AHI

Hardware independent audio for Amiga
AHI Developer's Guide for AHI version 4.20
Document version 4.11 (1999-03-28)

Martin 'Leviticus' Blom

1 Overview

This document was written in order to make it easier for developers to understand and use AHI in their own productions, and write Software That Works(TM).

`ahi.device` has two different API's; one library-like function interface (low-level), and one "normal" device interface (high-level). Each of them serves different purposes. The low-level interface is targeting music players, games and real-time applications. The high-level interface is targeting applications that just want to have a sample played, play audio streams or record samples as easily as possible.

As with everything else, it is important that you chose the right tool for the job—you'll only get frustrated otherwise.

Not everything about AHI is documented here; for more information, see AHI *User's Guide* and the autodocs.

2 Distribution

AHI is available as *freeware*. That is, it may be freely distributed in unmodified form with no changes what so ever, but you may not charge more than a nominal fee covering distribution costs. However, donations are welcome (see AHI *User's Guide*).

If you use this software in a commercial or shareware product, please consider giving the author (see Chapter 3 [The Author], page 5)—and preferably each one of the contributors too (see AHI *User's Guide*)—an original or registered version of your work. Should you want to distribute the AHI software with your own product, there is really nothing to consider, is it?

If you wish to distribute this software with a hardware product, contact the author (see Chapter 3 [The Author], page 5). Distribution of AHI with hardware products is *not* free.

3 The Author

The author can be reached at the following addresses:

Electronic mail
`martin@blom.org`

Standard mail
Martin Blom
Ålsättersgatan 15A:24
SE-584 35 Linköping
SWEDEN

World-Wide Web
`http://martin.blom.org`

4 Definitions

Following are some general definitions of terms that are used in this document.

Sample A sample is one binary number, representing the amplitude at a fixed point in time. A sample is often stored as an 8 bit signed integer, a 16 bit signed integer, a 32 bit floating point number etc. AHI only supports integers.

Sample frame
In mono environments, a sample frame is the same as a sample. In stereo environments, a sample frame is a tuple of two samples. The first member is for the left channel, the second for the right.

Sound Many sample frames stored in sequence as an array can be called a sound. A sound is, however, not limited to being formed by samples, it can also be parameters to an analog synth or a MIDI instrument, or be white noise. AHI only supports sounds formed by samples.

5 Function Interface

The device has, in addition to the usual I/O request protocol, a set of functions that allows the programmer to gain full control (at least as much as possible with device independence) over the audio hardware. The advantages are low overhead and much more advanced control over the playing sounds. The disadvantages are greater complexity and only one user per sound card.

If you want to play music or sound effects for a game, record in high quality or want to do realtime effects, this is the API to use.

5.1 Guidelines

5.1.1 Follow The Rules

It's really simple. If I tell you to check return values, check sample types when recording, not to trash d2-d7/a2-a6 in hooks, or not to call `AHI_ControlAudio()` with the `AHIC_Play` tag from interrupts or hooks, you do as you are told.

5.1.2 The Library Base

The `AHIBase` structure is private, so are the sub libraries' library base structures. Don't try to be clever.

5.1.3 The Audio Database

The implementation of the database is private, and may change any time. `ahi.device` provides functions access the information in the database (`AHI_NextAudioID()`, `AHI_GetAudioAttrsA()` and `AHI_BestAudioIDA()`).

5.1.4 User Hooks

All user hooks must follow normal register conventions, which means that d2-d7 and a2-a6 must be preserved. They may be called from an interrupt, but you cannot count on that; it can be your own process or another process. Don't assume the system is in single-thread mode. Never spend much time in a hook, get the work done as quick as possible and then return.

5.1.5 Function Calls From Other Tasks, Interrupts Or User Hooks

The `AHIAudioCtrl` structure may not be shared with other tasks/threads. The task that called `AHI_AllocAudioA()` must do all other calls too (except those callable from interrupts).

