the Part appears

Each Part represented in a scorefile can have only one Part info statement. Like the scorefile's **info** statement, interpretation and use of a Part info's parameters is left to the application designer.

tagRange Statement

The **tagRange** statement declares the range of noteTags used in the body of the scorefile:

tagRange *anInteger* **to** *aHigherInteger* ;

This is an optional statement that optimizes the noteTag renumbering that occurs when you mix two or more scorefiles together or when you merge a scorefile into an existing Score object.

It isn't an error to use a tag that's outside the range specified by a **tagRange** statement, but the renumbering optimization applies only to tags that are within the declared range. A scorefile can have more than one **tagRange** statement although each subsequent statement cancels the previous one.

Body Statements

Note Statements

When a scorefile is read by an application, a single Note object is created for each note statement in the file. Note statements take the following form:

partName , (*typeAndTag*) [, *parameters*] ;

partName is the name of the Part to which the Note belongs. It must be declared in a **part** statement in the header.

typeAndTag provides noteType and noteTag information; its form depends on the noteType:

- For a noteDur, it takes the form

  (*duration* [*noteTag*])

  duration is a **double** expression that specifies the duration of the Note in beats; *tag* is an integer expression that assigns the Note's noteTag.

- For a noteOn or noteOff, the noteTag is required:

  (**noteOn** *noteTag*)
  (**noteOff** *noteTag*)

- The noteTag is optional for a noteUpdate:

  (**noteUpdate** [*noteTag*])

A noteUpdate with a noteTag is applied to the specified noteTag stream. Without a noteTag, it's applied to all noteTag streams that are currently being realized on the same Instrument as the noteUpdate.

- Finally, a mute never takes a noteTag:

  (**mute**)

parameters is a list of parameters separated by commas. A parameter takes the form:

parameterName : *expression*

parameterName is the name of the parameter. Its form is that of a Music Kit parameter identifier minus the "MK_" prefix. For example, MK_freq becomes, in a scorefile, **freq**. In a scorefile you can create your own parameters simply by including them in a note statement. When the scorefile is read by an application, a parameter identifier is automatically created and named for each of your invented parameters.

expression is computed as the value assigned to the parameter. An expression can include variable assignments:

parameterName **:** **(** *variable* = *expression* **)**

Time Statements

A time statement specifies the performance time in beats for all subsequent Note statements until another time statement is encountered. A time statement takes the form:

t [+] *expression* **;**

The keyword **t** is a special symbol; its value is the current time, in beats, in the scorefile. At the start of the scorefile, the value of **t** is 0.0. If *expression* is preceded by **+**, **t** is incremented by the value of *expression*. Otherwise, **t** takes the value of *expression* directly. Time always moves forward in a scorefile—the value of **t** must never decrease.

t can be used as a read-only variable in an expression.

Header or Body Statements

Variable Declarations and Assignments

Variable declaration is the same as in C:

- When you declare a variable you must specify its type.
- More than one variable of the same type can be declared in the same declaration.
- A variable's value may be set when it's declared.

The variable declaration statement takes the following form:

dataType identifier [= *expression*] [**,** *identifier* [= *expression*]] ... **;**

Assignment is also like C:

identifier = *expression*

Variable assignments can be nested and can appear in parameter value expressions.

ScoreFile provides seven data types:

double
int
string
env
wave
object
var

The **double** and **int** types are the same as in C; **string** takes a string value:

string = "*text*";

env, **wave**, and **object** take Envelope, WaveTable, and object values, respectively, as described in the following sections. **var** is a wild card: A variable so declared automatically matches the type of its assigned data. In general, **var** obviates the need for the other six types; however, the others can be used for clarity, or to cast a value to a particular type.

envelope Statements

You can create an Envelope in a scorefile by using an **envelope** statement:

envelope *envelopeName* = *envelopeConstant* ;

When the scorefile is read, an Envelope object is created and named for each **envelope** statement in the file. *envelopeName* can be any previously undeclared identifier and can be used as the value in a variable assignment (the variable's type must be **env** or **var**):

env = *envelopeName* ;

envelopeConstant contains a list of the Envelope's breakpoints. Each breakpoint is described by its x, y, and (optional) smoothing values. Breakpoint descriptions are in parentheses and the entire Envelope is delimited by brackets:

[(*xValue* , *yValue* [, *smoothingValue*]) , ...]

A scorefile can contain any number of Envelopes.

waveTable Statements

WaveTables are created with the **waveTable** statement:

waveTable *waveTableName* = *waveTableConstant* **;**

Similar to the **envelope** statement, an object is created and named for each **waveTable** statement in a scorefile when the file is read. The created object is either a Partials or a Samples object, depending on the specification in *waveTableConstant*. A Partials object is described as a series frequency ratio, amplitude ratio, and (optional) phase values.

