Mini-BAE C API

Developer Reference

Document Version 1.0.1
Modified 5/19/00

Reflects Mini-BAE Version 1.0.0

Synopsis

This document is intended to teach programmers how to use the portable C language application program interface (API) of the Beatnik Audio Engine, Mini Edition (Mini-BAE) library in their software projects.

Related Documentation

For musicians' and sound artists' information on preparing music, dialog, and sound effects content for Mini-BAE playback, please refer to:

http://www.beatnik.com/

Contents

An Introduction to Mini-BAE

The platform-neutral Beatnik Audio Engine, Mini Edition (Mini-BAE) is an exceptionally mature, well-rounded, and reliable computer music and sound system specially customized for small-footprint and embedded applications.

Mini-BAE is…

- **Flexible**
  It can play several leading industry-standard music and audio file formats from disk or from memory.
- **Interactivity-oriented**
  You can pause and resume any media playback at will, change pitch while a sound is playing, etc.The feature-rich Mini-BAE API is full of opportunities for innovative integration of sound design into interactive virtual worlds– for example, Mini-BAE includes a MIDI synthesizer that your application – or user – can play in real-time.
- **Great-Sounding**
  The audio engine DSP quality is very good, comparable to high-end PC wavetable sound cards even though the processing is entirely software-based, and audio quality is selectable all the way from 8-bit 8 kHz up to 16-bit 44.1 kHz.
- **Powerful and Extensible**
  The Mini-BAE music synthesizer is a high-quality wavetable instrument with LFOs, ADSRs, and filters. It can be delivered with a complete General MIDI patch bank, or for small footprints a ‘light’ General MIDI approximation can be used. Custom instrument banks can be built with the Beatnik Editor application.
- **Efficient**
  Despite Mini-BAE’s power, the per-voice processor load is impressively low.
- **Hardware Independent – Sound Card-Agnostic**
  Some other sound systems require consumers to buy expensive (and quickly out-dated) hardware sound cards – but Mini-BAE avoids nearly all sound card support issues because it’s 100% software. Mini-BAE requires no particular sound hardware in the host system, only an audio output.
- **Consistent Across Platforms**
  Because it’s a software-only solution, consistently great-sounding music is much easier to achieve with Mini-BAE than with many other computer music systems.
- **An Inexpensive Cross-Platform Audio Solution**
  Mini-BAE provides the same music and sound capabilities on every supported platform, so porting music and sound content is a no-brainer, requiring literally no additional sound artist work– simply re-use the files.

Before the was the Mini-BAE, there was the original Beatnik Audio Engine – which started out in life as a game sound system.

In March of 1991, game development veterans Steve Hales and Jim Nitchals joined together to create a licensable sound effects and music package for the Macintosh games industry. Their first goal was reasonable playback of standard MIDI files via software wave table synthesis on a Macintosh Plus and Macintosh II. By fall the technology was ready and they had their first licensee.

Brøderbund software had committed musical resources, and Presage Software programming resources to bring **Prince of Persia** to the marketplace in early 1992, which was a critical and financial success for both companies. Next came another well known title, **Lemmings**, for the Macintosh, and then **Out of This World** from Interplay.

Apple Computer wanted to license the technology for QuickTime, which led to Jim deciding to work with Apple on QuickTime Music Architecture.

Steve continued to license the technology to third parties, and created a company to improve this technology by hiring professional musicians. This led to a variety of music authoring projects including After Dark's **Disney Screen Saver Collection**. In early 1993, General Magic licensed the technology and commissioned a sound and music management API for their **MagicCap** operating system and all of the MagicCap applications. Later that year Logitech wanted a software MIDI file player to drive their **AudioMan** serial digital audio output device. We worked together with Sapien Technologies for the Windows version, and the product shipped that fall.

By late 1993, SoundMusicSys, as it had come to be called, had evolved into a full-blown sound management solution that included MIDI

Who Uses BAE?

Embedded & Operating Systems

BeOS
WebTV
MagicCap OS *from General Magic /Sony MagicLink*
QuickTime Music *from Apple*
AudioMan *from Logitech*
Beatnik Player for web browsers *from Beatnik*

Games

Hexen *from id Software*
Descent *from Interplay Productions, Inc.*
Legend of Kyrandia *from Westwood Studios/Interplay Productions*
Wolfenstien 3D
Flashback
Out of this World
Mario Teaches Typing
Star Trek:25th anniversary
Wing Commander 3 and 4 *from Origin Systems and Lion Entertainment*
3D Ultra Pinball *from Sierra Online*
The Incredible Machine
Even More Incredible Machine
Incredible Machine 3.0
Lode Runner: The Legend Returns
Prince of Persia, One and Two *from Brøderbund Software*
Where in the World is Carmen Sandiego?
Where in Time is Carmen Sandiego?
Where in the USA is Carmen Sandiego?
Where in Space is Carmen Sandiego?
Carmen Sandiego Junior Detective
Math Workshop
The Backyard
Alien Tales
James Discovers Math
Lemmings *from Psygnosis/Sony*
Oh no more Lemmings *from Psygnosis/Sony*
RoboSport *from Maxis*
SimCity 2000
SimAnt
SimEarth
SimLife
Widget Workshop
SimTown
Elfish
Zurk's Learning Safari
Zurk's Rainforest Lab
Zoop *from Viacom New Media*
Spectre Supreme *from Velocity*
Spectre VR
Out of the Sun *from Domark*
Flying Nightmares
Return to Zork *from Activision*
7th Guest *from Virgin Interactive*
Dinonauts *from Virgin Sound and Vision*
Dinosaur Adventure *from Knowledge Adventure*
Falcon MC *from Spectrum HoloByte,*
Breakthru
Star Trek: The Next Generation - “A Final Unity”
Take-a-break Crosswords
ClockWerx
Amazon Trail *from MECC Software*
Odell Down Under
SnapDragon
Dino Park Tycoon
Oregon Trail II

Screen Savers

After Dark 3.0 *from Berkeley Systems*
Simpsons Screen Saver Collection
Disney Screen Saver Collection
Star Trek: The Next Generation
Totally Twisted Collection
Looney Tunes Collection
Star Wars Screen Entertainment *from LucasArts*

playback, digital streaming, and one-shot sound effects management. With over 50 published titles, it had become the de facto standard for the Macintosh gaming community, and Steve's company had grown to 15 employees.

RIP Igor

In July of 1994 Steve Hales left that company, taking the SoundMusicSys technology to continue to pursue the goal of a cross-platform software MIDI sampler and sound effects system, under the new name Igor's Software Laboratories. At Igor, the cross-platform aspect of SoundMusicSys really took off: the technology was ported to Wintel with the help of Sapien Technologies, and to a variety of dedicated tube-top web browser appliances.

In January of 1997, Igor's Software Laboratories was acquired by Headspace, Inc., a leading provider of audio content and technology for the Internet and multimedia led by a certain Thomas Dolby Robertson who, it's rumored, was once blinded with Science and failed in Geometry by an unspecified female person. As a creator of musical content, Headspace has sold work to many customers, including Netscape Communications, whose Navigator 3.0 section of its web site was sonified by a team of composers from Headspace; and SegaSoft, who used Headspace content for their highly-lauded CD-ROM **Obsidian**.

By coupling the Igor Labs interactive sound drivers (now renamed the Headspace Audio Engine) to Headspace's first-class interactive music and sound design techniques, Headspace is seeking to elevate interactive audio to a new level. "For too long, sound has been a second-class citizen in the corporate world of desktop computing. Music and sound are central to all forms of communication and entertainment, yet computing still seems so cold and impersonal," noted Thomas at the time. "I couldn't figure out why 200 million people are still staring at silent monitor screens, so instead of yelling at computer companies, I decided to build a better mousetrap. With Igor Labs' excellent technology and staff on board, Headspace will forever change the way computers sound."

In 1999, Headspace changed its name to Beatnik, Inc., and the Headspace Audio Engine became the Beatnik Audio Engine. And late in 1999 the compact, embed-able Mini-BAE was created.

Moving Ahead

OK, enough hype already! Let's go have a look at how Mini-BAE works.

Functional Architecture

This chapter presents a rundown of the music and sound media types that the Mini-BAE plays, and outlines the internal infrastructure used to play them. Details on audio signal flow and the instrument access mechanism are also presented.

Types of Sound Media: MIDI Music and Digital Audio

Mini-BAE can play music and sound media in the following formats:

Usage	Media Type	Filename Extension
MIDI Music	**RMF** (Rich Music Format)	.rmf
	Standard MIDI File (Musical Instrument Digital Interface)	.mid, .midi
	Direct MIDI Messages (Musical Instrument Digital Interface)	(calls, not a file)
Digital Audio	**WAV** (Windows WAV files) (2 variations of this basic format, in either 8-bit or 16-bit sample bit depth) Standard uncompressed PCM IMA 4:1 compressed	.wav
	AIFF (Audio Interchange File Format)	.aif, .aiff
	AU (Unix Audio)	.au

Mini-BAE is able to play different media types together, at the same time[1]. All of their audio is heard at once, mixed together.

MIDI Music Rendering: The Mini-BAE Synthesizer

Mini-BAE plays MIDI music with its MIDI Synthesizer, which accepts MIDI event messages and emits an audio stream. It's a full-featured, high-quality, software-only MIDI synth with features very like the internals of a keyboard synthesizer musical instrument. This instrument can be supplied with Beatnik's great-sounding General MIDI instrument set, or with a small-footprint simulation subset. You can drive the MIDI Synthesizer from two sources – either from music files (Standard MIDI files or RMF files), or with a direct stream of individual MIDI event messages from your application. You can do both at once, if you like.

To play music files (MIDI or RMF), Mini-BAE includes a playback-only sequencer that reads the files, either in memory or straight from disk, and sends the recorded musical events to the MIDI Synthesizer. In the case of direct MIDI messages, it's up to your application to perform an appropriate series of MIDI event messages to play the MIDI Synthesizer like fingers on a keyboard.

Wherever the MIDI events come from, the Mini-BAE MIDI Synthesizer uses instrument definitions to render the notes it plays. These instruments[1] are typically organized into indexed collections called 'banks' or 'MIDI patch[2] banks' using the Beatnik Editor application. When one of the MIDI sources sends the synthesizer a MIDI 'Program Change' event, the Synthesizer scans its patch banks for an instrument definition with that same number, and uses it to render subsequent notes. In other words, the instrument is specified indirectly – the Program Change event simply references a slot number, and doesn't include any actual instrument definition data.

Internally, the MIDI Synthesizer uses software digital signal processing (DSP) code to shape the stored samples according to the pitch and timing requirements of each MIDI NoteOn message it receives. At a minimum this DSP involves transposing the pitch by altering the sampling rate and interpolating, and imposing a volume envelope by multiplying each individual sample with the instantaneous volume value of the envelope.

About the Standard MIDI File Format

The Standard MIDI File format is directly based on the same kinds of MIDI event messages that are streamed over MIDI cables between keyboards and instruments during musical performances – for example, Note On, Note Off, or Program Change. A Standard MIDI File records a musical performance as a series of such MIDI events interspersed with time-delay messages. Because of MIDI's origin in music performance – rather than composition or recording – support for compositional requirements like musical structures, or repeating a section of the performance, is primitive or absent in the MIDI File standard.

(The Standard MIDI File format standard is administered by the MIDI Manufacturer's Association.)

---

1. Within certain limits: It's possible to ask Mini-BAE to attempt to play more media at once than the processor can handle.

1. Which at heart are specialized tables of audio samples with optional loop points and keyboard mapping tables, and with parameter tables containing optional modulations like amplitude and pitch envelopes, and periodic variations like vibrato.

2. The term 'patch' survives from the days of analog electronic music synthesizers, some of which required 'patch' cables like old mechanical telephone plugboards. In Mini-BAE, the term 'patch' can be regarded as a synonym for 'instrument definition'.

About the RMF File Format

An RMF file is a small, efficient music file that bundles all resources for a song – performance, instrument, sample, and copyright information – into a single file with data compression and encryption, optimized for real-time playback. You create RMF files using the Beatnik Editor application. The performance portion of RMF is based on the MIDI file model, but is extended with new embedded commands for looping, track muting, and other real-time playback variations. Data compression for MIDI data and audio samples is also provided – audio data compression options include IMA 4:1, Sun mu-law, and Lossless Beatnik.

(The Rich Music Format standard is administered by Beatnik.)

Sending Direct MIDI Event Messages

In addition to playing back pre-recorded MIDI performances from files, Mini-BAE also allows you to create your own music in real time by sending a live stream of MIDI events into the Mini-BAE music synthesizer. Sending MIDI messages like Note On and Note Off through API functions lets you build music “on the fly”.

This opens the door to a rich range of interactive sound design possibilities. You could easily create a professional-sounding (and quite playable) onscreen keyboard instrument, complete with ranks of instrument select buttons. Or you could attach musical gestures like short note sequences to onscreen objects, and trigger them on rollovers or clicks. Or you could allow your user to play along live with a pre-composed song being played from a file[1].

Depending upon the requirements of your particular design, you might want to consider handling your game sound effects with the Mini-BAE MIDI synthesizer (rather than individual digital audio files) by creating a custom instrument bank containing sound effect samples instead of instrument recordings. That would allow you to manage all the sounds as a single file, and let you trigger any sound just by sending the synth the right Note On event.

Digital Audio Playback

Unlike MIDI or RMF, which store instructions for a synthesizer on how to recreate a piece of music at the note level, a digital audio file is the recorded waveform of an actual sound– if you were to stream the list of individual sample values to a DAC at the right rate, you’d hear the recorded sound again. Mini-BAE can play digital audio data straight from a disk file, or from an image in memory.

Digital Audio Files

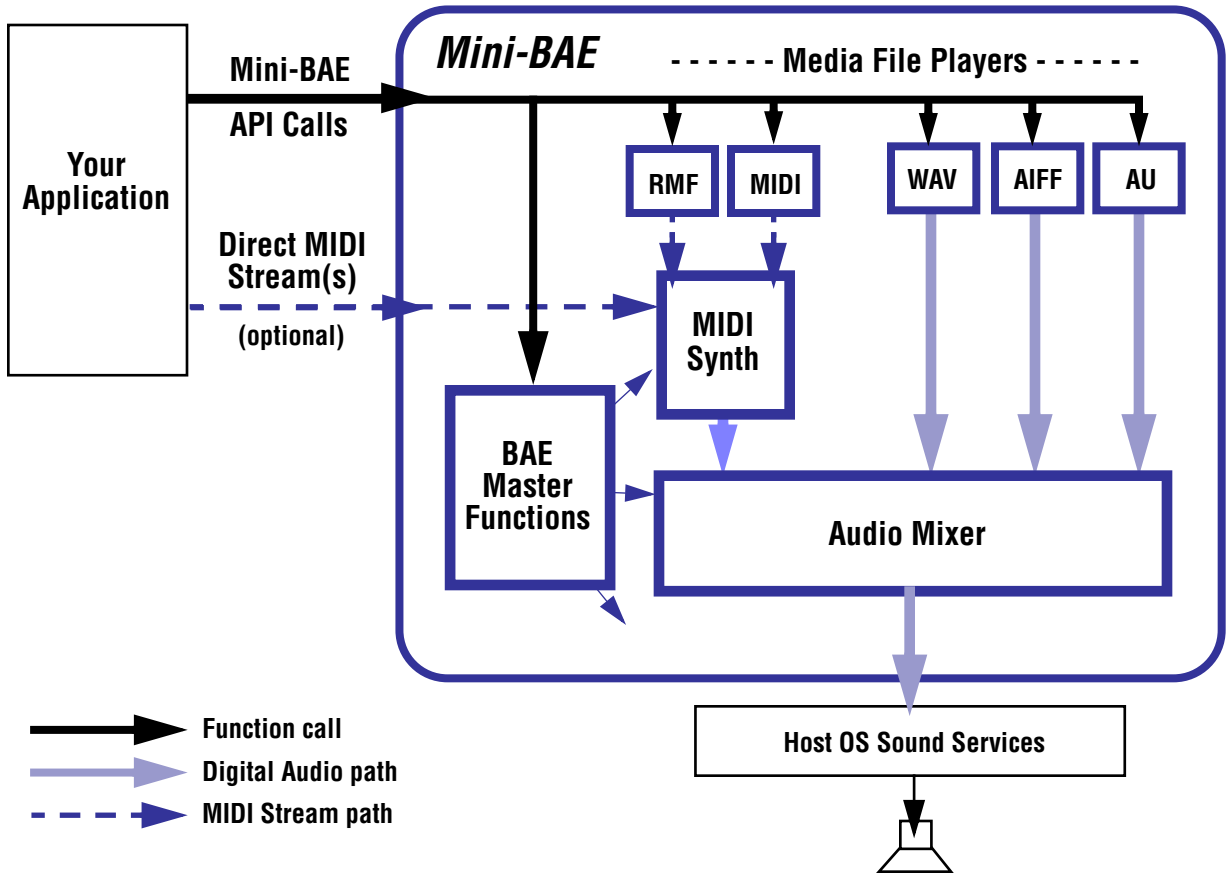
For convenience in working with your sound artists, Mini-BAE is able to directly play a variety of popular digital audio file formats. For flexibility in tuning runtime performance and RAM and disk space, a wide range of sampling rates is also supported. And because you already have plenty of other things to get headaches over, at run-time Mini-BAE can play multiple digital audio files together at the same time, even if they all have different file formats and sampling rates. The files can either be pre-loaded in RAM, or loaded from disk for playback.

1. Further scenarios are left to you, creative reader, as an exercise.

Mini-BAE Functional Blocks

Before getting too deeply into the low-level programming specifics, you may find it helpful to know a little about how Mini-BAE is organized.

Mini-BAE furnishes playback services for a variety of music and sound media formats. Major internal functions include piping the audio output stream to the host computer audio hardware, performing DSP for music synthesis and audio mixing, and coordinating timing. In order to do all that, Mini-BAE is organized into several functional blocks: an Audio Mixer, a MIDI Synthesizer, several media file players, and a set of master control and information functions:



The Audio Mixer

In general terms, an audio mixer is a device that accepts some number of audio signal inputs arriving from a variety of sound-generating sources, and combines them into just one output signal. The large consoles you see in photos of recording studios and film mix stages are (among other things) audio mixers.

Mini-BAE's audio mixer is in many ways the center of the system, the connecting point where all sound sources are merged into a single signal. At any given moment, the mixer's inputs are fed by whatever Mini-BAE audio sources are active at the time – for example, the MIDI Synthesizer and two WAV files might all be playing at the same time. By default, the audio mixer is configured to accept up to 8 audio input sources. A compile `#define` allows the audio mixer to accommodate more or fewer inputs, up to a maximum of 64.

The output of the audio mixer is fed to the host device's sound output hardware (or to the OS sound system software), which in turn drives the connected amplifiers and speakers.

In the Mini-BAE C API, the audio mixer and MIDI Synthesizer are presented together as a single pseudo-object called the BAEMixer_. Every Mini-BAE client application needs to create exactly one BAEMixer_, which is used to access all mixer and synthesizer functions.

Master Functions

Mini-BAE also furnishes a number of overall system control, timing, and query functions. You can set several general operating modes, including the sample rate and what method is used to interpolate values between the stored sample values. Several system status **Get…()** functions are also furnished.

All master functions are accessed through the same BAEMixer_ pseudo-object.