Only calls specifically said to be callable from interrupts may be called from user hooks or interrupts. Note that `AHI_ControlAudioA()` has some tags that must not be present when called from an interrupt.

5.1.6 Multitasking

Most audio drivers need multitasking to be turned on to function properly. Don't turn it off while using the device.

5.2 Opening And Closing `ahi.device` For Low-level Access

Not too hard. Just open `ahi.device` unit `AHI_NO_UNIT` and initialize `AHIBase`. After that you can access all the functions of the device just as if you had opened a standard shared library.

5.2.1 Assembler

For the assembler programmer there are two handy macros: `OPENAHI` and `CLOSEAHI`. Here is a small example how to use them:

```
        OPENAHI 4                 ;Open at least version 4.
        lea     _AHIBase(pc),a0
        move.l  d0,(a0)
        beq     error

; AHI's functions can now be called as normal library functions:
        move.l  _AHIBase(pc),a6
        moveq   #AHI_INVALID_ID,d0
        jsr     _LVOAHI_NextAudioID(a6)

error:
        CLOSEAHI
        rts
```

Note that you **have** to execute the `CLOSEAHI` macro even if `OPENAHI` failed!

5.2.2 C

For the C programmer, here is how it should be done:

```
struct Library    *AHIBase;
struct MsgPort    *AHImp=NULL;
struct AHIRequest *AHIio=NULL;
BYTE               AHIDevice=-1;

if(AHImp = CreateMsgPort())
{
  if(AHIio = (struct AHIRequest *) CreateIORequest(
      AHImp, sizeof(struct AHIRequest)))
  {
    AHIio->ahir_Version = 4;  /* Open at least version 4. */
    if(!(AHIDevice = OpenDevice(AHINAME, AHI_NO_UNIT,
        (struct IORequest *) AHIio, NULL)))
    {
      AHIBase = (struct Library *) AHIio->ahir_Std.io_Device;

// AHI's functions can now be called as normal library functions:
      AHI_NextAudioID(AHI_INVALID_ID);

      CloseDevice((struct IORequest *) AHIio);
      AHIDevice = -1;
    }
    DeleteIORequest((struct IORequest *) AHIio);
    AHIio = NULL;
  }
```

```
    DeleteMsgPort(AHImp);
    AHImp = NULL;
  }
```

5.3 Obtaining The Hardware

If you wish to call any other function than

- `AHI_AllocAudioRequestA()`
- `AHI_AudioRequestA()`
- `AHI_BestAudioIDA()`
- `AHI_FreeAudioRequest()`
- `AHI_GetAudioAttrsA()`
- `AHI_NextAudioID()`
- `AHI_SampleFrameSize()`

...you have to allocate the actual sound hardware. This is done with `AHI_AllocAudioA()`. `AHI_AllocAudioA()` returns an `AHIAudioCtrl` structure, or `NULL` if the hardware could not be allocated. The `AHIAudioCtrl` structure has only one public field, `ahiac_UserData`. This is unused by AHI and you may store anything you like here.

If `AHI_AllocAudioA()` fails it is important that you handle the situation gracefully.

When you are finished playing or recording, call `AHI_FreeAudio()` to deallocate the hardware and other resources allocated by `AHI_AllocAudioA()`. `AHI_FreeAudio()` also deallocates all loaded sounds (see Section 5.4 [Declaring Sounds], page 12).

5.3.1 `AHI_AllocAudioA()` Tags

`AHI_AllocAudioA()` takes several tags as input.

`AHIA_AudioID`
: This is the audio mode to be used. You must not use any hardcoded values other than `AHI_DEFAULT_ID`, which is the user's default fallback ID. In most cases you should ask the user for an ID code (with `AHI_AudioRequestA()`) and then store the value in your settings file.