Each specification defines a single partial and is surrounded by braces; like an Envelope, the entire object is delimited by brackets:

[{ *frequencyRatio* **,** *amplitudeRatio* **[** **,** *phase* **] } ,** ... **]**

A Samples object is defined by a soundfile:

[{ "*soundfileName*" **}]**

waveTableName can be used in a **wave** or **var** assignment.

object Statements

You can use an **object** statement to add your own objects to a scorefile:

object *objectName* = *objectConstant* **;**

objectConstant contains, in brackets, the name of the object's class followed by a description of the object:

[*className objectDescription*]

objectDescription can be any text except "]". *className* must implement the methods **readASCIIStream:**, and **writeASCIIStream:** to define how to read and write the object description.

include Statements

When an **include** statement is encountered, the specified file is immediately read and interpreted:

include "*scorefileName*"**;**

print Statements

A **print** statement is used to print information to a stream pointer (NXStream *):

print *expression* [, *expression*] ... ;

The information is displayed when the scorefile is interpreted. The **setScorefilePrintStream:** method, defined by Score and ScorefilePerformer, lets you set the stream to which a scorefile's messages are printed. By default, they're printed to standard error.

tune Statements

The **tune** statement lets you create a tuning system other than the default twelve-tone equal-temperament:

tune *pitchVariable* = *expression* ;
tune *expression* ;

The first form of the statement tunes *pitchVariable*, a predeclared ScoreFile variable, to *expression,* taken as a frequency in hertz. All pitch variables of the same pitch class as *pitchVariable* are tuned to the appropriate octave transposition of *expression.* Pitch variables are described in the next section, "Predeclared Variables, Constants, and Special Symbols." The second form transposes all pitch variables by *expression* half-steps. A negative value transposes down; a fractional value transposes by less than a half step.

comment and endComment Statements

In addition to supporting the C and Objective-C language comment syntax, ScoreFile supplies its own comment construction:

comment;
commentedCode
endComment;

Predeclared Variables, Constants, and Special Symbols

Pitch Variables

ScoreFile reserves a number of words as predefined pitch variables. Pitch variables represent the frequencies of pitches over a ten and a half octave range. A pitch variable name takes the following form (spaces between components aren't allowed):

pitchLetter[*sharpOrFlat*]*octave*

pitchLetter is a lowercase letter from **a** to **g**.

sharpOrFlat is **s** for sharp and **f** for flat. (Double sharps and double flats aren't supported.)

octave is **00** or an integer from **0** to **9**. Octaves are placed such that **c4** is middle C. **c00** is the lowest pitch, **g9** is the highest.

A pitch variable can be assigned an arbitrary value in an assignment statement or assignment expression. The value assigned to a pitch variable is taken as a frequency in hertz:

pitchVariable = *expression* **;**

By assigning a value to a pitch variable, only the value of that pitch variable is changed; this is different from using a pitch variable in a **tune** statement, where all pitch variables of the same pitch class are affected.

Key Number Constants

Key numbers are similar in appearance to pitch variables, but have an appended **k** (again, embedded spaces aren't allowed):

pitchLetter[*sharpOrFlat*]*octave***k**

Unlike a pitch variable, which represents a frequency, a key number is an integer that represents the ordinal number of a key on a MIDI synthesizer.

MIDI Constants

A number of MIDI constants defined as values for MIDI parameters are provided by ScoreFile:

resetControllers	monoMode	sysActiveSensing
localControlModeOn	polyMode	sysReset
localControlModeOff	sysClock	sysUndefined0xf9
allNotesOff	sysStart	sysUndefined0xfd
omniModeOff	sysContinue	
omniModeOn	sysStop	

Other Constants

ScoreFile also defines the integer constants **YES** (1) and **NO** (2).

Special Symbols

ScoreFile defines two special symbols, **t** and **ran**. These are read-only variables that should never be assigned a value in an assignment statement. The **t** symbol was described in the section "Time Statements," earlier in this chapter.

ran is a random number (a **double**) between 0 and 1. The seed for the random number generator is randomly set to produce a different series of random numbers every time the file is read.

Operators

ScoreFile provides its own set of operators in addition to supporting a subset of C arithmetic operators. The following table shows all the available operators in order of decreasing priority. The operators unique to ScoreFile are discussed below.

Operator	Operation
( )	Grouping
–	Unary minus
dB	Decibel computation
^, ~	Exponentiation, pitch transposition
*, /, %	Multiplication, division, modulus
+, –	Addition, subtraction
@	Envelope lookup
&	String concatenation
=	Assignment
,	Sequence separator

Decibel Computation Operator

The postfix decibel operator **dB** is used to specify an amplitude value in units of decibels:

expression **dB**

The computation used by the **dB** operator is:

$10^{(expression\ /20)}$

0 dB is the maximum amplitude.

Exponentiation Operator

In ScoreFile, the expression

expression ^ *expression*

calculates the left expression raised to the power of the right expression.

Pitch Transposition Operator

The pitch transposition operator **~** is designed to transpose a pitch variable:

pitchVariable **~** *expression*

The computed value is the frequency of *pitchVariable* raised or lowered by *expression* half-steps (a negative value lowers the pitch). The pitch variable's value isn't affected.

Envelope Lookup Operator

The Envelope lookup operator **@** retrieves a discrete value from an envelope:

envelopeName **@** *xValue*

The calculation returns the y value in *envelopeName* that corresponds to *xValue*. The operation performs a linear interpolation between breakpoints, if necessary.

String Concatenation Operator

The string concatenation operation takes the form:

expression **&** *expression*

The two expressions are converted to text and concatenated to produce a new string, regardless of the data types of the original expressions.

Appendix A
Summary of ScoreFile Language Syntax

Appendix A
Summary of ScoreFile Language Syntax

This appendix gives a succinct summary of the syntax of the ScoreFile language. Chapter 4, "ScoreFile Language Reference," provides a general description and explanation of ScoreFile syntax and ScoreFile program organization.

