VM LABS

NUON **Audio Library Functions**

The new Programmer's Guide

Version 1.0

VM Labs, Inc.
520 San Antonio Road
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Copyright notice

Contents

1. Overview

This document describes the NUON audio libraries. Currently two different audio libraries are part of the SDK:

- NISE (spell: *nice*) - **N**UON **I**nteractive **S**ound **E**ngine : libnise
- NUON Synth - General MIDI synthesizer : libsynth

The following markers indicate in this document if a specific API call is available in a certain library:

SYNTH NISE

The fundamental difference between the two libraries is shown in the below table:

	NISE	**Synth**
MPE usage	top half of MPE 0 part of mini-BIOS	MPE 1 or 2
In-game music	streaming off disc 16 bit stereo 32 kHz	GMIDI 31 voices 32 kHz
Interactive Sound	31 voices Basic API plus: Echo Prologic LFO Filter Distortion	31 voices Basic API plus: Reverb Prologic
Header	nise.h	synth.h

The NUON Synth is a high-quality wavetable synthesizer with the following features:

- full General MIDI set
- 31 voices
- quadruple interpolator
- filter
- high-end reverb and chorus effects

The NUON Synth uses an entire 4k-MPE. You're relativly free in the choice of that MPE you want to use as an audio co-processor. The PCM API offered by the NUON Synth is a subset of the NISE PCM API. The NISE engine on the other hand gives more advanced PCM features and streaming audio. It also ”lives” in MPE 0 mini-BIOS, and leaves MPE 1 and 2 free for other purposes.
Both audio engines run at a sample rate of 32kHz.

2. Audio API Description

2.1 General API

2.1.1 AUDIOInit

SYNTH NISE

`void AUDIOInit(void);`

This call initializes the audio system. Depending on which library you're using...

libsynth ...it starts up the synth on a MPE assigned by the BIOS.

libnise ...it installs NISE into the mini-BIOS on MPE 0

Parameter	Values	Comments
Return value	int	0 = Error Bitfield of available APIs

Bit 0 PCM Engine API

Bit 1 Synth High Level API

Bit 2 Synth MIDI Direct API

Bit 3 Synth Low Level API

Bit 4 Streaming Audio API

2.1.2 AUDIOInitX

SYNTH NISE

`void AUDIOInitX(AUDIO_RESOURCES *res);`

Like **`AUDIOInit`**, but allows to be more flexible in the resource allocation.

```
typedef struct
{
   int audioMPE;
   long sysramBuffer;
   long sdramBuffer;
   long ramWavetableStart;
} AUDIO_RESOURCES;
```

- **audioMPE** What MPE to use for audio. Must be set to a valid MPE number. For LIBNISE this should be the MPE where the mini-BIOS is running (usually MPE 0), for LIBSYNTH it should be the first free MPE.

- **sysramBuffer** SysRAM buffer (220KB for libsynth, 4KB for libnise) – Can be set to NULL if you want the audio library to allocate the buffer for you. Otherwise must be set to a valid SYSRAM address. Do not set to -1.
- **sdramBuffer** SDRAM buffer (8KB on a 8KB boundary!) – Can be set to NULL if you want the audio library to allocate the buffer for you. Otherwise must be set to a valid SDRAM address. Do not set to -1.
- **ramWavetableStart** Start of a RAM based GMIDI wavetable. – This should be set to the address where the SYNTH wavetable data is located in memory. If this is set to NULL, the library will automatically use the data that's linked into the library. This field is only used with the LIBSYNTH library. For LIBNISE it is unused.

2.1.3 AUDIOExit

SYNTH NISE

`void AUDIOExit();`

Stops all audio and frees all resources allocated by **`AUDIOInit`**[2.1.1] or **`AUDIOInitX`**[2.1.2].

2.1.4 AUDIOMixer

SYNTH NISE

`void AUDIOMixer(long musicVolume, long fxVolume);`

Set the master volume for sound effects and background music.