Music and Sound Files

Unless you're able to achieve your sound design objectives solely by sending MIDI messages to the Mini-BAE synthesizer, you'll be dealing with music and sound media like WAV and RMF files. Although not shown in the above diagram, these files contain the data that drive all the media players.

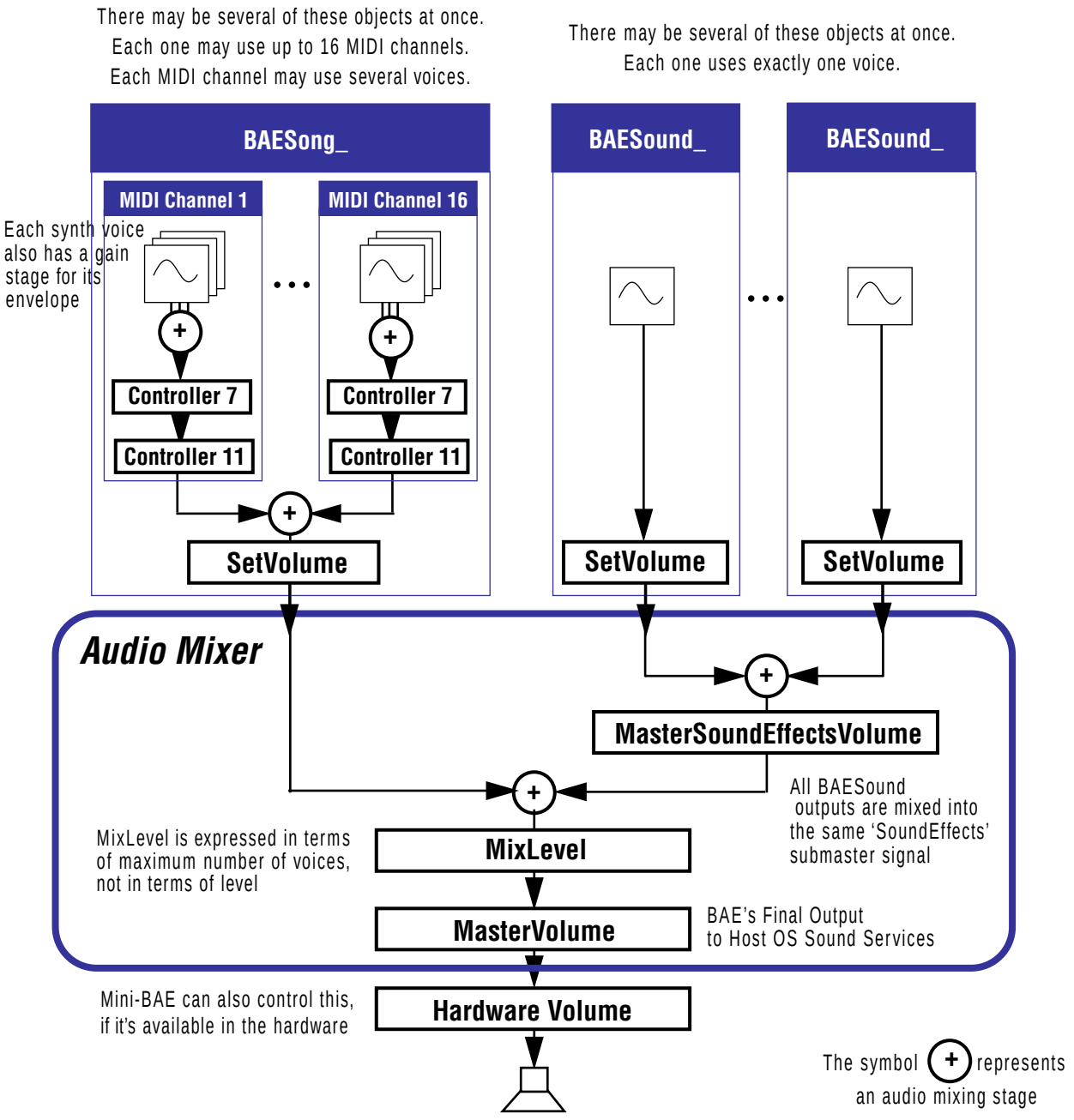
Mini-BAE also represents music and sound files by pseudo-objects – a BAESong_ is used for every MIDI or RMF music file, and a BAESound_ is used for every digital audio file.

Note that the distinction between BAESong_ and BAESound_ is based on the file format, not the style of audible content stored in the file. For example, playing a BAESound_ can produce music if the stored audio was recorded from a musical source like a rock CD, and an RMF file can contain sound effects or speech, not just music.

Mini-BAE's Gain Structure

Mini-BAE can be seen as a fairly involved audio system. Like most audio systems it has a multi-stage, hierarchical gain structure in which the final volume of a sound or musical note is influenced by several volume controls all operating simultaneously, in series. It's important to understand this gain structure, because if any one of those volume controls is set too low, you'll lose all sound – and if any one of them is too high, you'll risk digital overload distortion. You can set and get any of these volume controls through the API at any time.

The following diagram illustrates the gain stages in the Mini-BAE signal path.



At the start of the chain, individual pieces of audio media and individual MIDI channels each have their own volume setting facilities. All have a **SetVolume()** call, and BAESong_ objects also have two MIDI Continuous Controllers per MIDI channel affecting gain: Controller 7 is Volume, and Controller 11 is Expression. (Continuous controllers can be set both from MIDI data and via the API.)

The next several gain stages happen in the audio mixer (BAEMixer_), but as the diagram shows, what happens depends upon the media type.

All digital audio voices – that is, samples loaded into BAESound_ objects – are submixed together, and a master volume control for that submix is provided (called **MasterSoundEffectsVolume**[1]).

By contrast, all audio generated by music synthesizer objects (BAESong_) bypasses the **MasterSoundEffects** submix and volume control, and is instead mixed together with the output of the **MasterSoundEffectsVolume** submix.

The output of this second mix is attenuated to avoid overflow according to the BAEMixer_'s `mixLevel` property[1]. Mini-BAE then provides an overall volume control called **MasterVolume** for its final audio mix output. If the host computer's sound hardware (or the host OS sound software) offers control over a further or hardware-based master volume control in hardware, Mini-BAE provides **HardwareVolume** functions to set and get that too.

MIDI Programs and Beatnik Instruments

The relationship between MIDI Program numbers, Beatnik Instruments, and Beatnik Samples is potentially confusing, so we'd like to explain it before you get too deeply into coding. It's important to understand how Beatnik Instruments relate to traditional MIDI Programs, and that this multi-layered asset access scheme uses three separate numbering spaces: standard MIDI Program numbering, a different number space for Beatnik Instruments, and a third number space for Beatnik Samples.

About Programs and Banks

As you probably know, each MIDI channel uses one 'instrument' at a time to render musical notes occuring on that channel. You select a channel's instrument with MIDI Program Change messages. These Program Changes have just a single parameter – a MIDI Program number ranging from 0 through 127. Program Change messages select a whole Program, they don't adjust individual instrument characteristics such as waveform or envelope. In the Beatnik platform, these MIDI Program numbers access Beatnik Instrument definitions that music and sound artists have created using the Beatnik Editor application.

A MIDI Program number selects an instrument definition within the context of a "bank" of instruments with 128 slots, also numbered 0–127. Related instrument definitions are usually grouped into such banks, rather than being managed individually. When a MIDI player allows access to multiple banks at the same time (as Mini-BAE does), the banks are also accessed by number, and you can set each MIDI channel's bank independently with MIDI Bank Select messages (MIDI Continuous Controller 0, where the controller value is the desired bank number). Program Change messages always access instruments from the channel's currently selected bank. In the absence of any Bank Select message, the default bank is always Bank 0.

The set of available banks is not specified in the MIDI standards, however. Mini-BAE is generally delivered with a reduced version of Beatnik's standard 'Headspace Bank' instrument library, which contains two Banks of instruments: Bank 0 is a General MIDI sound set (where each program number accesses a standard, prescribed instrument), and Bank 1 is the Beatnik Special sound set, containing interesting and useful variations. RMF files can also include instruments for use with the specific song, and these instruments behave as though included in Bank 2 of the

---

1. Despite the name **MasterSoundEffectsVolume**, its scope isn't tied to sound effects *per se* – it would also affect digital dialog and sampled music files.
1. Note that `mixLevel` works differently from the other volume settings. By default, `mixLevel` is set to 8, meaning that the mixer can handle 8 simultaneous voices with full-scale 16-bit signals without breaking into overflow distortion. You can set `mixLevel` differently if you want or need to.

instrument library.

Note that selecting a program without an instrument – that is, an empty slot – produces silent notes for that channel (and only that channel). In Bank 2, or when not using the Headspace Bank, there is no guarantee that any given MIDI Program number will actually access a valid instrument definition.

Channel 10 'Percussion' Programs

The previous description is a slight oversimplification, in that that scheme is not used for MIDI channel 10, which instead follows the General MIDI 'Drum Channel' guidelines for accessing MIDI Programs. The drum channel concept also adds a new element to the bank container, extending the basic 128-slot Bank with a further 128 slots for 'Percussion' programs, which are numbered from 128-255. Any bank may contain percussion programs in addition to the 128 'melodic' instrument slots.

The term 'percussion instruments' is somewhat misleading, as these slots can be used for any kind of instrument, voice, or sound effect – not just percussive sounds. However, percussion programs can be accessed only from the Drum Channel (MIDI channel 10).

On MIDI channel 10, the Programs in a bank are accessed differently from the method that normal melodic channels use. The drum channel selects different programs not with Program Change messages, but instead with MIDI Note On events – each of the 127 possible MIDI note numbers both selects the correspondingly numbered percussion instrument and plays at its natural pitch – that is, the note's pitch is ignored. By contrast, on any other MIDI channel a typical instrument transposes every note according to the MIDI note number, and the instrument is selected with a Program Chage messages.

A Bank Select message on MIDI channel 10 determines which bank's percussion instruments will be used for that channel. (A side effect of this is that you can't play percussion instruments from different banks together at the same time.)

768 Program Slots

To pull all the above information together into a single picture, Mini-BAE offers a total of 768 slots for instrument definitions – three banks of 256 slots each, with each bank split into 128 slots for the non-Drum channels and 128 percussion slots for the Drum Channel (MIDI Channel 10):

Bank Name	MIDI Bank Select Numbers	MIDI Program Numbers	API Program Numbers	Instruments
General MIDI (Headspace Bank)	**0**	**0 - 127**	**0 - 127**	**Melodic**
		128 - 255	**128 - 255**	**Percussion**
Beatnik Special (Headspace Bank)	**1**	**0 - 127**	**256 - 367**	**Melodic**
		128 - 255	**368 - 511**	**Percussion**
User (in RMF File)	**2**	**0 - 127**	**512 - 639**	**Melodic**
		128 - 255	**640 - 767**	**Percussion**

Accessing Instruments in Mini-BAE

In Mini-BAE, there are two ways to pick a given MIDI channel's Program from these 768 slots: with MIDI messages stored in your music media files, and from your application via calling functions in the Mini-BAE C API.

- To select a Program from your music files (Standard MIDI File or RMF), use a Program Change message to select a melodic instrument from the current bank, using MIDI Program numbers 0–127.

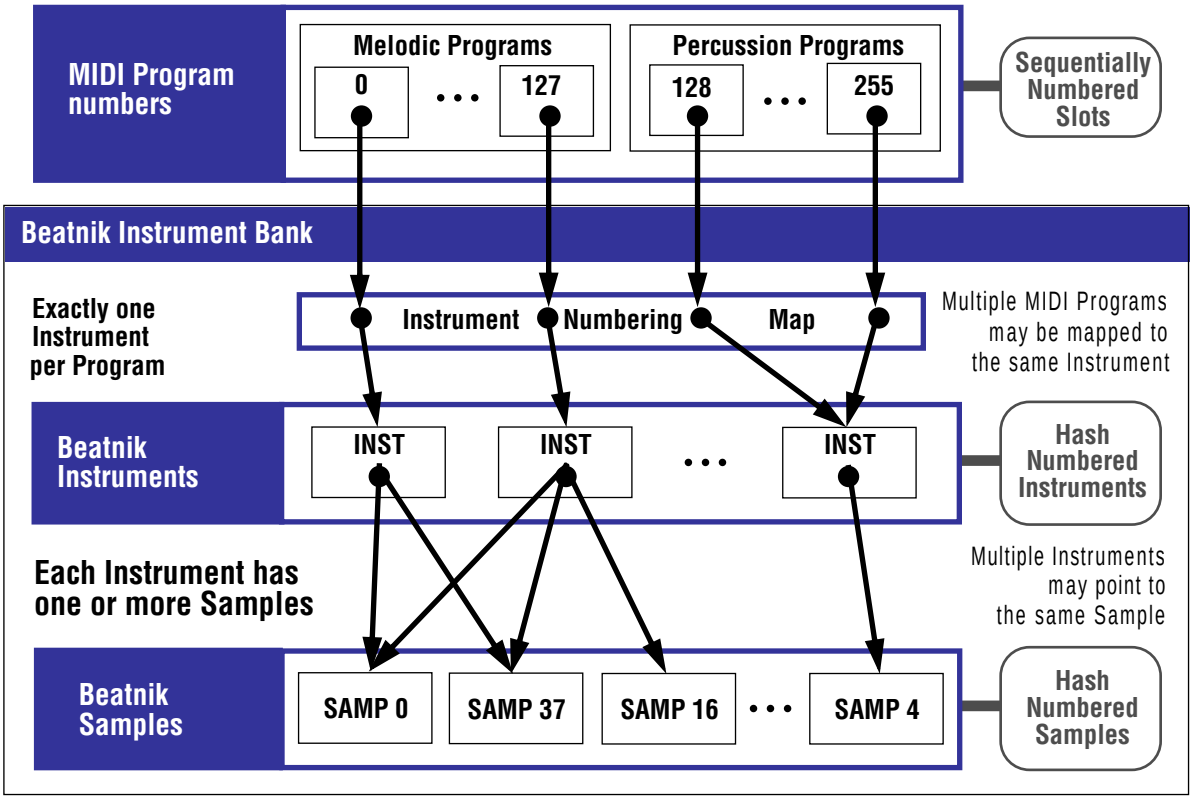
  The default bank is Bank 0, which will be General MIDI when using the Headspace Bank. To access programs from a bank other than Bank 0, use a Bank Select message before sending the Program Change message (Continuous Controller 0 whose value is the desired bank number).

- To access a Program from the Mini-BAE API, either call **BAESong_ProgramChange()** with an API Program number (0–767), or call **BAESong_ProgramBankChange()** with a Bank Select number (0–2) and a MIDI Program number (0–127).

Note that Percussion programs cannot be selected with MIDI Program Change events or the C API functions – only by playing notes on MIDI channel 10.

Program, Instrument, and Sample Numbering

The following diagram attempts to illustrate the relationships between these three number spaces, for a given bank:



MIDI Program numbers – MIDI provides a number space of 128 slots per Bank, numbered from 0 to 127 (although some MIDI instruments call them 1-128).

Beatnik Instrument numbering – These are the instrument definition data blocks that the MIDI Program Change events access – each Instrument definition contains playback parameters for pitch, envelopes, etc., and one or more links to Beatnik Samples. You map MIDI Program numbers to Beatnik Instruments using the Beatnik Editor (in addition to being the instrument editor, its also the Bank assembly tool), and a single Beatnik Instrument can be mapped to more than one MIDI Program numbers if desired. Internally, Beatnik Instruments have their own resource numbers which are assigned by the Beatnik Editor in creation order (and so usually appear fairly random). These Beatnik Instrument resource numbers bear no meaningful relationship to the MIDI Program numbers that call them up. (Your application can also load and otherwise manage separate Instruments.)

Beatnik Sample numbering – These are the digital audio data blocks that serve as waveform sources for the Beatnik Instruments. Beatnik Sample numbers constitute yet a third separate number space, and again the numbers are assigned by the Beatnik Editor (upon inspection they'll appear random because they're hashed). You can use the same Beatnik sample in as many Beatnik Instrument definitions as you like.

Moving Ahead

You should now have a general understanding of what Mini-BAE does, and how it's organized internally. Next, let's take a detailed look at how to program with it.

Programming with Mini-BAE

This chapter presents an overview of how you program with the Mini-BAE library, both at a high level and in practical detail. A separate **Functions Reference** (see page 36) details every individual function.

Quick Start

Mini-BAE is a C library with an object-oriented design.

You may have the source code or not, depending on the terms of your company's Mini-BAE license. If you have the source code, you can build the library with various compile switches, and/or modify the code yourself. The codebase is in ANSI C for maximum portability.

To use Mini-BAE in your application, you'll include the header file `MiniBAE.h`, link with the Mini-BAE library, and then in your program create and manipulate three pseudo-classes of objects (BAEMixer_, BAESound_, and BAESong_) via Mini-BAE functions. These 'pseudo-classes' are implemented via `struct`s.

The programming basically breaks down into three phases:

- **I. Starting Mini-BAE**
- **II. Managing Media** (audio, MIDI, and RMF data) **and Controlling Playback**
- **III. Shutting Down**

> **Note:** What follows is a merely a quick rundown of the most basic Mini-BAE functions. More details are provided a little later on in this chapter, and again, a separate **Functions Reference** (see page 36) details every individual function.

I. Starting Mini-BAE

The runtime core of the Mini-BAE system – that is, the music synthesizer, the audio mixer, and all central services – is represented by the BAEMixer_ pseudo-class. Before attempting any other Mini-BAE operations, you must always begin by creating exactly one BAEMixer_ object and initializing it, in three steps:

1. Call **BAEMixer_New()** to allocate a BAEMixer_.
2. Call **BAEMixer_Open()** to initialize the BAEMixer_.

   BAEMixer_Open has several parameters for setting Mini-BAE's initial operating modes. When you open the mixer, keep these points in mind:

   - **BAEMixer_Open()** mode parameters include the system sample rate (8, 11, 16, 22, 24, 32, 40, 44, or 48 kHz), system sample bit depth (8 or 16 bit), and system sample interpolation mode (none, linear, or 3-point). You can query and change any of these modes later, using **BAEMixer_...()** functions.
   - If you want to hear any RMF or MIDI music play (BAESong_ objects), then don't set the **maxSongVoices** parameter to **0** – that would prevent the Mini-BAE MIDI Synthesizer from allocating any voices.
   - Likewise, if you want to hear any digital audio play (BAESound_ objects), then don't set the **maxSoundVoices** parameter to **0**.
3. If you intend to use the Mini-BAE MIDI Synthesizer – that is, to play MIDI or RMF files or to execute direct MIDI messages – call either **BAEMixer_AddBankFromFile()** or **BAEMixer_AddBankFromMemory()** to provide the synthesizer with a default set of instrument definitions to use.

   Attempting to load any MIDI or RMF music without first setting the instrument bank will fail and produce the error code **BAE_BAD_INSTRUMENT**[1]. You can query and change instrument banks later, using **BAEMixer_...()** functions. In most cases, once your first MIDI sound has been played, all of the instruments it requires will have been copied into memory, and the original can be disposed.

After you take these steps, Mini-BAE will be ready to begin handling music and sound media, in preparation for playback.

II. Managing Media and Controlling Playback

Two other Mini-BAE pseudo-classes represent individual pieces of playable music and sound media: BAESound_ is used to hold digital audio media, and BAESong_ is for MIDI and RMF music. These pseudo-classes furnish functions for loading and purging media data, setting and getting playback properties, and controlling playback. Here's how to use them.

