AUDIOSCIENCE

SPECIFICATION

HPI

Hardware Programming Interface

V2.96 – NOV-11 2005

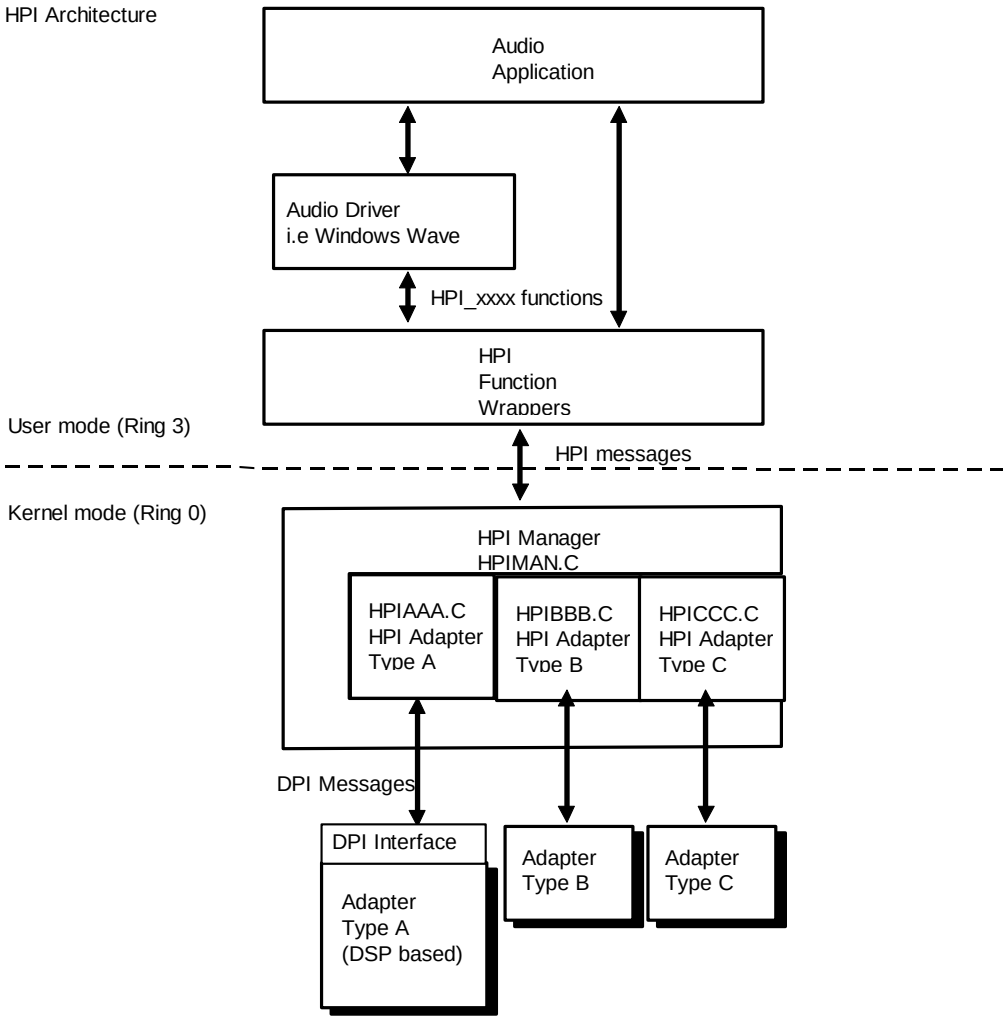
History:

Sep-02-97	SGT	V1.00	Changed HPI_SubSysOpen to _SubSysCreate. Add HPI_SubSysFree
Jan-14-98	SGT	V1.01	Changed definition of Revision in AdapterGetInfo, Added Out/InStreamGetInfoEx()
Aug-13-98	AGE	V1.02	Added VOX control. Added ms bounds for volume autofade. Made up a max of 1 hour, although we could theoretically do more ?
Nov-10-98	AGE	V1.03	Added description for HPI_AESEBU_Transmitter_SetClockSource( ).
May-3-99	SGT	V1.04	Start to add OS specific section
Jul-26-99	AGE	V1.05	Added `HPI_OutStreamSetVelocity()`. Turned off "auto-date" in title. Corrected description of pdwVersion field of HPI_SubSysGetVersion().
Aug-06-99	AGE	V1.06	Added HPI_ChannelMode().
Feb-14-00	SGT	V1.07	Change Gpio to Gpio
Apr-20-00	SGT	V1.08	Add ref to Win2000 WDM driver and change SubsysOpen to SubsysCreate example code
May-31-00	EWB	V1.09	Add bitstream controls
July-10-00	SGT	V1.10	Add examples on how to use various controls
Jan-10-01	EWB	V1.11	Add bitstream subsection in Input Stream section
Jan-31-01	AGE	V1.12	Add adapter mode select.
Feb-05-01	AGE	V1.13	Add HPI_AdapterGetMode().
Feb-13-01	AGE	V1.14	Fix a couple of HPI_AapterGetMode() typos. Feb 16 more typos fixed.
Jun-18-01	SGT	V1.15	Add Sync control, text to GPIO
Aug-15-01	SGT	V1.16	Add MPEG Ancillary functions
Dec-10-01	AGE	V1.17	More MPEG Ancillary data updates.
Dec-13-01	AGE	V1.18	Updated HPI_OutStreamAncillaryXXXX calls.
Dec-14-01	AGE	V1.19	More MPEG ancillary data updates.
Feb-26-02	SGT	V1.20	Add AES Transmitt set Format
Mar-07-02	SGT	V1.21	Add analog/digital mux on linein example
Mar-28-02	AGE	V1.22	Added bit/byte ordering description to 3.4.12 HPI_OutStreamAncillaryRead.
Apr-12-02	AGE	V1.23	Added HPI_ProfileGetIdleCount() which can be used to compute DSP utilization.
May-02-02	EWB/SGT	V1.24	Added HPI_VolumeAutoFadeProfile
May-06-02	AGE	V1.25	Change to Note1 in HPI_InStreamAncillaryReset(). Must now be called before HPI_InStreamSetformat(). Added specification of default ancillary data settings imposed by HPI_InStreamOpen(). Added description of error returns by NvMem operations.
May-28-02	EWB	V1.26	Added how to activate MP3 VBR encoding
May-30-02	AGE	V1.27	Added HPI_AdapterSetModeEx(). Updated mode descriptions, including current defines.
Jun-03-02	AGE	V2.66	Version number changed to match HPI.H.
July-01-02	EWB	V2.67	HPI_MeterGetRms, and description of meter ballistics in meter controls section
July-16-02	AGE	V2.68	Clarification of energy data in HPI_OutStreamAncillaryRead().
Aug-02-02	AGE	V2.69	Clarification of energy data in HPI_OutStreamAncillaryRead() (again!)
Aug-21-02	AGE	V2.70	Added HPI_Multiplexer_GetSource() and HPI_Multiplexer_QuerySource().
Aug-23-02	AGE	V2.70	Added HPI_ChannelModeGet(). Updated the OStream DRAINED state description.
Sep-25-02	AGE	V2.73	Added HPI_OutStreamSetTimeScale().
Oct-31-02	EWB	V2.74	Added HPI_Meter[Set | Get][Peak | Rms]Ballistics().
Oct-31-02	AGE	V2.74	Added HPI_Microphone_SetPhantomPower() & HPI_VolumeGetRange()
Nov-25-02	SGT	V2.74	Clarify ChannelMode, add example code to Microphone control
Dec-13-02	SGT	V2.75	Add Equalizer, Compander. Clarify lower limits on Peak meters
Jan-06-03	EWB	V2.76	Update ancillary data descriptions for MP3 encode, Added OnOff parameter to HPI_ParametricEQ_GetInfo
Mar-10-03	SGT	V2.77	fix typo in HPI_SampleClock_SetSource description. Add HPI_SampleClockGetSource
May-13-03	EWB	V2.79	Document tuner controls. Reorder mixer control sections alphabetically. Add HPI_MixerGetControlByIndex.
July-16-03	SGT	V2.81	Add compression format section.
Sep-02-30	AGE	V2.83	Updated Tuner specd. Get RF level now spec'd to return dBuV (dB micro volts). Added documentation of HPI_SubSysCreateAdapter() and HPI_SubSysDeleteAdapter().
Jan-19-04	EWB	V2.85	Added Tuner_GetGain, Control_QuerySetting. Fix some cut and paste errors. Rename HPI_VolumeGetRange to HPI_VolumeQueryRange
Feb-12-04	EWB	V2.85	Update HPI_MixerGetControlByIndex return values
Mar-08-04	AGE	V2.86	Correction to typos in Hpi_OutStreamAncillaryReset().
Mar-08-04	SGT	V2.86	Add ref to HPI_ControlQuery in Tuner control section. Add another example in HPI_ControlQuery
May-28-04	EWB	V2.88	In and Out stream host buffer functions. Expand on SubSysFindAdapters error return.
July-12-04	SGT	V2.88	Added HPI_Tuner_GetStatus(), which replaces Tuner_GetVideoStatus()
Sep-02-04	TFE	V2.90	Added HPI_WaveFormatToHpiFormat() and HPI_HpiFormatToWaveFormat() utility runctions.
Oct-07-04	AGE	V2.90	Added HPI_TunerSetMode() and HPI_Tuner_GetInstantaneousRFLevel().
Oct-12-04	AGE	V2.90	Change GetInstantaneousRFLevel() to GetRawRFLevel()
Oct-12-04	AGE	V2.90	document GetRawRFLevel() return values
Nov-2-04	EWB	V2.90	Update GetInfoEx to reflect BBM changes.
Jan-4-05	AGE	V2.92	Add HPI_ProfileGetUtilization() function.
May-5-05	AGE	V2.94	Add additional HPI_SampleClock functions.
May-5-05	SGT	V2.94	Update ADAPTER_MODES to include MULTICHANNEL
AUG-8-05	SGT	V2.94	Tweaks to HPI_SampleClock section
Nov-11-05	AGE	V2.96	Correct wListLength description for HPI_SubSysFindAdapters().

1. INTRODUCTION

The AudioScience Hardware Programming Interface (HPI) provides an interface to AudioScience's hardware for operating system device drivers. This interface "abstracts" the audio hardware in a manner similar to WindowsNT's HAL (Hardware Abstraction Layer). Typical hardware would consist of ISA or PCI based audio adapters ("sound cards") , but external audio adapters based on TCP/IP networks or high speed serial interfaces (USB/FireWire) are also possible.

Adapters that contain a DSP communicate with the HPI via the DSP Programming Interface or DPI. The DPI is essentially a subset of the HPI that only deals with functions specific to an adapter.



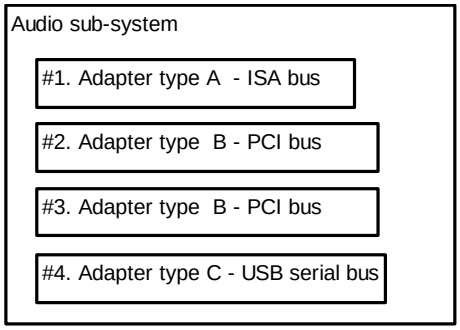
2. HPI MODEL OF THE HARDWARE

The HPI views the audio hardware as being made up of the following **objects**: sub-system, adapter, stream mixer and control

2.1 SubSystem and Adapters

In a computer there is an audio **sub-system.** The sub-system contains different types of audio **adapters**. An adapter may be a bus-based plug-in card (ISA/PCI) or a port based external adapter (serial/parallel/USB).