`AHIA_MixFreq`
: This is the mixing frequency to be used. The actual frequency will be rounded to the nearest frequency supported by the sound hardware. To find the actual frequency, use `AHI_GetAudioAttrsA()`. If omitted or `AHI_DEFAULT_FREQ`, the user's preferred fallback frequency will be used. In most cases you should ask the user for a frequency (with `AHI_AudioRequestA()`) and then store the value in your settings file.

`AHIA_Channels`
: All sounds are played on a *channel*, and this tag selects how many you wish to use. In general it takes more CPU power the more channels you use and the volume gets lower and lower.

`AHIA_Sounds`
: You must tell AHI how many different sounds you are going to play. See Section 5.4 [Declaring Sounds], page 12 for more information.

`AHIA_SoundFunc`
: With this tag you tell AHI to call a hook when a sound has been started. It works just like Paula's audio interrupts. The hook receives an `AHISoundMessage` structure as message. `AHISoundMessage->ahism_Channel` indicates which channel the sound that caused the hook to be called is played on.

`AHIA_PlayerFunc`
: If you are going to play a musical score, you should use this "interrupt" source instead of VBLANK or CIA timers in order to get the best result with all audio drivers. If you cannot use this, you must not use any "non-realtime" modes (see `AHI_GetAudioAttrsA()` in the autodocs, the `AHIDB_Realtime` tag).

`AHIA_PlayerFreq`
: If non-zero, it enables timing and specifies how many times per second `PlayerFunc` will be called. This must be specified if `AHIA_PlayerFunc` is! It is suggested that you keep the frequency below 100-200 Hz. Since the frequency is a fixpoint number `AHIA_PlayerFreq` should be less than 13107200 (that's 200 Hz).

`AHIA_MinPlayerFreq`
: The minimum frequency (`AHIA_PlayerFreq`) you will use. You should always supply this if you are using the device's interrupt feature!

`AHIA_MaxPlayerFreq`
: The maximum frequency (`AHIA_PlayerFreq`) you will use. You should always supply this if you are using the device's interrupt feature!

`AHIA_RecordFunc`
: This hook will be called regularly when sampling is turned on (see `AHI_ControlAudioA()`). It is important that you always check the format of the sampled data, and ignore it if you can't parse it. Since this hook may be called from an interrupt, it is not legal to directly `Write()` the buffer to disk. To record directly to harddisk you have to copy the samples to another buffer and signal a process to save it. To find out the required size of the buffer, see `AHI_GetAudioAttrsA()` in the autodocs, the `AHIDB_MaxRecordSamples` tag.

`AHIA_UserData`
: Can be used to initialize the `ahiac_UserData` field. You do not have to use this tag to change `ahiac_UserData`, you may write to it directly.

5.4 Declaring Sounds

Before you can play a sample array, you must `AHI_LoadSound()` it. Why? Because if AHI knows what kind of sounds that will be played later, tables and stuff can be set up in advance. Some drivers may even upload the samples to the sound cards local RAM and play all samples from there, drastically reducing CPU and bus load.

You should `AHI_LoadSound()` the most important sounds first, since the sound cards RAM may not be large enough to hold all your sounds.

`AHI_LoadSound()` also associates each sound or sample array with a number, which is later used to refer to that particular sound.

There are 2 types of sounds, namely `AHIST_SAMPLE` and `AHIST_DYNAMICSAMPLE`.

`AHIST_SAMPLE`
: This is used for static samples. Most sounds that will be played are of this type. Once the samples has been "loaded", you may not alter the memory where the samples are located. You may, however, read from it.

`AHIST_DYNAMICSAMPLE`
: If you wish to play samples that you calculate in realtime, or load in portions from disk, you must use this type. These samples will never be uploaded to a sound cards local RAM, but always played from the normal memory. There is a catch, however. Because of the fact that the sound is mixed in chunks, you must have a certain number of samples in memory before you start a sound of this type. To calculate the size of the buffer (in samples), use the following formula:

$$size = samples * Fs / Fm$$

where samples is the value returned from `AHI_GetAudioAttrsA()` when called with the `AHIDB_MaxPlaySamples` tag, Fs is the highest frequency the sound will be played at and Fm is the actual mixing frequency (`AHI_ControlAudioA()/AHIC_MixFreq_Query`).