To play a MIDI file:

1. Call **BAESong_New()** to create a BAESong_ object and associate it with the current BAEMixer_.
2. Call **BAESong_LoadMidiFromMemory()** or **BAESong_LoadMidiFromFile()** to associate the BAESong_

1. For situations in which you want some music to play even if some of the instruments aren't available, the **BAESong_Load...()** function has an **ignoreBadInstruments** parameter.

with the MIDI file data you want to play.

If you want to override any of the properties stored in the MIDI file, this is the time to do it.

3. Call **BAESong_Preroll()** to prepare the song for playback.
4. Call **BAESong_Start()** to commence playback.

To play an RMF file:

This is identical to the sequence for MIDI files, except that the Load function you call is tailored for the RMF file format.

1. Call **BAESong_New()** to create a BAESong_ object and associate it with the current BAEMixer_.
2. Call **BAESong_LoadRmfFromMemory()** or **BAESong_LoadRmfFromFile()** to associate the BAESong_ with the RMF file you want it to play.

   If you want to override any of the properties stored in the RMF file, such as volume, this is the time to do it.
3. Call **BAESong_Preroll()** to prepare the song for playback.
4. Call **BAESong_Start()** to commence playback.

When working with MIDI and RMF files, keep these points in mind:

- While you can create, load, and manage any number of BAESong_objects, a maximum of 2 BAESong_ objects can play at the same moment. **BAESong_Start()** will return an error if you try to play more than that.
- Whenever the total number of simultaneous voices needed to service all your currently playing BAESong_ objects exceeds the value of the BAEMixer_'s current **maxMidiVoices**[1] property, "voice stealing" will kick in – degrading the music by causing notes to cut off prematurely.
- While a BAESong_ is playing, your application can also send the synthesizer MIDI messages in real-time – to modify the composition or arrangement by adding musical lines, overriding mutes, and so forth.

To send MIDI messages to the MIDI Synthesizer directly:

1. Call **BAESong_New()** to create a BAESong_ object and associate it with the current BAEMixer_
2. If you know what instruments you want to use for your direct notes, call **BAESong_LoadInstrument()** once per instrument, to pre-load them into the synthesizer's instrument cache.
3. At this point, you can begin calling the several MIDI Event functions that the BAESong_ class furnishes – such as **BAESong_NoteOn()**, **BAESong_NoteOff()**, and **BAESong_ProgramChange()**. If you plan to play any notes, begin with a Program Change to select an instrument.

   If you're not certain that the instrument you've requested with your most recent **BAESong_ProgramChange()** has been loaded, use **BAESong_NoteOnWithLoad()** instead of **BAESong_NoteOn()**.

---

1. Each simultaneously sounding note consumes one voice; by default Mini-BAE is compiled for 8 voices.

When working with direct MIDI messages, remember that every MIDI Event function includes a `time` parameter, expressed in microseconds, that determines when the Mini-BAE MIDI Synthesizer will handle the event. If `time` is `0`, or is in the past, the event will be interpreted immediately; if `time` is in the future, the event will be scheduled for interpretation when that time arrives. To help you determine how best to schedule your MIDI events, a range of **BAESong_Get…()** functions is provided – such as **BAESong_GetMicrosecondPosition()** and **BAESong_GetTempo()**.

To play a digital audio file:

1. Call **BAESong_New()** to create a BAESound_ object and associate it with the current BAEMixer_.
2. Call **BAESong_LoadMemorySample()** or **BAESong_LoadFileSample()** to associate the BAESound_ with the digital audio file you want it to play. You'll need to specify the file's format as WAV, AIFF, or AU.
3. Call **BAESong_Start()** to commence playback.

When working with digital audio files, keep these points in mind:

- If you have raw audio sample data rather than one of the standard digital audio file formats, you can use **BAESong_LoadCustomSample()**.
- Unloading a BAESound_ object while it's playing stops the sound immediately.
- If the number of simultaneously playing BAESound_ objects exceeds the BAEMixer_'s current `maxSoundVoices`, Mini-BAE will return the error code `BAE_NO_FREE_VOICES`, and some of the BAESound_s won't be heard.

III. Shutting Down

When you're finished playing music and sound, you must shut down Mini-BAE. This has two parts:

1. Get rid of all of your BAESound_, and BAESong_ objects by calling **BAESong_Delete()** and **BAESong_Delete()**. This will also unload any music and sound media data still in memory.
2. Deactivate and remove your BAEMixer_ object by calling BAEMixer_**Delete**.

Development System Requirements

There are no special development system requirements for building or running Mini-BAE – any machine that you normally use to build your code should be fine. Mini-BAE's simple audio-out requirements mean your development and test systems won't need to be equipped with advanced or expensive sound cards – anything with an audio output should work.

Adding Mini-BAE to Your Project

Depending on your particular licensing arrangement, Mini-BAE may arrive either as the full ANSI C source code base, or as a compiled C library and its associated header file (`MiniBAE.h`). For details on the mechanics of how to work with your particular Mini-BAE package, please refer to the file `MiniBaeBuildNotes.txt`.

Function Return Codes

Every Mini-BAE function returns a `BAEResult` code. All returned function parameters are passed via pointers. The return codes are listed in `MiniBAE.h` and in the Appendix **Return Codes** (page 118).

Introduction to the Mini-BAE Pseudo-Classes

About Pseudo-Classes

For modularity, and to approximate the organization of the full BAE's C++ API, the Mini-BAE C API has an object-oriented design. Because the C programming language does not directly support objects, we've used 'pseudo-classes' to represent the three major runtime objects:

- BAEMixer_:The Mini-BAE system core
- BAESound_: Digital audio files (WAV, AIFF, and AU)
- BAESong_: Music files (MIDI and RMF)

Where possible, the same function names are used for similar purposes in BAESound_ and BAESong_.

To work around C's lack of data members, we use a `struct` for the data storage for each object instance. A **New** function for each pseudo-class allocates the `struct`.

To work around C's lack of member functions, we use function naming and an instance parameter. That is, all function names in the Mini-BAE API start with either BAEMixer_, BAESound_, or BAESong_, indicating the pseudo-class to which each function belongs. And the first parameter in every Mini-BAE function is a pointer to the desired pseudo-object.

So, rather than calling member functions in the conventional object-oriented way, like this:

```
myWaveFile.setRate( 22050.0 );
```

you tie member functions to object instances like this:

```
BAESound_SetRate( &myWaveFile, 22050.0 );
```

Note: a collection of general utility functions is also provided, with names beginning "BAEUtil_" – however, BAEUtil_ is just a group of functions, not a pseudo-object, and no **New()** function is needed to use them functions.

BAEMixer_ Functions

Here's a quick rundown of what the BAEMixer_ can do, broken down by functional area. For full details on specific functions, consult the **Functions Reference** (see page 36).

Existential Functions

New – Open – IsOpen – GetMixerVersion – Close – Delete

Before Mini-BAE will operate, you have to create exactly one **New()** BAEMixer_ instance and **Open()** it, setting several configuration parameters in the call. At that point you can begin to use all the rest of the Mini-BAE C API functions. When it's time to turn Mini-BAE off, call **BAEMixer_Close()** once. (See **Quick Start** for details.)

Audio Output Management

SetCurrentDevice – GetCurrentDevice – GetMaxDeviceCount – GetDeviceName
Mini-BAE can manage multiple, named audio output devices, and drive any one of them at a time.

DisengageAudio – ReengageAudio – IsAudioEngaged
If you need to temporarily suspend Mini-BAE's audio output to the host's sound system, call **BAEMixer_DisengageAudio()**. While disengaged Mini-BAE will remain mute, but will still update and process normally. When ready to resume sound output, call **BAEMixer_ReengageAudio()**.

Timing & Instrumentation

GetTick – GetAudioSampleFrame
Mini-BAE maintains a constantly incrementing microsecond timebase counter that you can query with **BAEMixer_GetTicks()**. There's also a sample frame counter, which increments each time Mini-BAE emits one final output audio sample, irrespective of sample rate. These functions can be useful for sound/picture synchronization, and for determining how to time-tagged MIDI events for execution at specific future times.

SetAudioLatency – GetAudioLatency
We define 'AudioLatency' as the number of microseconds that elapse between a **BAEMixer_Start()** call and the first appearance of audio at the host device's audio output. You may be able to use this information to improve picture/sound synchronization, or to back-time gameplay events relative to sound events.

GetCPULoadInMicroseconds – GetCPULoadInPercent
Because Mini-BAE is processor-intensive, we provide these instrumentation functions to allow your application to dynamically monitor Mini-BAE's CPU load. If you experience excessive loads, see **Managing Mini-BAE Performance** (page 26) for tips on corrective steps.

Instrument Bank Management

Beatnik Instrument Banks are needed for RMF and MIDI music playback. Banks are collections of Instrument data, including the audio samples that the synth plays back at different rates to produce pitched musical notes, and several sound-shaping parameters.

AddBankFromFile – AddBankFromMemory
BringBankToFront – SendBankToBack
UnloadBank – UnloadBanks
These functions let you associate Beatnik Instrument Banks with Mini-BAE's MIDI Synthesizer. A front-to-back ordering scheme allows certain instruments to override others when desired (see page 47).

GetBankVersion – GetGroovoidNameFromBank
These functions return text information from Bank files.

Configuration and Mixing Properties

ChangeAudioModes – ChangeSystemVoices
Although **BAEMixer_Open()** sets the most important Mini-BAE operating modes, you can also use these functions to change them at any time.

GetMidiVoices – GetSoundVoices
GetMixLevel – GetModifiers – GetTerpMode – GetQuality
Is8BitSuported – Is16BitSupported
GetRealtimeStatus – IsAudioActive
These functions return current BAEMixer_ configuration parameters and status properties, and allow you to determine when Mini-BAE is generating pure silence.

SetMasterSoundEffectsVolume – GetMasterSoundEffectsVolume
SetMasterVolume – GetMasterVolume
SetHardwareVolume – GetHardwareVolume
Mini-BAE offers several volume controls, many of which act in series (see **Mini-BAE's Gain Structure**, page 9).

BAESound_ Functions

Here's a quick rundown of what BAESound_ can do, broken down by functional area.

Existential Functions

New – Delete
SetMixer – GetMixer
You have to create a **New()** BAESound_ instance and associate it with the BAEMixer_ before you can load any digital audio file data. When you're done with the sound, **Delete()** it. You can also query or change the mixer association at any time.

Media Management

LoadMemorySample – LoadFileSample – LoadCustomSample
Unload
These functions control the loading, setup, and disposal of the actual sound media data.

Playback Control

Start – Stop – Pause – Resume
IsPaused – IsDone – GetInfo
You can control the sound's playback with simple 'transport' controls, like the buttons on a CD player., and query its playback state.

SetVolume – GetVolume – Fade
You can play the song back louder or quieter than it was recorded, and you can make the volume fade in or out over a period of time.

SetRate – GetRate
You can control and query the BAESound_'s playback sample rate.

SetLowpassAmountFilter – GetLowpassAmountFilter
SetResonanceAmountFilter – GetResonanceAmountFilter
SetFrequencyAmountFilter – GetFrequencyAmountFilter
You can set and get the parameters of the lowpass filter used for every BAESound_.

SetSampleLoopPoints – GetSampleLoopPoints
You can adjust and query the start and end points of the looping section of the sample.

BAESong_ Functions

Here's a quick rundown of what BAESong_ can do, broken down by functional area.

Existential Functions

New – Delete
SetMixer – GetMixer
You have to create a **New()** BAESong_ instance and associate it with the BAEMixer_ before you can load any MIDI or RMF file data. When you're done with the sound, **Delete()** it. You can also query or change the mixer association at any time.

Media Management

These functions control the loading, setup, and disposal of MIDI and RMF media data.

LoadMidiFromMemory – LoadMidiFromFile
LoadRmfFromMemory – LoadRmfFromFile
LoadGroovoid
You can load any MIDI file or RMF file from disk or memory, or load a Groovoid song from the current Banks, in preparation for playback.

GetMicrosecondLength
You can get the playing time of a loaded song.

Instrument Management

LoadInstrument – UnloadInstrument – IsInstrumentLoaded
It takes instruments to render musical notes. These functions allow you to manage instrument loading on an individual basis. Because loading a MIDI or RMF file causes its required instruments to be loaded, this level of instrument management is usually needed only when playing notes directly via the MIDI message functions (see below).

Playback Control

Preroll – Start – Stop – Pause – Resume
SetMicrosecondPosition – GetMicrosecondPosition
IsPaused – IsDone
You can control the song's playback with simple 'transport' controls, like the buttons on a CD player. You can also query the song's playback position and state, and move the playback position anywhere you like.

SetVolume – GetVolume – Fade
You can play the song back louder or quieter than it was recorded, and you can make the volume fade in or out over a period of time.

SetMasterTempo – GetMasterTempo
You can play the song back faster or slower than it was recorded.

SetTranspose – GetTranspose – AllowChannelTranspose – DoesChannelAllowTranspose
You can play any MIDI channel back at a higher or lower pitch than was recorded.

SetLoops – GetLoops
You can set the song to automatically repeat any number of times – or forever.

MuteChannel – UnmuteChannel – GetChannelMuteStatus
SoloChannel – UnSoloChannel – GetChannelSoloStatus
You can mute or solo any combination of the 16 channels in the MIDI synthesizer.

MuteTrack – UnmuteTrack – GetTrackMuteStatus
SoloTrack – UnSoloTrack – GetTrackSoloStatus
You can mute or solo any combination of the up to 64 tracks in a MIDI or RMF file.

MIDI Events

NoteOn – NoteOnWithLoad – NoteOff – AllNotesOff
You can play MIDI notes on any channel directly via function calls, as opposed to playing back note sequences stored in a MIDI or RMF file.

ProgramChange – ProgramBankChange – GetProgramBank
You set the instrument for each MIDI channel, and query what instrument each channel is using.

PitchBend – GetPitchBend
ControlChange – GetControlValue
KeyPressure – ChannelPressure
You can set and query several MIDI controllers in real time, to add expression to your musical performance.

ParseMidiData – AreMidiEventsPending
Utility functions let you send any arbitrary short MIDI message, and determine whether the MIDI synthesizer has finished its work.

BAEUtil_ Functions

Unlike the other function groups, the BAEUtil_ functions don't simulate a class, and so need no **New()** function. Instead, they're a set of general utility functions for dealing with your BAEMixer_, BAESong_s, and BAESound_s.

Instrument Management Functions

TranslateBankProgramToInstrument
Returns the Beatnik instrument number being used for a given MIDI bank and program

number.

RMF Metadata Functions

BAEUtil_GetRmfSongInfo – BAEUtil_GetInfoSize
BAEUtil_IsRmfSongEncrypted – BAEUtil_IsRmfSongCompressed
BAEUtil_GetRmfVersion

These functions return metadata about RMF files and the songs stored in them.

Moving Ahead

This concludes our introduction to Mini-BAE's facilities. Now let's look at some practical implementation details.

Issues

Memory, Performance, Limitations

This chapter examines a few Mini-BAE issues, and offers a few tips that developers should be aware of.

Memory Management

A function return code of `BAE_MEMORY_ERR` means the requested operation failed because the required memory couldn't be allocated. If you're getting this error, here's some information to help you consider Mini-BAE memory usage strategies.

Mini-BAE Minimum Memory Requirements

When Mini-BAE is active but idle, it uses anywhere from 20 kBytes to 150 kBytes of RAM, depending on the feature `#define` settings in effect at compile time. Playing any media will dynamically increase the memory requirements, as described in the following section.

Additional Memory Requirements for Media Objects

In addition to the above, each media object to be played by Mini-BAE requires the indicated amount of memory:

Class	Size in bytes
BAESound_	60 bytes + size of the digital audio sample
BAESong_	10172 bytes + size of the MIDI data + 1060 bytes per instrument used + sum of the sizes of all samples used by those instruments

Managing Mini-BAE Performance

Mini-BAE is a flexible, software-only system with few hard boundaries on its functionality. As a result, it's possible to task Mini-BAE heavily enough to degrade your overall system performance. For example, if you feed Mini-BAE too much data or set its modes injudiciously, it may monopolize the CPU and crowd out the rest of your application. Mini-BAE CPU usage should therefore be planned, and in some cases monitored. This section explains how Mini-BAE CPU overuse can arise, how to detect it at run-time, and how to correct it.

How Mini-BAE Spends Its Cycles

Primarily, Mini-BAE performs a significant amount of DSP to develop every audio sample it emits. This DSP involves music synthesis, pitch shifting, and audio mixing. The CPU load for these operations is directly proportional to the number of voices Mini-BAE is actively processing at any given moment.

Estimating and Measuring CPU Load

You can use this table to estimate Mini-BAE CPU usage for a given processor type, processor speed, Mini-BAE system sampling rate, and number of simultaneously playing audio voices:

Processor Type	% of CPU bandwidth per Voice	Typical Load % (16 voices MIDI/ 2 voices digital audio)
P90	1.00	18
603e	0.9	16
MMX 200	0.35	6
604e	0.35	6

Reducing CPU Load

If you determine that your Mini-BAE CPU usage is excessive, there are a number of corrective steps that could be taken. These steps fall into two categories:

1. **Programmer Steps –** These are steps you as programmer can take on your own.
2. **Media Changes –** These steps touch on the music or sound content and design, and so may involve your music and/or sound artists, product designers, art directors, and so forth.

These two categories are detailed below.

1. Programmer Steps

All options in this group involve adjusting Mini-BAE global system operating modes.

– Reduce System Sample Rate

Reducing the Mini-BAE system sample rate reduces CPU usage proportionally. Typically this involves a drop from 44.1 kHz to 22.05 kHz, cutting CPU usage rate literally in half. Mini-BAE can run at 8, 11, 16, 22, 24, 32, 40, 44, or 48 kHz, and automatically compensates for changes in the sampling rate at runtime – so you don't have to worry that changing

the system sample rate will change the pitch of the audio media being played. Also, feel free to change the sample rate at runtime if you wish – any audio glitches that may result will be minor.

Note however that because higher sampling rates generally sound better than lower sampling rates, reducing the Mini-BAE system sample rate may[1] reduce audio quality. For best fidelity, we recommend staying at or above 22.05 kHz.

If you do choose to reduce your sample rate, note that you may be able to recover some RAM and disk space by then reducing the sample rate of your sound media (digital audio files and custom instrument samples in RMF files) because there is no fidelity advantage to using sound data with a sample rate higher than the system that will be playing it[2]. For example, if your system sample rate is reduced to 22.05 kHz but your music is in 44.1 kHz digital audio files, you can convert the music to 22.05 kHz files which are only half as big.

To set the sampling rate, use the function **BAEMixer_Open()** (see page 39), or **BAEMixer_ChangeAudioModes()** (see page 51).

– Choose a Less Expensive Pitch Transposition Mode

When Mini-BAE plays a piece of sound at anything other than its original sample rate, it uses DSP code to transpose the pitch on a per-sample basis. Three different interpolation techniques are available for this, ranging from fast but crude, to smooth but slow – and you can select which technique the engine uses, as a way of controlling the quality/performance trade-off.

To set the sampling rate, use the function **BAEMixer_Open()** or **BAEMixer_ChangeAudioModes()**. The available interpolation types are:

```
enum {
   BAE_DROP_SAMPLE = 0,
   BAE_2_POINT_INTERPOLATION,
   BAE_LINEAR_INTERPOLATION
} BAETerpMode;
```

BAE_LINEAR_INTERPOLATION is the most processor-intensive option, and sounds best

BAE_2_POINT_INTERPOLATION is less expensive, but still sounds fairly good

BAE_DROP_SAMPLE is much cheaper, but can sound a little crummy in some cases – for example, when large pitch shifts are required. In drop-sample interpolation we simply re-use the last data sample, rather than interpolating a value between the data samples.