Program Structure

scorefile:
    [*header*] [**BEGIN ;** [*body* [**END** ;]]]

header:
    headerStatement **;** [*header*]

headerStatement:
    scoreInfoStatement
    partDeclaration
    partInfoStatement
    tagRangeDeclaration
    headerOrBodyStatement

body:
    bodyStatement **;** [*body*]

bodyStatement:
    timeStatement
    noteStatement
    headerOrBodyStatement

headerOrBodyStatement:
    variableDeclaration
    envelopeDeclaration
    waveTableDeclaration
    objectDeclaration
    assignmentStatement
    includeStatement
    printStatement
    tuneStatement
    commentStatement
    endCommentStatement

Header Statements

scoreInfoStatement:
    info [, *parameters*]

partDeclaration:
    part *partList*

partList:
    partName [, *partList*]

partName:
    identifier

partInfoStatement:
    partName [, *parameters*]

tagRangeDeclaration:
    tagRange *integer* **to** *integer*

Body Statements

timeStatement:
    t [+] *expression*

noteStatement:
    partName , (*typeAndTag*) [, *parameters*]

typeAndTag:
    duration [, *noteTag*]
    noteOn , *noteTag*
    noteOff , *noteTag*
    noteUpdate [, *noteTag*]
    mute

duration:
    expression

noteTag:
    integerExpression

parameters:
    parameter [, *parameters*]

parameter:
    parameterName : *parameterValue*

parameterName:
    identifier

parameterValue:
    expression

Header or Body Statements

The large, bold brackets, braces, and parentheses in the components of the envelope, waveTable, and object declarations below are to be typed where shown.

variableDeclaration:
    dataType initVariableList

dataType:
    double
    int
    string
    var
    obj
    wave
    env

initVariableList:
    initVariable [, *initVariableList*]

initVariable:
    identifier [= *expression*]

envelopeDeclaration:
    envelope *envelopeName* = *envelopeConstant*

envelopeName:
    identifier

envelopeConstant:
    [*envelopePointList* **]**

envelopePointList:
    envelopePoint [, *envelopePointList*]

envelopePoint:
    (*xValue* , *yValue* [, *smoothingValue*] **)**

waveTableDeclaration:
    waveTable *waveTableName* = *waveTableConstant*

waveTableName:
    identifier

waveTableConstant:
    [*partialsList*]
    [{ *soundfileName* }]

partialsList:
    partial [, *partialsList*]

partial:
    { *frequencyRatio* , *amplitudeRatio* [, *phase*] }

soundfileName:
    fileName

fileName:
    "*fileName*"

objectDeclaration:
    object *objectName* = *objectConstant*

objectName:
    identifier

objectConstant:
    [*className objectDescription*]

objectDescription:
    defined by *className*; can contain anything except]

assignmentStatement:
    identifier = *expression*

includeStatement:
    include *fileName*

printStatement:
    print *expressionList*

expressionList:
    expression [, *expressionList*]

tuneStatement:
    tune *pitchVariable* = *expression*
    tune *expression*

commentStatement:
    comment

endCommentStatement:
    endComment

Constants, Predeclared Variables, and Special Symbols

midiConstants:
    channelModeConstant
    systemRealTimeConstant

channelModeConstant:
    resetControllers
    localControlModeOn
    localControlModeOff
    allNotesOff
    omniModeOff
    omniModeOn
    monoMode
    polyMode

systemRealTimeConstant:
    sysClock
    sysUndefined0xf9
    sysStart
    sysContinue
    sysStop
    sysUndefined0xfd
    sysActiveSensing
    sysReset

otherConstants:
    keyNumber
    NO (equal to 0)
    YES (equal to 1)

predeclaredVariable:
    pitchVariable

ScoreFile reserves more than 200 keywords for the representation of pitch names and key numbers. Rather than list the entire set of these keywords, formulas are given here that describe the form of the *keyNumber* and *pitchVariable* names.

keyNumber:
    pitchVariable

pitchVariable:
    pitchLetter [*sharpOrFlat*] *octave*

pitchLetter:
c
d
e
f
g
a
b

sharpOrFlat:
s
f

octave:
00
0
1
2
3
4
5
6
7
8
9

The ScoreFile special symbols are read-only variables that can manipulate their own value. Special symbols should never be assigned a value in an assignment statement.

specialSymbols:
t
ran

Operators

Operators are shown in descending priority. Operators on the same line are of equal priority; they're processed in the order that they occur in the scorefile.

operator:
groupingOperator
prefixOperator postfixOperator
arithmeticOperator
envelopeLookupOperator stringConcatenationOperator
assignmentOperator
sequenceSeparator

groupingOperator:
    ( )

prefixOperator:
    –

postfixOperator:
    dB

arithmeticOperator:
    ^ ~
    *** / %**
    + –

envelopeLookupOperator:
    @

stringConcatenationOperator:
    &

assignmentOperator:
    =

sequenceSeparator:
    ,

Appendix B
Music Tables

Appendix B
Music Tables

Pitches and Frequencies

The following table shows the correspondence between pitch name variables, key numbers, and the frequencies (in Hz) that they represent.

Key number constants, not explicitly listed here, are the same as pitch name variables, but with an appended "k".

The rightmost column, "Upper Limit," shows the highest frequency (inclusive) that corresponds to the key number. A key number's lowest frequency (non-inclusive) is the upper limit of the previous key number.