The samples can be in one of four different formats, named `AHIST_M8S`, `AHIST_S8S`, `AHIST_M16S`, and `AHIST_S16S`.

`AHIST_M8S`
: This is an 8 bit mono sound. Each sample frame is just one signed byte.

`AHIST_S8S`
: This is an 8 bit stereo sound. Each sample frame is one signed byte representing the left channel, followed by another one for the right channel.

`AHIST_M16S`
: This is a 16 bit mono sound. Each sample frame is just one signed 16 bit word, in big endian/network order format (most significant byte first).

`AHIST_S16S`
: This is a 16 bit stereo sound. Each sample frame is one signed 16 bit word, in big endian/network order format (most significant byte first) representing the left channel, followed by another one for the right channel.

If you know that you won't use a sound anymore, call `AHI_UnloadSound()`. `AHI_FreeAudio()` will also do that for you for any sounds left when called.

There is no need to place a sample array in *Chip memory*, but it **must not** be swapped out! Allocate your sample memory with the `MEMF_PUBLIC` flag set. If you wish to have your samples in virtual memory, you have to write a double-buffer routine that copies a chunk of memory to a `MEMF_PUBLIC` buffer. The *SoundFunc* should signal a task to do the transfer, since it may run in supervisor mode (see `AHI_AllocAudioA()`).

5.5 Making Noise

After you have allocated the sound hardware and declared all your sounds, you're ready to start playback. This is done with a call to `AHI_ControlAudioA()`, with the `AHIC_Play` tag set to `TRUE`. When this function returns the *PlayerFunc* (see `AHI_AllocAudioA()`) is active, and the audio driver is feeding silence to the sound hardware.

5.5.1 Playing A Sound

All you have to do now is to set the desired sound, it's frequency and volume. This is done with `AHI_SetSound()`, `AHI_SetFreq()` and `AHI_SetVol()`. Make sure the `AHISF_IMM` flag is set for all these function's *flag* argument. And don't try to modify a channel that is out of range! If you have allocated 4 channels you may only modify channels 0-3.

The sound will not start until both `AHI_SetSound()` and `AHI_SetFreq()` has been called. The sound will play even if `AHI_SetVol()` was not called, but it will play completely silent. If you wish to temporary stop a sound, set its frequency to 0. When you change the frequency again, the sound will continue where it were.

When the sound has been started it will play to the end and then repeat. In order to play a one-shot sound you have use the `AHI_PlayA()` function, or install a sound interrupt using the `AHIA_SoundFunc` tag with `AHI_AllocAudioA()`. For more information about using sound interrupts, see below.

A little note regarding `AHI_SetSound()`: *Offset* is the first sample that will be played, both when playing backwards and forwards. This means that if you call `AHI_SetSound()` with *offset*

0 and *length* 4, sample 0,1,2 and 3 will be played. If you call `AHI_SetSound()` with *offset* 3 and *length* −4, sample 3,2,1 and 0 will be played.

Also note that playing very short sounds will be very CPU intensive, since there are many tasks that must be done each time a sound has reached its end (like starting the next one, calling the *SoundFunc*, etc.). Therefore, it is recommended that you "unroll" short sounds a couple of times before you play them. How many times you should unroll? Well, it depends on the situation, of course, but try making the sound a thousand samples long if you can. Naturally, if you need your *SoundFunc* to be called, you cannot unroll.

5.5.2 Playing One-shot Sounds And Advanced Loops

Some changes has been made since earlier releases. One-shot sounds and sounds with only one loop segment can now be played without using sample interrupts. This is possible because one of the restrictions regarding the `AHISF_IMM` flag has been removed.