See also: **BAEMixer_Open()**.....39 **BAEMixer_ChangeAudioModes()**.....51

– Reduce MIDI Synthesizer Polyphony

In some situations the average CPU usage during playback may be fine, but occasional spikes become problematic. One way to cap such spikes is to reduce the maximum number of voices the MIDI synthesizer is allowed to play at once[3]. To limit synthesizer polyphony,

1. We say ‘may’ because if all of your media was created at a lower sampling rate than you’re starting from, you may not be able to notice the difference in audio quality resulting from a drop to a lower sampling rate.
2. Except for sounds intended to play back at a different pitch, such as instrument bank samples.

use the function **BAEMixer_Open()** (see page 39).

Note: You may wish to check with your music and sound artists before taking this step, to double-check that Mini-BAE will still have enough voices to play the music as it's intended to sound. (Few things sound as jarring as music whose notes cut off too soon.)

2. Media Changes

While the following steps can also help reduce your Mini-BAE CPU load, they all involve adjusting the music and/or sound media, and/or how that media is creatively used in your product. Consequently, you may want to consider consulting with your music/sound artists, product manager, producer, or art director before altering the music and sound files yourself.

– Avoid Overlapping Music

One good way to make a lot of voices active – and thus burn more CPU cycles – is to play music files. A single RMF or MIDI file will typically use several simultaneous voices, with no pre-defined upper limit[1]. Playing multiple music files at the same time will of course increase this voice consumption and CPU usage – so you may want to minimize the number of pieces of music that you play at once.

For example, one common CPU-expensive situation is a crossfade between two pieces of music– that is, while you're fading the volume of one piece of music out in order to avoid an abrupt end, you load and start a second piece of music, and start fading its volume up from silence. The processor bandwidth problem arises because during the crossfade period, although the perceived volume isn't necessarily any louder, you actually have two separate pieces of music running at the same time, typically doubling Mini-BAE's CPU load until the first piece ends.

The same problem applies when you want to layer up more than one piece of music at once.

If Mini-BAE CPU usage becomes a problem in these situations, consider redesigning your segues to avoid overlapping playback. That is, fade the first piece to silence, and then stop it; then start the second piece with the volume at zero, and finally start fading the second piece in. This way, only one music file is executing at a time and the peak CPU load isn't excessive.

– Re-Voice Your Music Files

Another way to reduce Mini-BAE CPU load is to have your music rearranged to use fewer voices. This work should be done by a music person, preferably the same one who arranged it the first time.

Note that Mini-BAE's customizable instrument sound sets can be a help here. For example, it might in some cases be possible to replace a doubled, detuned line being played using 2 voices and a pitchbend with a single line playing a doubled, detuned sample. Or, a string

3. This metric of a musical instrument is referred to as its *polyphony.*

1. Although Standard MIDI files have a limit of 16 MIDI *channels*, the NoteOn/NoteOff model that MIDI uses makes it possible for any number of *voices* to be active at once. And although the number of rendered voices is limited by the BAEMixer_'s **maxMidiVoices** property (which you can set as high as you like), Mini-BAE will otherwise process all of the voices.

arrangement that's scored as many separate parts might be replaced by fewer voices, each playing a sample of a string section performance.

– Play Samples Back at Natural Pitch

Transposing the pitch of a sample at playback time is CPU-intensive. To reduce CPU loads spent on transposition, try to play your percussion instruments at their natural, sampled pitch where possible, rather than transposing to a different note.

With non-percussion instruments this is rarely an issue, because the ordinary use of melodic notes, pitch bend, and vibrato all necessitate some pitch transposition anyway – to realize any CPU benefit from playing at natural pitch, the instrument can't use any of these common features.

Mini-BAE System Limits

Mini-BAE has a few general operational limits to keep in mind:

- There is a limit on the maximum number of RMF or MIDI songs and digital audio sounds that can play at once. This cap is determined at compile time, using a `#define` which is set to 2 by default.
- There is a limit on the maximum number of voices available at a time to play and mix all digital audio samples and MIDI and RMF musical notes. This cap is determined at compile time, using the symbol `BAE_MAX_VOICES` which is set to 8 by default. The maximum possible number is 64.
- In addition, Mini-BAE provides facilities for capping how many of those 64 voices may be used for MIDI synthesizer notes (`maxMidiVoices`), and for digital audio playback (`maxSoundVoices`). You can use these to make sure that neither group of sounds gets squeezed out by the other. See **BAEMixer_Open()** (see page 39), and **BAEMixer_ChangeSystemVoices()** (see page 51).
- When using the interpolation mode `BAE_2_POINT_INTERPOLATION`, BAESound_ samples are each limited to 1 megasample (not 1 megabyte) of data, due to address math issues.
- A BAESound_'s loop can't be less than 100 samples long.
- Only mono and stereo media is supported. Attempting to play any digital audio media containing more than two channels will produce an error code rather than playing the data.

Tip: Queue Media Before It's Needed

Sometimes fast start-up of a piece of music or sound is crucial – for example, a mouse click sound, or a musical instrument control surface. A good way to minimize start-up latencies for a piece of music or sound media is to queue the media before it's needed, rather than waiting until it's needed to begin the loading process. By "Queuing" we mean to create the objects and load them with data from the music or sound files *before* you need them to play. This way, they'll be "armed" and ready to start immediately whenever you need them– no waiting for a disk load or data copy, etc.

Queuing is particularly recommended for situations in which you'll be assembling continuous

soundtracks at runtime by stringing together smaller pieces of media. The quick start-up time for a queued media object makes it much easier to maintain a continuous musical beat (in the case of music) or avoid silences (in the case of sound effects) at the seams between the pieces.

Moving Ahead

In case you experience any difficulties coming up to speed with Mini-BAE, the next chapter is our guide to troubleshooting, in the form of a FAQ.

Troubleshooting / FAQ

Frequently Asked Questions

Here, to save your time and our time, are answers to the questions we're most often asked about installing and using Mini-BAE. If these your question isn't addressed here, please ask your designated Beatnik contact person.

What file formats does Mini-BAE play?

Standard MIDI File, WAV, AIFF, Sun AU, and RMF (Rich Music Format.)

What is RMF?

RMF (Rich Music Format) is a hybrid file type that encapsulates MIDI and audio samples along with some interactive performance settings, plus encrypted copyright data.

How do you make an RMF file?

With the Beatnik Editor.

Is RMF an open standard?

Not at this time.

What copyright protection does RMF offer?

You can encode 40-bit encrypted copyright and licensing information into a file, and it can be easily displayed anywhere an RMF file plays. It will help a publisher or composer keep track of who is using their music.

Will Beatnik support DLS (the proposed MIDI Manufacturers Association standard for downloadable samples)?

We do intend to support DLS in future, although we can't yet announce any specific date.

What's MIDI, and what advantages does Beatnik's software MIDI synthesizer have?

Mini-BAE leverages the economy of MIDI by playing Standard MIDI files (SMF) and furnishing a software synthesizer with optional General MIDI (GM) instrument set to play them. But Mini-BAE goes further than plain MIDI and GM – through the use of Beatnik's RMF file format, you can extend MIDI's power with your own cus-

tom instruments and samples. And whereas MIDI has unpredictable results when played on different systems, Beatnik guarantees you the same high-fidelity playback on all supported platforms.

What does the Beatnik Editor do?

It manages Mini-BAE sound banks and imports user samples; it edits instruments settings such as LFOs for vibrato, ADSR for volume shaping, and filters to adjust the tone; it imports and manages Standard MIDI Files; it allows the addition of tamper-proof Copyright and Licensing info; and you can save out your work as a Beatnik Session (for further editing) or as an RMF file with several compression and playback options.

In what sense is Mini-BAE 'interactive'?

Any piece of music can respond to user input and change its tempo, key, instrumentation, mix, or song structure. Multiple themes can be layered. Individual notes, events and sound effects can be triggered from user input. Your application can be written to randomize or step through multiple pieces of music, keeping the user experience fresh every time.

How does one create these interactions?

It's largely a function of good authoring. There are some shortcuts, which you can learn through our documentation. Beatnik also supplies example source code which you can borrow. Basically, your application can issue Mini-BAE commands and respond to callbacks, and Mini-BAE has a rich set of musical functions that can respond to your application.

Who creates interactions, the composer or the programmer?

Either. Ideally, the musician creates not only linear music files, but also a range of alternatives, variations, and interchangeable short themes. He or she supplies these to the programmer, who then writes code to integrate them into the application. Please note, however, that with virtually no programming at all, you can simply embed a music file in a game and use it as a great-sounding background track.

Does Mini-BAE do algorithmic variations, like Koan? Does it make stylistic choices like auto-accompaniment keyboards?

No. With Mini-BAE, the composer is always responsible for the musical form.

Why doesn't my song play?

Check your MIDI data: Make sure that the very first command in every MIDI track is a program change.

How come one of my instruments won't play, even though the rest will?

- Is the volume scaled to zero?
- Does the instrument point to the right samples?

- Check your MIDI data: Did you mute that instrument's channel in the sequencer?

My instrument's playing the right set of notes, but in the wrong key. Why?

The sample is probably set to the wrong root key. For instance, if you had a sample of a piano at middle C, but the root key were set to the F above Middle C, the driver will treat the sample as the F. You can set the root key in the Beatnik Editor.

Mini-BAE's using too much CPU time. What can I do to reduce it?

A number of things. Try:

- Running the tune on a more powerful machine
- Reducing the playback rate, i.e. from 22kHz to 11kHz
- Turning off interpolation, or choosing a cheaper interpolation technique
- Reducing the number of voices in the MIDI data

This last is the most drastic solution, since it usually entails a rearrangement of the music; you can temporarily set `MaxNotes` lower, but obviously this runs the risk that some number of notes will get cut off or not played if your music needs more voices than you've allowed.

For more information consult the **Mini-BAE Issues** chapter (see **Reducing CPU Load**, page 26).

What's up with the volume? It keeps changing, inappropriately.

Make sure your `mixLevel` is set appropriately when you first Open the mixer.

Why are notes are not being held long enough (i.e. they're getting cut short)?

Check your mixer: Is the `mixLevel` set up correctly, per the previous answer?

And check the envelopes on your custom instruments: Do they cut off too soon?

Why do Percussion notes keep getting cut off?

If the instrument in question uses the Beatnik Editor's default natural envelope ('no envelope'), then check your MIDI file for short notes.

Remember that a MIDI note is treated not as a trigger occurring at a single point in time, but as an event with a duration. If your cymbal sample is 1 second long, and a MIDI note associated with that sample has a duration of 1/10 of a second, the cymbal sample will only be played for the duration of the MIDI event, 1/10 of a second, cutting off the end of the sample.

My percussion instrument only has one MIDI event, so why am I hearing several out-of-sync attacks?

Using the Beatnik Editor, make sure looping is turned off for this instrument.

Weird things are happening with my instruments, and they seem to be related to the MIDI channel numbers. What's going on?

Check your MIDI file (or the MIDI info being streamed to the MIDI Synthesizer) for missing Program Changes; each track or channel should ordinarily start with one.

When playing a Standard MIDI File, Mini-BAE will by default use instrument number (MIDI channel number - 1) for each channel; for example, Channel 1 gets played with instrument 0, Channel 2 gets played with instrument 1, and so forth. Ordinarily you want to override this.

How do I make a channel of the Mini-BAE MIDI Synthesizer play in Mono mode instead of Poly?

Sorry, you can't. The Mini-BAE MIDI Synthesizer's channels are always in Poly mode.

Mini-BAE C API Functions Reference

BAEMixer_
Functions Reference

Alphabetical Index

Subject Index

Existential Functions

See also: **The Audio Mixer** ..... 8 **Existential Functions** overview 19

BAEMixer_New()

```
BAEMixer BAEMixer_New( void );
```

'Create' a new `BAEMixer` structure. Actually returns a pointer to the global single `BAEMixer` structure. Can only be one mixer object per sound card.

Returns: (always succeeds)

Note: The `BAEMixer` must be initialized via **BAEMixer_Open()** before it can be used.

See also:

BAEMixer_Open()

```
BAEResult BAEMixer_Open(
        BAEMixer          mixer,
        BAEQuality        q,
        BAETerpMode       t,
        BAEAudioModifiers am,
        short int         maxMidiVoices,
        short int         maxSoundVoices,
        short int         mixLevel,
        BAE_BOOL          engageAudio );
```

Initializes the indicated BAEMixer_ in preparation for all sound generation, and sets several operating modes. You must call **BAEMixer_New()** and **BAEMixer_Open()** before calling any other Mini-BAE functions.

Params:	`mixer`	The BAEMixer_
	`q`	Quality mode (combination of sample rate and interpolation mode; see `BAEQuality`)
	`t`	Interpolation mode (see `BAETerpMode`)
	`am`	Miscellaneous modes (see `BAEAudioModifiers`)
	`maxSongVoices`	Maximum number of rendered notes playing at once.
	`maxSoundVoices`	Maximum number of Sound objects playing at once
	`mixLevel`	Total number of full-scale voices before distortion (Song notes plus Sound objects)
	`engageAudio`	Whether to send mixer audio output to the host device
Returns:	`BAE_NULL_OBJECT`	Null `mixer` object pointer
	`BAE_PARAM_ERR`	Bad parameters
	`BAE_NOT_REENTERANT`	Attempt to re-enter Mini-BAE
	`BAE_GENERAL_BAD`	Header file and built code versions don't match
Note:	If the platform is not capable of providing the requested level of service, Mini-BAE will fall back to a lower level during the **BAEMixer_Open()** call.	
See also:		

BAEMixer_IsOpen()

```
BAEResult BAEMixer_IsOpen(
          BAEMixer  mixer,
          BAE_BOOL *outIsOpen );
```

Upon return, parameter **outIsOpen** will point to a **BAE_BOOL** indicating whether the indicated BAEMixer_ is currently open (**TRUE**) or closed (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_GetMixerVersion()

```
BAEResult BAEMixer_GetMixerVersion(
          BAEMixer   mixer,
          short int *outVersionMajor,
          short int *outVersionMinor,
          short int *outVersionSubMinor );
```

Upon return, parameters **outVersionMajor**, **outVersionMinor**, and **outVersionSubMinor** will point to **short int**s indicating the Mini-BAE version number for the indicated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_Close()

```
BAEResult BAEMixer_Close( BAEMixer mixer );
```

Causes the indicated BAEMixer_ to stop functioning, delete its data, and free its memory.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
Note:	In Mini-BAE split implementations, **BAEMixer_Close()** frees DSP-side memory.	
See also:		

BAEMixer_Delete()

```
BAEResult BAEMixer_Delete( BAEMixer mixer );
```

Closes and deactivates the indicated BAEMixer_, effectively deleting it.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
Note:	In Mini-BAE split implementations, **BAEMixer_Delete()** frees MCU-side memory.	
See also:		

Audio Output Management

See also: **Audio Output Management** overview...20

BAEMixer_SetCurrentDevice()

```
BAEResult BAEMixer_SetCurrentDevice(
          BAEMixer mixer,
          long     deviceID,
          void     *deviceParameter );
```

Causes the indicated BAEMixer_ to begin sending its audio output to the indicated device, with any optional device-specific parameters as pointed to by **deviceParameter**. On platforms not supporting multiple devices, this call has no effect.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_GetCurrentDevice()

```
BAEResult BAEMixer_GetCurrentDevice(
          BAEMixer mixer,
          void     *deviceParameter,
          long     *outDeviceID );
```

Upon return, parameter **outDeviceID** will point to a **long** containing the device ID of the audio output device to which the indicated BAEMixer_ is currently sending its audio output; any optional device-specific parameters being used for that device will be pointed to by **deviceParameter**.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_GetMaxDeviceCount()

```
BAEResult BAEMixer_GetMaxDeviceCount(
          BAEMixer mixer,
          long     *outMaxDeviceCount );
```

Upon return, parameter **outMaxDeviceCount** will point to a long indicating the maximum number of audio output devices to which the indicated BAEMixer_ is able to send its output. On platforms not supporting multiple devices, this will be **0**.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_GetDeviceName()

```
BAEResult BAEMixer_GetDeviceName(
          BAEMixer      mixer,
          long          deviceID,
          char         *cName,
          unsigned long cNameLength );
```

Upon return, parameter **cName** will point to a character string containing the name of the audio output device specified by device ID number **deviceID** for the indicated BAEMixer_. Provide the maximum string length in bytes, including the terminating **NULL**, in **cNameLength**. On platforms not supporting multiple devices, this call has no effect.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_DisengageAudio()

```
BAEResult BAEMixer_DisengageAudio( BAEMixer mixer );
```

Causes the indicated BAEMixer_ to temporarily suspend audio output to the host. This allows for cooperative sharing of the output services with any other sound generating entities. All Mini-BAE processing continues to operate in real time while disengaged. Use **BAEMixer_ReengageAudio()** to resume audio output.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
Note:		
See also:		

BAEMixer_ReengageAudio()

```
BAEResult BAEMixer_ReengageAudio( BAEMixer mixer );
```

Resumes audio output from the indicated BAEMixer_ to the host, following a **BAEMixer_DisengageAudio()**.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
Note:		
See also:	**BAEMixer_DisengageAudio()**	

BAEMixer_IsAudioEngaged()

```
BAEResult BAEMixer_IsAudioEngaged(
          BAEMixer mixer,
          BAE_BOOL  *outIsEngaged );
```

Upon return, parameter **outIsEngaged** will point to a **BAE_BOOL** indicating whether the indicated BAEMixer_ is currently engaged (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_NOT_SETUP	Indicated **mixer** not initialized
Note:		
See also:		

Timing & Instrumentation

See also: **Timing & Instrumentation** overview.....20

BAEMixer_GetTick()

```
BAEResult BAEMixer_GetTick(
          BAEMixer       mixer,
          unsigned long *outTick );
```

Upon return, parameter **outTick** will point to the indicated BAEMixer_'s current time, expressed in microseconds elapsed since initialization.

Returns:		
	BAE_NULL_OBJECT	Null **mixer** object pointer
	BAE_PARAM_ERR	Null mixer parameter

See also:

BAEMixer_GetAudioSampleFrame()

```
BAEResult BAEMixer_GetAudioSampleFrame(
          BAEMixer   mixer,
          short int *pLeft,
          short int *pRight,
          short int *outFrame );
```

Upon return, parameters **pLeft** and **pRight** will point to the indicated BAEMixer's current left and right master audio output sample buffers, and parameter **outFrame** will point at a **short int** containing the BAEMixer_'s current write index (as used in writing to the left and right buffers).

Returns:		
	BAE_NULL_OBJECT	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters

See also:

BAEMixer_SetAudioLatency()

```
BAEResult BAEMixer_SetAudioLatency(
          BAEMixer      mixer,
          unsigned long requestedLatency );
```

Reconfigures the current Mini-BAE output device buffers to achieve the requested audio output latency, if possible. Latency is expressed in integer milliseconds (1000 = 1 second).

Returns:		
	BAE_NULL_OBJECT	Null **mixer** object pointer
	BAE_NOT_SETUP	Function not available on this platform.