Parameter	**Values**	**Comments**
musicVolume	2.30	Master volume for libnise streaming audio libsynth MIDI
fxVolume	2.30	Master volume for sound effects

2.1.5 AUDIOScale

NISE

`void AUDIOScale(long scale);`

Sets the post-scale level. Since NISE has no dynamic limiter, it is possible that too many simultaneous voices exceed the 16-bit range of the audio samples. This will result in fairly ugly noises.
Warning: The post-scale value is simple way to avoid or at least decrease the problem. However, from a signal processing point of view this method is not correct, since it also scales old (chronological earlier) voices. As result a big post-scale value can have unexpected effects: Overall volume decreases with more voices or voices seem to disappear and re-appear as a kind of echo!
Reasonable values are $0 - 3$.

Parameter	**Values**	**Comments**
scale	0-3	Post-Scale 0: Off 3: Max. scale

2.1.6 *AUDIOInitStreamingAudio*

NISE

`long AUDIOInitStreamingAudio(int handle,long StreamBuffer,`
`long StreamBufferSize);`

Initialize streaming audio device and buffer.

Parameter	**Values**	**Comments**
device	(returned by _MediaOpen)	Device handle used for streaming
StreamBuffer	SysRAM Addr	Start of the stream buffer
StreamBufferSize		Size of the stream buffer

The size of the stream buffer determines how often the audio library has to access the media. It makes sure that streaming audio can survive longer periodes of stopped input data, e.g.

- Scratched disks
- User initiated MediaReads
- ...

Example:
A 2MB stream buffer holds $\frac{2MB}{4\times32000Hz} \approx 16s$. Since the stream buffer is used as a double buffer, media accesses are initiated every 8 seconds.
NOTE: **`AUDIOInitStreamingAudio`**[2.1.6] has to be called once before streaming audio can be started!

2.1.7 *AUDIOSetupStreamingAudio*

NISE

`long AUDIOSetupStreamingAudio(long startSector,long endSector);`

This call starts prefilling the streaming buffer. The **`startSector`** and **`endSector`** define the stream to play. The sector numbers are normally given relative to the **`NUON.DAT`** data file. The file format has to be **Raw, 16-bit, Stereo, Big Endian**!
The following example shows a typical start-up sequence:

```
#define STREAMBUFFERSECTORS 128
#define SECTORSIZE 2048

buffer=(long)malloc(STREAMBUFFERSECTORS*SECTORSIZE);
deviceHandle=_MediaOpen(MEDIA_DVD, "", unused, &unused);

if (AUDIOInitStreamingAudio(deviceHandle,buffer,STREAMBUFFERSECTORS))
{
  for(;;)
  {
      AUDIOSetupStreamingAudio(startSector,startSector+size-1);
      do {
          stat=AUDIOStreamingAudioStatus();
```

```
       } while (!stat);
       if (stat == 1)
          break;
  }
  AUDIOStartStreamingAudio();
}
```

2.1.8 *AUDIOSetupExternalStreamFeeder*

NISE

`void AUDIOSetupExternalStreamFeeder(void (* feeder)(unsigned char *, long))`

This call is an **alternative** to **`AUDIOSetupStreamingAudio`**[2.1.7]. This allows to install a callback function that will deliver the streaming audio data. In this case NISE will not use _MediaRead to load the data by itself:

```
#define STREAMBUFFERSECTORS 128
#define SECTORSIZE 2048

void myFeeder(unsigned char *adr, int bytesNeeded)
{
   // Copy "bytesNeeded" bytes of audio data at "adr"
}

main()
{

  buffer=(long)malloc(STREAMBUFFERSECTORS*SECTORSIZE);

  if (AUDIOInitStreamingAudio(0,buffer,STREAMBUFFERSECTORS))
  {
    AUDIOSetupExternalStreamFeeder(myFeeder);
    AUDIOStartStreamingAudio();
  }
}
```

The feeder callback gets called with a pointer to a buffer and the number of bytes needed. The format of the audio data has to be PCM/32kHz/stereo. **NOTE:** The callback is called at interrupt time, so it should not take too long to complete. The data is picked up by the sound engine via DMA. That means the data has to be copied into the buffer by DMA or the cache has to be flushed!