The `AHISF_IMM` flag determines if `AHI_SetSound()`, `AHI_SetFreq()` and `AHI_SetVol()` should take effect immediately or when the current sound has reached its end. The rules for this flags are:

- If used inside a sample interrupt (*SoundFunc*): Must be cleared.
- If used inside a player interrupt (*PlayerFunc*): May be set or cleared.
- If used elsewhere: Must be set.

What does this mean? It means that if all you want to do is to play a one-shot sound from inside a *PlayerFunc*, you can do that by first calling `AHI_SetSound()`, `AHI_SetFreq()` and `AHI_SetVol()` with `AHISF_IMM` set, and then use `AHI_SetSound(ch, AHI_NOSOUND, 0, 0, actrl, 0L)` to stop the sound when it has reached the end. You can also set one loop segment this way.

`AHI_PlayA()` was added in AHI version 4, and combines `AHI_SetSound()`, `AHI_SetFreq()` and `AHI_SetVol()` into one tag-based function. It also allows you to set one loop and play one-shot sounds.

To play a sound with more than one loop segment or ping-pong looping, a sample interrupt needs to be used. AHI's *SoundFunc* works like Paula's interrupts and is very easy to use.

The *SoundFunc* hook will be called with an `AHIAudioCtrl` structure as object and an `AHISoundMessage` structure as message. `ahism_Channel` indicates which channel caused the hook to be called.

An example *SoundFunc* which handles the repeat part of an instrument can look like this (SAS/C code):

```
__asm __saveds ULONG SoundFunc(register __a0 struct Hook *hook,
    register __a2 struct AHIAudioCtrl *actrl,
    register __a1 struct AHISoundMessage *chan)
{
  if(ChannelDatas[chan->ahism_Channel].Length)
    AHI_SetSound(chan->ahism_Channel, 0,
        (ULONG) ChannelDatas[chan->ahism_Channel].Address,
        ChannelDatas[chan->ahism_Channel].Length,
        actrl, NULL);
  else
    AHI_SetSound(chan->ahism_Channel, AHI_NOSOUND,
        NULL, NULL, actrl, NULL);
  return NULL;
}
```

This example is from an old version of the AHI NotePlayer for *DeliTracker 2*. `ChannelDatas` is an array where the start and length of the repeat part is stored. Here, a repeat length of zero indicates a one-shot sound. Note that this particular example only uses one sound (0). For applications using multiple sounds, the sound number would have to be stored in the array as well.

Once again, note that the `AHISF_IMM` flag should **never** be set in a *SoundFunc* hook!

5.5.3 Tricks With The Volume

Starting with V4, `AHI_SetVol()` can take both negative volume and pan parameters. If you set the volume to a negative value, the sample will, if the audio mode supports it, invert each sample before playing. If pan is negative, the sample will be encoded to go to the surround speakers.

6 Device Interface

The I/O request protocol makes it very easy to play audio streams, sounds from disk and non time-critical sound effects in a multitasking friendly way. Recoding is just as easy, on behalf of quality. Several programs can play sounds at the same time, and even record at the same time if your hardware is full duplex.

If you want to write a sample player, play (warning?) sounds in your applications, play an audio stream from a CD via the SCSI/IDE bus, write a voice command utility etc., this is the API to use.

Note that while all the low-level functions (see Chapter 5 [Function Interface], page 9) count lengths and offsets in sample frames, the device interface—like all Amiga devices—uses bytes.

6.1 Opening And Closing `ahi.device` For High-level Access

Four primary steps are required to open ahi.device:

- Create a message port using `CreateMsgPort()`. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type `AHIRequest` using `CreateIORequest()`. `CreateIORequest()` will initialize the I/O request to point to your reply port.
- Specify which version of the device you need. The lowest supported version is 4. Version 1 and 3 are obsolete, and version 2 only has the low-level API.
- Open `ahi.device` unit `AHI_DEFAULT_UNIT` or any other unit the user has specified with, for example, a *UNIT* tooltype. Call `OpenDevice()`, passing the I/O request.