Note:

See also:

BAEMixer_GetAudioLatency()

```
BAEResult BAEMixer_GetAudioLatency(
          BAEMixer        mixer,
          unsigned long *outLatency );
```

Upon return, parameter outLatency will point to the current Mini-BAE audio output latency for the indicated BAEMixer_, expressed in milliseconds (1000 = 1 second).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters.
Note:		
See also:		

BAEMixer_GetCPULoadInMicroseconds()

```
BAEResult BAEMixer_GetCPULoadInMicroseconds(
          BAEMixer        mixer,
          unsigned long *outLoad );
```

Upon return, parameter **outLoad** will point to an **unsigned long** containing an estimate of the number of microseconds the indicated BAEMixer_ is taking to generate each audio output buffer.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAEMixer_GetCPULoadInPercent()

```
BAEResult BAEMixer_GetCPULoadInPercent(
          BAEMixer        mixer,
          unsigned long *outLoad );
```

Upon return, parameter **outLoad** will point to an **unsigned long** containing an integer in the range 0-100 reporting what percentage of the available processor time the indicated BAEMixer_ is using.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

Instrument Bank Management

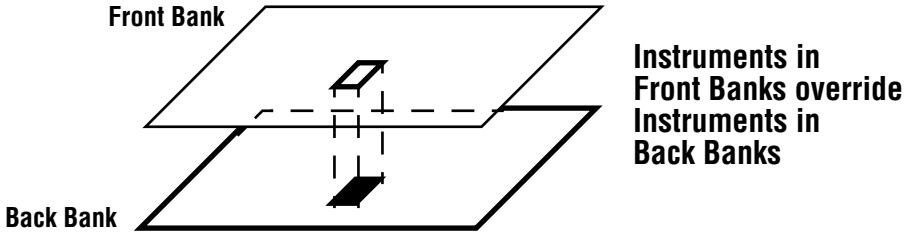
Mini-BAE requires access to Beatnik instrument definition resources in order to render musical notes. Custom instruments (accessed as MIDI Program Bank 2) may travel in RMF files, however General MIDI (MIDI program Bank 0), Beatnik Special (MIDI Program Bank 1), or any other customized sound banks must ordinarily be associated with the BAEMixer_ directly using the following functions. Instrument resources are usually kept in instrument Bank files (or file images) for easy management, and a bank file may contain instruments for any combination of the 768 slots in MIDI Program Banks 0, 1, and 2.

The following Patch Bank Management functions facilitate loading and unloading of these instrument banks from disk or memory.

Bank Search Paths

A single mixer may use more than one bank file at a time. If so, instrument requests are handled using a 'search path' mechanism similar to a graphics depth-ordering scheme, where any instrument in the 'back' bank can be overridden by a like-numbered instrument in a bank that's closer to the 'front'.

For example, if your back bank contains all 768 instruments, and you then add a second bank closer to the front containing one instrument in slot number 7, then all subsequent requests for instrument number 7 will access the instrument 7 from the front bank, not the instrument 7 from the back bank.



The following Patch Bank Management functions also allow you to specify where in this search path each bank will appear.

See also: **Instrument Bank Management** overview20 **Instrument Management**..... 82

BAEMixer_AddBankFromFile()

```
BAEResult BAEMixer_SetBankToFile(
          BAEMixer      mixer,
          BAEPathName   pAudioPathName,
          BAEBankToken *outToken );
```

Causes the indicated BAEMixer_ to load and begin using the instrument bank file at path **pAudioPathName** for note rendering, in addition to any previously added instrument banks. The new bank is placed at the front of the search path. Upon return, parameter **outToken** will point at a reference ID **BAEBankToken** for use in manipulating or getting information about the bank.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_BAD_FILE	Bad file or path spec
Note:		
See also:		

BAEMixer_AddBankFromMemory()

```
BAEResult BAEMixer_SetBankToMemory(
          BAEMixer      mixer,
          void         *pAudioFile,
          unsigned long fileSize,
          BAEBankToken  *outToken );
```

Causes the indicated BAEMixer_ to begin using the instrument bank file image at address **pAudioFile** for note rendering, in addition to any previously added instrument banks. The new bank is placed at the front of the search path. Parameter **fileSize** must indicate the length in bytes of the instrument bank resource. Upon return, parameter **outToken** will point at a reference ID **BAEBankToken** for use in manipulating or getting information about the bank.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_BAD_FILE	Bad instrument bank resource
Note:		
See also:		

BAEMixer_BringBankToFront()

```
BAEResult BAEMixer_BringBankToFront(
          BAEMixer     mixer,
          BAEBankToken bank );
```

Places the indicated bank at the front of the search path (first), so that instruments present in this bank override like-numbered instruments in all other currently used banks.

Returns:	`BAE_NULL_OBJECT`	Null `mixer` object pointer
See also:		

BAEMixer_SendBankToBack()

```
BAEResult BAEMixer_SendBankToBack(
          BAEMixer     mixer,
          BAEBankToken bank );
```

Places the indicated bank at the back of the search path (last), so that any like-numbered instruments present in other currently used banks override instruments in this bank.

Returns:	`BAE_NULL_OBJECT`	Null `mixer` object pointer
See also:		

BAEMixer_UnloadBank()

```
BAEResult BAEMixer_UnloadBank(
          BAEMixer     mixer,
          BAEBankToken bank );
```

Causes the indicated BAEMixer_ to stop using and close the one indicated instrument bank.

Returns:	`BAE_NULL_OBJECT`	Null `mixer` object pointer
Note:	Do not free or dispose of any bank while it's being used by a BAEMixer_; always call **BAEMixer_UnloadBank()** or **BAEMixer_UnloadBanks()** before disposing.	
See also:	**BAEMixer_UnloadBank()**	

BAEMixer_UnloadBanks()

```
BAEResult BAEMixer_UnloadBanks( BAEMixer mixer );
```

Causes the indicated BAEMixer_ to stop using and close all current instrument banks.

Returns:	`BAE_NULL_OBJECT`	Null `mixer` object pointer
Note:	Do not free or dispose of any bank while it's being used by a BAEMixer_ – always call **BAEMixer_UnloadBank()** or **BAEMixer_UnloadBanks()** before disposing.	
See also:	**BAEMixer_UnloadBanks()**	

BAEMixer_GetBankVersion()

```
BAEResult BAEMixer_GetBankVersion(
          BAEMixer     mixer,
          BAEBankToken bank,
          short int    *outVersionMajor,
          short int    *outVersionMinor,
          short int    *outVersionSubMinor );
```

Upon return, parameters **outVersionMajor**, **outVersionMinor**, and **outVersionSubMinor** will point to the version number of the indicated instrument **bank** for the indicated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	This function is named **GetVersionFronAudioFile()** in the full versions of BAE.	
See also:		

BAEMixer_GetGroovoidNameFromBank()

```
BAEResult BAEMixer_GetGroovoidNameFromBank(
          BAEMixer mixer,
          long     index,
          char     *cSongName );
```

Upon return, parameter **cSongName** will point to the name of Groovoid number **index**, as found via the indicated BAEMixer_'s instrument bank search path. That is, if Groovoids with the requested **index** appear in more than one of the instrument banks associated with the BAEMixer_, this function will return the name of the frontmost one.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_NOT_SETUP	The indicated mixer has not been initialized
Note:	Known as **GetSongNameFromAudioFile()** in the full versions of BAE	
See also:		

Configuration and Mixing

See also: **Configuration and Mixing Properties** ...21
BAESound_ Functions overview.....21
Configuration and Mixing Properties...21
Instrument Bank Management.....20

BAEMixer_ChangeAudioModes()

```
BAEResult BAEMixer_ChangeAudioModes(
          BAEMixer          mixer,
          BAEQuality        q,
          BAETerpMode       t,
          BAEAudioModifiers am );
```

Changes the operating modes of the indicated BAEMixer_ to the indicated **BAEQuality**, **BAETerpMode**, and **BAEAudioModifiers**.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Bad parameters.
Note:		
See also:		

BAEMixer_ChangeSystemVoices()

```
BAEResult BAEMixer_ChangeSystemVoices(
          BAEMixer  mixer,
          short int maxMidiVoices,
          short int maxSoundVoices,
          short int mixLevel );
```

Changes the maximum number of note rendering voices (**maxSongVoices**), maximum number of digital audio voices (**maxSoundVoices**), and maximum number of full-scale voices before clipping (**mixLevel**) for the indicated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Bad parameters
Note:		
See also:		

BAEMixer_GetMidiVoices()

```
BAEResult BAEMixer_GetMidiVoices(
          BAEMixer   mixer,
          short int *outNumMidiVoices );
```

Upon return, parameter **outNumMidiVoices** will point to a **short int** containing the indicated BAEMixer_'s current maximum number of voices available for MIDI and RMF note rendering.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_GetSoundVoices()

```
BAEResult BAEMixer_GetSoundVoices(
          BAEMixer   mixer,
          short int *outNumSoundVoices );
```

Upon return, parameter **outNumSoundVoices** will point to a **short int** containing the indicated BAEMixer_'s current maximum number of voices available for BAESound_ objects (samples).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_GetMixLevel()

```
BAEResult BAEMixer_GetMixLevel(
          BAEMixer mixer,
          short int *outMixLevel );
```

Upon return, parameter **outMixLevel** will point to a **short int** containing the indicated BAEMixer_'s current maximum number of simultaneous full-scale voices before distortion (combined BAESound_ and BAESong_ voices).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_GetModifiers()

```
BAEResult BAEMixer_GetModifiers(
          BAEMixer           mixer,
          BAEAudioModifiers *outMods );
```

Upon return, parameter **outMods** will point to a **BAEAudioModifiers struct** containing the indicated BAEMixer_'s current **modifiers** flags, which control various system-wide BAE operating modes. See **struct BAEAudioModifiers** for the flags and their interpretation.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
	BAE_NOT_SETUP	Indicated **mixer** not initialized
Note:		
See also:		

BAEMixer_GetTerpMode()

```
BAEResult BAEMixer_GetTerpMode(
          BAEMixer     mixer,
          BAETerpMode *outTerpMode );
```

Upon return, parameter **outTerpMode** will point to a **BAETerpMode struct** containing the indicated BAEMixer_'s current interpolation mode. See **struct BAETerpMode** for interpretation.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_NOT_SETUP	Indicated **mixer** not initialized
See also:		

BAEMixer_GetQuality()

```
BAEResult BAEMixer_GetQuality(
          BAEMixer    mixer,
          BAEQuality  *outQuality );
```

Upon return, parameter **outQuality** will point to a **BAEQuality struct** containing the indicated BAEMixer_'s current quality mode (combination of sample rate and interpolation mode). See **struct BAEQuality** for interpretation.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
	BAE_NOT_SETUP	Indicated **mixer** not initialized
Note:		
See also:		

BAEMixer_Is8BitSupported()

```
BAEResult BAEMixer_Is8BitSupported(
          BAEMixer  mixer,
          BAE_BOOL *outIsSupported );
```

Upon return, parameter **outIsSupported** will point to a **BAE_BOOL** indicating whether the indicated BAEMixer_ supports 8-bit audio output (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_Is16BitSupported()

```
BAEResult BAEMixer_Is16BitSupported(
          BAEMixer  mixer,
          BAE_BOOL *outIsSupported );
```

Upon return, parameter **outIsSupported** will point to a **BAE_BOOL** indicating whether the indicated BAEMixer_ supports 16-bit audio output (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAEMixer_IsAudioActive()

```
BAEResult BAEMixer_IsAudioActive(
          BAEMixer  mixer,
          BAE_BOOL *outIsActive );
```

Upon return, parameter **outIsActive** will point to a **BAE_BOOL** indicating whether the indicated BAEMixer_ is currently generating audio, as determined by looking for active voices and taking a snapshot of the audio output stream.

Returns:	**BAE_PARAM_ERR**	Null parameter
	BAE_NULL_OBJECT	Null **mixer** object pointer
Note:	Just because songs are 'playing' doesn't necessarily mean the mixer is emitting audio. For example, **BAEMixer_IsAudioActive()** can return **FALSE** during a rest inside a piece of music – a time when **BAESong_IsDone()** would indicate that the song is still playing.	
See also:		

BAEMixer_GetRealtimeStatus()

```
BAEResult BAEMixer_GetRealtimeStatus(
          BAEMixer     mixer,
          BAEAudioInfo *pStatus );
```

Upon return, parameter **pStatus** will point to a **BAEAudioStatus struct** containing the indicated BAEMixer_'s current status variables (see **struct BAEAudioStatus** for fields).

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null mixer parameter
Note:		
See also:		

BAEMixer_SetMasterSoundEffectsVolume()

```
BAEResult BAEMixer_SetMasterSoundEffectsVolume(
          BAEMixer           mixer,
          BAE_UNSIGNED_FIXED theVolume );
```

Sets the shared master volume for all BAESound_ objects played by the indicated BAEMixer_ to the indicated volume.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

- **FLOAT_TO_UNSIGNED_FIXED**
- **UNSIGNED_FIXED_TO_FLOAT**
- **LONG_TO_UNSIGNED_FIXED**
- **UNSIGNED_FIXED_TO_LONG**
- **UNSIGNED_FIXED_TO_LONG_ROUNDED**
- **UNSIGNED_FIXED_TO_SHORT**
- **UNSIGNED_FIXED_TO_SHORT_ROUNDED**

See also:

BAEMixer_GetMasterSoundEffectsVolume()

```
BAEResult BAEMixer_GetMasterSoundEffectsVolume(
        BAEMixer            mixer,
        BAE_UNSIGNED_FIXED *outVolume );
```

Upon return, parameter **outVolume** will point to the shared master volume for all BAESound_ objects played by the indicated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAEMixer_SetMasterVolume()

```
BAEResult BAEMixer_SetMasterVolume(
        BAEMixer            mixer,
        BAE_UNSIGNED_FIXED theVolume );
```

Sets the master volume of the indicated BAEMixer_ to the indicated volume.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null mixer parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAEMixer_GetMasterVolume()

```
BAEResult BAEMixer_GetMasterVolume(
          BAEMixer           mixer,
          BAE_UNSIGNED_FIXED *outVolume );
```

Upon return, parameter **outVolume** will point to the current master volume of the indicated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameters

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAEMixer_SetHardwareVolume()

```
BAEResult BAEMixer_SetHardwareVolume(
          BAEMixer mixer,
          BAE_UNSIGNED_FIXED theVolume );
```

Sets the hardware-based final output volume of the audio output device currently being used by the indicated BAEMixer_ to the indicated volume, if available on this platform.

Returns:	**BAE_NULL_OBJECT**	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAEMixer_GetHardwareVolume()

```
BAEResult BAEMixer_GetHardwareVolume(
          BAEMixer            mixer,
          BAE_UNSIGNED_FIXED *outVolume );
```

Upon return, parameter **outVolume** will point to the hardware-based final output volume of the audio output device currently being used by the indicated BAEMixer_, if available on this platform.

Returns:		
	BAE_NULL_OBJECT	Null **mixer** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

[End of BAEMixer_ Function Reference]

BAESound_
Functions Reference

Alphabetical Index to Functions

Subject Index to Functions

Existential Functions

See also: **II. Managing Media and Controlling Playback Existential Functions** overview 21
16

BAESound_New()

```
BAESound BAESound_New( BAEMixer mixer );
```

Creates a new **BAESound** structure and associates it with the indicated BAEMixer_.

Returns:		(always succeeds)
Note:	You must use **BAESound_New()** and one of the **BAESound_Load...()** functions before you can play a sample with a BAESound_ object.	
See also:		

BAESound_Delete()

```
BAEResult BAESound_Delete( BAESound sound );
```

Deactivates the indicated BAESound_, unloads its sample media data, and frees its memory. Call this when done with the BAESound_ object.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
Note:		
See also:		

BAESound_SetMixer()

```
BAEResult BAESound_SetMixer(
          BAESound sound,
          BAEMixer mixer );
```

Associates the indicated BAESound_ with the indicated BAEMixer_, replacing the previously associated BAEMixer_.

Returns:	**BBAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESound_GetMixer()

```
BAEResult BAESound_GetMixer(
          BAESound  sound,
          BAEMixer *outMixer );
```

Upon return, the BAEMixer pointed at by parameter **outMixer** will contain the address of the BAEMixer_ with which the indicated BAESound_ is associated.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

Media Management

See also: II. Managing Media and Controlling Playback Media Management overview 22
overview 16

BAESound_LoadMemorySample()

```
BAEResult BAESound_LoadMemorySample(
          BAESound    sound,
          void        *pMemoryFile,
          unsigned    long memoryFileSize,
          BAEFileType fileType             );
```

Loads the indicated BAESound_ with the in-memory sample media data at the indicated address and in AIFF, WAV, or AU format (as indicated via parameter **fileType**). Also sets sample playback properties according to the file header. The sample data is used in place, not copied; however, if any decompression is needed to access the data, memory allocation and decompression will occur during this call.

Params:	**sound**	The BAESound_
	pMemoryFile	Address of sample file image to load
	memoryFileSize	Size in bytes of file image at **pMemoryFile**
	fileType	File format (see **BAEFileType**)
Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_BAD_FILE	Bad or missing sample data
	BAE_BAD_FILE_TYPE	Unknown audio file format
	BAE_PARAM_ERR	**sound** object not initialized
	BAE_NOT_SETUP	Feature not supported on this platform
Note:	On some platforms, Mini-BAE does not support this feature. In those cases, this function has no effect and returns **BAE_NOT_SETUP**.	
See also:		

BAESound_LoadFileSample

```
BAESound_LoadFileSample(
        BAESound    sound,
        BAEPathName filePath,
        BAEFileType fileType  );
```

Loads the indicated BAESound_ from the media file at the indicated **filePath** and of the indicated format (AIFF, WAV, or AU as indicated via parameter **fileType**). Also sets sample playback properties according to the file header. Memory allocation and any sample data decompression will occur during this call.

Params:	**sound**	The BAESound_
	filePath	Path of sample file to load
	fileType	File format (see **BAEFileType**)
Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_MEMORY_ERR	Can't allocate memory for sample file image
	BAE_BAD_FILE	Bad or missing sample data
	BAE_BAD_FILE_TYPE	Unknown audio file format
	BAE_NOT_SETUP	Feature not supported on this platform
Note:	On some platforms, Mini-BAE does not support this feature. In those cases, this function has no effect and returns **BAE_NOT_SETUP**.	
Note:		
See also:		

BAESound_LoadCustomSample()

```
BAEResult BAESound_LoadCustomSample(
          BAESound           sound,
          void               *sampleData,
          unsigned long      frames,
          unsigned short int bitSize,
          unsigned short int channels,
          BAE_UNSIGNED_FIXED rate,
          unsigned long      loopStart,
          unsigned long      loopEnd    );
```

Loads the indicated BAESound_ with a copy of the in-memory raw sample media data at the indicated address, and sets several sample properties as per the parameters.

Params:	**sound**	The BAESound_
	sampleData	Address of sample data to load
	frames	Number of sample frames of data at **sampleData**
	bitSize	Depth in bits of sample data (always 8 or 16)
	channels	1 for mono data, 2 for stereo data.
	rate	Sample rate in Hz, in 16.16 fixed-point format
	loopStart	frame number of first sample in loop
	loopEnd	frame number of last sample in loop
Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_MEMORY_ERR	Can't allocate memory for sample copy
	BAE_PARAM_ERR	**sound** object not initialized

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_Unload()

```
BAEResult BAESound_Unload( BAESound sound );
```

Unloads any previously loaded sample media data from the indicated BAESound_ object.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null **sound** parameter

Note:

See also:

Playback Control

See also: **II. Managing Media and Controlling Playback Playback Control** overview.....21
16
Audio Output Management.....42

BAESound_Start()

```
BAEResult BAESound_Start(
          BAESound            sound,
          short int           priority,
          BAE_UNSIGNED_FIXED  sampleVolume,
          unsigned long       startOffsetFrame );
```

Causes playback of the indicated BAESound_ to begin, at the indicated priority and volume, and optionally beginning at the indicated sample frame. Normal volume is **1.0**. If no voices are available at the indicated priority level, this function fails and returns **NO_FREE_VOICES**.