2.1.9 *AUDIOGetStreamingAudioInfo*

NISE

`long AUDIOGetStreamingAudioInfo(int selector)`

This call is used to retrieve additional information about the state of streaming audio. The selector defines which information is returned:

Selector	Return value
kStreamingInfoCurrentSector	Start sector of next _MediaRead
kStreamingInfoLoopFlag	Looping flag
kStreamingInfoReadError	Number of read errors
kStreamingInfoRetryCount	Number of read retries

NOTE: The returned values are only meaningful when **not** using an external feeder!

2.1.10 AUDIOSetStreamingAudioLooping

NISE

```
void AUDIOSetStreamingAudioLooping(int state)
```

This call turns the looping of streaming audio on or off (default is *On*)

2.1.11 AUDIOStartStreamingAudio

NISE

```
void AUDIOStartStreamingAudio(void);
```

This call start streaming audio. If the loop flag is set the stream loops forever until **`AUDIOStop-StreamingAudio`**[2.1.12] is called.
NOTE: **`AUDIOStartStreamingAudio`** starts streaming at the current position in the stream. To restart from beginning call **`AUDIOSetupStreamingAudio`**[2.1.7] first!

2.1.12 AUDIOStopStreamingAudio

NISE

```
void AUDIOStopStreamingAudio(void);
```

This call stops streaming audio.

2.1.13 AUDIOStreamingAudioStatus

NISE

```
int AUDIOStreamingAudioStatus(void);
```

This call returns the current status of streaming audio:

Values	Comments
0	Start-up, prefilling buffer
1	Stopped, ready to start
2	Running
3	Reached end of stream
-1	Disk error

2.2 PCM API

SYNTH NISE

2.2.1 *PCMPlaySample*

`long PCMPlaySample (long voice, PCMHEAD *sample, PCMPOS *pos, long vol, long echoAmount);`

Turns on a voice, given by a predefined PCMHEAD structure:

```
typedef struct
{
unsigned long  PCMWaveBegin;
unsigned long  PCMLength;
unsigned long  PCMLoopBegin;
unsigned long  PCMLoopEnd;
unsigned long  PCMBaseFreq;
unsigned long  PCMControl;
unsigned long  PCMDistortionAmount;
unsigned long  PCMFilterInputTrim;
unsigned long  PCMFilterFrequency;
unsigned long  PCMFilterResonance;
unsigned long  PCMFilterType;
} PCMHEAD;
```

All the positioning parameters are stored in the PCMPOS structure:

```
typedef struct
{
unsigned long  PCMPanLR;
unsigned long  PCMPanFR;
unsigned long  PCMPanUD;
unsigned long  Reserved1;    //set to zero!!
} PCMPOS;
```

Offset	Name	Format	Description
0	Sample Address	32 bit	
4	Sample Length	32 bit	Length in samples
8	Sample Frequency	3.13	$0x2000 = 48kHz$
			$0x1000 = 24kHz$
12	Sample Control	Bitfield	*see below*
16	Loop Start	32 bit	Loop start in samples
20	Loop End	32 bit	Loop end in samples

Sample control bitfield:

Bit	Name	Description
0	Loop Bit	1 = Looping on
1-3	Format	000 = 16bit PCM mono
		001 = 8bit NSC mono
		all other values are reserved
4-31	Reserved	Set to zero

Parameter	**Values**	**Comments**
voice		Voice number to use for this sample $0 \dots 30$
		if you want to automatically assign a voice: -1
sample	SysRAM Addr	Pointer to a PCMHEAD structure
pos	SysRAM Addr	Pointer to a PCMPOS structure
vol	2.30	Gain for this voice $max = 0x40000000 = 1.0$
echoAmount	2.30	Echo/Reverb amount $max = 0x40000000 = 1.0$
Returns:		
	0 to 30	Voice Handle

You can define a front/rear panning parameter. The encoding is Dolby Prologic compatible and produces a 3D audio effect even on a two speaker stereo setup. Due to the restrictions of ProLogic you can place the voice only within a triangle. There is only one rear channel, even when you use two rear speakers. As a result the left/right panning has less effect when moving towards the rear channel (and has no effect when your voice is panned completly to the rear channel).
The up/down panning parameter has no function in the current version of the audio libraries. The reverb parameter defines the reverb depth added to the voice. In *libsynth* this is using a real reverb, in *libnise* it's just a simple echo!