Pitch Name	Frequency	Key #	Upper Limit
C00	8.175625	0	8.418725
Cs00/Df00	8.661875	1	8.919375
D00	9.176875	2	9.449775
Ds00/Ef00	9.722812	3	10.011812
E00/Ff00	10.300937	4	10.607137
F00/Es00	10.913438	5	11.237738
Fs00/Gf00	11.562188	6	11.905888
G00	12.249688	7	12.613988
Gs00/Af00	12.978438	8	13.364138
A00	13.75	9	14.1587
As00/Bf00	14.5675	10	15.0007
B00/Cf0	15.434062	11	15.892562
C0/Bs00	16.35125	12	16.83745
Cs0/Df0	17.32375	13	17.83865
D0	18.35375	14	18.89965
Ds0/Ef0	19.445625	15	20.023725
E0/Ff0	20.601875	16	21.214275
F0/Es0	21.826875	17	22.475575
Fs0/Gf0	23.124375	18	23.811775
G0	24.499375	19	25.228075
Gs0/Af0	25.956875	20	26.728375
A0	27.5	21	28.3174
As0/Bf0	29.135	22	30.0015
B0/Cf1	30.868125	23	31.785225
C1/Bs0	32.7025	24	33.6749
Cs1/Df1	34.6475	25	35.6775
D1	36.7075	26	37.7993
Ds1/Ef1	38.89125	27	40.04745
E1/Ff1	41.20375	28	42.42875

(continued)

Pitch Name	Frequency	Key #	Upper Limit
F1/Es1	43.65375	29	44.95125
Fs1/Gf1	46.24875	30	47.62375
G1	48.99875	31	50.45625
Gs1/Af1	51.91375	32	53.45685
A1	55.0	33	56.635
As1/Bf1	58.27	34	60.0031
B1/Cf2	61.73625	35	63.57055
C2/Bs1	65.405	36	67.35
Cs2/Df2	69.295	37	71.3549
D2	73.415	38	75.5987
Ds2/Ef2	77.7825	39	80.0949
E2/Ff2	82.4075	40	84.8574
F2/Es2	87.3075	41	89.9024
Fs2/Gf2	92.4975	42	95.2475
G2	97.9975	43	100.9125
Gs2/Af2	103.8275	44	106.9137
A2	110.0	45	113.2699
As2/Bf2	116.54	46	120.0062
B2/Cf3	123.4725	47	127.1412
C3/Bs2	130.81	48	134.7
Cs3/Df3	138.59	49	142.71
D3	146.83	50	151.1975
Ds3/Ef3	155.565	51	160.19
E3/Ff3	164.815	52	169.715
F3/Es3	174.615	53	179.805
Fs3/Gf3	184.995	54	190.495
G3	195.995	55	201.825
Gs3/Af3	207.655	56	213.8275
A3	220.0	57	226.54
As3/Bf3	233.08	58	240.0125
B3/Cf4	246.945	59	254.2824
C4/Bs3	261.62	60	269.4
Cs4/Df4	277.18	61	285.42
D4	293.66	62	302.395
Ds4/Ef4	311.13	63	320.38
E4/Ff4	329.63	64	339.43
F4/Es4	349.23	65	359.61
Fs4/Gf4	369.99	66	380.99
G4	391.99	67	403.65
Gs4/Af4	415.31	68	427.655
A4	440.0	69	453.08
As4/Bf4	466.16	70	480.025
B4/Cf5	493.89	71	508.565
C5/Bs4	523.24	72	538.8
Cs5/Df5	554.36	73	570.84
D5	587.32	74	604.79
Ds5/Ef5	622.26	75	640.76
E5/Ff5	659.26	76	678.86
F5/Es5	698.46	77	719.22
Fs5/Gf5	739.98	78	761.98
G5	783.98	79	807.3

(continued)

Pitch Name	Frequency	Key #	Upper Limit
Gs5/Af5	830.62	80	855.31
A5	880.0	81	906.16
As5/Bf5	932.32	82	960.05
B5/Cf6	987.78	83	1017.13
C6/Bs5	1046.48	84	1077.6
Cs6/Df6	1108.72	85	1141.68
D6	1174.64	86	1209.58
Ds6/Ef6	1244.52	87	1281.52
E6/Ff6	1318.52	88	1357.72
F6/Es6	1396.92	89	1438.44
Fs6/Gf6	1479.96	90	1523.96
G6	1567.96	91	1614.6
Gs6/Af6	1661.24	92	1710.62
A6	1760.0	93	1812.32
As6/Bf6	1864.64	94	1920.1
B6/Cf7	1975.56	95	2034.26
C7/Bs6	2092.96	96	2155.1999
Cs7/Df7	2217.44	97	2283.36
D7	2349.28	98	2419.1599
Ds7/Ef7	2489.04	99	2563.04
E7/Ff7	2637.04	100	2715.44
F7/Es7	2793.84	101	2876.8799
Fs7/Gf7	2959.92	102	3047.92
G7	3135.92	103	3229.1999
Gs7/Af7	3322.48	104	3421.2399
A7	3520.0	105	3624.6399
As7/Bf7	3729.28	106	3840.2
B7/Cf8	3951.12	107	4068.52
C8/Bs7	4185.92	108	4310.3999
Cs8/Df8	4434.88	109	4566.7199
D8	4698.56	110	4838.3199
Ds8/Ef8	4978.08	111	5126.0799
E8/Ff8	5274.08	112	5430.8799
F8/Es8	5587.68	113	5753.7599
Fs8/Gf8	5919.84	114	6095.8399
G8	6271.84	115	6458.3999
Gs8/Af8	6644.96	116	6842.4799
A8	7040.0	117	7249.2799
As8/Bf8	7458.56	118	7680.3999
B8/Cf9	7902.24	119	8137.0399
C9/Bs8	8371.84	120	8620.8
Cs9/Df9	8869.76	121	9133.44
D9	9397.12	122	9676.64
Ds9/Ef9	9956.16	123	10252.1599
E9/Ff9	10548.16	124	10861.76
F9/Es9	11175.36	125	11507.52
Fs9/Gf9	11839.68	126	12191.6799
G9	12543.68	127	12543.68

Music Kit Parameters

This section lists and describes the parameters that are recognized by the Music Kit SynthPatches. The following information is given for each parameter:

- Print name
- Typical value range
- Description of use

Keep in mind that a parameter's print name is used when the parameter is written to a scorefile. In an application, a parameter is known as an integer identifier that's represented as **MK_***printName*. For example, the parameter listed below as **amp** is identified in an application as **MK_amp**.