Returns:		
	BAE_NULL_OBJECT	Null **sound** object pointer
	BAE_NO_FREE_VOICES	Couldn't allocate a voice at this priority
	BAE_PARAM_ERR	Null parameters

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_Stop()

```
BAEResult BAESound_Stop(
          BAESound sound,
          BAE_BOOL startFade );
```

Stops playback of the indicated BAESound_ in one of two ways, depending upon the value of the **startFade** parameter: either stop immediately (**FALSE**), or stop after smoothly fading the sound out over a period of about 2.2 seconds (**TRUE**).

Returns:		
	BAE_NULL_OBJECT	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note: Returns immediately, not at the end of the fade-out period.

See also:

BAESound_Pause()

```
BAEResult BAESound_Pause( BAESound sound );
```

Pauses playback of the indicated BAESound_. If already paused, this function will have no effect. To resume playback, call **BAESound_Resume()** or **BAESound_Start()**.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer

See also:

BAESound_Resume()

```
BAEResult BAESound_Resume( BAESound sound );
```

If the indicated BAESound_ is paused at the time of this call, resumes playback from the point at which it was most recently paused. If not paused, this function will have no effect.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer

Note: Another way to resume playback after a pause is to call **BAESound_Start()**.

See also:

BAESound_IsPaused()

```
BAEResult BAESound_IsPaused(
          BAESound sound,
          BAE_BOOL *outIsPaused );
```

Upon return, parameter **outIsPaused** will point to a **BAE_BOOL** indicating whether the indicated BAESound_ is currently in a paused state (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

See also:

BAESound_IsDone()

```
BAEResult BAESound_IsDone(
          BAESound sound,
          BAE_BOOL *outIsDone );
```

Upon return, the **BAE_BOOL** pointed at by parameter **outIsDone** will indicate whether the indicated BAESound_ object has (**TRUE**) or has not (**FALSE**) played all the way to its end and stopped on its own.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESound_GetInfo()

```
BAEResult BAESound_GetInfo(
          BAESound       sound,
          BAESampleInfo *outInfo );
```

Upon return, the **BAESampleInfo struct** pointed to by parameter **outInfo** will contain a copy of the current sample playback property set of the indicated **BAESound** object.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
	BAE_NOT_SETUP	No sound data loaded
Note:		
See also:		

BAESound_SetVolume()

```
BAEResult BAESound_SetVolume(
          BAESound           sound,
          BAE_UNSIGNED_FIXED newVolume );
```

Sets the playback volume of the indicated BAESound_ object to the indicated level. Normal volume is **1.0**.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_GetVolume()

```
BAEResult BAESound_GetVolume(
          BAESound            sound,
          BAE_UNSIGNED_FIXED *outVolume );
```

Upon return, the **BAE_UNSIGNED_FIXED** pointed to by parameter **outVolume** will hold a copy of the indicated BAESound_'s current playback volume.

Returns:		
	BAE_NULL_OBJECT	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_Fade()

```
BAEResult BAESound_Fade(
          BAESound  sound,
          BAE_FIXED sourceVolume,
          BAE_FIXED destVolume,
          BAE_FIXED timeInMiliseconds );
```

Fades the volume of the indicated BAESound_ smoothly from **sourceVolume** to **destVolume**, over a period of **timeInMilliseconds**. Note that this may be either a fade up or a fade down.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_FIXED** numbers:

FLOAT_TO_FIXED
FIXED_TO_FLOAT
LONG_TO_FIXED
FIXED_TO_LONG
FIXED_TO_LONG_ROUNDED
FIXED_TO_SHORT
FIXED_TO_SHORT_ROUNDED
RATIO_TO_FIXED
UNSIGNED_RATION_TO_FIXED

See also:

BAESound_SetRate()

```
BAEResult BAESound_SetRate(
          BAESound           sound,
          BAE_UNSIGNED_FIXED newRate );
```

Sets the playback sample rate of the indicated BAESound_ object to the indicated rate, in Hertz.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_GetRate()

```
BAEResult BAESound_GetRate(
          BAESound           sound,
          BAE_UNSIGNED_FIXED *outRate);
```

Upon return, the **BAE_UNSIGNED_FIXED** pointed to by parameter **outRate** will hold a copy of the indicated BAESound_'s current sample rate, in Hertz.

Returns:		
	BAE_NULL_OBJECT	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameters

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESound_SetLowPassAmountFilter()

```
BAEResult BAESound_SetLowPassAmountFilter(
          BAESound  sound,
          short int lowPassAmount );
```

Sets the depth of the lowpass filter effect for the indicated BAESound_ object.

Returns:		
	BAE_NULL_OBJECT	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note:

See also:

BAESound_GetLowPassAmountFilter()

```
BAEResult BAESound_GetLowPassAmountFilter(
          BAESound  sound,
          short int *outLowPassAmount );
```

Upon return, the **short int** pointed to by parameter **outLowPassAmount** will hold a copy of the indicated BAESound_ object's current lowpass filter effect's depth setting.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note:

See also:

BAESound_SetResonanceAmountFilter()

```
BAEResult BAESound_SetResonanceAmountFilter(
          BAESound  sound,
          short int resonanceAmount );
```

Sets the resonance of the lowpass filter effect for the indicated BAESound_ object.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note:

See also:

BAESound_GetResonanceAmountFilter()

```
BAEResult BAESound_GetResonanceAmountFilter(
          BAESound  sound,
          short int *outResonanceAmount );
```

Upon return, the **short int** pointed to by parameter **outResonanceAmount** will hold a copy of the indicated BAESound_ object's current lowpass filter effect’s resonance setting.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter

Note:

See also:

BAESound_SetFrequencyAmountFilter()

```
BAEResult BAESound_SetFrequencyAmountFilter(
          BAESound  sound,
          short int frequencyAmount );
```

Sets the frequency of the lowpass filter effect for the indicated BAESound_ object.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESound_GetFrequencyAmountFilter()

```
BAEResult BAESound_GetFrequencyAmountFilter(
          BAESound   sound,
          short int *outFrequencyAmount );
```

Upon return, the **short int** pointed to by parameter **outFrequencyAmount** will hold a copy of the indicated BAESound_ object's current lowpass filter effect's frequency setting.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESound_SetSampleLoopPoints()

```
BAEResult BAESound_SetSampleLoopPoints(
          BAESound      sound,
          unsigned long start,
          unsigned long end    );
```

Sets the loop start and end bounds for the indicated BAESound_, both expressed in terms of sample frame numbers.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Parameters null or out of range
	BAE_BUFFER_TOO_SMALL	Loop too short (see **MIN_LOOP_SIZE**)
	BAE_NOT_SETUP	No sound data loaded
Note:		
See also:		

BAESound_GetSampleLoopPoints()

```
BAEResult BAESound_GetSampleLoopPoints(
          BAESound       sound,
          unsigned long *outStart,
          unsigned long *outEnd    );
```

Upon return, the **unsigned long**s pointed at by parameters **outStart** and **outEnd** will hold copies of the current loop start and end bounds (respectively) of the indicated BAESound_, both expressed in terms of sample frame numbers.

Returns:	**BAE_NULL_OBJECT**	Null **sound** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_NOT_SETUP	No sound data loaded
Note:		
See also:		

[End of BAESound_ Functions Reference]

BAESong_
Functions Reference

Alphabetical Index

Subject Index

Existential Functions

See also: **II. Managing Media and Controlling Playback Existential Functions** overview 22
16

BAESong_New()

```
BAESong BAESong_New( BAEMixer mixer );
```

Creates a new BAESong_ structure and associates it with the indicated BAEMixer_. The new BAESong_ is able to execute direct MIDI event commands immediately (see **MIDI Events**, page 98).

Returns:		(always succeeds)
Note:	You must use **BAESong_New()** and one of the **BAESong_Load...()** functions before you can play a MIDI or RMF song with a BAESong_ object.	
See also:		

BAESong_Delete()

```
BAEResult BAESong_Delete( BAESong song );
```

Deactivates the indicated BAESong_, unloads its MIDI or RMF media data, and frees its memory. Call this when done with the BAESong_ object.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
Note:		
See also:		

BAESong_SetMixer()

```
BAEResult BAESong_SetMixer(
          BAESong  song,
          BAEMixer mixer );
```

Associates the indicated BAESong_ with the indicated BAEMixer_, replacing the previously associated BAEMixer_.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_GetMixer()

```
BAEResult BAESong_GetMixer(
          BAESong   song,
          BAEMixer *outMixer );
```

Upon return, parameter **outMixer** will point to a copy of the BAEMixer_ with which the indicated BAESong_ is associated.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

Media Management

See also: II. Managing Media and Controlling Playback Media Management overview 22
overview 16

BAESong_LoadMidiFromMemory()

```
BAEResult BAESong_LoadMidiFromMemory(
          BAESong        song,
          void const*    pMidiData,
          unsigned long  midiSize,
          BAE_BOOL       ignoreBadInstruments );
```

Associates the indicated BAESong_ with an in-memory image of a Standard MIDI File located at the indicated address and having the indicated length (in bytes). Parameter **ignoreBadInstruments** controls whether any failures to load instruments required to play the MIDI song will (**TRUE**) or will not (**FALSE**) be reported in the returned **BAEResult**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_GENERAL_BAD	Song internally inconsistent
	BAE_BAD_FILE	Bad MIDI data
	BAE_MEMORY_ERR	Couldn't allocate memory
	BAE_PARAM_ERR	Null parameter

Note: Do not free the data at **pMidiData** until the BAESong is done using it.

See also:

BAESong_LoadMidiFromFile()

```
BAEResult BAESong_LoadMidiFromFile(
          BAESong      song,
          BAEPathName  filePath,
          BAE_BOOL     ignoreBadInstruments );
```

Loads the indicated BAESong_ with a copy of the indicated Standard MIDI File. Parameter **ignoreBadInstruments** controls whether any failures to load instruments required to play the indicated MIDI file will (**TRUE**) or will not (**FALSE**) be reported in the returned **BAEResult**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_GENERAL_BAD	Song internally inconsistent
	BAE_BAD_FILE	Bad MIDI data
	BAE_MEMORY_ERR	Couldn't allocate memory
	BAE_PARAM_ERR	Null parameters

Note:

See also:

BAESong_LoadRmfFromMemory()

```
BAEResult BAESong_LoadRmfFromMemory(
          BAESong       song,
          void          *pRMFData,
          unsigned long rmfSize,
          short int     songIndex,
          BAE_BOOL      ignoreBadInstruments );
```

Associates the indicated BAESong_ with the indicated song from an in-memory image of an RMF File located at the indicated address and having the indicated length (in bytes). Songs in an RMF File are numbered consecutively, starting from **0**. Parameter **ignoreBadInstruments** controls whether any failures to load instruments required to play the indicated RMF data will (**TRUE**) or will not (**FALSE**) be reported in the returned **BAEResult**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_GENERAL_BAD	Song internally inconsistent
	BAE_BAD_FILE	Bad MIDI data
	BAE_PARAM_ERR	Null parameters
	BAE_RESOURCE_NOT_FOUND	Couldn't find the requested **songIndex**
Note:	Do not free the data at **pRMFData** until the BAESong_ is done using it.	
See also:		

BAESong_LoadRmfFromFile()

```
BAEResult BAESong_LoadRmfFromFile(
          BAESong     song,
          BAEPathName filePath,
          short int   songIndex,
          BAE_BOOL    ignoreBadInstruments );
```

Loads the indicated BAESong_ with a copy of the indicated song from the indicated RMF File. Songs in an RMF File are numbered consecutively, starting from **0**. Parameter **ignoreBadInstruments** controls whether any failures to load instruments required to play the indicated RMF data will (**TRUE**) or will not (**FALSE**) be reported in the returned **BAEResult**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_GENERAL_BAD	Song internally inconsistent
	BAE_BAD_FILE	Bad MIDI data
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAESong_LoadGroovoid()

```
BAEResult BAESong_LoadGroovoid(
          BAESong  song,
          char     *cName,
          BAE_BOOL ignoreBadInstruments );
```

Loads the indicated BAESong_ with the MIDI data contained in the Groovoid with the indicated name, if that name is available in the instrument bank currently being used by the BAEMixer_ with which the BAESong_ is associated. Parameter **ignoreBadInstruments** controls whether any failures to load instruments required to play the indicated Groovoid will (**TRUE**) or will not (**FALSE**) be reported in the returned **BAEResult**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Song not initialized
	BAE_GENERAL_BAD	Song internally inconsistent
Note:		
See also:		

BAESong_GetMicrosecondLength()

```
BAEResult BAESong_GetMicrosecondLength(
          BAESong         song,
          unsigned long *outLength );
```

Upon return, parameter **outLength** will point to an **unsigned long** containing the length in microseconds of the indicated BAESong_'s currently loaded MIDI or RMF song data. The result assumes that the song would be played at the tempo stored in the song data, so any changes made via **BAESong_SetTempo()** would not be reflected.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

Instrument Management

See also: **II. Managing Media and Controlling Playback** **Instrument Management** overview 22
overview 16
Instrument Bank Management 47

BAESong_LoadInstrument()

```
BAEResult BAESong_LoadInstrument(
          BAESong        song,
          BAE_INSTRUMENT instrument );
```

Loads the indicated instrument (and all samples it uses) from the current instrument bank(s) into the indicated BAESong_, unless already loaded.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
	BAE_NOT_SETUP	BAESong_ not initialized
Note:		
See also:		

BAESong_UnloadInstrument()

```
BAEResult BAESong_UnloadInstrument(
          BAESong        song,
          BAE_INSTRUMENT instrument );
```

Deletes the indicated instrument (and any sample data not needed by the remaining loaded instruments), and frees that memory.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
	BAE_NOT_SETUP	BAESong_ not initialized
	BAE_STILL_PLAYING	Data is locked, try again later
Note:	Unloading an instrument during playback may prevent some or all notes from being heard.	
See also:		

BAESong_IsInstrumentLoaded()

```
BAEResult BAESong_IsInstrumentLoaded(
          BAESong         song,
          BAE_INSTRUMENT  instrument,
          BAE_BOOL        *outIsLoaded );
```

Upon return, the **BAE_BOOL** pointed to by parameter **outIsLoaded** will indicate whether the requested instrument is currently loaded into the indicated BAESong_ (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

Playback Control

See also: **II. Managing Media and Controlling Playback Playback Control** overview.....22
16
Audio Output Management.....42

BAESong_Preroll()

```
BAEResult BAESong_Preroll( BAESong song );
```

Prepares the indicated BAESong_ for later instant playback by performing any and all lengthy resource setup operations.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
Note:		
See also:		

BAESong_Start()

```
BAEResult BAESong_Start(
          BAESong   song,
          short int priority );
```

Causes playback of the indicated BAESong_ to begin, at the indicated priority.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_Stop()

```
BAEResult BAESong_Stop(
          BAESong  song,
          BAE_BOOL startFade );
```

Stops playback of the indicated BAESong_ in one of two ways, depending upon the value of the **startFade** parameter: either stop immediately (**FALSE**), or stop after smoothly fading the song out over a period of about 2.2 seconds (**TRUE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	Returns immediately, not at the end of the fade-out period.	
See also:		

BAESong_Pause()

```
BAEResult BAESong_Pause( BAESong song );
```

Pauses playback of the indicated BAESong_. If already paused, this function will have no effect. To resume playback, call **BAESong_Resume()** or **BAESong_Start()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
Note:		
See also:		

BAESong_Resume()

```
BAEResult BAESong_Resume( BAESong song );
```

If the indicated BAESong_ is paused at the time of this call, causes playback to resume from the point at which it was most recently paused. If not paused, this function will have no effect. Another way to resume playback after a pause is to call **BAESong_Start()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
Note:		
See also:		

BAESong_SetMicrosecondPosition()

```
BAEResult BAESong_SetMicrosecondPosition(
        BAESong       song,
        unsigned long ticks );
```

Sets the current playback position of the indicated BAESong_ to the requested time offset, expressed in microseconds from the beginning of the MIDI or RMF song data.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_GetMicrosecondPosition()

```
BAEResult BAESong_GetMicrosecondPosition(
          BAESong         song,
          unsigned long *outTicks );
```

Upon return, parameter **outTicks** will point to an **unsigned long** containing the current playback position of the indicated BAESong_, expressed in microseconds.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_IsPaused()

```
BAEResult BAESong_IsPaused(
          BAESong    song,
          BAE_BOOL *outIsPaused );
```

Upon return, parameter **outIsPaused** will point to a **BAE_BOOL** indicating whether the indicated BAESong_ is currently in a paused state (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_IsDone()

```
BAEResult BAESong_IsDone(
          BAESong    song,
          BAE_BOOL *outIsDone );
```

Upon return, the **BAE_BOOL** pointed at by parameter **outIsDone** will indicate whether the indicated BAESong_ object has (**TRUE**) or has not (**FALSE**) played all the way to its end and stopped on its own.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_SetVolume()

```
BAEResult BAESong_SetVolume(
          BAESong             song,
          BAE_UNSIGNED_FIXED volume );
```

Sets the playback volume of the indicated BAESong_ object to the indicated level. Normal volume is `1.0`.

Returns:	`BAE_NULL_OBJECT`	Null `song` object pointer
	`BAE_GENERAL_BAD`	song internally inconsistent
	`BAE_BAD_FILE`	Bad MIDI data
	`BAE_PARAM_ERR`	Null parameter

Note: The `BAE_UNSIGNED_FIXED` data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using `BAE_UNSIGNED_FIXED` numbers:

`FLOAT_TO_UNSIGNED_FIXED`
`UNSIGNED_FIXED_TO_FLOAT`
`LONG_TO_UNSIGNED_FIXED`
`UNSIGNED_FIXED_TO_LONG`
`UNSIGNED_FIXED_TO_LONG_ROUNDED`
`UNSIGNED_FIXED_TO_SHORT`
`UNSIGNED_FIXED_TO_SHORT_ROUNDED`

See also:

BAESong_GetVolume()

```
BAEResult BAESong_GetVolume(
          BAESong             song,
          BAE_UNSIGNED_FIXED *outVolume );
```

Upon return, the `BAE_UNSIGNED_FIXED` pointed to by parameter `outVolume` will hold a copy of the indicated BAESong_'s current playback volume.

Returns:	`BAE_NULL_OBJECT`	Null `song` object pointer
	`BAE_PARAM_ERR`	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESong_Fade()

```
BAEResult BAESong_Fade(
          BAESong   song,
          BAE_FIXED sourceVolume,
          BAE_FIXED destVolume,
          BAE_FIXED timeInMiliseconds );
```

Fades the volume of the indicated BAESong_ smoothly from **sourceVolume** to **destVolume**, over a period of **timeInMilliseconds**. Note that this may be either a fade up or a fade down.