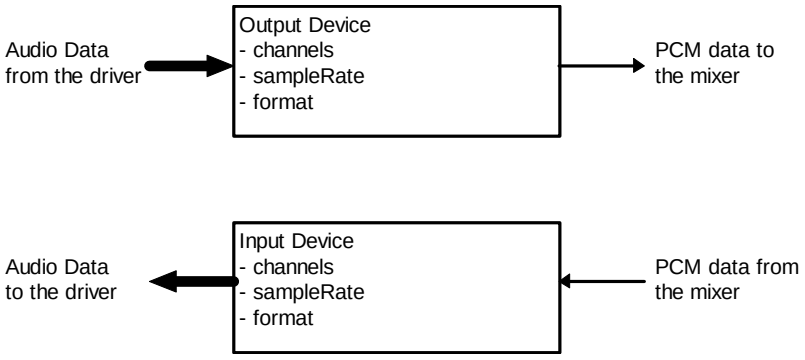
For example there may be 4 adapters in the sub-system. 1 of adapter type A, 2 of adapter type B, and 1 of adapter type C. Note that the audio sub-system is only concerned with AudioScience adapters.



All AudioScience adapters have a unique adapter number associated with them. This is physically set on the adapter using a jumper or switch, in a manner similar to the device ID setting on SCSI devices.

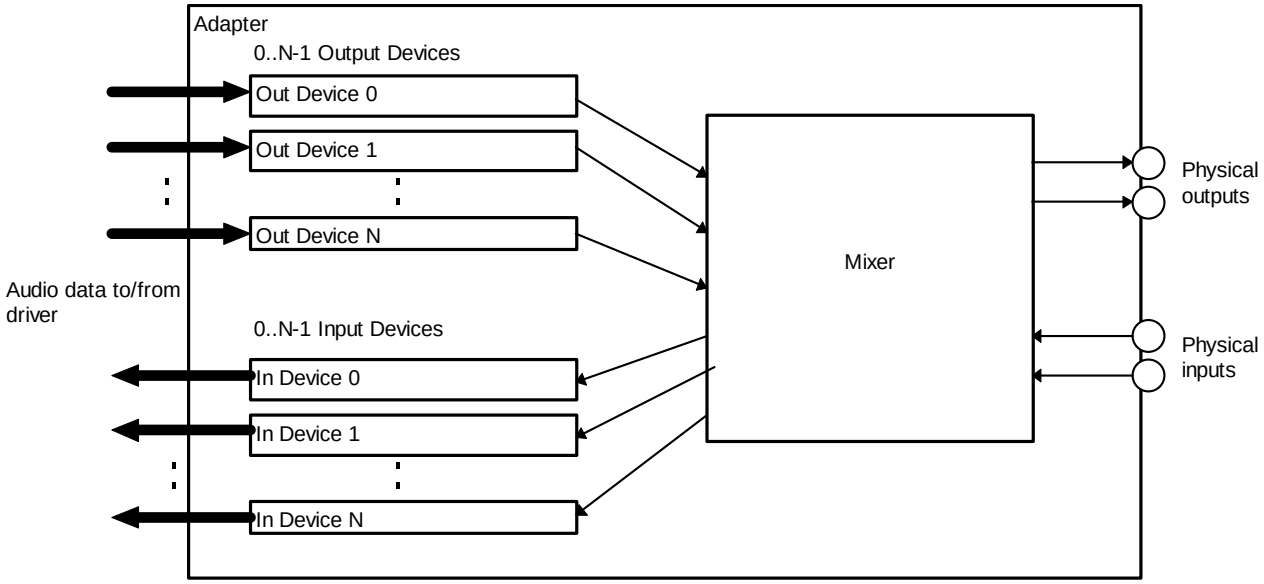
2.2 Streams

Each audio adapter is made up of 0 or more **output_stream**(s) and 0 or more **input_stream**(s). An input_stream is a source of digital audio (record) and a output_stream is a destination for digital audio (playing). A stream can record/play digital audio with specific attributes of **channels**, **sample rate** and **format.**



The number of channels a stream is usually either be (mono) or 2 (stereo). In the case of a forrmat such as DolbyAC3, the number of channels would be 6.

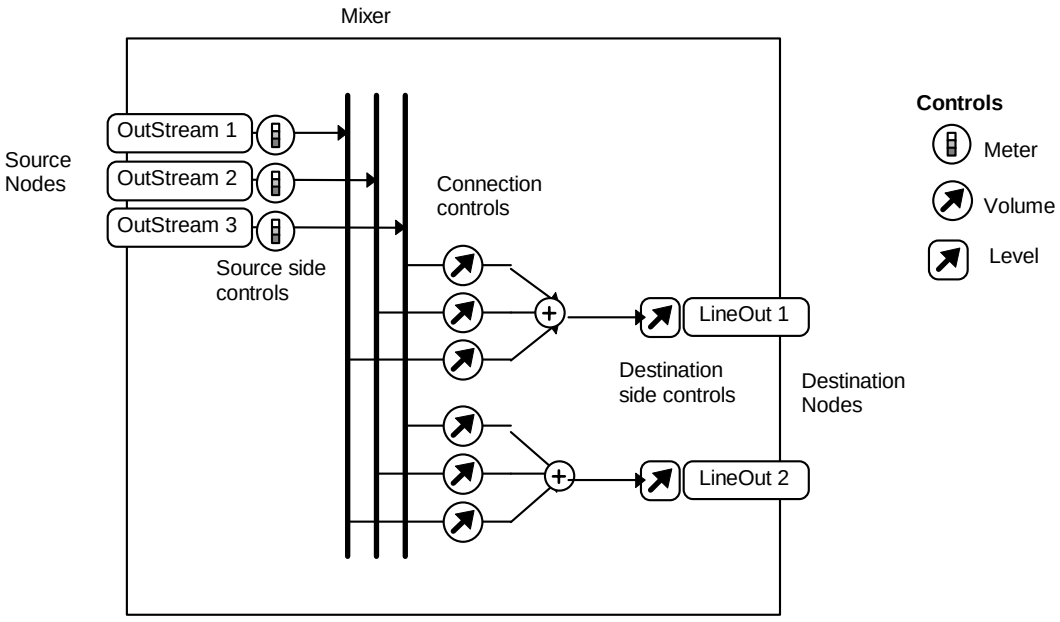
The sampleRate refers to the number of samples per second the stream is playing or recording. Common sample rates are 8kHz, 11.025kHz, 32kHz, 44.1kHz and 48kHz, although the HPI allows almost any sample rate to be specified (to 1Hz resolution)



2.3 Mixers

Each adapter also contains a **mixer**, which routes audio signals between the in/out streams and the **physical** inputs and outputs on the adapter. A mixer contains **controls** that serve to alter and route the audio signals as they pass through. Examples of controls include volume and meter.

For example an adapter may have three output streams, but only two physical (stereo) outputs. The mixer would combine the audio signals from the three streams to send to the two physical outs. This is shown in the following diagram.



Mixer controls can be located in one of three places: source side, on a connection (between a source and destination) and destination side. In the previous diagram you can see that the meter controls are source side, the volume controls are on a connection between a particular source and destination and the level controls are on the destination side.

3. HPI FUNCTION/STRUCTURE REFERENCE

3.1 Basic Data Types

The following data types are used extensively in the HPI:

```
typedef unsigned char   HW8;    /* 8 bit word = byte */
typedef unsigned short  HW16;    /* 16 bit word */
typedef unsigned long   HW32;    /* 32 bit word */

Where a parameter is a gain or level, it is expressed in milliBels.  This can be
converted to deciBels (dB) by dividing by HPI_UNITS_PER_dB (==100).
This allows integer values to represent increments of 0.01dB
```

3.2 HPI Errors

Most HPI functions return an error code (in a HW16). The error codes are defined in hpi.h. Error numbers can be converted into text using the function HPI_GetErrorText();

3.3 Sub-System

3.3.1 HPI_SubSysCreate

Syntax:
```
HPI_HSUBSYS* HPI_SubSysCreate( void );
```

Input Parameters:
none

Output Parameters:
*phSubSysHandle Pointer to HPI subsystem handle, NULL if error

Description:
Create, opens and initialises the audio subsystem. Must be called before any other HPI functions are called.

3.3.2 HPI_SubSysFree

Syntax:
```
void HPI_SubSysCreate(HPI_HSUBSYS *phHpiSubSys );
```

Input Parameters:
*phSubSysHandle Pointer to HPI subsystem handle, NULL if error

Output Parameters:
none

Description:
Closes the HPI subsystem, freeing any resources allocated. Must be the last HPI function called.

3.3.3 HPI_SubSysGetVersion

Syntax:

```
HW16 HPI_SubSysGetVersion(
            HPI_HSUBSYS *phSubSysHandle,
            HW32        *pdwVersion
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*pdwVersion	32 bit word containing version of HPI. Upper 24bits is major version number and lower 8 bits is minor version number i.e. 0x00000104 is version 1.04.

Description:

Obtains the version of the HPI.

3.3.4 HPI_SubSysFindAdapters

Syntax:

```
HW16 HPI_SubSysFindAdapters(
            HPI_HSUBSYS *phSubSysHandle,
            HW16 *pwNumAdapters,
            HW16  awAdapterList[],
            HW16  wListLength
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wListLength	Size (in elements) of *pawAdapterList array

Output Paramters:

*pwNumAdapters	Number of initialised adapters in the audio sub-system
awAdapterList[]	Array of adapter types (see description)

Description:

This function will find and initialise all AudioScience adapters in the computer. Info returned is the number and type of adapters present in the subsystem.

The number of adapters is returned in pwNumAdapters.

The type and adapter number of each adapter is returned in an array of HW16 pointed to by pawAdapterList. Each position in the array identifies an adapter with an adapter index of the corresponding array index. The value of the array indicates the adapter type. A value of zero indicates that no adapter exists for that adapter number.

For example if pawAdapterList had a 1200 in position 0, a 0 in position 1 and a 4501 in position 2, that would indicate an 1200 adapter set to adapter number 1 and a 4501 adapter set to adapter number 3 in the system. Note that the Adapter number (as set on the card/adapter) will be one more than the array index.

Index:	0	1	2	3
PawAdapterList->	0x6244	0	0x4346	0

Return value
The return value is the last error that occurred when initializing all the adapters. As there is only one error return, when there are multiple adapters some of the adapters may have initialized correctly and still be usable. This is indicated by wNumAdapters > 0
It is up to the application whether to continue or fail, but the user should be notified of the error in any case.
Some possible return values are
HPI_DUPLICATE_ADAPTER_NUMBER – two adapters have the same jumper setting. Only the first one will be accessible
HPI_ERROR_DSP_BOOTLOAD – something went wrong with starting the adapter DSP
HPI_ERROR_DSP_FILE_NOT_FOUND – probably an old driver and a new card, or asidsp.bin is missing.
Other errors in the 900 range are adapter specific.

3.3.5 HPI_SubSysCreateAdapter

Syntax:
```
HW16 HPI_SubSysCreateAdapter(
            HPI_HSUBSYS *phSubSysHandle,
            HPI_RESOURCE *pResource,
            HW16         *pwAdapterIndex
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
*pResource	Pointer to the resources used by this adapter.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
wAdapterIndex	The index of the adapter that was just created.

Description:
This is used by PnP OSes to create an audio adapter.

3.3.6 HPI_SubSysDeleteAdapter

Syntax:
```
HW16 HPI_SubSysDeleteAdapter(
            HPI_HSUBSYS *phSubSys,
            HW16          wAdapterIndex
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	The index of the adapter to delete.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
This is used by PnP OSes to delete an audio adapter.