As described in Chapter 3 of the *Concepts* manual, a constant-valued parameter can be retrieved as any data type, regardless of how it was set. However, the Music Kit SynthPatches always retrieve parameter values as specific types. The type by which a particular parameter is retrieved is implied by the value range. For example, a value range of [0.0, 1.0] implies that the parameter value is retrieved as a **double**; the range [0, 127] means that the type is **int**.

The parameters are organized according to synthesis technique. Many of the parameters, such as those that affect frequency, are common to more than one synthesis technique. Thus, for completeness, the same parameter description may be given in more than one section.

Frequency Modulation Parameters

There are nine subclasses of SynthPatch that perform frequency modulation (fm). The simplest of these are single-modulator instruments:

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

There's also a simple fm that has access to the Music Kit WaveTable database for use in the carrier oscillator:

SynthPatch	Description
DBFm1vi	Simple fm with interpolation, vibrato, and WaveTable database

The database contents are listed in "WaveTable Database," later in this appendix.

The rest of the fm SynthPatches use two modulators. All of these classes provide frequency interpolation and vibrato:

SynthPatch	Description
Fm2cvi	Cascade fm
Fm2cnvi	Cascade fm with random modulation (noise) on the modulators
Fm2pvi	Parallel fm
Fm2pnvi	Parallel fm with noise

The parameters recognized by the fm SynthPatches are listed below; parameters that are recognized by only a subset of the SynthPatches are so noted.

Name	Value Range	Description
amp	[0.0, 1.0] or [−∞, 0.0] dB	Basic amplitude (but see **amp1**). The dB (decibel) scaling is obtained through the **MKdB()** C function; in a scorefile, with the **dB** postfix operator.
amp0	same as **amp**	Amplitude when **ampEnv** = 0.0.
amp1	same as **amp**	Amplitude when **ampEnv** = 1.0; synonym for **amp**.
ampAtt	[0.0, ∞]	**ampEnv** attack duration in seconds.
ampEnv	Envelope object	Amplitude envelope.
ampRel	[0.0, ∞]	**ampEnv** release duration in seconds.
bearing	[–45.0, 45.0]	Stereophonic placement in degrees. 0.0 is center, –45.0 is hard left, and 45.0 is hard right.
bright	[0.0, 1.0]	Modulation index scaler (defaults to 1.0).
cRatio	[0.0, ~10.0]	Carrier frequency as a ratio of the fundamental.
c1Ratio	same as **cRatio**	Synonym for **cRatio**
freq	[~15.0, ~11000.0] or pitch variable	Fundamental frequency (but see **freq1**).
freq0	same as **freq**	Frequency when **freqEnv** = 0.0.
freq1	same as **freq**	Frequency when **freqEnv** = 1.0; synonym for **freq**.
freqAtt	[0.0, ∞]	**freqEnv** attack duration in seconds.

(continued)

Name	Value Range	Description
freqEnv	Envelope object	Frequency envelope.
freqRel	[0.0, ∞]	**freqEnv** release duration in seconds.
keyNum	[0, 127] or key number	Specifies pitch as an index into the default TuningSystem, an array of frequencies. Used only in the absence of **freq**.
m1Ind	[0.0, ~10.0]	Index of modulator 1 (but see **m1Ind1**).
m1Ind0	same as **m1Ind**	Index of modulator 1 when **m1IndEnv** = 0.0.
m1Ind1	same as **m1Ind**	Index of modulator 1 when **m1IndEnv** = 1.0; synonym for **m1Ind**.
m1IndAtt	[0.0, ∞]	**m1IndEnv** attack duration in seconds.
m1IndEnv	Envelope object	Index Envelope for modulator 1.
m1IndRel	[0.0, ∞]	**m1IndEnv** release duration in seconds.
m1Phase	[–180.0, 180]	Initial phase of modulator 1.
m1Ratio	[0.0, ~10.0]	Modulator 1 frequency as a ratio of the fundamental.
m1Waveform	WaveTable object	Waveform of modulator 1.
m2Ind *through* m2Waveform	Same as the similarly named parameters above, applied to modulator 2. Used by the two-modulator SynthPatches only (Fm2cvi, Fm2cnvi, Fm2pvi, and Fm2pnvi).	
noiseAmp	[0.0, 1.0]	Amplitude of noise. This parameter and the following noise parameters are used by Fm2cnvi and Fm2pnvi only.
noiseAmp0	same as **noiseAmp**	Amplitude of noise when **noiseAmpEnv** = 0.0.
noiseAmp1	same as **noiseAmp**	Amplitude of noise when **noiseAmpEnv** = 1.0; synonym for **noiseAmp**.
noiseAmpAtt	[0.0, ∞]	**noiseAmpEnv** attack duration in seconds.
noiseAmpEnv	Envelope object	Noise amplitude Envelope.
noiseAmpRel	[0.0, ∞]	**noiseAmpEnv** release duration in seconds.