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
	BAE_NOT_SETUP	No song loaded

Note: The **BAE_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_FIXED** numbers:

FLOAT_TO_FIXED
FIXED_TO_FLOAT
LONG_TO_FIXED
FIXED_TO_LONG
FIXED_TO_LONG_ROUNDED
FIXED_TO_SHORT
FIXED_TO_SHORT_ROUNDED
RATIO_TO_FIXED
UNSIGNED_RATION_TO_FIXED

See also:

BAESong_SetMasterTempo()

```
BAEResult BAESong_SetMasterTempo(
          BAESong            song,
          BAE_UNSIGNED_FIXED tempoFactor );
```

Sets the tempo of the indicated BAESong_, expressed as a ratio relative to the tempo stored in the Standard MIDI File or RMF file data. For example, a **tempoFactor** of **2.0** would cause the song to play at doublespeed.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter

Note: If called while the song is stopped, this function will appear to have no effect because starting a song resets the tempo to the value stored in the MIDI or RMF data.

The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESong_GetMasterTempo()

```
BAEResult BAESong_GetMasterTempo(
          BAESong            song,
          BAE_UNSIGNED_FIXED *outTempoFactor );
```

Upon return, parameter **outTempoFactor** will point at a **BAE_UNSIGNED_FIXED** containing a copy of the indicated BAESong_'s current tempo, as set by **BAESong_SetMasterTempo()**. This tempo is expressed as a ratio relative to the tempo stored in the Standard MIDI File or RMF file data. Example, a **tempoFactor** of **2.0** indicates playback at doublespeed.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter

Note: The **BAE_UNSIGNED_FIXED** data type consists of a 16-bit integer part and a 16-bit fractional part. Beatnik recommends using the following BAE macros when forming or using **BAE_UNSIGNED_FIXED** numbers:

FLOAT_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_FLOAT
LONG_TO_UNSIGNED_FIXED
UNSIGNED_FIXED_TO_LONG
UNSIGNED_FIXED_TO_LONG_ROUNDED
UNSIGNED_FIXED_TO_SHORT
UNSIGNED_FIXED_TO_SHORT_ROUNDED

See also:

BAESong_SetTranspose()

```
BAEResult BAESong_SetTranspose(
          BAESong song,
          long    semitones );
```

Sets the indicated BAESong_'s transposition (pitch offset), in terms of a signed number of MIDI note numbers (semitones). Positive transposition produces higher note numbers and higher pitches; negative transposition produces lower note numbers and pitches. The current transposition offset is always added to note numbers played with the BAESong_ at the time each note is rendered (rather than modifying stored MIDI data). However, each MIDI channel of the BAESong_ can independently enable or disable transposition.

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter

See also:

BAESong_GetTranspose()

```
BAEResult BAESong_GetTranspose(
          BAESong song,
          long    *outSemitones );
```

Upon return, the **long** pointed to by parameter **outTranspose** will hold a copy of the indicated BAESong_'s current transposition amount. (See **BAESong_SetTranspose()**)

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter

See also:

BAESong_AllowChannelTranspose()

```
BAEResult BAESong_AllowChannelTranspose(
          BAESong             song,
          unsigned short int  channel,
          BAE_BOOL            allowTranspose );
```

Enables (**TRUE**) or disables (**FALSE**) pitch transposition for the indicated MIDI channel of the indicated BAESong_. (See **BAESong_SetTranspose()**)

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter

See also:

BAESong_DoesChannelAllowTranspose()

```
BAEResult BAESong_DoesChannelAllowTranspose(
          BAESong             song,
          unsigned short int  channel,
          BAE_BOOL            *outAllowTranspose );
```

Upon return, the **BAE_BOOL** pointed to by parameter **outAllowTranspose** will indicate whether the indicated MIDI channel of the indicated BAESong_ has transposition enabled (**TRUE**) or disabled (**FALSE**). (See **BAESong_SetTranspose()**)

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters

See also:

BAESong_SetLoops()

```
BAEResult BAESong_SetLoops(
          BAESong song,
          short   numLoops );
```

Sets the loop repeat counter for the indicated BAESong_ to the indicated value. To prevent or cancel looping, call with **numLoops** equal to **0**. Please read all notes below.

Returns:		
	BAE_NULL_OBJECT	Null **song** object pointer
	BAE_PARAM_ERR	Bad **numLoops** – must be non-negative

Note: Calling **BAESong_Start()** will ordinarily reset the BAESong_'s repeat counter variable to 0, replacing your **numLoops** value. To prevent this, call **BAESong_Preroll()** before calling **BAESong_SetLoops()**.

Note: **About Song Looping –** Looping behavior for each BAESong_ – that is, whether or not the song will begin again from its start each time playback reaches the end of the MIDI or RMF data – is controlled by an internal repeat counter variable, which you can override with this function at any time. If the value of the loop repeat counter is equal to zero when the end of the song is reached, then the song doesn't repeat any further; otherwise, the counter is decremented and the song restarts.

Note that this means your song will play a total of **numLoops** + 1 times, not **numLoops**. For example, if you call **BAESong_SetLoops(yourSong, 1);** while the song is playing, you'll hear the song play twice: a first pass, followed by a loop back to the start and a second playback pass.

Note: **BAESong_SetLoops()** controls only RMF whole-song looping, as set in the Beatnik Editor's Song Settings window. Looping within individual MIDI File tracks using the Beatnik marker and controller techniques is not affected by this function.

See also:

BAESong_GetLoops()

```
BAEResult BAESong_GetLoops(
          BAESong    song,
          short int *outNumLoops );
```

Upon return, parameter **outNumLoops** will point to a **short int** containing a copy of the indicated BAESong_'s loop repeat setting, as set by **BAESong_SetLoops()**. This is the number of times the song will restart when playing back, so the song will be heard that number of times plus one.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	This function returns the value of the repeat setting (which does not change during playback), not the current value of the internal loop counter (which does change during playback). Consequently, the returned value will not change during BAESong_ playback.	
See also:		

BAESong_MuteChannel()

```
BAEResult BAESong_MuteChannel(
          BAESong            song,
          unsigned short int channel );
```

Mutes the indicated MIDI channel of the indicated BAESong_. To restore normal output, use **BAESong_UnmuteChannel()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	'Mute' is an audio production term meaning to temporarily turn off an audio signal; muting a MIDI channel turns off the audio output of all notes rendered on that channel.	
See also:		

BAESong_UnmuteChannel()

```
BAEResult BAESong_UnmuteChannel(
          BAESong            song,
          unsigned short int channel );
```

Unmutes the indicated MIDI channel of the indicated BAESong_, reversing the effect of **BAESong_MuteChannel()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
See also:		

BAESong_GetChannelMuteStatus()

```
BAEResult BAESong_GetChannelMuteStatus(
          BAESong   song,
          BAE_BOOL *outChannels );
```

Upon return, the array of 16 **BAE_BOOLs** pointed to by parameter **outChannels** will indicate whether each of the 16 MIDI channels of the indicated BAESong_ is currently muted (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_SoloChannel()

```
BAEResult BAESong_SoloChannel(
          BAESong             song,
          unsigned short int channel );
```

Solos the indicated MIDI channel of the indicated BAESong_. To restore normal output, use **BAESong_UnSoloChannel()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	‘Solo’ is an audio production term meaning to temporarily turn off all audio signals other than the soloed signal. Soloing a MIDI channel turns off the audio output of all notes rendered on all other MIDI channels.	
See also:		

BAESong_UnSoloChannel()

```
BAEResult BAESong_UnSoloChannel(
          BAESong             song,
          unsigned short int channel );
```

Un-solos the indicated MIDI channel of the indicated BAESong_, reversing the effect of **BAESong_SoloChannel()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_GetChannelSoloStatus()

```
BAEResult BAESong_GetChannelSoloStatus(
          BAESong   song,
          BAE_BOOL *outChannels );
```

Upon return, the array of 16 **BAE_BOOLs** pointed to by parameter **outChannels** will indicate whether each of the 16 MIDI channels of the indicated BAESong_ is currently soloed (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_MuteTrack()

```
BAEResult BAESong_MuteTrack(
          BAESong             song,
          unsigned short int track );
```

Mutes the indicated Standard MIDI File data track or RMF file data track for the indicated BAESong_. To restore normal output, use **BAESong_UnmuteTrack()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	'Mute' is an audio production term meaning to temporarily turn off an audio signal; muting a MIDI File track turns off the audio output of all notes stored in that track.	
See also:		

BAESong_UnmuteTrack()

```
BAEResult BAESong_UnmuteTrack(
          BAESong             song,
          unsigned short int track );
```

Unmutes the indicated Standard MIDI File data track or RMF file data track for the indicated BAESong_, reversing the effect of **BAESong_MuteTrack()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_GetTrackMuteStatus()

```
BAEResult BAESong_GetTrackMuteStatus(
          BAESong   song,
          BAE_BOOL *outTracks );
```

Upon return, the array of 16 **BAE_BOOL**s pointed to by parameter **outTracks** will indicate whether each of the 16 Standard MIDI File or RMF file data tracks for the indicated BAESong_ is currently muted (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_SoloTrack()

```
BAEResult BAESong_SoloTrack(
          BAESong            song,
          unsigned short int track );
```

Solos the indicated Standard MIDI File or RMF file data track for the indicated BAESong_. To restore normal output, use **BAESong_UnSoloTrack()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:	'Solo' is an audio production term meaning to temporarily turn off all audio signals other than the soloed signal. Soloing a MIDI File track turns off the audio output of all notes stored in all other MIDI File tracks.	
See also:		

BAESong_UnSoloTrack()

```
BAEResult BAESong_UnSoloTrack(
          BAESong            song,
          unsigned short int track );
```

Un-solos the indicated Standard MIDI File or RMF file data track for the indicated BAESong_, reversing the effect of **BAESong_SoloTrack()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_GetSoloTrackStatus()

```
BAEResult BAESong_GetSoloTrackStatus(
        BAESong   song,
        BAE_BOOL *outTracks );
```

Upon return, the array of 16 **BAE_BOOL**s pointed to by parameter **outTracks** will indicate whether each of the 16 Standard MIDI File or RMF file data tracks for the indicated BAESong_ is currently soloed (**TRUE**) or not (**FALSE**).

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

MIDI Events

See also: **MIDI Music Rendering: The Mini-BAE Synthesizer** 6

BAESong_NoteOn()

```
BAEResult BAESong_NoteOn(
          BAESong       song,
          unsigned char channel,
          unsigned char note,
          unsigned char velocity,
          unsigned long time      );
```

Renders a note on the indicated MIDI channel of the indicated BAESong_, using the indicated MIDI note number and note velocity. If you supply a value of **0** for the **time** parameter the note is started immediately, otherwise the note is started when the BAESong_'s current playback position reaches (or passes) **time**. The note will be rendered using the MIDI program (instrument) number and bank number in effect for the indicated MIDI channel of the BAESong_ at the **time** the note is started. Once started, the note will be maintained (and perhaps audibly sustained) until ended with a corresponding **BAESong_NoteOff()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters

Note: If the required instrument is not loaded at the time the note is started, the note may produce unpredictable sound or silence. If there is any question that the instrument you need may not be loaded, use **BAESong_NoteOnWithLoad()**.

See also:

BAESong_NoteOnWithLoad()

```
BAEResult BAESong_NoteOnWithLoad(
          BAESong       song,
          unsigned char channel,
          unsigned char note,
          unsigned char velocity,
          unsigned long time      );
```

Renders a new note on the indicated MIDI channel of the indicated BAESong_, using the indicated MIDI note number and note velocity. If you supply a value of **0** for the **time** parameter the note is started immediately, otherwise the note is started when the BAESong_'s current playback position reaches (or passes) **time**. The note will be rendered using the MIDI program (instrument) number and bank number in effect for the indicated MIDI channel of the BAESong_ at the time the note is started. If that instrument is not yet loaded, it will be loaded in time to start the note. Once started, the note will be maintained (and perhaps audibly sustained) until ended with a corresponding **BAESong_NoteOff()**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	If the required instrument is not loaded at the time the note is started, the note may produce unpredictable sound or silence. If there is any question that the instrument you need may not be loaded, use **BAESong_NoteOnWithLoad()**.	
See also:		

BAESong_NoteOff()

```
BAEResult BAESong_NoteOff(
          BAESong        song,
          unsigned char  channel,
          unsigned char  note,
          unsigned char  velocity,
          unsigned long  time      );
```

Causes any and all notes with matching MIDI channel and MIDI note number currently rendering on the indicated BAESong_ to “key off” at the indicated time, with the indicated “key off” velocity. This leads to termination of the note's envelope either immediately or at a later time (depending upon the design of the instrument being used), and upon envelope termination all rendering and maintenance of the note will end. If you supply a value of **0** for the **time** parameter the “key off” occurs immediately, otherwise it occurs when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAESong_AllNotesOff()

```
BAEResult BAESong_AllNotesOff(
          BAESong       song,
          unsigned long time  );
```

Causes any and all notes rendering on the indicated BAESong_ to “key off” at the indicated **time**. This leads to termination of those notes’ envelopes either immediately or at a later time (depending upon the design of the instrument being used), and upon envelope termination all rendering and maintenance of the notes will end. If you supply a value of **0** for the **time** parameter the “key offs” occurs immediately, otherwise they occur when the BAESong_’s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameter
Note:		
See also:		

BAESong_ProgramChange()

```
BAEResult BAESong_ProgramChange(
          BAESong       song,
          unsigned char channel,
          unsigned char programNumber,
          unsigned long time           );
```

Sends a MIDI Program Change event on the indicated MIDI channel of the indicated BAESong_, selecting the indicated instrument from the channel's currently selected instrument bank. If you supply a value of **0** for the **time** parameter the program change event occurs immediately, otherwise the event occurs when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	When the BAESong_ is initialized all MIDI channels’ bank numbers are set to **0** (the General MIDI bank), but they can be changed individually via the **BAESong_ProgramBankChange()** function, or a MIDI continuous controller **0** event (which can be either stored in a MIDI or RMF file, or sent via the function **BAESong_ControlChange()**).	
See also:		

BAESong_ProgramBankChange()

```
BAEResult BAESong_ProgramBankChange(
          BAESong       song,
          unsigned char channel,
          unsigned char programNumber,
          unsigned char bankNumber,
          unsigned long time           );
```

Selects the indicated MIDI instrument bank and sends a MIDI Program Change event on the indicated MIDI channel of the indicated BAESong_, thus selecting the indicated instrument from the indicated instrument bank. If you supply a value of **0** for the **time** parameter the Program Change event occurs immediately, otherwise the event occurs when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	Mini-BAE supports three bank numbers: **0** for General MIDI, **1** for Beatnik Special, and **2** for User instruments directly contained within RMF files.	
See also:		

BAESong_GetProgramBank()

```
BAEResult BAESong_GetProgramBank(
          BAESong       song,
          unsigned char channel,
          unsigned char *outProgram,
          unsigned char *outBank     );
```

Upon return, the **unsigned char**s pointed to by parameters **outProgram** and **outBank** will contain copies of the current MIDI program (instrument) number and instrument bank number, respectively, for the indicated MIDI channel of the indicated BAESong_.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	MIDI program number values range from **0** through **127**, and Beatnik supports three bank numbers: **0** for General MIDI, **1** for Beatnik Special, and **2** for User instruments directly contained within RMF files.	
See also:		

BAESong_PitchBend()

```
BAEResult BAESong_PitchBend(
          BAESong       song,
          unsigned char channel,
          unsigned char lsb,
          unsigned char msb,
          unsigned long time );
```

Sets the Pitch Bend value for the indicated MIDI channel of the indicated BAESong_, expressed as a 14 bit (plus sign) Least Significant Byte / Most Significant Byte parameter pair. This Pitch Bend control detunes all notes being rendered on the indicated channel at the time of the pitch bend event, in an amount determined by the design of the MIDI channel's current instrument at the time. If you supply a value of **0** for the **time** parameter the pitch bend event is rendered immediately, otherwise the event is rendered when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	To produce a continuous pitch sweep effect, you must call **BAESong_PitchBend()** repeatedly with a smoothly changing **msb** + **lsb** value.	
See also:		

BAESong_GetPitchBend()

```
BAEResult BAESong_GetPitchBend(
          BAESong       song,
          unsigned char channel,
          unsigned char *outLSB,
          unsigned char *outMSB  );
```

Upon return, the **unsigned char**s pointed to by parameters **outLSB** and **outMSB** will contain a copy of the current MIDI pitchbend value for the requested MIDI channel of the indicated BAESong_, expressed as a 14 bit (plus sign) Least Significant Byte / Most Significant Byte parameter pair

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAESong_ControlChange()

```
BAEResult BAESong_ControlChange(
          BAESong       song,
          unsigned char channel,
          unsigned char controlNumber,
          unsigned char controlValue,
          unsigned long time          );
```

Sets the indicated MIDI continuous controller for the indicated MIDI channel of the indicated BAESong_ to the indicated value. If you supply a value of **0** for the **time** parameter the control change event occurs immediately, otherwise the event occurs when the BAESong_'s current playback position reaches (or passes) **time**.

The large table below describes the MIDI continuous controllers that Mini-BAE implements.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	MIDI continuous controller values range from **0** through **127**.	
Note:	A Continuous Controller number **123** message resets the values of all controllers for that MIDI channel to the **Reset Value** shown in the above table.	
See also:		

Controller Number	Reset Value	Function and Controller Value Effect
0	0	**Channel instrument Bank Select** Note: Responds to bank number MSB only. 0 – General MIDI bank (in sample library) 1 – Beatnik's Special bank (in sample library) 2 – User bank (in RMF file)
1	0	**Channel Modulation Wheel** Range: 0 (no modulation) – 127 (maximum modulation)
6	0	**Data Entry** Used after NRPN 640 (Continuous Controllers 98, 99) to set the MIDI Channel's response to MIDI Program Change and Note On messages: 0 – General MIDI Channel Mode: On Channel 10, selects Percussion Non-Transpose Mode; on all other channels, selects Melodic Transpose Mode 1 – Percussion Transpose Mode: Programs 0-127 select Percussion instruments 128-255, notes transpose them. 2 – Percussion Non-Transpose Mode: (ala General MIDI Channel 10) Note numbers 0-127 trigger Percussion instruments at natural pitch. 3 – Melodic Transpose Mode: Programs 0-127 select Melodic bank instruments, notes transpose them.
7	127	**Channel Volume** Note: This behavior is different from most other MIDI synthesizers. 0 – Prevents any notes from starting on this channel Normal range: 1 (minimum volume) – 127 (maximum volume)
10	64	**Channel Pan** (stereo position) Range: 0 (full left) – 127 (full right). Center is 64.
11	0	**Channel Expression** Note: This behavior is different from most other MIDI synthesizers. 127 – Volume is boosted by 25% 0 – 126: Normal volume
64	0	**Channel Sustain Pedal** ("Hold1")
98, 99	0	**Non-Registered Parameter Number (NRPN)** – See Controller 6 98 – Least Significant Byte (LSB) / 99 – Most Significant Byte (MSB)
100, 101	0	**Registered Parameter Number (RPN)** – Reserved
123	(n/a)	**Reset All Controllers & All Notes Off** – see Note above

BAESong_GetControlValue()

```
BAEResult BAESong_GetControlValue(
          BAESong       song,
          unsigned char channel,
          unsigned char controller,
          char          *outValue   );
```

Upon return, the **char** pointed to by parameter **outValue** will contain a copy of the current value of the indicated MIDI continuous controller for the indicated MIDI channel of the indicated BAESong_.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:	MIDI continuous controller values range from **0** through **127**.	
See also:		

BAESong_KeyPressure()

```
BAEResult BAESong_KeyPressure(
          BAESong       song,
          unsigned char channel,
          unsigned char note,
          unsigned char pressure,
          unsigned long time      );
```

Sets the MIDI polyphonic key pressure value for the indicated MIDI note number on the indicated MIDI channel of the indicated BAESong_. If you supply a value of **0** for the **time** parameter the key pressure event is rendered immediately, otherwise the event is rendered when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAESong_ChannelPressure()

```
BAEResult BAESong_ChannelPressure(
          BAESong       song,
          unsigned char channel,
          unsigned char pressure,
          unsigned long time      );
```

Sets the Channel Key Pressure value for the indicated MIDI channel of the indicated BAESong_. If you supply a value of **0** for the **time** parameter the Channel Key Pressure event is rendered immediately, otherwise the event is rendered when the BAESong_'s current playback position reaches (or passes) **time**.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

BAESong_ParseMidiData()

```
BAEResult BAESong_ParseMidiData(
          BAESong       song,
          unsigned char commandByte,
          unsigned char data1Byte,
          unsigned char data2Byte,
          unsigned char data3Byte,
          unsigned long time );
```

Sends the BAESong_ object any arbitrary short MIDI message, consisting of the indicated MIDI **commandByte** and up to three MIDI data bytes, at the indicated **time**. The Mini-BAE MIDI synthesizer responds to the following **commandByte** values, where '**n**' represents the MIDI channel nybble:

```
0x8n   Note Off
0x9n   Note On
0xAn   Key Pressure (aftertouch)
0xBn   Continuous Controller
0xCn   Program Change
0xDn   Channel Pressure (aftertouch)
0xEn   Pitch Bend
```

If you supply a value of **0** for the **time** parameter the event occurs immediately, otherwise it occurs when the BAESong_'s current playback position reaches (or passes) **time**.