2.2.2 PCMVoiceOn

`long PCMVoiceOn (void *startAddr, long frequency, long len, long vol, long panLR, long panFR);`

Turns on a voice. This call is useful for simple sound effects without looping (e.g. gunshots). You don't have to setup a PCMHEAD structure.

Parameter	**Values**	**Comments**
startAddr	SysRAM addr	Start of the sample in SysRAM
frequency	3.13	Playback frequency
		$0x2000 = 48kHz$
		$0x1000 = 24kHz$
len		Length in samples
vol	2.30	Gain for this voice $max = 0x40000000 = 1.0$
panLR	2.30	Left/Right Pan $middle = 0$
panFR	2.30	Front/Rear Pan $middle = 0$
Returns:		
	Voice Handle	

2.2.3 PCMVoiceOff

`void PCMVoiceOff (long voiceHandle);`

Kills a voice

Parameter	**Values**	**Comments**
voiceHandle	0 to 30	Voice Handle

2.2.4 PCMGetUsedVoices

`long PCMGetUsedVoices (void);`

Returns a bitfield indicating which voices are currently playing.

Bit 0	Voice 0
Bit 1	Voice 1
...	
Bit 31	Voice 31

2.2.5 PCMSetVolume

`void PCMSetVolume (long voiceHandle, long vol);`

Changes the volume of a running voice

Parameter	**Values**	**Comments**
voiceHandle	0 to 30	Voice Handle
vol	2.30	New gain for this voice $max = 0x40000000 = 1.0$

2.2.6 PCMSetPanLR

`void PCMSetPanLR (long voiceHandle, long panLR);`

Changes the left/right pan of a running voice

Parameter	**Values**	**Comments**
voiceHandle	0 to 30	Voice Handle
panLR	2.30	New left/right pan $middle = 0$ $left = 0xC0000000 = -1.0$ $right = 0x40000000 = 1.0$

2.2.7 PCMSetPanFB

`void PCMSetPanFB (long voiceHandle, long panFR);`

Changes the front/back pan of a running voice

Parameter	**Values**	**Comments**
voiceHandle	0 to 30	Voice Handle
panFR	2.30	New front/back pan $middle = 0$ $back = 0xC0000000 = -1.0$ $front = 0x40000000 = 1.0$

2.2.8 PCMSetPitch

`void PCMSetPitch (long voiceHandle, long pitch);`

Changes the pitch of a running voice

Parameter	**Values**	**Comments**
voiceHandle	0 to 30	Voice Handle
pitch	3.13	New playback frequency $0x2000 = 48kHz$ $0x1000 = 24kHz$

The following API calls are only available with the *libnise* library! NISE

2.2.9 PCMSetEchoAmount

`void PCMSetEchoAmount (long voice, long echoAmount);`

Sets the amount of echo for given voice. The amount value is actually pan value between the dry (no echo) signal and the wet (echo) signal.

Parameter	Values	Comments
voice	0 to 30	Voice Handle
echoAmount	2.30	Echo amount $0x00000000 = 0.0$ no echo $0x40000000 = 1.0$ only echo

NOTE: Echo has to be enabled. See **`PCMEnableFeature`**[2.2.13] and **`PCMEnableGlobalFeature`**[2.2.15].

2.2.10 PCMSetLFO

`void PCMSetLFO (long voice, long amplitude, long freq);`

This call sets the LFO (Low Frequency Oscillator) parameters. The LFO produces a triangular waveform with a given amplitude.

Parameter	Values	Comments
voice	0 to 30	Voice Handle
amplitude	2.30	Amplitude
freq		Frequency in Hz

The LFO output can be added to various other parameters, but no range checking is performed! The LFO generator has to be enabled. See **`PCMEnableFeature`**[2.2.13].

2.2.11 PCMSetDistortionAmount

`void PCMSetDistortionAmount (long voice, long distAmount);`

This call sets the distortion amount.

Parameter	Values	Comments
voice	0 to 30	Voice Handle
distAmount	16.16	Disortion amount $0x00010000 = 0.0$ no distortion $0x00040000 = 4.0$ max distortion

NOTE: The distortion module has to be enabled. See **`PCMEnableFeature`**[2.2.13].