3.4 Adapter

3.4.1 HPI_AdapterOpen

Syntax:

```
HW16 HPI_AdapterOpen(
            HPI_HSUBSYS *phSubSysHandle,
            HW16        wAdapterIndex
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Opens an adapter for use. The adapter is specified by the wAdapterIndex which corresponds to the adapter index on the adapter hardware (set using jumpers or switch)

3.4.2 HPI_AdapterClose

Syntax:

```
HW16 HPI_AdapterClose(
            HPI_HSUBSYS *phSubSysHandle,
            HW16        wAdapterIndex
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Closes the adapter associated with the wAdapterIndex

3.4.3 HPI_AdapterGetInfo

Syntax:

```
HW16 HPI_AdapterGetInfo(
      HPI_HSUBSYS *phSubSysHandle,
      HW16        wAdapterIndex,
      HW16        *pwNumOutStreams,
      HW16        *pwNumInStreams,
      HW16        *pwVersion,
      HW32        *pdwSerialNumber,
      HW16        *pwAdapterType
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*pwNumOutStreams	Number of output streams (play) on the adapter
*pwNumInStreams	Number of input streams (record) on the adapter
*pwVersion	Adapter hardware and DSP software version information. The 16bit word contains the following information:

Bits	Description	Range
b15-13	DSP software major version	0..7
b12-b7	DSP software minor version	0..63
b6-b3	Adapter PCB revision	A..P represented as 0..15
b2-b0	Adapter assembly revision	0..7

For example if *pwVersion = 0x2782 would represent adapter revision A2 and DSP software version 1.15

*pdwSerialNumber	Adapter serial number. Starts at 1 and goes up
*pwAdapterType	Adapter ID code, defined in HPI.H. Examples are HPI_ADAPTER_ASI1201 (0x1201)

Description:
Obtains information about the specified adapter, including the number of output streams and number of input streams, version, serial number and it's type. The adapter is assumed to have one mixer.

3.4.4 HPI_AdapterSetMode

Syntax:

```
HW16 HPI_AdapterSetMode(
            HPI_HSUBSYS *phSubSysHandle,
            HW16        wAdapterIndex,
            HW32        dwAdapterMode
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
dwAdapterMode	The adapter mode. This is used to changed how an adapter operates. Currently defined modes are:

HPI_ADAPTER_MODE_4OSTREAM	ASI4401 4 stereo/mono ostreams -> 1 stereo line out ASI4215 4 ostreams - > 4 line outs ASI6114 4 ostreams -> 4 line outs
HPI_ADAPTER_MODE_6OSTREAM	(ASI4401 6 mono ostreams -> 2 mono line outs)
HPI_ADAPTER_MODE_8OSTREAM	ASI6114 8 ostreams -> 4 line outs ASI6118 8 ostreams -> 8 line outs
HPI_ADAPTER_MODE_9OSTREAM	ASI6044 9 ostreams -> 4 lineouts
HPI_ADAPTER_MODE_12OSTREAM	ASI504X 12 ostreams -> 4 lineouts
HPI_ADAPTER_MODE_16OSTREAM	ASI6118 16 ostreams -> 8 line outs
HPI_ADAPTER_MULTICHANNEL	ASI504X 2 ostreams -> 4 line outs (1 to 8 channel streams), 4 lineins -> I instream (1 to 8 channel streams) at 0-48kHz For more info see the SSX Specification
HPI_ADAPTER_MODE1	ASI504X - 12 ostream, 4 instream 0 to 48kHz sample rates (see ASI504X datasheet for more info)
HPI_ADAPTER_MODE2	ASI504X - 4 ostream, 4 instream 0 to 192kHz sample rates. (see ASI504X datasheet for more info)
HPI_ADAPTER_MODE3	ASI504X - 4 ostream, 4 instream 0 to 192kHz sample rates. (see

	ASI504X datasheet for more info)

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the operating mode of an adapter. The adapter must be restarted for the mode to take affect. Under Windows this means that the computer must be rebooted.

3.4.5 HPI_AdapterSetModeEx

Syntax:

```
HW16 HPI_AdapterSetMode(
            HPI_HSUBSYS *phSubSysHandle,
            HW16        wAdapterIndex,
            HW32        dwAdapterMode,
            HW16        wSetOrQuery
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
dwAdapterMode	The adapter mode. This is used to changed how an adapter operates. Currently defined modes are: HPI_ADAPTER_MODE_4OSTREAM ASI4401 4 stereo/mono ostreams -> 1 stereo line out ASI4215 4 ostreams - > 4 line outs ASI6114 4 ostreams -> 4 line outs HPI_ADAPTER_MODE_6OSTREAM (ASI4401 6 mono ostreams -> 2 mono line outs) HPI_ADAPTER_MODE_8OSTREAM ASI6114 8 ostreams -> 4 line outs ASI6118 8 ostreams -> 8 line outs HPI_ADAPTER_MODE_16OSTREAM ASI6118 16 ostreams -> 8 line outs
wSetOrQuery	Should be set to either HPI_ADAPTER_MODE_SET or HPI_ADAPTER_MODE_QUERY.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets or queries the operating mode of an adapter. When wSetOrQuery is set to HPI_ADAPTER_MODE_QUERY, the adapter is queried to see if the specified mode is supported without changing any adapter settings. A return value of 0 indicates that the requested mode is supported. When wSetOrQuery is set to HPI_ADAPTER_MODE_SET the mode of the adapter is changed. A return value of 0 indicates success. The adapter must be restarted for the mode to take affect. Under Windows this means that the computer must be rebooted.

3.4.6 HPI_AdapterGetMode

Syntax:

```
HW16 HPI_AdapterGetMode(
            HPI_HSUBSYS *phSubSysHandle,
            HW16        wAdapterIndex,
```

```
        HW32        *pdwAdapterMode
        );
```

Input Parameters:

*phSubSysHandle — Pointer to HPI subsystem handle.

pdwAdapterMode — The adapter mode. This will be set to the current adapter mode. Currently defined modes are:

HPI_ADAPTER_MODE_4OSTREAM
- ASI4401 4 stereo/mono ostreams -> 1 stereo line out
- ASI4215 4 ostreams - > 4 line outs
- ASI6114 4 ostreams -> 4 line outs

HPI_ADAPTER_MODE_6OSTREAM
- ASI6012 6 ostreams -> 2 line outs
- ASI6122 6 ostreams -> 2 line outs
- ASI4401 6 mono ostreams -> 2 mono line outs

HPI_ADAPTER_MODE_8OSTREAM
- ASI6114 8 ostreams -> 4 line outs
- ASI6118 8 ostreams -> 8 line outs

HPI_ADAPTER_MODE_16OSTREAM
- ASI6118 16 ostreams -> 8 line outs

Output Parameters:

Error — 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Gets the operating mode of an adapter (ASI4401 specific).

3.5 Stream

Functions that apply to both Out and In Streams.

3.5.1 HPI_StreamEstimateHostBufferSize

Syntax:

```
HW16 HPI_StreamEstimateBufferSize(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hStreamHandle,
      HPI_FORMAT *pF,
      HW32 dwHostPollingRateInMilliSeconds,
      HW32 *dwRecommendedBufferSize)'
```

Input Parameters:

*phSubSysHandle — Pointer to HPI subsystem handle.

*pF — pointer to a HPI_FORMAT containing the format that is going to be used for the stream.

`dwHostPollingRateInMilliSeconds` — The rate at which the stream info will be polled and data read or written to the stream.

Output Parameters:

`*dwRecommendedBufferSize` — The recommended minimum size for the host buffer.

Return value:

Error — 0 upon success,
HPI_ERROR_INVALID_FORMAT

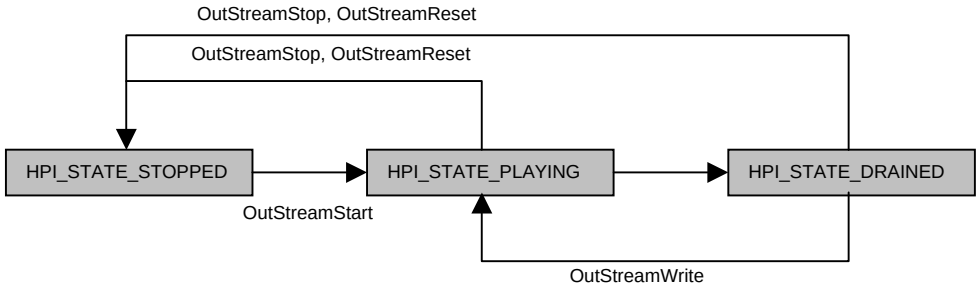
Description:
Calculate the minimum buffer size for a stream given the audio format that the stream will be set to use and the rate at which the host polls the stream state and reads or writes data.
The buffer size returned by this function should be used as the minimum value passed to HPI_OutStreamHostBufferAllocate() or HPI_InStreamHostBufferAllocate() If different formats and samplerates will be used on the stream, buffer size should be calculated for the highest datarate format, or the buffer should be freed and allocated again for each format change.

Enabling background bus mastering places some additional constraints on your application.

In order to allow the BBM to catch up if the host app has got behind, it must be possible to transfer data faster than it is being acquired. This means that the buffer needs to be bigger than minimum. Recommended size is at least 2x minimum. A buffer size of Nx4096 – 20 makes the best use of memory.

3.6 Output Stream

3.6.1 OutStream States



The state HPI_STATE_DRAINED indicates that the stream has run out of decoded data.

This can happen for two reasons:
Intentionally, when the end of the currently playing audio is reached.
A transient error condition when the adapter DSP was unable to decode audio fast enough.

It is possible that HPI_OutStreamGetInfoEx() indicates that there is still a small amount data to play, but the stream enters the DRAINED state, and the amount of data to play never gets to zero.
The dwDataToPlay count measures the amount of encoded data (MPEG, PCM etc) in the stream's buffer. When there is less than a whole frame of encoded data left, it cannot be decoded for mixing and output.

The size of a frame varies depends on the audio format.
Compressed formats such as MPEG require whole frames of data in order to decode audio. The size of the input frame depends on the samplerate and bitrate (E.g. MPEG frame_bytes = bitrate/8 * 1152/samplerate).

AudioScience's implementation of PCM decoding also requires a minimum amount of input data. ASI4xxx adapters require 4608 bytes, whereas ASI6xxx adapters require 1536 bytes for stereo, half this for mono.

Conservative conditions to detect end of play
Have done the final OutStreamWrite
Stream state is HPI_STATE_DRAINED

dwDataToPlay < 4608

Input data requirements for different algorithms.

Bytes	PCM	MP1	MP2	MP3	AC2
AX	4608	<3456	<3456	-	380
AX4	4608	<3456	<3456	3456	-
AX6	1536	<=700	<=700	1100	380

3.6.2 HPI_OutStreamOpen

Syntax:
```
HW16 HPI_OutStreamOpen(
      HPI_HSUBSYS *phSubSysHandle,
      HW16 wAdapterIndex,
      HW16 wOutStreamIndex,
      HPI_HOSTREAM *phOutStreamHandle );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.
wOutStreamIndex	Index of the OutStream to be opened. Ranges from 0 to wNumOutStreams-1 (returned by HPI_AdapterGetInfo()

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*phOutStreamHandle	Handle to the OutStream.

Description:
Open and initializes an output stream. An Adapter index and OutStream index are passed in and a OutStream handle is passed back. The handle is used for all future calls to that OutStream. A particular output stream may only be open by one application at one time.