Example: **BAESong_ParseMidiData(0x92, 80, 127, 0)** immediately sends a Note On for channel 2, note 80, velocity 127.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
See also:		

BAESong_AreMidiEventsPending()

```
BAEResult BAESong_AreMidiEventsPending(
          BAESong  song,
          BAE_BOOL *outPending );
```

Upon return, the **BAE_BOOL** pointed to by parameter **outPending** will indicate whether any MIDI events are currently pending. That is, whether this BAESong_ has queued any MIDI events with a future **time** for later execution.

Returns:	**BAE_NULL_OBJECT**	Null **song** object pointer
	BAE_PARAM_ERR	Null parameters
Note:		
See also:		

[End of BAESong_ Functions Reference]

BAEUtil_
Functions Reference

Alphabetical Index

Subject Index

Instrument Management Utilities

See also: **Instrument Management Functions** overview23

BAEUtil_TranslateBankProgramToInstrument()

```
BAE_INSTRUMENT   TranslateBankProgramToInstrument(
        unsigned short bank,
        unsigned short program,
        unsigned short channel,
        unsigned short note );
```

Returns the **BAE_INSTRUMENT** ID of the Beatnik instrument being used on the indicated MIDI channel number for the indicated MIDI Program Bank and Program Number. For MIDI channel 10, the MIDI note number is also considered (the note number is ignored for all MIDI channels other than 10).

Returns:	**BAE_INSTRUMENT** ID
Note:	Beatnik supports three bank numbers: 0 for General MIDI, 1 for Beatnik Special, and 2 for User instruments directly contained within RMF files.
Note:	Mini-BAE conforms to the General MIDI standard, whereby MIDI channel 10 (**PERCUSSION_CHANNEL**) is considered the ‘drum channel’ and handles MIDI note numbers differently from the other 15 channels. Specifically, each MIDI note number accesses a separate instrument, rather than transposing a single instrument to different pitches as in the ‘melodic’ channels.
See also:	

RMF Metadata Utilities

See also: **RMF Metadata Functions** overview.....24

BAEUtil_GetRmfSongInfo()

```
BAEResult BAEUtil_GetRmfSongInfo(
          void           *pRMFData,
          unsigned long  rmfSize,
          short          songIndex,
          BAEInfoType    infoType,
          char           *targetBuffer,
          unsigned long  bufferBytes );
```

If the RMF file image at address **pRMFData** contains a song with index **songIndex**, and that song includes a text info field of type **infoType**, then upon return the null-terminated character string pointed at by parameter **targetBuffer** will contain a copy of that text info field. You must supply the size in bytes of the RMF file image, and the size in bytes of your **targetBuffer**.

Returns:	**BAE_PARAM_ERR**	Bad **infoType** requested
	BAE_NOT_SETUP	RMF info feature not supported
Note:		
See also:		

BAEUtil_GetInfoSize()

```
unsigned long BAEUtil_GetInfoSize(
          void           *pRMFData,
          unsigned long  rmfSize,
          short          songIndex,
          BAEInfoType    infoType );
```

If the RMF file image at address **pRMFData** contains a song with index **songIndex**, and that song includes a text info field of type **infoType**, returns the size in bytes of the contents of that field. You must supply the size in bytes of the RMF file image.

Returns:	Size in bytes of the contents of the indicated Text info field.
Note:	
See also:	

BAEUtil_IsRmfSongEncrypted()

```
BAE_BOOL BAEUtil_IsRmfSongEncrypted(
         void          *pRMFData,
         unsigned long rmfSize,
         short         songIndex );
```

If the RMF file image at address **pRMFData** contains a song with index **songIndex**, returns a **BAE_BOOL** indicating whether that song is (**TRUE**) or is not (**FALSE**) encrypted. You must supply the size in bytes of the RMF data.

Returns: **TRUE** (encrypted) or **FALSE** (not encrypted)

Note: While Beatnik RMF generation tools generally encrypt songs, the RMF file format also accommodates unencrypted songs.

See also:

BAEUtil_IsRmfSongCompressed()

```
BAE_BOOL BAEUtil_IsRmfSongCompressed(
         void *pRMFData,
         unsigned long rmfSize,
         short songIndex );
```

If the RMF file image at address **pRMFData** contains a song with index **songIndex**, returns a **BAE_BOOL** indicating whether that song is (**TRUE**) or is not (**FALSE**) data-compressed. You must supply the size in bytes of the RMF data.

Returns: **TRUE** (data-compressed) or **FALSE** (not data-compressed)

Note: While Beatnik RMF generation tools generally data-compress songs, the RMF file format also accommodates uncompressed songs.

See also:

BAEUtil_GetRmfVersion()

```
BAEResult BAEUtil_GetRmfVersion(
         void          *pRMFData,
         unsigned long rmfSize,
         short int     *pVersionMajor,
         short int     *pVersionMinor,
         short int     *pVersionSubMinor );
```

If the RMF file image exists at address **pRMFData**, then upon return the **short int**s pointed to by parameters **pVersionMajor**, **pVersionMinor**, and **pVersionSubMinor** will contain the version number of the RMF format in which that RMF file is encoded. You must supply the size in bytes of the RMF data.

Returns: **BAE_PARAM_ERR** Null parameters or bad RMF data

See also:

[End of BAEUtil_ Functions Reference]

Appendix 1:

Glossary

Terminology for Music, Audio, and Beatnik

AIFF – Audio Interchange File Format, the name of a PCM (Pulse Code Modulation) digital audio file format. Typically an AIFF file will have a filename extension of ".aiff". Can be loaded into a BAESound_ object.

.AU – AUdio file, the name of a PCM (Pulse Code Modulation) digital audio file format. Typically an AU file will have a filename extension of ".au". Can be loaded into a BAESound_ object.

Audio Stream – Any source of serial PCM (Pulse Code Modulation) digital audio data, i.e. a disk file, network connection, etc. Can be played via a BAESoundStream object.

Bank – A collection of instrument definitions for the Mini-BAE MIDI Synthesizer, consisting of up to 128 melodic instruments and up to 128 percussion instruments. Built in the Beatnik Editor and saved in an RMF File

Channel (MIDI 1-16) – In the basic realtime MIDI serial communication protocol, multiple messages streams are sent down the same wire, but kept distinct by the use of a 4-bit Channel field. Consequently there are 16 MIDI channels available in every MIDI connection. Typically these 16 MIDI channels each correspond to a separate logical MIDI musical instrument. These logical instruments may be physically separate devices, or in the case of multitimbral instruments, a single physical device may include multiple 'virtual instruments,' one per MIDI channel, with each channel able to play an instrument sound different from all other channels. (Further, individual virtual instruments may be polyphonic, i.e. able to play more than one note at a time; generally, it takes one voice for each simultaneously playing note.) So, all notes played on a given MIDI channel are played with the same instrument sound. That sound can be selected or changed with a MIDI **ProgramChange** event on that MIDI channel. A notable exception to this scheme is the custom of the Drum Channel, described below.

Drum Channel – In General MIDI, MIDI channel 9 is reserved as the drum channel and responds to MIDI Note On events differently from other channels. On the drum channel, each MIDI note number causes a different instrument to play at its own natural pitch, whereas on all other channels all MIDI notes use the same instrument but each MIDI note number plays that instrument at a different pitch. This scheme makes it easy to access many different percussion sounds (which a lot of popular music requires; hence the name 'drum channel') without having to use a MIDI Program Change event at every instrument change.

Igor – The night was dark and fiercely storming as young, stoic, and ever so slightly goofy Igor once again reached for the engraved antique brass brakesman's grip that would slowly, inexorably, and pneumatically raise the rusty operating table on which the now-inanimate husk that had in its prior, pre-reconfiguration life been known as SoundMusicSys upwards, up through the mechanically openable, many-paned skylight in what the now increasingly frequent bursts of lightning-light revealed to be the improbably distant ceiling of the improbably large Acoustic Lab, exposing

SoundMusicSys to all the brute raw blind inhuman power of the Castle's great sky, bait-like, to fool Nature into that most un-Natural of all possible transformations, Reanimation. Igor pulled the lever, and giggled.

Instrument – In Mini-BAE, an Instrument definition occupies one of the 256 slots in a Bank, and consists of a number of parameters and one or more pointers to the Samples it requires. A Mini-BAE Instrument is analogous to a 'patch' or 'program' in a typical electronic musical instrument. You can edit Instrument definitions in the Beatnik Editor, as well as collecting Instruments into Banks.

Interpolation – It turns out that in order to play a sampled sound back at a pitch other than that at which it was recorded, you need to compute sample values that would fall between the samples you actually have. This is generically called Interpolation, and there are different ways to do it. Mini-BAE offers three different ways: `Mini-BAE_DROP_SAMPLE`, `Mini-BAE_2_POINT_INTERPOLATION`, and `Mini-BAE_LINEAR_INTERPOLATION`. Consult **Choose Cheaper Pitch Transposition Mode** for details of each of these interpolation types (see page 27).

Loop – To create an instrument able to hold a note for an arbitrarily long time, you 'loop' a section of the sample the instrument uses– i.e. play the loop section over and over and over as long as you still need it. The looped section is defined by a pointer to the first sample in the looped section (loop start), and a pointer to the last sample in the looped section (loop end); when playback reaches the loop end, the player jumps back to the beginning of the loop. Note that it's usually important and sometimes difficult to choose your loop section to minimize discontinuities at the loop point.

MIDI – Musical Instrument Digital Interface. The various MIDI standards are based on a realtime serial communication protocol which is generally used to connect a controller device, such as a keyboard or a computer running a sequencer program, to a sound-generating musical instrument device such as a synthesizer or sampler. From this realtime control origin, various related but non-realtime standards such as MIDI Files and the General MIDI instrument set have grown. The MIDI standards are maintained by the International MIDI Association.

MIDI File – The name of a musical data file format. Typically a MIDI file will have a filename extension of ".mid" or ".mdi". Can be loaded into a BAESong_ object.

MIDI Synthesizer – Real or virtual musical instrument that accepts a stream of MIDI messages and generates an audio output. These instruments can all be thought of as MIDI Synthesizers, although some people prefer to use a more precise terminology depending upon the exact sound generating technology used by the device. For example, a sampler is sometimes different from a synthesizer, but this gets tricky in the cases of many recent instruments and Mini-BAE, where the instrument has the sound-shaping capabilities of a synthesizer but is able to apply them to samples as well as the simpler waveforms typical of most synthesizers. The Mini-BAE MIDI Synthesizer is a multitimbral MIDI instrument, implemented in software rather than hardware. (See also glossary entry for Channel.)

Mixer – In general, a Mixer is a device that combines some number of audio inputs into a smaller number of audio outputs. In Mini-BAE, the BAEMixer_ does that and much, much more. Consult **Mini-BAE Audio Mixer** for details (see page 8).

MixLevel – In the BAEMixer_, the **mixLevel** property is an integer that sets the maximum number of full-scale audio sources that Mini-BAE can play simultaneously without lapsing into digital overflow distortion.

Mute – To mute a sound source or mixer channel is to temporarily turn its volume off, so that it is no longer heard. The obverse of 'to mute' is 'to unmute.'

Pan – A sound's position in the left-to-right stereo field; in Mini-BAE, this is expressed as an integer with -63 producing full left, 0 producing center, and 63 producing full right, with intermediate values producing proportional positions.

RMF – Rich Music Format, the name of a musical data file format invented by Beatnik. Typically an RMF file will have a filename extension of ".rmf". Can be loaded into a BAESong_ object. Like the MOD format, RMF has the ability to encapsulate its own instrument data, including the sound samples needed to play the sound as the arranger intended it to sound.

Sample – In Mini-BAE, a Sample is a table of linear PCM digital audio samples that is used by an Instrument definition. The Beatnik Editor allows you to link Samples to Instruments. Generically, 'sample' also refers to any sampled audio data, so you may also hear your digital audio media (AIFF, AU, and WAV files of music, sound effects, or dialog) referred to as 'samples.'

SDII or SD2 – Sound Designer II, the name of a Pulse Code Modulation (PCM) digital audio file format (and a trademark of Digidesign, Inc.). Typically a SDII file will have no filename extension, or a filename extension of ".sd2". Cannot be loaded into any Mini-BAE objects for playback; you have to convert a SDII file to AIFF, AU, or WAV first, and then load the converted file into a BAESound_ object.

Solo – To solo a sound source or mixer channel is to temporarily turn off the volume of all other sound sources or mixer channels, so that only the soloed items can be heard. The obverse of 'to solo' is 'to unsolo.'

SoundMusicSys – Archaic name for much earlier versions of Mini-BAE.

Stereo Filtering – For applications requiring the highest audio output quality, the BAEMixer_ object furnishes a stereo smoothing reconstruction filter (a lowpass filter). Beatnik recommends not using Stereo Filtering unless you're sure you have cycles to spare, as this feature uses about an extra 1% of your CPU's time.

Tempo – Music playback speed, usually expressed in beats per minute.

Track – A MIDI file can have multiple simultaneously playing tracks, each of which is analogous to a single MIDI connection. Recall that each MIDI connection may have up to 16 channels, and that each MIDI channel may have multiple simultaneously sounding notes. Note that the channels do not span the tracks: A note playing on channel 2 of track 3 will not necessarily use the same instrument as a note playing on channel 2 of track 15.

Voice – In most electronic musical instruments, a voice is whatever hardware and/or software infrastructure is required to keep a single monaural note sounding. Consequently, the number of voices in an instrument is the maximum number of notes the instrument can hold simultaneously. In the case of Mini-BAE and other software synthesizers, a voice consists of a data table slice and a timeslice. You can set Mini-BAE's number of available voices with the symbol **Mini-**

BAE_MAX_VOICES; default is 64 stereo voices.

.WAV – The name of a Pulse Code Modulation (PCM) digital audio file format. Typically a WAV file will have a filename extension of ".wav". Can be loaded into a BAESound_ object.

Appendix 2:

MIDI Implementation

For the Beatnik Software Synthesizer used in Mini-BAE

Model: Beatnik Mini-BAE
Software Wavetable Synthesizer

Date: April 2, 2000
Version: Mini-BAE

Function		Transmitted	Recognized	Remarks
Basic Channel	Default	x	1 - 16	
	Changed	x	1 - 16	
Mode	Default	x	Mode 3	Can't change
	Messages	x	x	
	Altered	**********		
Note Number		x	0-127	
	True Voice	**********	0-127	
Velocity	Note on	x	O	
	Note off	x	x	
After Touch	Keys	x	x	
	Channels	x	x	
Pitch Bend	Pitch Bend	x	*1, *2	Resolution 12 bit
Change Control	0	x	*1, *2	Bank Select (MSB Only)
	1	x	*1, *2	Modulation
	6	x	*1, *2	Data Entry
	7	x	*1, *2	Volume
	10	x	*1, *2	Panpot
	11	x	*1, *2	Expression
	64	x	*1, *2	Hold1 (Sustain)
	85, 86, 87	x	*1, *2	Beatnik Looping & Muting (from files only)
	98, 99	x	*1, *2	NRPN (LSB, MSB)
	100, 101	x	*1, *2	RPN (LSB, MSB) – Reserved
	121	x	*1, *2	Reset All Controllers
	123	x	O	All Notes Off
Program Change		x	*1	
	True Number	**********	0-127	
System Exclusive		x	x	
System Common	Song Position	x	x	
	Song Select	x	x	
	Tune Request	x	x	
System Real Time	Clock	x	x	
	Commands	x	x	
Aux. Messages	Local On/Off	x	x	
	All Notes Off	x	O (123)	
	Active Sensing	x	x	
	System Reset	x	x	
Notes		*1 O x can be selectable O : Yes x : No *2 See BAESong_SetController() function for details on Beatnik's response to Controllers.		
Mode 1: OMNI ON, POLY Mode 2: OMNI ON, MONO Mode 3: OMNI OFF, POLY Mode 4: OMNI OFF, MONO				

Appendix 3:

Return Codes

Interpreting Function Return Values

Here are Mini-BAE's error return codes:

```
/* Common errors returned from the system */
typedef enum {

   BAE_NO_ERROR = 0,

   BAE_PARAM_ERR = 10000,
   BAE_MEMORY_ERR,
   BAE_BAD_INSTRUMENT,
   BAE_BAD_MIDI_DATA,
   BAE_ALREADY_PAUSED,
   BAE_ALREADY_RESUMED,
   BAE_DEVICE_UNAVAILABLE,
   BAE_NO_SONG_PLAYING,
   BAE_STILL_PLAYING,
   BAE_TOO_MANY_SONGS_PLAYING,
   BAE_NO_VOLUME,
   BAE_GENERAL_ERR,
   BAE_NOT_SETUP,
   BAE_NO_FREE_VOICES,
   BAE_STREAM_STOP_PLAY,
   BAE_BAD_FILE_TYPE,
   BAE_GENERAL_BAD,
   BAE_BAD_FILE,
   BAE_NOT_REENTERANT,
   BAE_BAD_SAMPLE,
   BAE_BUFFER_TOO_SMALL,
   BAE_BAD_BANK,
   BAE_BAD_SAMPLE_RATE,
   BAE_TOO_MANY_SAMPLES,
   BAE_UNSUPPORTED_FORMAT,
   BAE_FILE_IO_ERROR,
   BAE_SAMPLE_TOO_LARGE,
   BAE_UNSUPPORTED_HARDWARE,
   BAE_ABORTED,
   BAE_FILE_NOT_FOUND,
   BAE_RESOURCE_NOT_FOUND,
   BAE_NULL_OBJECT,
   BAE_ALREADY_EXISTS,

   BAE_ERROR_COUNT
} BAEResult;
```