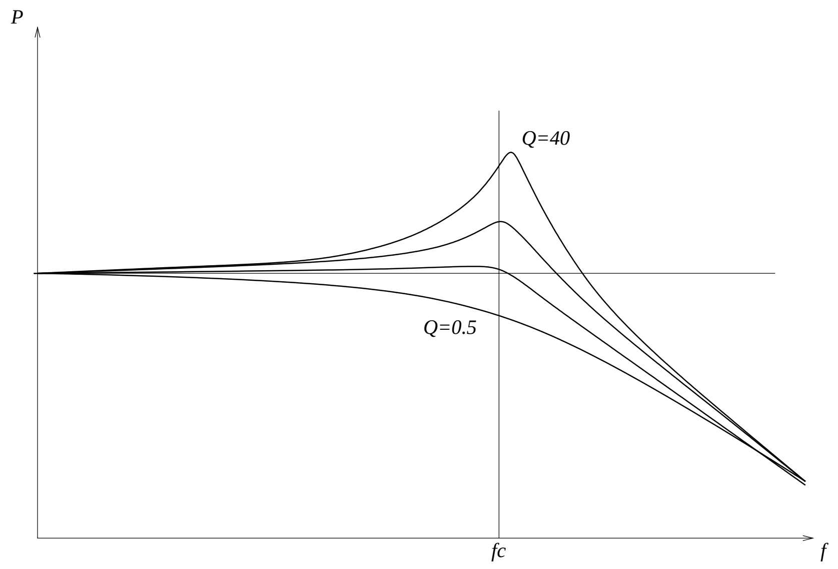
2.2.12 *PCMSetFilter*

`void PCMSetFilter (long voice, long f1, long Q1, long type);`

This call sets the filter parameters.

Parameter	**Values**	**Comments**
voice	0 to 30	Voice Handle
f1	2.30	Cut off frequency $0x40000000 = 1.0$ = 5kHz
Q1	2.30	Resonance factor Range 2.0 to 0
type	0,1,2	Filter type 0: Low Pass 1: High Pass 2: Band Pass

The following two formulas show how to calculate the real f and Q from the given F_1 and Q_1 parameters:
$f = \frac{F_1 F_s}{2\pi}$
$Q = \frac{1}{Q_1}$
The below graph shows the effect of the resonance Q in a low pass filter:



The NISE filter module has fairly good low pass filter characteristics, but consider the high pass and band pass filter type to be more ”experimental”.
NOTE: The filter module has to be enabled. See **`PCMEnableFeature`**[2.2.13].

2.2.13 *PCMEnableFeature*

`void PCMEnableFeature (long voice, long feature);`

This call is used to enable a certain feature on a given voice.

Parameter	Values	Comments
voice	0 to 30	Voice Handle
feature		Feature to enable

The following features are available:

`kNISELFO`	Enable LFO
`kNISEDistortion`	Enable Distortion
`kNISEFilter`	Enable Filter
`kNISEInterpolation`	Enable Lineare Interpolation (default: on)
`kNISELFOModFilterF`	Add LFO to filter frequency
`kNISELFOModFilterQ`	Add LFo to filter resonance
`kNISELFOModGain`	Add LFO to gain
`kNISELFOModPan`	Add LFO to left/right pan
`kNISELFOModDist`	Add LFO to distortion amount

NOTE: No range checking is done after adding the LFO value to any of the above parameters. Make sure they don't exceed the specified range!

2.2.14 *PCMDisableFeature*

```
void PCMDisableFeature (long voice, long feature);
```

This call is used to disable a certain feature on a given voice.

Parameter	Values	Comments
voice	0 to 30	Voice Handle
feature		Feature to disable

2.2.15 *PCMEnableGlobalFeature*

```
void PCMEnableGlobalFeature(long feature);
```

This call is used to enable global features.

Parameter	Values	Comments
feature		Global feature to enable

The following features are available:

`kNISEEcho`	Enable Echo
`kNISEPrologic`	Enable Prologic compatible encoding
`kNISEStreamingAudio`	Enable Streaming Audio

Note: Streaming audio is normally enabled and disabled by **`AUDIOStartStreamingAudio`**[2.1.11] anf **`AUDIOStopStreamingAudio`**[2.1.12].

2.2.16 PCMDisableGlobalFeature

`void PCMDisableGlobalFeature(long feature);`

This call is used to disable global features.