(continued)

Name	Value Range	Description
phase	[–180.0, 180]	Initial phase of the carrier, in degrees.
portamento	[0.0, ∞]	Phrase rearticulation time in seconds. Resets the x value of the second point of *all* Envelopes.
rvibAmp	[0.0, 1.0]	Random vibrato amplitude; unused by Fm1 and Fm1i.
svibAmp	[0.0, 1.0]	Sinusoidal vibrato amplitude; see above.
svibFreq	[0.0, ~15.0]	Sinusoidal vibrato frequency; see above.
waveform	WaveTable object or database string	Waveform of the carrier (sine wave by default). The database string is used only by DBFm1vi.
waveLen	power of 2	**waveform** length (optimal value by default).

Wave Table Synthesis Parameters

There are seven subclasses of SynthPatch that implement wave table synthesis. The first four use WaveTable objects that you supply:

SynthPatch	Description
Wave1	One WaveTable
Wave1v	One WaveTable with vibrato
Wave1i	One WaveTable with frequency interpolation
Wave1vi	One WaveTable with vibrato and interpolation

The others have access to the Music Kit WaveTable database:

SynthPatch	Description
DBWave1v	One database WaveTable with vibrato
DBWave1vi	One database WaveTable with vibrato and interpolation
DBWave2vi	Two database WaveTables with vibrato and interpolation

The wave table synthesis parameters are:

Name	Value Range	Description
amp	[0.0, 1.0] or [–∞, 0.0] dB	Basic amplitude (but see **amp1**). The dB (decibel) scaling is obtained through the **MKdB()** C function; in a scorefile, with the **dB** postfix operator.
amp0	same as **amp**	Amplitude when **ampEnv** = 0.0.

(continued)

Name	Value Range	Description
amp1	same as **amp**	Amplitude when **ampEnv** = 1.0; synonym for **amp**.
ampAtt	[0.0, ∞]	**ampEnv** attack duration in seconds.
ampEnv	Envelope object	Amplitude envelope.
ampRel	[0.0, ∞]	**ampEnv** release duration in seconds.
bearing	[–45.0, 45.0]	Stereophonic placement in degrees. 0.0 is center, –45.0 is hard left, and 45.0 is hard right.
freq	[~15.0, ~11000.0] or pitch variable	Fundamental frequency (but see **freq1**).
freq0	same as **freq**	Frequency when **freqEnv** = 0.0.
freq1	same as **freq**	Frequency when **freqEnv** = 1.0; synonym for **freq**.
freqAtt	[0.0, ∞]	**freqEnv** attack duration in seconds.
freqEnv	Envelope object	Frequency envelope.
freqRel	[0.0, ∞]	**freqEnv** release duration in seconds.
keyNum	[0, 127] or key number	Specifies pitch as an index into the default TuningSystem, an array of frequencies. Used only in the absence of **freq**.
phase	[–180.0, 180]	Initial phase of the first WaveTable, in degrees.
portamento	[0.0, ∞]	Phrase rearticulation time in seconds. Resets the x value of the second point of *all* Envelopes.
rvibAmp	[0.0, 1.0]	Random vibrato amplitude; unused by Fm1 and Fm1i.
svibAmp	[0.0, 1.0]	Sinusoidal vibrato amplitude; see above.
svibFreq	[0.0, ~15.0]	Sinusoidal vibrato frequency; see above.
waveform	WaveTable object or database string	First WaveTable (but see **waveform1**). The database string is used only by database SynthPatches.

(continued)

Name	Value Range	Description
waveform0	same as **waveform**	WaveTable when **waveformEnv** = 0.0. Used by DBWave2vi only.
waveform1	same as **waveform**	WaveTable when **waveformEnv** = 1.0; synonym for **waveform**.
waveformEnv	Envelope object	Envelope that pans between the two WaveTables. Used by DBWave2vi only.
waveLen	power of 2	**waveform** length (optimal value by default).

Pluck Parameters

There's only one Pluck. An example of physical modeling synthesis, Pluck derives much of its characteristic sound naturally, without requiring much attention. As such, it recognizes fewer parameters than the other SynthPatches:

Name	Value Range	Description
amp	[0.0, 1.0] or [−∞, 0.0] dB	Basic amplitude (but see **amp1**). The dB (decibel) scaling is obtained through the **MKdB()** C function; in a scorefile, with the **dB** postfix operator.
ampRel	[0.0, ∞]	noteOff release time in seconds (to −60 dB).
bearing	[-45.0, 45.0]	Stereophonic placement in degrees. 0.0 is center, −45.0 is hard left, and 45.0 is hard right.
decay	[0.0, ∞]	Initial decay in seconds (to −60 dB). 0.0 means no decay.
freq	[~15.0, ~11000.0] or pitch variable	Fundamental frequency.
keyNum	[0, 127] or key number	Specifies pitch as an index into the default TuningSystem, an array of frequencies. Used only in the absence of **freq**.
lowestFreq	same as **freq**	Lowest frequency among subsequent noteUpdates.
pickNoise	[0.0, ~0.06]	Duration of initial noise burst in seconds.
sustain	[0.0, 1.0]	Level of sustain.

WaveTable Database

The Music Kit provides a library of WaveTable objects that can be accessed by the DB family of SynthPatches: DBFm1vi, DBWave1v(i), and DBWave2vi. The entries in the database are referred to by string names, both in an application and in a scorefile. The precise WaveTable that's used depends on the frequency of the Note that's being synthesized. By default, the value of **freq** (or its synonym **freq1**) is used to determine the Note's frequency; you can find an entry based on the value of **freq0** by preceding the database string name with the character "0" (zero); for example, "0BA".