3.6.3 HPI_OutStreamHostBufferAllocate

Syntax:
```
HW16 HPI_OutStreamHostBufferAllocate(
      HPI_HSUBSYS *phSubSysHandle,
      HW32 dwSizeInBytes);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
dwSizeInBytes	Size of bus mastering buffer to allocate.

Return value:

Error	0 upon success, HPI_ERROR_INVALID_DATASIZE if memory can't be allocated (retrying the call with a smaller size may succeed) HPI_ERROR_INVALID_OPERATION if virtual address of buffer can't be found.

Description:
Allocates a buffer inside the driver for bus mastering transfers. Once the buffer is allocated, OutStream data will be transferred from it in the background. (rather than the application having to wait for the transfer)
This function is provided so that the application can match the size of its transfers to the size of the buffer.

From now on, HPI_OutStreamGetInfoEx will return the size and data available in host buffer rather than the buffers on the adapter. However, while there is space in the adapter buffers, data will be quickly transferred to the adapter, providing additional buffering against delays in sending more audio data.

Note that there is a minimum buffer size that will work with a given audio format and polling rate. An appropriate size for the buffer can be calculated using HPI_StreamEstimateHostBufferSize()

If an error occurs or the adapter doesn't support host buffering then no buffer is allocated. Stream transfers still take place using foreground transfers (all drivers pre 2004). Performance will be relatively worse.

3.6.4 HPI_OutStreamHostBufferFree

Syntax:
```
HW16 HPI_OutStreamHostBufferFree(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStream);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStream	Handle of the OutStream

Return value:

Error	0 upon success, HPI_ERROR_INVALID_FUNC if the function is not implemented

Description:
Free any buffers allocated by HPI_OutStreamHostBufferAllocate.

3.6.5 HPI_OutStreamClose

Syntax:
```
HW16 HPI_OutStreamClose(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Closes an output stream. Deallocates host buffers if they are being used.

3.6.6 HPI_OutStreamWrite

Syntax:
```
HW16 HPI_OutStreamWrite(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      HPI_DATA    *pData
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream
pData	Pointer to an HPI_DATA structure containing a description of the audio data to be written to the OutStream and played.

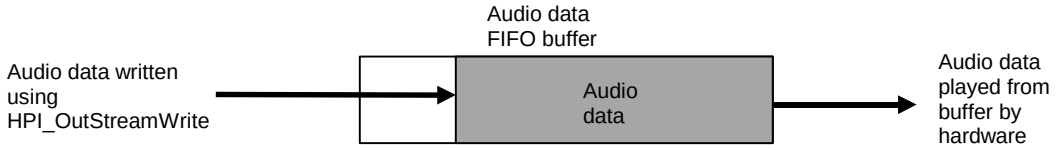
Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Writes a block of audio data to the specified output stream. The data to write is specified by pData. pData also contains the format of the data (channels, compression, sampleRate). The HPI will copy the data in pData to the Output stream hardware. Once the data has been written to the stream, the memory used to hold that data may be reused.

A different format will only be accepted in the first write after the stream is opened or reset.

The size of the data block that may be written is limited to half the size of the streams internal data buffer (specified by dwBufferSize returned by HPI_OutStreamGetInfo()).



Example:

```
wHE = HPI_FormatCreate(
              &hpiFormat,
              2,                    // stereo channel
              HPI_FORMAT_MPEG_L2,   // MPEG Layer II
              44000L,               //sample rate
              128000L,              //128k bits/sec
              0                     // no attributes
              );

wHE = HPI_DataCreate( &Data, &hpiFormat, gabBuffer, BLOCK_SIZE );
wHE = HPI_OutStreamWrite( phSubSys, hOutStream, &Data);
```

3.6.7 HPI_OutStreamStart

Syntax:
```
HW16 HPI_OutStreamStart(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Starts an output stream playing audio data. Data is taken from the circular buffer on the adapter hardware that has already been partially filled by HPI_OutStreamWrite commands. Audio is played from the current position in the buffer (which may be reset using HPI_OutStreamReset)

3.6.8 HPI_OutStreamStop

Syntax:
```
HW16 HPI_OutStreamStop(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Stops an output stream playing audio data. The audio data buffer is not cleared, so a subsequent OutStreamStart will resume playing at the position in the buffer where the playback had been stopped.

3.6.9 HPI_OutStreamReset

Syntax:
```
HW16 HPI_OutStreamReset(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Clears the audio data buffer of an output stream. If the stream was playing at the time, it will be stopped.

3.6.10 HPI_OutStreamGetInfo

Syntax:
```
HPI_OutStreamGetInfo(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      HW16  *pwState,
      HW32  *pdwBufferSize,
      HW32  *pdwDataToPlay
      )
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

*pwState	State of stream = HPI_STATE_STOPPED, or HPI_STATE_PLAYING
*pdwBufferSize	Size (in bytes) of stream data buffer
*pdwDataToPlay	Amount of data (in bytes) left in the buffer to play.

Description:

NOTE: This function has been superseded by HPI_OutStreamGetInfoEx()

Returns information about the Output stream. This includes the size of the streams hardware buffer returned in pdwBufferSize. Also includes whether the stream is currently playing (the state) and the amount of audio data left to play.

3.6.11 HPI_OutStreamGetInfoEx

Syntax:

```
HPI_OutStreamGetInfoEx(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      HW16  *pwState,
      HW32  *pdwBufferSize,
      HW32  *pdwDataToPlay,
      HW32  *pdwSamplesPlayed,
      HW32  *pdwAuxiliaryDataToPlay
      )
```

```
HW16 HPI_OutStreamGetInfoEx(
        HPI_HSUBSYS    *phSubSysHandle,
        HPI_HOSTREAM   hOutStreamHandle,
        HW16           *pwState,
        HW32           *pdwBufferSize,
        HW32           *pdwDataToPlay,
        HW32           *pdwSamplesPlayed,
        HW32           *pdwAuxiliaryDataToPlay);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

*pwState	State of stream = HPI_STATE_STOPPED,HPI_STATE_PLAYING or HPI_STATE_DRAINED
*pdwBufferSize	Size (in bytes) of stream data buffer
*pdwDataToPlay	Amount of data (in bytes) left in the buffer to play.
*pdwSamplesPlayed	Number of samples played since an HPI_OutStreamReset was last issued.
*pdwAuxiliaryDataToPlay	Amount of data (in bytes) in on-card buffer

Description:
Returns information about the Output stream. This is similar to HPI_OutStreamGetInfo but contains extended information. This includes the size of the streams buffer returned in pdwBufferSize. Also includes whether the stream is currently playing (the state) and the amount of audio data left to play.

The SamplesPlayed parameter returns the number of samples played since the last HPI_OutStreamReset command was issued. It reflects the number of stereo samples for a stereo stream and the number of mono samples for a mono stream. This means that if a 44.1kHz stereo and mono stream were both playing they would both return SamplesPlayed=44100 after 1 second.

AuxiliaryDataToPlay is only relevant when BBM is active (see HPI_OutStreamHostBufferAllocate). In this case DataToPlay refers to the host side buffer state while AuxiliaryDataToPlay refers to the data remaining in the cards own buffers.

3.6.12 HPI_OutStreamQueryFormat

Syntax:
```
HW16 HPI_OutStreamQueryFormat(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HOSTREAM      OutStreamHandle,
      HPI_FORMAT        *pFormat
      );
```

Description:
Queries an OutStream to see whether it supports a certain audio format, described in pFormat. The result, returned in the error code, is 0 if supported and HPI_ERROR_INVALID_FORMAT if not supported.

3.6.13 HPI_OutStreamSetTimeScale

Syntax:
```
HW16 HPI_OutStreamSetTimeScale(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HOSTREAM      OutStreamHandle,
      HPI32             dwTimeScale
      );
```

Input Parameters:

wVelocity	The velocity in units HPI_OSTREAM_TIMESCALE_UNITS
hOutStreamHandle	Handle to the OutStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

NOTE: This functionality is only available on the ASI6100 and ASI6200 series adapters

Sets the playback time scale with pitch and content preservation. Range is 0.8-1.2 (80% to 120%) of original time. wTimeScale in set by:

```
dwTimeScale = (HW16)(fTimeScale * HPI_OSTREAM_TIMESCALE_UNITS);
```

where `fTimeScale` in a floating point number in the range of 0.8 < `fTimeScale` < 1.2. The actual granularity of this setting is 1 / 2048 or approximately 0.05% (approx 5 `HPI_OSTREAM_TIMESCALE_UNITS`).
The first call to HPI_ OutStreamSetTimeScale should be made while the stream is reset so that time scaling can be performed after starting playback. Subsequent calls to HPI_ OutStreamSetTimeScale can be made when the stream is playing to modify the timescale factor.

3.6.14 HPI_OutStreamSetVelocity

Syntax:
```
HW16 HPI_OutStreamSetVelocity(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HOSTREAM      OutStreamHandle,
      HPI16             wVelocity
      );
```

Input Parameters:

wVelocity	The velocity in units HPI_VELOCITY_UNITS
hOutStreamHandle	Handle to the OutStream

Output Parameters:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

NOTE: This functionality is only available on the ASI4300 series adapters

Sets the playback velocity for scrubbing. Velocity range is +/- 4.0. wVelocity in set by
`wVelocity = (HW16)(fVelocity * HPI_VELOCITY_UNITS);`
where
`fVelocity` in a floating point number in the range of -4.0 to +4.0.
This call puts the stream in “scrub” mode. The first call to HPI_OutStreamSetVelocity should be made while the stream is reset so that scrubbing can be performed after starting playback.

3.6.14.1 Call sequence

A typical playback call sequence without scrubbing is:

Write
Write
Start
Write
.....

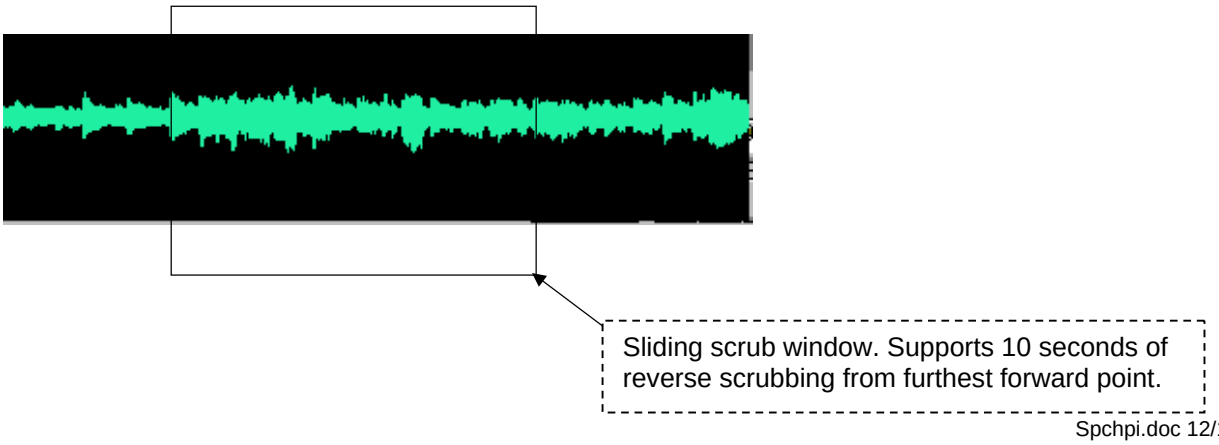
Stop

A typical playback sequence with scrubbing is:

Write
Write
SetVelocity
Start
Write
SetVelocity
.....
Stop - automatically turns of velocity/scrub mode.

3.6.14.2 Data flow

The scrubbing approach taken here is to decode audio to a “scrub buffer” that contains many seconds of PCM that can be traversed in at a variable rate.



Forward scrubbing does not have any limitations what-so-ever, apart from the maximum speed, as specified by `HPI_OutStreamSetVelocity()`.

Reverse scrubbing operates under the following constraints:
1) The user may not scrub on audio prior to the `HPI_OutStreamStart( )` data point.
2) The user may not reverse scrub further than -10 seconds from the forward most scrub position.
If either of these constaints is broken, the stream state will return HPI_STATE_DRAINED and audio playback will cease. The user can then forward scrub again after this error condition.

3.6.15 HPI_OutStreamAncillaryReset

Syntax:
```
HW16 HPI_OutStreamAncillaryReset(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HOSTREAM      OutStreamHandle,
      HW16              wMode,
      );
```

Input Parameters:

hOutStreamHandle	Handle to the OutStream
wMode	The mode for the ancillary data extraction to operate in. Valid settings are HPI_MPEG_ANC_RAW and HPI_MPEG_ANC_HASENERGY. The “RAW” mode indicates that the entire ancillary data field is taken up by data from the Anc data buffer. The “HASENERGY” option tells the decoder that the MPEG frames have energy information stored in them (5 bytes per stereo frame, 3 per mono).

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Initializes the MPEG Layer II / III Ancillary data channel to support the extraction of wBytesPerFrame bytes from the MPEG bitstream.

Note1 (Calling order):
This call must be made after Open or Reset and before first OutStreamWrite call.

3.6.16 HPI_OutStreamAncillaryGetInfo

Syntax:
```
HW16 HPI_OutStreamAncillaryGetInfo(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      HW32  *pdwFramesAvailable)
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStreamHandle	Handle to the OutStream