Each `OpenDevice()` must eventually be matched by a call to `CloseDevice()`. When the last close is performed, the device will deallocate all resources.

All I/O requests must be completed before `CloseDevice()`. Abort any pending requests with `AbortIO()`.

Example:

```
struct MsgPort    *AHImp      = NULL;
struct AHIRequest *AHIio      = NULL;
BYTE               AHIDevice  = -1;
UBYTE              unit       = AHI_DEFAULT_UNIT;

/* Check if user wants another unit here... */

if(AHImp = CreateMsgPort())
{
  if(AHIio = (struct AHIRequest *)
      CreateIORequest(AHImp, sizeof(struct AHIRequest)))
  {
    AHIio->ahir_Version = 4;
    if(!(AHIDevice = OpenDevice(AHINAME, unit,
        (struct IORequest *) AHIio, NULL)))
    {


      /* Send commands to the device here... */


      if(! CheckIO((struct IORequest *) AHIio))
```

```
        {
          AbortIO((struct IORequest *) AHIio);
        }

        WaitIO((struct IORequest *) AHIio);

        CloseDevice((struct IORequest *) AHIio);
        AHIDevice = -1;
      }
      DeleteIORequest((struct IORequest *) AHIio);
      AHIio = NULL;
    }
    DeleteMsgPort(AHImp);
    AHImp = NULL;
  }
```

6.2 Reading From The Device

You read from `ahi.device` by passing an `AHIRequest` to the device with `CMD_READ` set in `io_Command`, the number of bytes to be read set in `io_Length`, the address of the read buffer set in `io_Data`, the desired sample format set in `ahir_Type` and the desired sample frequency set in `ahir_Frequency`. The first read command in a sequence should also have `io_Offset` set to 0. `io_Length` must be an even multiple of the sample frame size.

6.2.1 Double Buffering

To do double buffering, just fill the first buffer with `DoIO()` and `io_Offset` set to 0, then start filling the second buffer with `SendIO()` using the same I/O request (but don't clear `io_Offset`!). After you have processed the first buffer, wait until the I/O request is finished and start over with `SendIO()` on the first buffer.

6.2.2 Distortion

The samples will automatically be converted to the sample format set in `ahir_Type` and to the sample frequency set in `ahir_Frequency`. Because it is quite unlikely that you ask for the same sample frequency the user has chosen in the preference program, chances that the quality is lower than expected are pretty high. The worst problem is probably the anti-aliasing filter before the A/D converter. If the user has selected a higher sampling/mixing frequency than you request, the signal will be distorted according to the Nyquist sampling theorem. If, on the other hand, the user has selected a lower sampling/mixing frequency than you request, the signal will not be distorted but rather bandlimited more than necessary.

6.3 Writing To The Device

You write to the device by passing an `AHIRequest` to the device with `CMD_WRITE` set in `io_Command`, the precedence in `io_Message.mn_Node.ln_Pri`, the number of bytes to be written in `io_Length`, the address of the write buffer set in `io_Data`, the sample format set in `ahir_Type`, the desired sample frequency set in `ahir_Frequency`, the desired volume set in `ahir_Volume` and the desired stereo position set in `ahir_Position`. Unless you are doing double buffering, `ahir_Link` should be set to `NULL`. `io_Length` must be an even multiple of the sample frame size.

6.3.1 Double Buffering

To do double buffering, you need two I/O requests. Create the second one by making a copy of the request you used in `OpenDevice()`. Start the first with `SendIO()`. Set `ahir_Link` in the second request to the address of the first request, and `SendIO()` it. Wait on the first, fill the first buffer again and repeat, this time with `ahir_Link` of the first buffer set to the address of the second I/O request.

6.3.2 Distortion

The problems with aliasing are present but not as obvious as with reading. Just make sure your source data is bandlimited correctly, and do not play samples at a lower frequency than they were recorded.