Name	Description
"BA"	Bass voice "ah"
"BE"	Bass voice "eh"
"BO"	Bass voice "oh"
"BU"	Bass voice "oo"
"TA"	Tenor voice "ah"
"TE"	Tenor voice "eh"
"TI"	Tenor voice "ee"
"TO"	Tenor voice "oh"
"TU"	Tenor voice "oo"
"SA"	Soprano voice "ah"
"SE"	Soprano voice "eh"
"SI"	Soprano voice "ee"
"SO"	Soprano voice "oh"
"SU"	Soprano voice "oo"
"VCA"	Cello attack
"VCS"	Cello sustain
"VNA"	Violin attack
"VNS"	Violin sustain
"TR"	Trumpet
"BN"	Bassoon
"AS"	Alto saxophone
"SS"	Soprano saxophone
"BC"	Bass clarinet
"CL"	Clarinet
"EH"	English horn
"OB"	Oboe
"PN"	Piano
"TW"	Triangle wave
"SW"	Sawtooth wave
"IW"	Impulse wave

Appendix C
Details of the DSP

Appendix C
Details of the DSP

Memory Map

The following table describes the memory map for the DSP private RAM (8K words).

Start	End	Name
p:0	p:$1FF	On-chip program RAM ('$' denotes hex)
p:$2000	p:$3FFF	Off-chip program RAM, image 1
p:$A000	p:$BFFF	Off-chip program RAM, image 2
x:0	x:$FF	On-chip data RAM, x bank
x:$100	x:$1FF	On-chip data ROM, x bank (Mu-Law, A-law tables)
x:$2000	x:$3FFF	Off-chip data RAM, x bank, image 1
x:$A000	x:$AFFF	Off-chip data RAM, x bank, image 2
y:0	y:$FF	On-chip data RAM, y bank
y:$100	y:$1FF	On-chip data ROM, y bank (Sine wave cycle)
y:$2000	y:$3FFF	Off-chip data RAM, y bank, image 1
y:$A000	y:$AFFF	Off-chip data RAM, y bank, image 2

Off-chip memory exists in two "images" for each space. In image 1, all three memory spaces occupy the same physical memory (in other words, the X/Y~, PS~, and DS~ pins of the DSP56001 are not connected when address line A15 is low). In image 2, x and y are split into separate 4K banks, and p overlays them both with an 8K image (that is, X/Y~ is used as address line A12 and PS~ and DS~ are not connected when A15 is high). External memory starts at 8K ($2000) instead of 512 ($200) because address line A13 in the DSP must be high to enable external DSP RAM. (Note that there is another enable for this RAM in the System Control Register 2.)

DSP D-15 Connector Pinouts

The following describes the output pins of the DSP D-15 connector at the back of the cube. The left column is the connector pin number, and the right column is the signal name as it appears in the Motorola *DSP56000/DSP56001 Digital Signal Processor User's Manual.*

D-15	DSP
1	SCK
2	SRD
3	STD
4	SCLK
5	RXD
6	TXD
7	+12V, 500mA
8	−12V, 100mA
9	GND
10	GND
11	GND
12	SC2
13	SC1
14	SC0
15	GND

Figure C-1 shows the circuit through which signals are sent from the DSP to the D-15 connector.

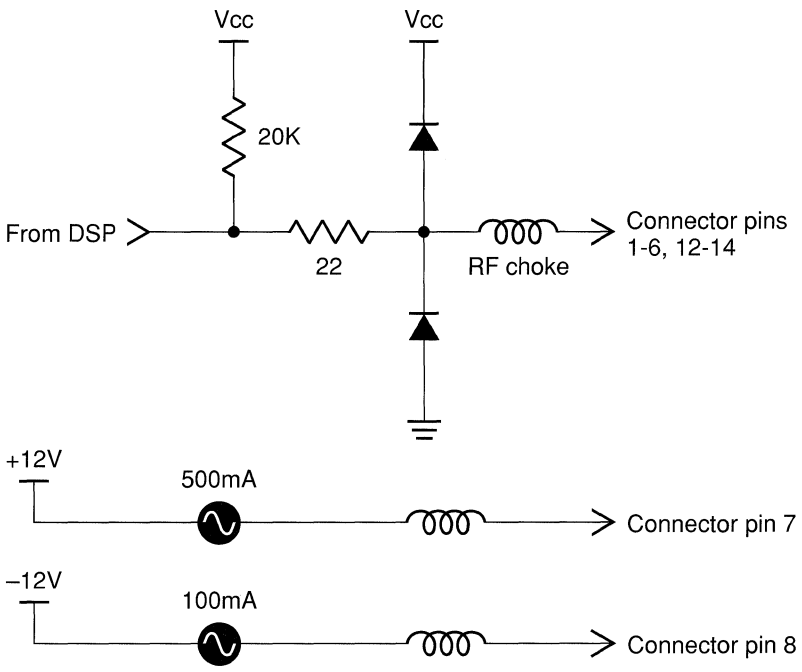


Figure C-1. D-15 Connector

There's a series RF choke on each connector signal that doesn't affect its steady-state level.