Parameter	**Values**	**Comments**
feature		Global feature to disable

2.2.17 PCMEchoFeedback

`void PCMEchoFeedback(long feedback);`

This call defines the global echo feedback parameter:

Parameter	**Values**	**Comments**
feedback	2.30	Echo Feedback $0x00000000 = 0.0$ no feedback $0x40000000 = 1.0$ max. feedback

2.3 Synth High Level API

2.3.1 SYNTHConfig

`void SYNTHConfig(long maxVoices, long mpeUsage, long reverbOn, long chorusOn);`

Configuration of synth performance parameters. They can be used to reduce the system resource usage of the synth:

Parameter	Values	Comments
maxVoices	0 to 32	Maximum number of voices (default: 32)
mpeUsage	0 to 100	Maximum usage of the MPE in % (default: 100%)
reverbOn	0/1	1 = Reverb on, 0 = Reverb off (default: On)
chorusOn	0/1	1 = Chorus on, 0 = Chorus off (default: On)

2.3.2 SYNTHInstallCB

`void SYNTHInstallCB(void (*SYNTHCallback) (long p0,long p1,long p2,long p3));`

Installs a commbus callback. The synth sends commbus packets to the application to signal certain events: e.g. end of midi file, midi feedback etc. **NOTE:** The callback is called from the BIOS level 1 interrupt service routine. Make sure you're spending not too much time in this callback!

p0 - p3 commbus packet (4 long values)

The generic format of a response is

p0	data0 (depends on ID)
p1	data1 (depends on ID)
p2	data2 (depends on ID)
p3	Bits 0-23: data3 (depends on ID) Bits 24-31: ID

The ID specifies the type of packet and defines the format for the rest of the packet.

ID 0x00 End of MIDI track

p0	0
p1	0
p2	0
p3	Bits 0-23: 0 Bits 24-31: 0x00

ID 0x10 MIDI Event Feedback

p0	MIDI parameter 1
p1	MIDI parameter 2
p2	MIDI channel
p3	Bits 0-23: MIDI command Bits 24-31: 0x10

ID 0x80 Lyrics event

p0	text bytes 0-4
p1	text bytes 5-7
p2	text bytes 8-11
p3	Bits 0-23: 0 Bits 24-31: 0x80

2.3.3 SYNTHInfo

`void SYNTHInfo(long extended,long answer[]);`

Returns a packet containing various synth infos and states:

extended 0 = standard info, 1 = extended info

answer This parameter should point to a buffer with room for one comm bus packet (four 32bit values)

extended = 0	
answer[0]	Protocol Version
answer[1]	First address used by the synth
answer[2]	Start of the database
answer[3]	Database Revision

extended = 1	
answer[0]	Used Voices
answer[1]	*only for internal use!*
answer[2]	*only for internal use!*
answer[3]	Parser State

Protocol Version:

Currently V 1.20

Database Revision:

Currently Rev. 1

2.3.4 SYNTHInstallBank

`void SYNTHInstallBank(long bankNumber, long *databaseStart);`

Integrates a synth bank (created by the voiceing tool) into the MIDI parser.

Parameter	Values	Comments
bankNumber	0 to 127	MIDI bank number
addr	SysRAM addr	Address of the bank in SysRAM

2.3.5 *SYNTHNextMidiFile*

`void SYNTHNextMidiFile(long addr);`

Specifies the address of the next MIDI file to play after the current one has finished. **Note:** This call has no effect if no MIDI file is running.

Parameter	Values	Comments
addr		Address of a MIDI file in SysRAM or SDRAM

NOTE: The MIDI file has to be in format 0, otherwise it will be ignored!
The address 0 turns off this feature - **`SYNTHStartMidiParser`**[2.3.7] is doing that automatically.
To loop a MIDI file endless, just call **`SYNTHStartMidiParser`**[2.3.7] and **`SYNTHNextMidiFile`**[2.3.5] with the same address.
To loop through a number of MIDI files initialize **`SYNTHStartMidiParser`** [2.3.7] and **`SYNTHNextMidiFile`** with the addresses of the first and second MIDI file. Use the **`SYNTHInstallCB`** [2.3.2] function to install a callback. Within this callback you check for the **`END_OF_TRACK_ID`** packet and call **`SYNTHNextMidiFile`** [2.3.5] with the next address.