Output Parameters:

*pdwFramesAvailable	Number of HPI_ANC_FRAMEs in the hardware buffer available for reading.

Description:

Returns information about the Ancillary stream.

3.6.17 HPI_OutStreamAncillaryRead

Syntax:
```
HW16 HPI_OutStreamAncillaryRead(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hOutStreamHandle,
      HPI_ANC_FRAME *pdwAncFrameBuffer,
      HW32  dwAncFrameBufferSizeInBytes,
      HW32  dwNumberOfAncillaryFramesToRead
      )
```

Input Parameters:

*phSubSysHandle Pointer to HPI subsystem handle.
hOutStreamHandle Handle to the OutStream
pdwAncFrameBuffer A pointer to a buffer where the read Ancillary data frames should be placed.
dwAncFrameBufferSize The size of the Ancillary data buffer in bytes (used for a sanity check)
dwNumberOfAncillaryFramesToRead How many frames to read.

Description:
Reads frames of ancillary data from a stream's ancillary data buffer to pdwBuffer. Note that in the situation where energy level information is embedded in the ancillary data stream along with ancillary data, only the ancillary data will be returned in the ancillary data buffer.

Bytes are filled in the bData[] array of the HPI_ANC_FRAME structures in the following order.

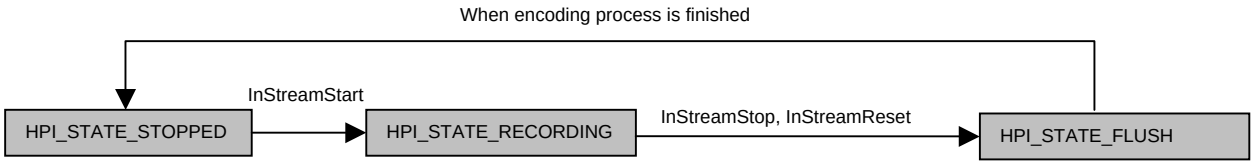
The first bit of ancillary information that follows the vaild audio data is placed in bit 7 of bData[0]. The first 8 bits of ancillary information following valid audio data are all placed in bData[0]. In the case where there are 6 bytes total of ancillary information (48 bits) the last byte filled is bData[5].

Note:

- If OutStreamAncillaryReset () was called with mode=HPI_MPEG_ANC_RAW, the ancillary data in its entirety is placed in the AncFrameBuffer, so if the file was recorded with energy information in the ancillary data field, the energy information will be included in the extracted ancillary data.
- If OutStreamAncillaryReset () was called with mode=HPI_MPEG_ANC_HASENERGY, the ancillary data minus the energy informastion is placed in the AncFrameBuffer.

3.7 Input Stream

3.7.1 InStream States



3.7.2 HPI_InStreamOpen

Syntax:
```
HW16 HPI_InStreamOpen(
      HPI_HSUBSYS *phSubSysHandle,
      HW16 wAdapterIndex,
      HW16 wInStreamIndex,
      HPI_HISTREAM*phInStreamHandle );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
wAdapterIndex	Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.
wInStreamIndex	Index of the InStream to be opened. Ranges from 0 to wNumInStreams-1 (returned by HPI_AdapterGetInfo()

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*phInStreamHandle	Handle to the InStream.

Description:
Open and initializes an output stream. An Adapter index and InStream index are passed in and a InStream handle is passed back. The handle is used for all future calls to that InStream. A particular output stream may only be open by one application at one time.

Note:
A side effect of HPI_InStreamOpen() is that the following default ancillary data settings are made:
Ancillary Bytes Per Frame = 0
Ancillary Mode = HPI_MPEG_ANC_HASENERGY
Ancillary Alignment = HPI_MPEG_ANC_ALIGN_RIGHT

3.7.3 HPI_InStreamHostBufferAllocate

Syntax:
```
HW16 HPI_InStreamHostBufferAllocate(
      HPI_HSUBSYS *phSubSysHandle,
      HW32 dwSizeInBytes);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
dwSizeInBytes	Size of bus mastering buffer to allocate.

Return value:

Error	0 upon success, HPI_ERROR_INVALID_DATASIZE if memory can't be allocated (retrying the call with a smaller size may succeed) HPI_ERROR_INVALID_OPERATION if virtual address of buffer can't be found.

Description:
Assuming no error:
Allocates a buffer on the host PC for bus mastering transfers. Once the buffer is allocated, InStream data will be transferred to it in the background. (rather than the application having to wait for the transfer)
This function is provided so that the application can match the size of its transfers to the size of the buffer.

From now on, HPI_InStreamGetInfoEx will return the size and data available in host buffer rather than the buffers on the adapter. However, if the host buffer is allowed to fill up, the adapter buffers will then begin to fill up. In other words you still have the benefit of the larger adapter buffers

Note that there is a minimum buffer size that will work with a given audio format and polling rate. An appropriate size for the buffer can be calculated using HPI_StreamEstimateHostBufferSize()

3.7.4 HPI_InStreamHostBufferFree

Syntax:
```
HW16 HPI_InStreamHostBufferFree(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HOSTREAM hInStream);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hOutStream	Handle of the InStream

Return value:

Error	0 upon success, HPI_ERROR_INVALID_FUNC if the function is not implemented

Description:
Free any buffers allocated by HPI_InStreamHostBufferAllocate

3.7.5 HPI_InStreamClose

Syntax:
```
HW16 HPI_InStreamClose(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAM hInStreamHandle );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream to close

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Closes an input stream. Deallocates allocated host buffer.

3.7.6 HPI_InStreamRead

Syntax:
```
HW16 HPI_InStreamRead(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAM hInStreamHandle,
      HPI_DATA    *pData
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream to close
pData	Pointer to an HPI_DATA structure containing a description of the audio data to be read from the InStream.

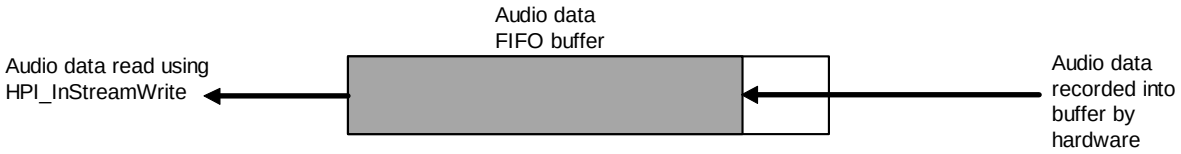
Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Reads a block of audio data from the specified InStream. The memory data buffer to read into is specified by pData. The HPI will copy the data in the InStream hardware buffer to memory buffer pointed to by pData->pbData.

The size of the data block that may be read may be any size upto the amount of data available in the hardware buffer (specified by dwDataAvailable returned by HPI_InStreamGetInfo()).



3.7.7 HPI_InStreamStart

Syntax:
```
HW16 HPI_InStreamStart(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAM hInStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Starts an input stream recording audio data. Audio data is written into the adapters hardware buffer using the currently selected audio format (channels, sample rate, compression format etc). Data is then read from the buffer using HPI_InStreamRead().

3.7.8 HPI_InStreamStop

Syntax:
```
HW16 HPI_InStreamStop(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAMhInStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Stops an input stream recording audio data. The audio data buffer is not cleared, so a subsequent InStreamStart will resume recording at the position in the buffer where the record had been stopped.

3.7.9 HPI_InStreamReset

Syntax:
```
HW16 HPI_InStreamReset(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAMhInStreamHandle,
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Clears the audio data buffer of an input stream. If the stream was recording at the time, it will be stopped.

3.7.10 HPI_InStreamGetInfo

Syntax:
```
HPI_InStreamGetInfo(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAMhInStreamHandle,
      HW16  *pwState,
      HW32  *pdwBufferSize,
      HW32  *pdwDataRecorded
      )
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hInStreamHandle	Handle to the InStream to close

Output Parameters:

*pwState	State of stream = HPI_STATE_STOPPED, or HPI_STATE_RECORDING
*pdwBufferSize	Size (in bytes) of stream data buffer
*pdwDataRecorded	Amount of data (in bytes) contained in the hardware buffer.

Description:
Returns information about the input stream. This includes the size of the stream's hardware buffer returned in pdwBufferSize. Also includes whether the stream is currently recording (the state) and the amount of audio data currently contained in the buffer.

3.7.11 HPI_InStreamGetInfoEx

Syntax:
```
HPI_InStreamGetInfoEx(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAMhInStreamHandle,
      HW16  *pwState,
      HW32  *pdwBufferSize,
      HW32  *pdwDataRecorded,
      HW32  *pdwSamplesRecorded,
      HW32  *pdwAuxilaryDataRecorded
      )
```

Input Parameters:

*phSubSysHandle Pointer to HPI subsystem handle.
hInStreamHandle Handle to the InStream to close

Output Parameters:
*pwState State of stream = HPI_STATE_STOPPED, or HPI_STATE_RECORDING
*pdwBufferSize Size (in bytes) of stream data buffer
*pdwDataRecorded Amount of data (in bytes) contained in the stream buffer.
*pdwSamplesRecorded Number of samples recorded since an HPI_InStreamReset was last issued.
*pdwAuxilaryDataRecorded Amount of data in on-card buffers when BBM is active.

Description:
Returns extended information about the input stream. This includes the size of the stream's hardware buffer returned in pdwBufferSize. Also includes whether the stream is currently recording (the state), the amount of audio data currently contained in the buffer and how many samples have been recorded.

The SamplesRecorded parameter returns the number of samples recorded since the last HPI_InStreamReset command was issued. It reflects the number of stereo samples for a stereo stream and the number of mono samples for a mono stream. This means that if a 44.1kHz stereo and mono stream were both recording they would both return SamplesRecorded=44100 after 1 second.

DataRecorded returns the amount of data available to be read back in the next HPI_InStreamRead call. When BBM is active this is the data in the host buffer, otherwise it is the amount in the on-card buffer.

AuxilaryDataRecorded is only valid when BBM is being used (see HPI_InStreamHostBufferAllocate).
In BBM mode it returns the amount of data left in the on-card buffers. This can be used to determine if a record overrun has occurred (both BBM and card buffers full), or at the end of recording to ensure that all recorded data has been read.

3.7.12 HPI_InStreamQueryFormat

Syntax:
```
HW16 HPI_InStreamQueryFormat(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HISTREAM      InStreamHandle,
      HPI_FORMAT        *pFormat
      );
```

Description:
Queries an input stream to see whether it supports a certain audio format, described in pFormat. The result, returned in the error code, is 0 if supported and HPI_ERROR_INVALID_FORMAT if not supported.

3.7.13 HPI_InStreamSetFormat

Syntax:
```
HW16 HPI_InStreamSetFormat(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HISTREAM      InStreamHandle,
      HPI_FORMAT        *pFormat
      );
```

Description:
Sets the recording format for an input stream. The format to set is described by pFormat. The result, returned in the error code, is 0 if supported and HPI_ERROR_INVALID_FORMAT if not supported.

For example, to set an InStream to stereo, MPEG Layer II @ 256kbs, 44.1kHz sample rate:

```
wHE = HPI_FormatCreate(
                &hpiFormat,
                2,                  // stereo channel
                HPI_FORMAT_MPEG_L2, // compression format
                44100,              // sample rate
                256000,             // bits/sec (only used for MPEG)
                0                   // currently no attributes
               );

wHE = HPI_InStreamSetFormat(
                phSubSys,
                hInStream,
                &hpiFormat
                );
```

3.7.13.1 MP3 Variable Bitrate

On adapters that support MP3 encoding, a quality factor between 0 and 100 controls variable bitrate encoding. The quality factor replaces the bitrate in the dwBitrate parameter of HPI_FormatCreate. Setting the “bitrate” to 100 or less automatically activates variable bitrate encoding.

```
wHE = HPI_FormatCreate(
                &hpiFormat,
                2,                  // stereo channel
                HPI_FORMAT_MPEG_L2, // compression format
                44100,              // sample rate
                75,                 // VBR quality setting
                0                   // currently no attributes
               );
```

3.7.14 InStream interaction with Bitstream

Where an instream HPI_DESTNODE_ISTREAM is connected to a bitstream input HPI_SOURCENODE_RAW_BITSTREAM, the bitstream input provides an unformatted stream of data bits. How this data is treated is affected by the format chosen when HPI_InStreamSetFormat is called. There are two valid choices:

HPI_FORMAT_RAW_BITSTREAM
The raw bits are copied into the stream without synchronization

The value returned by HPI_InStreamGetInfoEx into *pdwSamplesRecorded is the number of 32 bit words of data recorded.

HPI_FORMAT_MPEG_L2
After the InStream has been reset the incoming bitstream is scanned for the MPEG2 Layer 2 sync pattern (0xFFFE or 0xFFFC), and input (recording) of the bitstream data is inhibited until this pattern is found. The first word of recorded or monitored data will be this sync pattern, and following data will be word-aligned to it.

This synchronisation process only takes place once after stream reset. There is a small chance that the sync pattern appears elsewhere in the data and the mpeg sync in recorded data won't be byte aligned.

The value returned by HPI_InStreamGetInfoEx into *pdwSamplesRecorded is calculated from the number of bits recorded using the values for dwBitRate and dwSampleRate set by HPI_InStreamSetFormat

```
samples = bits recorded * dwSampleRate / dwBitRate
```

If the samplerate is 44.1kHz, then the samplecount will not be exact.

3.7.15 HPI_InStreamAncillaryReset

Syntax:

```
HW16 HPI_InStreamAncillaryReset(
       HPI_HSUBSYS       *phSubSysHandle,
```

```
HPI_HISTREAM      InStreamHandle,
HW16              wBytesPerFrame,
HW16              wMode,
HW16              wAlignment,
HW16              wIdleBit
);
```

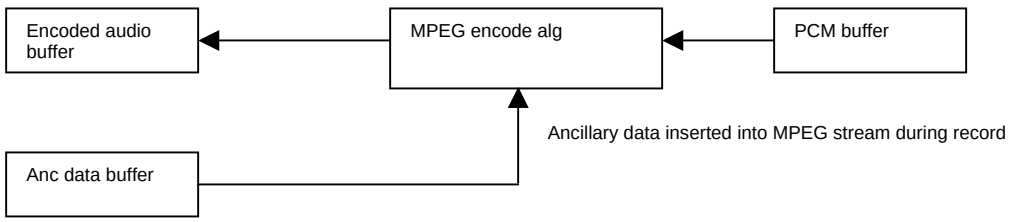
Input Parameters:

hInStreamHandle	Handle to the InStream
wBytesPerFrame	This variable specifies the rate at which data is inserted into the Ancillary data channel. Note that when (wMode== HPI_MPEG_ANC_HASENERGY, see below) an additional 5 bytes per frame are automatically allocated for energy information.
wMode	The mode for the ancillary data extraction to operate in. Valid settings are HPI_MPEG_ANC_RAW and HPI_MPEG_ANC_HASENERGY. The “RAW” mode indicates that the entire ancillary data field is taken up by data from the Anc data buffer. The “HASENERGY” option tells the encoder that the MPEG frames have energy information stored in them (5 bytes per stereo frame, 3 per mono). The encoder will insert the energy bytes before filling the remainder of the ancillary data space with data from the ancillary data buffer.
wAlignment	HPI_MPEG_ANC_ALIGN_LEFT – the wBytesPerFrame data immediately follow the audio data. Spare space is left at the end of the frame. HPI_MPEG_ANC_ALIGN_RIGHT – the wBytesPerFrame data is packed against the end of the frame. Spare space is left at the start of the frame. HPI_MPEG_ANC_ALIGN_FILL – all ancillary data space in the frame is used. wBytesPerFrame or more data is written per frame. There is no spare space. This parameter is ignored for MP3 encoding, effectively it is fixed at HPI_MPEG_ANC_ALIGN_FILL (See Note 2)
wIdleBit	This field tells the encoder what to set all the bits of the ancillary data field to in the case where there is no data waiting to be inserted. Valid values are 0 or 1. This parameter is ignored for MP3 encoding, if no data is available, no data will be inserted (See Note 2)

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Initializes the MPEG Layer II / III Ancillary data channel to support the embedding of wBytesPerFrame bytes into the MPEG bitstream. See the below table for the relationship between wBytesPerFrame and the datarate on the ancillary data channel.



For stereo recording, ancillary data is organised as follows:

Sample rate	Frame rate	Energy rate	Bitrate per byte of ancillary data
48 kHz	41.66 frames/sec	1333.33 bits/sec	333.33 bits/sec
44.1 kHz	38.28 frames/sec	1224.96 bits/sec	306.25 bits/sec
32 kHz	27.77 frames/sec	888.89 bits.sec	222.22 bits/sec

Ancillary data is embedded at the end of each MPEG frame as folllows:

Key:
e = ancillary energy bits (correct number of bits not shown)
d = ancillary data bits
a = audio bits
f = fill bits
x = don't care bits

HPI_MPEG_ANC_ALIGN_LEFT (fill is at end for "raw" case)
wMode = HPI_MPEG_ANC_RAW
a,a,a,d,d,d,d,d,d,…,f,f,f <eof>
wMode = HPI_MPEG_ANC_HASENERGY
a,a,a,d,d,d,d,..,f,f,f,e,e,e,e <eof>

HPI_MPEG_ANC_ALIGN_RIGHT (fill is at front)
wMode = HPI_MPEG_ANC_RAW
a,a,a,f,f,f,d,d,d,d,d,d<eof>
wMode = HPI_MPEG_ANC_HASENERGY
a,a,a,f,f,f,d,d,d,d,..,e,e,e,e <eof>

HPI_MPEG_ANC_ALIGN_FILL (all available ancillary data spots are used)
wMode = HPI_MPEG_ANC_RAW
a,a,a,d,d,d,d,d,d,d,d,d<eof>
wMode = HPI_MPEG_ANC_HASENERGY
a,a,a,d,d,d,d,d,d,d,d,..,e,e,e,e <eof>

Note1 (Calling order):
This call must be made before an HPI_InStreamSetFormat() call.

Note2 (MP3):
Embedded energy information in an MPEG1 layer III (MP3) stream is quite difficult to utilize because its position is not constant within the MPEG frame.
With MP2, the ancillary data field always appears at the end of the MPEG frame.
The parameters wIdleBit and wAlignment are ignored for MP3 due to the inherently more efficient data packing scheme used.

Note3 (Default settings):
A side effect of HPI_InStreamOpen() is that the following default ancillary data settings are made:
Ancillary Bytes Per Frame = 0
Ancillary Mode = HPI_MPEG_ANC_HASENERGY
Ancillary Alignment = HPI_MPEG_ANC_ALIGN_RIGHT

3.7.16 HPI_InStreamAncillaryGetInfo

Syntax:

```
HW16 HPI_InStreamAncillaryGetInfo(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAM hInStreamHandle,
      HW32  *pdwFrameSpace);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.

hInStreamHandle Handle to the InStream

Output Parameters:
*pdwFrameSpace Maximum number of ancillary data frames that can be written to the anc frame buffer.

Description:
Returns information about the ancillary data stream.

3.7.17 HPI_InStreamAncillaryWrite

Syntax:
```
HW16 HPI_InStreamAncillaryWrite(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HISTREAM hInStreamHandle,
      HPI_ANC_FRAME     *pAncFrameBuffer,
      HW32  dwAncFrameBufferSizeInBytes,
      HW32  dwNumberOfAncDataFramesToWrite);
```

Input Parameters:
*phSubSysHandle Pointer to HPI subsystem handle.
hInStreamHandle Handle to the InStream
pdwAncFrameBuffer A pointer to a buffer where the ancillary data frames to write should be placed.
dwAncFrameBufferSize The size of the Ancillary data buffer in bytes (used for a sanity check)
dwNumberOfAncillaryFramesToWrite How many frames to write.

Description:
Writes `dwNumberOfAncDataFramesToWrite` frames from `pAncFrameBuffer` to the stream's ancillary data buffer.

The first bit of ancillary information that follows the vaild audio data is bit 7 of bData[0]. The first 8 bits of ancillary information following valid audio data are from bData[0]. In the case where there are 6 bytes total of ancillary information (48 bits) the last byte inserted in the frame is bData[5].

3.8 Mixer

3.8.1 HPI_MixerOpen

Syntax:
```
HW16 HPI_MixerOpen(
      HPI_HSUBSYS *phSubSysHandle,
      HW16 wAdapterIndex,
      HPI_HMIXER *phMixerHandle );
```

Input Parameters:
*phSubSysHandle Pointer to HPI subsystem handle.
wAdapterIndex Index of adapter to be opened. Ranges from 0 to 15 and corresponds to the Adapter Index set on the adapter hardware.

Output Parameters:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs
*phMixerHandle Handle to the Mixer

Description:
Open and initializes an adapters mixer. An Adapter index is passed in and a Mixer handle is passed back. The handle is used for all future calls to that Mixer. A particular mixer may only be open by one application at one time.