DSP56001 Instruction Set Summary

The following notation is used in the summary:

Notation	Denotes
'*'	Instructions that don't allow parallel data moves
[a,b]	One of a or b
<a,b>	Either a,b or b,a
<n>	A nonnegative integer
#I<n>	n-bit immediate value
A<n>	n-bit absolute address
An	A0, A1, or A2 (similarly for Bn)
Xn	X0 or X1 (similarly for Yn)
Rn	R0, R1, R2, R3, R4, R5, R6, or R7 (similarly for Nn, Mn)
AnyEa	Addressing modes (Rn)[–[Nn]], (Rn+Nn), –(Rn) (similarly for An)
AnyXY	[x,y]:AnyEa
AnyIO	[x,y]:<<pp (x or y peripheral address, 6 bits, 1's extended)
Creg	Registers Mn, SR, OMR, SP, SSH, SSL, LA, LC
Dreg	Registers Xn, Yn, An, Bn, A, B
Areg	Registers Rn, Nn
AnyReg	Registers Dreg, Areg, Creg
cc	CC(HS) CS(LO) EC EQ ES GE GT LC LE LS LT MI NE NR PL NN

left-justified moves: → [A,B,Xn,Yn]
right-justified moves: → [An,Bn,Rn,Nn]

Arithmetic Instructions

ABS [A,B]	Absolute Value
ADC [X,Y],[A,B]	Add Long with Carry
ADD [X,Xn,Y,Yn,B,A],[A,B]	Add
ADDL [B,A],[A,B]	Shift Left then Add (D=2*D+S)
ADDR [B,A],[A,B]	Shift Right then Add (D=D/2+S)
ASL [A,B]	Arithmetic Shift Left (D1=D1*2)
ASR [A,B]	Arithmetic Shift Right (D1=D1/2)
CLR [A,B]	Clear Accumulator
CMP [Xn,Yn,B,A],[A,B]	Compare (CCR=Sign(D1−S))
CMPM [Xn,Yn,B,A],[A,B]	Compare magnitude (CCR=Sign(D−S))
*DIV [Xn,Yn],[A,B]	Divide Iteration (D/S iteration)
MAC ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply-Add (no X1*X1, Y1*Y1)
MACR ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply, Accumulate, and Round
MPY ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply (no X1*X1, Y1*Y1)
MPYR ±[Xn,Yn],[Xn,Yn],[A,B]	Signed Multiply-Round (no X1*X1, Y1*Y1)
NEG [A,B]	Negate Accumulator
*NORM [A,B]	Normalize Accumulator Iteration
RND [A,B]	Round Accumulator
SBC [X,Y],[A,B]	Subtract Long with Carry (D = D − S − C)
SUB [X,Xn,Y,Yn,B,A],[A,B]	Subtract (D = D − S)
SUBL [B,A],[A,B]	Shift Left then Subtract (D = 2*D − S)
SUBR [B,A],[A,B]	Shift Right then Subtract (D = D/2 − S)
*Tcc [Xn,Yn,B,A],[A,B]	Transfer Conditionally
TFR [Xn,Yn,B,A],[A,B]	Transfer Data ALU Register
TST [A,B]	Test Accumulator

Logical Instructions

AND [Xn,Yn],[A,B]	Logical AND (D1=D1&S)
*ANDI #I8,[MR,CCR,OMR]	AND Immediate with Control Register
EOR [Xn,Yn],[A,B]	Logical Exclusive OR (D1=D1 XOR S)
LSL [A,B]	Logical Shift Accumulator Left (D1=D1<<1)
LSR [A,B]	Logical Shift Accumulator Right (D1=D1>>1)
NOT [A,B]	Logical Complement on Accumulator (D1=~D1)
OR [Xn,Yn],[A,B]	Logical Inclusive OR (D1=D1S)
*ORI #I8,[MR,CCR,OMR]	OR Immediate with Control Register
ROL [A,B]	Rotate Accumulator Left ([C,D1] ROL)
ROR [A,B]	Rotate Accumulator Right ([D1,C] ROR)

Bit Manipulation Instructions

*BCLR #B5,AnyXY	Bit Test and Clear (C = Selected bit)
*BSET #B5,AnyXY	Bit Test and Set (C = Selected bit)
*BCHG #B5,AnyXY	Bit Test and Change (C = Selected bit)
*BTST #B5,AnyXY	Bit Test on Memory (C = Selected bit)
*JCLR #B5,[AnyXY,AnyIO],xxxx	Jump if Bit Clear
*JSET #B5,[AnyXY,AnyIO],xxxx	Jump if Bit Set
*JSCLR #B5,[AnyXY,AnyIO],xxxx	Jump to Subroutine if Bit Clear
*JSSET #B5,[AnyXY,AnyIO],xxxx	Jump to Subroutine if Bit Set

Loop Instructions

*DO [[x,y]:[AnyEa,A12],AnyReg],L	Start Hardware Loop (L=Label after end)
*ENDDO	Exit from Hardware Loop

Move Instructions

*LUA (Rn)[±[Nn]],[Rn,Nn]	Load Updated Register
MOVE (NOP)	Move Data
*MOVEC <AnyXY,Creg>	Move Control Register
*MOVEC [#I16,#I8],Creg	Move Control Register
*MOVEC <Creg,AnyReg>	Move Control Register
*MOVEM <p:AnyEa,AnyReg>	Move Program Memory
*MOVEP <[AnyReg,AnyXY],AnyIO>	Move Peripheral Data
*MOVEP #I24,AnyIO	Move Peripheral Data

Program Control Instructions

*Jcc [A12,AnyEa]	Jump Conditionally
*JMP [A12,AnyEa]	Jump
*JScc [A12,AnyEa]	Jump to Subroutine Conditionally
*JSR [A12,AnyEa]	Jump to Subroutine
*NOP	No Operation
*REP [AnyXY,#I12,AnyReg]	Repeat Next Instruction
*RESET	Reset Peripherals
RTI	Return from Interrupt
RTS	Return from Subroutine
*STOP	Stop Processing
*SWI	Software Interrupt
*WAIT	Wait for Interrupt

Index

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

Printed in U.S.A.
2911.00
12/90