2.3.6 *SYNTHStartMidiParserFeedback*

`void SYNTHStartMidiParserFeedback(long addr,long mpe,long channels,long commands);`

Starts the MIDI parser with additional feedback parameters. With the parameters you specify which MIDI events are send back by the MIDI parser over the comm bus. **Note:** The parser is sending these packets only if the receiving MPE is ready to receive the packets!

Parameter	Values	Comments
addr		Address of a MIDI file in SysRAM or SDRAM.
mpe	0 to 3	MPE which receives the comm bus packets
channels		A bitfield to specifiy the MIDI channels of which the commands are send. Bit 0 corresponds to MIDI channel 1, Bit 15 corresponds to MIDI channel 16.
commands		Specifies the biggest MIDI command which is send. Example: 0x80 would enable only Note-Off commands, 0x90 enables Note-On and Note-Offs, 0xa0 enables Note-on/off and Aftertouch ...

NOTE: The MIDI file has to be in format 0, otherwise it will be ignored! For the format of the packet see **`SYNTHInstallCB`**[2.3.2] :

2.3.7 SYNTHStartMidiParser

`void SYNTHStartMidiParser(long addr);`

Starts the MIDI parser.

addr Address of a MIDI file in SysRAM or SDRAM.

NOTE: The MIDI file has to be in format 0, otherwise it will be ignored!

2.3.8 SYNTHStopMidiParser

`void SYNTHStopMidiParser(void);`

Stops the MIDI parser and mutes all synth voices.

2.3.9 SYNTHMidiSendEvents

`void SYNTHMidiSendEvents(unsigned char *events,long len);`

Sends a MIDI event string directly to the MIDI state machine. This call will be ignored while the MIDI parser is running.

events Pointer to the MIDI event(s)

len Length of the MIDI events in bytes

Example Turns on a note (middle C, velocity 127)

```
unsigned char noteon[]={0x90,60,127};
SYNTHMidiSendEvents(noteon,3);
```

2.4 Synth MIDI Direct API

These calls are talking directly to the MIDI state machine. But compared to the **`SYNTHMidiSendEvents`**[2.3.9] call which uses the standard MIDI event format, every of these calls represents a MIDI event. There is no need to build a MIDI event in memory and there is no 7-bit notation limitation like in the MIDI events. This gives room for upcoming improvements like:

- more than 16 MIDI channels
- microtonal support (fractional note numbers)

NOTE While the parser is running **all** API calls to the MIDI Direct layer are ignored!
The format of all parameters (like note number, channel and controller values) in the following API calls, are according to the MIDI spec!

2.4.1 *SYNTHMidiNoteOn*

```
void SYNTHMidiNoteOn(long channel, long note, long velocity);
```

Turns on a note on a MIDI channel

channel	0 . . . 15 for MIDI channel 1 . . . 16
note	0 . . . 127 note number
velocity	0 . . . 127 velocity

2.4.2 *SYNTHMidiNoteOff*

```
void SYNTHMidiNoteOff(long channel, long note, long velocity);
```

Turns off a note on a MIDI channel

channel	0 . . . 15 for MIDI channel 1 . . . 16
note	0 . . . 127 note number
velocity	0 . . . 127 velocity (ignored)

2.4.3 *SYNTHMidiPitchBend*

```
void SYNTHMidiPitchBend(long channel, long value);
```

Send a pitch bend value to a MIDI channel

channel	0 . . . 15 for MIDI channel 1 . . . 16
value	0 . . . 16383 pitch bend value (8192 is center)

2.4.4 *SYNTHMidiControlChange*

```
void SYNTHMidiControlChange(long channel, long no,long value);
```

Changes a controller on a MIDI channel

channel	0 . . . 15 for MIDI channel 1 . . . 16
no	controller number (see below)
value	0 . . . 127 new value

Currently the following MIDI controllers are supported:

0	Bank select
1	Modulation
7	Volume
6	Data Entry
10	Pan (64 is center, 0 is left, 127 is right)

11	Expression
64	Sustain
68	Legato
91	Reverb
93	Chorus
100	RPN (only 0 = ’pitchbend scale’ supported)
121	Reset Controllers
123	All Notes Off

Please refer to the MIDI specification to get more information about this controllers.