3.8.2 HPI_MixerClose

Syntax:
```
HW16 HPI_MixerClose(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HMIXER hMixerHandle );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hMixerHandle	Handle to the adapter mixer to close

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Closes a mixer.

3.8.3 HPI_MixerGetControl

Syntax:
```
HW16 HPI_MixerGetControl(
      HPI_HSUBSYS *phSubSysHandle,
      HPI_HMIXER  hMixerHandle,
      HW16        wSrcNodeType,
      HW16        wSrcNodeIndex,
      HW16        wDstNodeType,
      HW16        wDstNodeIndex,
      HW16        wControlType,
      HPI_HCONTROL *phControlHandle
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
HMixerHandle	Mixer handle
wSrcNodeType	Source node type i.e HPI_SOURCENODE_OSTREAM
wSrcNodeIndex	Index of particular source node type i.e the 2nd HPI_SOURCENODE_OSTREAM would be specified as index=1
wDstNodeType	Destination node type i.e HPI_DESTNODE_LINEOUT
wDstNodeIndex,	Index of particular source node type i.e the 3rd HPI_DESTNODE_LINEOUT would be specified as index=2
wControlType	Type of mixer control i.e HPI_CONTROL_METER, _VOLUME, _METER or _LEVEL

Output Parameters:

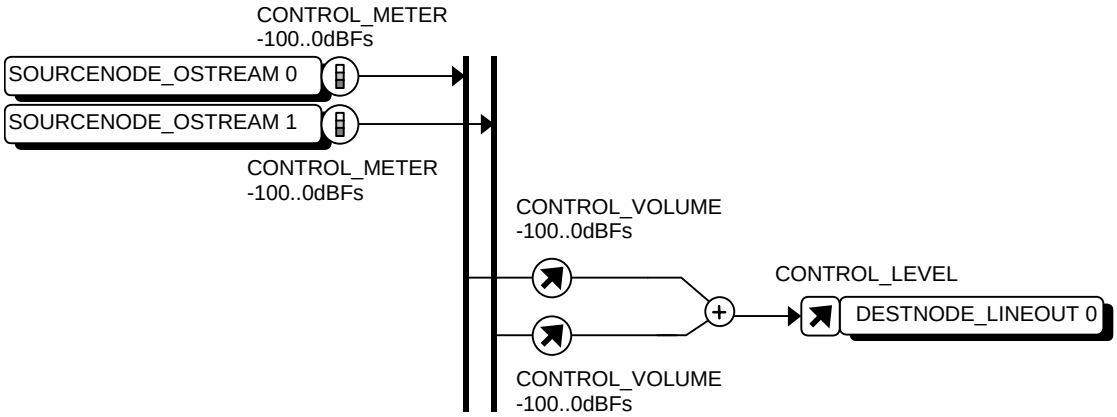
Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*phControlHandle	Handle to the particular control

Description:
Gets a mixer control. The control maybe located in one of three places:
1 - On a connection between a source node and a destination node
You specify both source and destination nodes (via type and type index)
2 - On a source node
You specify the source node and leave the destination node type and index as 0
3 - On a destination node
You specify the destination node and leave the source node type and index as 0

Note:

Not all adapters have controls at all nodes, or between all nodes. Consult the Mixer Map for your particular adapter to find out the location and type of controls in its mixer.

Example
Say you have an audio adapter with a mixer that has the following layout:



You can see that the mixer has two meter controls, located on each of the Outstream source nodes, two volume controls, located between the OutStream sources and the LineOut destination nodes and one level control located on the LineOut destination node.

You would use the following code to obtain a handle to the volume control that lies on the connection between OutStream#1 and LineOut#0:

```
wHE = HPI_MixerGetControl(
                &hSubSys,
                hMixer,
                HPI_SOURCENODE_OSTREAM,
                1,
                HPI_DESTNODE_LINEOUT,
                0,
                HPI_CONTROL_VOLUME,
                &hVolControl
                );
```

You would use the following code to obtain a handle to the level control that lies on the LineOut#0 node:

```
wHE = HPI_MixerGetControl(
                &hSubSys,
                hMixer,
                0,
                0,
                HPI_DESTNODE_LINEOUT,
                0,
                HPI_CONTROL_LEVEL,
                &hLevControl
                );
```

3.8.4 HPI_MixerGetControlByIndex

Syntax
```
HW16 HPI_MixerGetControlByIndex(
                HPI_HSUBSYS       *phSubSysHandle,
                HPI_HMIXER  hControlHandle,
                HW16  *pwSrcNodeType;
                HW16  *pwSrcNodeIndex;
                HW16  *pwDstNodeType;
                HW16  *pwDstNodeIndex;
                HW16  *pwControlType;
```

```
                    HPI_HCONTROL      *phControlHandle;
                    );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HMIXER	HMixer	Mixer handle
HW16	WControlIndex	Index of control to query

Output Parameters:

HW16	*pwSrcNodeType	Source node type i.e HPI_SOURCENODE_OSTREAM
HW16	*pwSrcNodeIndex	
HW16	*pwDstNodeType	
HW16	*pwDstNodeIndex	
HW16	*pwControlType	HPI_CONTROL_TUNER, HPI_CONTROL_….
HPI_HCONTROL	*phControlHandle	Handle of the control if it exists, else 0

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs HPI_ERROR_INVALID_OBJ_INDEX when wControlIndex > number of controls in the mixer being queried. HPI_ERROR_CONTROL_DISABLED when control is disabled.

Description:

Get the attributes and handle of a mixer control given its index.
This function is primarily intended to be used to enumerate all the controls in a mixer. Do this by iterating wControlIndex from 0 until the function returns HPI_ERROR_INVALID_OBJ_INDEX.
The iteration should not be terminated if error HPI_ERROR_CONTROL_DISABLED is returned. This indicates that a control that normally exists is disabled for some reason (possibly hardware failure). The application should not access such controls, but may for instance display a grayed-out GUI control.
For some adapters, a sourcenode type of 0 may be returned for some indices, indicating that there is no control with that index. In this case, continue immediately to the next index.

3.8.5 HPI_ControlQuery

Syntax HW16 HPI_ControlQuery(

```
                    const HPI_HSUBSYS    *phSubSysHandle,
                    const HPI_HCONTROL   hControlHandle,
                    const HW16           wAttrib,
                    const HW32           dwIndex,
                    const HW32           dwParam,
                    HW32                 *pdwSetting
);
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hMixer	Mixer handle
HW16	wAttrib	A control attribute
HW32	dwIndex	Index value
HW32	dwParam	Supplementary parameter

Output Parameters:

HW32	*pdwSetting	One of N possible settings for the control attribute, specified by dwIndex= (and possibly depending on the value of the supplementary parameter)

Return Value:

Error	0 upon success,

HPI_INVALID_OBJ_INDEX when dwIndex >= the number of possible settings of the control
HPI_ERROR_INVALID_OPERATION if the given attribute cannot be queried, or the control does not support the QuerySetting call at all.
HPI_ERROR_INVALID_CONTROL_ATTRIBUTE if the attribute does not exist
HPI_ERROR_INVALID_FUNCTION if the adapter family does not support this API.

Description:
Get the possible settings of an attribute of a mixer control given its index. This is done without disturbing the current setting of the control. Do this by iterating dwIndex from 0 until the function returns HPI_ERROR_INVALID_OBJ_INDEX.

For example, to determine how what bands are supported by a particular tuner, do the following:

```
for (dwIndex=0; dwIndex<10; dwIndex++) {
       wErr= HPI_ControlQuery(phSS,hC, HPI_TUNER_BAND, dwIndex, 0 , AvailableBands[dwIndex]);
       if (wErr !=0) break;
}
numBands=dwIndex;
```

For attributes that have a range, 3 values will be returned for indices 0 to 2: minimum, maximum and step. The supplementary parameter dwParam is used where the possible settings for one attribute depend on the setting of another attribute, e.g. a tuners frequency range depends on which band is selected.

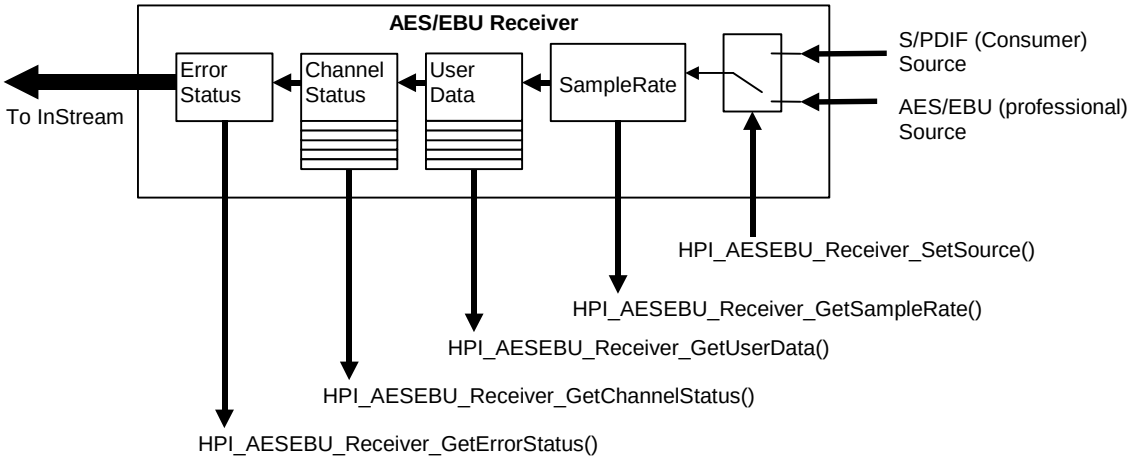
For example, to determine the frequency range of the AM band, do the following:

```
wErr= HPI_ControlQuery(phSS,hC, HPI_TUNER_FREQ, 0, HPI_TUNER_BAND_AM , pdwMinFreq);
wErr= HPI_ControlQuery(phSS,hC, HPI_TUNER_FREQ, 1, HPI_TUNER_BAND_AM , pdwMaxFreq);
wErr= HPI_ControlQuery(phSS,hC, HPI_TUNER_FREQ, 2, HPI_TUNER_BAND_AM , pdwFreqStep);
```

Supported attributes

Associated set function	Attribute	DwParam	Return values
HPI_Tuner_SetBand	HPI_TUNER_BAND	0	HPI_TUNER_BAND_*
HPI_Tuner_SetFrequency	HPI_TUNER_FREQ	HPI_TUNER_BAND_*	Range in kHz
HPI_Tuner_SetGain	HPI_TUNER_GAIN	0	Range in milliBels

3.9 AES/EBU Receiver Control



3.9.1 HPI_AESEBU_Receiver_SetSource

Syntax

```
HW16 HPI_AESEBU_Receiver_SetSource(
            HPI_HSUBSYS       *phSubSysHandle,
            HPI_HCONTROL      hControlHandle,
            HW16              wSource
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to control of type HPI_CONTROL_AESEBU_RECEIVER
wSource	Source of AES/EBU input stream. Can be either:
	HPI_AESEBU_SOURCE_SPDIF S/PDIF unbalanced "consumer" input
	HPI_AESEBU_SOURCE_AESEBU AES/EBU balanced "professional" input

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the source of the AES/EBU digital audio input to either the unbalanced S/PDIF or balancedAES/EBU. Note that not all audio adapters will have both kinds of inputs.

3.9.2 HPI_AESEBU_Receiver_GetSource

```
HW16 HPI_AESEBU_Receiver_GetSource(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16            *pwSource
                        );
```

Returns the receiver source which is the power on default or a value set by the last call to **HPI_AESEBU_Receiver_SetSource**. The variable indicated by **pwSource** will be set to either: **HPI_AESEBU_SOURCE_SPDIF** (indicating S/PDIF unbalanced "consumer" input) or

HPI_AESEBU_SOURCE_AESEBU (indicating AES/EBU balanced "professional" input).

3.9.3 HPI_AESEBU_Receiver_GetSampleRate