6.3.3 Playing multiple sounds at the same time

If you want to play several sounds at the same time, just make a new copy of the I/O request you used in `OpenDevice()`, and `CMD_WRITE` it. The user has set the number of channels available in the preference tool, and if too many requests are sent to the device the one with lowest precedence will be muted. When a request is finished, the muted request with the highest precedence will be played. Note that all muted requests continue to play silently, so the programmer will not have to worry if there are enough channels or not.

6.3.4 Suggested precedences

The precedences to use depend on what kind of sound you are playing. The recommended precedences are the same as for `audio.device`, listed in *AMIGA ROM Kernel Reference manual – Devices*. Reprinted without permission. So sue me.

```
Precedences  | Type of sound
-------------+------------------------------------------------------
   127       |  Unstoppable.  Sounds first allocated at lower
             | precedencies, then set to this highest level.
   90 - 100  |  Emergencies.  Alert, urgent situation that requires
             | immediate action.
   80 - 90   |  Annunciators.  Attention, bell (CTRL-G).
   75        |  Speech.  Synthesized or recorded speech
             | (narrator.device).
   50 - 70   |  Sonic cues.  Sounds that provide information that is not
             | provided by graphics.  Only the beginning of of each sound
             | should be at this level; the rest should ne set to sound
             | effects level.
  -50 - 50   |  Music program.  Musical notes in a music-oriented program.
             | The higher levels should be used for the attack portions
             | of each note.
  -70 - -50  |  Sound effects.  Sounds used in conjunction with graphics.
             | More important sounds should use higher levels.
  -100 - -80 |  Background.  Theme music and restartable background sounds.
  -128       |  Silence.  Lowest level (freeing the channel completely is
             | preferred).
```

Right. As you can see, some things do not apply to `ahi.device`. First, there is no way to change the precedence of a playing sound, so the precedences should be set from the beginning. Second, it is not recommended to use the device interface to play music. However, playing an audio stream from CD or disk comes very close. Third, there are no channels to free in AHI since they are dynamically allocated by the device.

7 Data Types And Structures

In this chapter some of the data types and structures used will be explained. For more information, please consult the autodocs and the include files.

7.1 Data Types

7.1.1 Fixed

`Fixed` is a signed long integer. It is used to represent decimal numbers without using floating point arithmetics. The decimal point is assumed to be in the middle of the 32 bit integer, thus giving 16 bits for the integer part of the number and 16 bits for the fraction. The largest number that can be stored in a `Fixed` is +32767.999984741, and the lowest number is −32768.

Example:

```
Decimal | Fixed
--------+----------
 1.0    | 0x00010000
 0.5    | 0x00008000
 0.25   | 0x00004000
 0      | 0x00000000
-0.25   | 0xffffc000
-0.5    | 0xffff8000
-1.0    | 0xffff0000
```

7.1.2 sposition

`sposition` (stereo position) is a `Fixed`, and is used to represent the stereo position of a sound. 0 is far left, 0.5 is center and 1.0 is far right.

7.2 Structures

7.2.1 AHIUnitPrefs And AHIGlobalPrefs

These structures are used in the `AHIU` and `AHIG` chunks, respective, which are part of the settings file ('`ENV:Sys/ahi.prefs`'), The file is read by AHI on each call to `OpenDevice()`, just before the audio hardware is allocated.

`AHIUnitPrefs` specifies the audio mode and its parameters to use for each device unit (currently 0-3 and `AHI_NO_UNIT`; unit 0 is also called `AHI_DEFAULT_UNIT`).

`AHIGlobalPrefs` contains some global options that can be used to gain speed on slow CPUs, the global debug level and a protection against CPU overload. The debug level specifies which of the functions in AHI should print debugging information to the serial port (the output can be redirected to a console window or a file with tools like *Sushi*[1]).

[1] Available from AmiNet, for example `ftp://ftp.germany.aminet.org/pub/aminet/dev/debug/Sushi.lha`.

Concept Index

Data Type Index

A

F

S

Function Index

A

Variable Index

A

C

I

L

Table of Contents