2.4.5 SYNTHMidiProgramChange

`void SYNTHMidiProgramChange(long channel,long patch);`

Changes the instrument on a MIDI channel

channel	0 . . . 15 for MIDI channel 1 . . . 16
patch	0 . . . 127 patch number

2.5 Synth Low Level API

This class of API calls are talking to the low level interface of the synth. It bypasses the MIDI parser and MIDI state machine. Every note you turn on is assigned to a handle which is returned by the **`SYNTH-NoteOn`** call. You use this handle to modify this voice with subsequent calls of the other API Low Level calls.

2.5.1 SYNTHNoteOn

`long SYNTHNoteOn(long patch, long note, long velocity);`

Turns on a voice

patch	0 . . . 127 patch number
note	0 . . . 127 note number
velocity	0 . . . 127 velocity
Returns:	
	voice handle

2.5.2 SYNTHChangePara

`void SYNTHChangePara(long voicelist,long no,long value);`

Turns off a voice

handle	voice handle

2.5.3 SYNTHNoteOff

```
void SYNTHNoteOff(long voicelist);
```

Turns off a voice

handle	voice handle

2.5.4 SYNTHPitchBend

```
void SYNTHPitchBend(long voicelist,long value);
```

Changes the pitch bend value for a voice

handle	voice handle
value	0 . . . 16383 new pitch bend value (8192 is center)

2.5.5 SYNTHModWheel

```
void SYNTHModWheel(long voicelist,long value);
```

Changes the modulation wheel value for a voice

handle	voice handle
value	0 . . . 127 new wheel value

2.5.6 SYNTHVolume

```
void SYNTHVolume(long voicelist,long value);
```

Changes the volume for a voice

handle	voice handle
value	0 . . . 127 new volume

2.5.7 SYNTHPan

```
void SYNTHPan(long voicelist,long value);
```

Changes the pan value for a voice

handle	voice handle
value	0 . . . 127 new pan value (64 is center, 0 is left, 127 is right)

2.5.8 SYNTHChorus

```
void SYNTHChorus(long voicelist,long value);
```

Changes the chorus amount for a voice

handle	voice handle
value	0 . . . 127 new chorus value

2.5.9 SYNTHReverb

```
void SYNTHReverb(long voicelist,long value);
```

Changes the reverb amount for a voice

handle	voice handle
value	0 . . . 127 new reverb value

3. Inside NISE

This chapter gives some insides into the NISE library. This should help to tune the application and make optimal use of the library. It also shows limitation that have to be considered by the application programmer.

3.1 Performance

The following table gives a performance estimate of various NISE modules. The number represent the time to generate or process 64 samples:

	Ticks
Plain voice	1120
Module	**additional Ticks**
NSC decompression	+260
Lin. interpolation	+60
LFO	+20
Distortion	+900
Filter	+1736
Echo	+332
Prologic	+332
Streaming Audio	+338
max.	5098

$MIPS = \frac{ticks \times 32000}{64}$:

	Ticks	MIPS	MPE usage	max. number of voices
Plain voice	1120 ticks	.56 MIPS	1%	100 voices (limited to 31 by NISE)
Full featured voice	5098 ticks	2.5 MIPS	4.7%	21 voices

3.2 Limitations

A few things should be mentioned about the implementation of NISE. In order to be as less intrusive as possible and given the limited amount of resources, we had to take a few short cuts a programmer has to be aware of. The main limitation is the lack of a dynamic limiter. Normally a dynamic limiter makes sure that the internal 32-bit wide registers do not overflow (better: wrap) when too many voices are enabled. In NISE it's the responsibility of the programmer to balance the individual gains of the PCM voices and streaming audio. However, the **`AUDIOScale`** function (NISE Version 1.52 and up) can be used to cheat and possibly hide the problem.
The second limitation is that there is no dynamic voice killing scheme. It's possible to overload MPE 0 with too many voices and features. The result is a stalling user task and probably unintended sounds. Again it's the responsibility of the programmer to balance the available MIPS in MPE 0.

Index