```
HW16 HPI_AESEBU_Receiver_GetSampleRate(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW32              *pdwSampleRate
                        );
```

Returns the sample rate of the incoming AES/EBU digital audio stream in ***pdwSampleRate**. This information is obtained from the channel status bits in the digital audio bitstream.

3.9.4 HPI_AESEBU_Receiver_GetUserData

```
HW16 HPI_AESEBU_Receiver_GetUserData(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16              wIndex,       /* ranges from 0..23 */
                        HW16              *pwData       /* returned user data */
                        );
```

Returns one of 24 possible user data bytes pointed to by **wIndex** in ***pwData**.

3.9.5 HPI_AESEBU_Receiver_GetChannelStatus

```
HW16 HPI_AESEBU_Receiver_GetChannelStatus(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16              wIndex,       /* ranges from 0..23 */
                        HW16              *pwData       /* returned channel data */
                        );
```

Returns one of 24 possible channel status bytes pointed to by **wIndex** in ***pwData**.

3.9.6 HPI_AESEBU_Receiver_GetErrorStatus

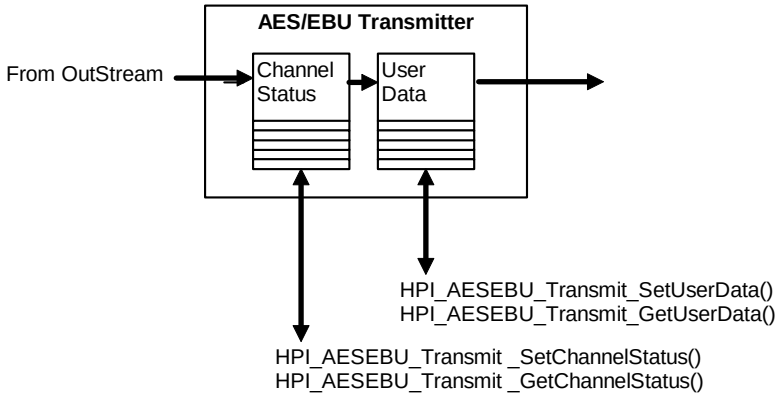
```
HW16 HPI_AESEBU_Receiver_GetErrorStatus(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16              *pwErrorData       /returned error data
                        );
```

Returns the error status of a control of type **HPI_AESEBU_RECEIVER.**
***pwErrorData** has the following information in it:

Bit	Define	Meaning
0	HPI_AESEBU_ERROR_NOT_LOCKED	1 when the receiver PLL (Phase Locked Loop)

		is not locked. This occurs when there is no input or if the received sample frequency is out of range (25-55kHz)
1	HPI_AESEBU_ERROR_POOR_QUALITY	1 when the input signal quality is poor. This indicates that the AES/EBU or S/PDIF bitstream does not meet specifications or that there is excessive noise on the signal.
2	HPI_AESEBU_ERROR_PARITY_ERROR	1 when there is a parity error
3	HPI_AESEBU_ERROR_BIPHASE_VIOLATION	1 when a bi-phase coding error is detetced
4	HPI_AESEBU_ERROR_VALIDITY	1 when the received validity bit is high

3.10 AES/EBU Transmitter Control



3.10.1 HPI_AESEBU_Transmitter_SetUserData

```
HW16 HPI_AESEBU_Transmitter_SetUserData(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16              wIndex,       /* ranges from 0..23 */
                        HW16              wData         /* user data to set */
                        );
```

Sets a byte of the AES/EBU user data of a control of type **HPI_AESEBU_TRANSMITTER**.

3.10.2 HPI_AESEBU_Transmitter_SetChannelStatus

```
HW16 HPI_AESEBU_Transmitter_SetChannelStatus(
                        HPI_HSUBSYS       *phSubSysHandle,
                        HPI_HCONTROL      hControlHandle,
                        HW16              wIndex,       /* ranges from 0..23 */
                        HW16              wData         /* user data to set */
                        );
```

Sets a byte of the AES/EBU channel status data of a control of type **HPI_AESEBU_TRANSMITTER**.

3.10.3 HPI_AESEBU_Transmitter_GetChannelStatus

```
HW16 HPI_AESEBU_Transmitter_GetChannelStatus(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16            wIndex,
                        HW16            *pwData
                        );
```

Provides read access to the channel status data.

3.10.4 HPI_AESEBU_Transmitter_SetClockSource

```
HW16 HPI_AESEBU_Transmitter_SetClockSource(
                        HPI_HSUBSYS       *phSubSysHandle,
                        HPI_HCONTROL      hControlHandle,
                        HW16              wClockSource
                        );
```

Sets the clock source for the AESEBU transmitter. Valid sources are one of ;
HPI_AESEBU_CLOCKSOURCE_ADAPTER and HPI_AESEBU_CLOCKSOURCE_AESEBU_SYNC.

3.10.5 HPI_AESEBU_Transmitter_SetFormat

```
HW16 HPI_AESEBU_Transmitter_SetFormat(
                        HPI_HSUBSYS       *phSubSysHandle,
                        HPI_HCONTROL      hControlHandle,
                        HW16              wOutputFormat
                        );
```

Sets the output electrical format for the AESEBU transmitter. Valid formats are:

HPI_AESEBU_SOURCE_SPDIF	S/PDIF unbalanced "consumer" input
HPI_AESEBU_SOURCE_AESEBU	AES/EBU balanced "professional" input

3.10.6 HPI_AESEBU_Transmitter_GetFormat

```
HW16 HPI_AESEBU_Transmitter_GetFormat(
                        HPI_HSUBSYS     *phSubSysHandle,
                        HPI_HCONTROL    hControlHandle,
                        HW16            *pwOutputFormat
                        );
```

Provides read access to the AESEBU transmitter electrical format (see **HPI_AESEBU_Transmitter_SetFormat**).

3.11 Bitstream Control

The bitstream control is always attached to a node of type HPI_SOURCENODE_RAW_BITSTREAM. The expected clock and data polarities can be set, and a simple measurement of the activity of the received bitstream can be made.

3.11.1 HPI_Bitstream_SetClockEdge

Syntax

```
HW16 HPI_Bitstream_SetClockEdge(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      HW16            wEdgeType
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
wEdgeType	HPI_POLARITY_POSITIVE== data is clocked in on the rising edge of the clock, HPI_POLARITY_NEGATIVE== data is clocked in on the falling edge of the clock. (default)

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the clock edge on which the incoming data is valid.

3.11.2 HPI_Bitstream_SetDataPolarity

Syntax:

```
HW16 HPI_Bitstream_SetDataPolarity(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      HW16            wPolarity
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
wPolarity	HPI_POLARITY_POSITIVE== data bits are read normally (default) HPI_POLARITY_NEGATIVE== data bits are inverted

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Controls whether incoming data is inverted or not.

3.11.3 HPI_Bitstream_GetActivity

Syntax:

```
HW16 HPI_Bitstream_GetActivity(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
```

```
        HW16            * pwClkActivity,
        HW16            * pwDataActivity
);
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*pwClkActivity	1==clock is active, 0==clock is inactive
*pwDataActivity	1 word sampled from the incoming raw data

Description:
Returns two *indicative* measurements of the incoming data stream.
The clock input is deemed inactive if no data bytes are received within a certain number of calls to a polling routine. The time this takes varies according to the number of active streams.
If there is clock activity, the data activity indicator is a sample of 16 bits from the incoming data. If this is persistently 0 or 0xFFFF this may indicate that the data input is inactive.

3.12 Channel Mode Control

3.12.1 HPI_ChannelModeSet

Syntax

```
HW16 HPI_ChannelModeSet(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              wMode
      );
```

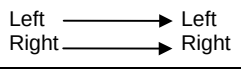
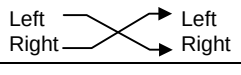
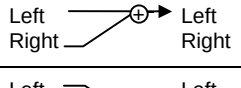
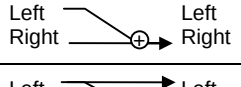
Input Parameters:

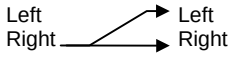
*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to channel mode control
wMode	HPI_CHANNEL_MODE_NORMAL, HPI_CHANNEL_MODE_SWAP, HPI_CHANNEL_MODE_STEREO_TO_LEFT, HPI_CHANNEL_MODE_STEREO_TO_RIGHT.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the channel mode so that one can swap left and right or use only the left and right channels.

HPI_CHANNEL_MODE_NORMAL	Left channel out = left channel in. Right channel out = right channel in	
HPI_CHANNEL_MODE_SWAP	Left channel out = right channel in. Right channel out = left channel in	
HPI_CHANNEL_MODE_STEREO_TO_LEFT (was HPI_CHANNEL_MODE_LEFT_ONLY)	Left channel out = (left channel in + right channel in)/2. Right channel out = mute	
HPI_CHANNEL_MODE_STEREO_TO_RIGHT (was HPI_CHANNEL_MODE_RIGHT_ONLY)	Left channel out = mute Right channel out = (right channel in + left channel in)/2.	
HPI_CHANNEL_MODE_LEFT_TO_STEREO	Left channel out = left channel in.	

(future implementation)	Right channel out = left channel in	
HPI_CHANNEL_MODE_RIGHT_TO_STEREO (future implementation)	Left channel out = right channel in. Right channel out = right channel in	

3.12.2 HPI_ChannelModeGet

Syntax

```
HW16 HPI_ChannelModeSet(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              *wMode
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to channel mode control.
*wMode	Returns one of; HPI_CHANNEL_MODE_NORMAL, HPI_CHANNEL_MODE_SWAP, HPI_CHANNEL_MODE_LEFT_ONLY, HPI_CHANNEL_MODE_RIGHT_ONLY.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Gets the current channel mode setting.

3.13 Compander (Compressor/Expander) Control

The Compander control applies an input-level dependent gain to a signal.

3.13.1 HPI_Compander_Set

Syntax HW16 HPI_Compander_Set(
HPI_HSUBSYS *phSubSysHandle,
HPI_HCONTROL hControlHandle,
HW16 wAttack;
HW16 wDecay;
HW16 wRatio100;
short nThreshold0_01dB;
short nMakeupGain0_01dB;
);

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control
HW16	wAttack	Attack time of compander in milliseconds
HW16	wDecay	Decay or release time of compander in milliseconds
HW16	wRatio100	Compression/expansion level x 100
HW32	nThreshold0_01dB	Threshold level in 100ths of dB
short	nMakeupGain0_01dB;	Makeup gain 100ths of dB

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Set up the operating parameters of a compander control. For example

```
wHE= HPI_Compander_Set(
       phSubsys,
       hCompanderControl,
       5,                    // 5ms attack time
       100,                  // 100ms decay time
       2000,                 // 2:1 compression
       -2000,                // -20dB threshold
       700                   // 7dB makeup gain
       );
```

wAttack, wDecay The timeconstants of the level measurement process that controls the compander. See <section: Meter ballistics> for details.
The output of an RMS detector with these timeconstants is compared with the threshold

wRatio100: Actual compression ratio x 100. In the following table R=wRatio/100

R	ratio	
R>0	R:1	For every R dB above the threshold, the output increases by 1dB
R< 0	1:R	For every 1 dB below the threshold, the output decreases by R dB
R=0		Compander is disabled
1 < R		Compressor: Above the threshold, signal is attenuated

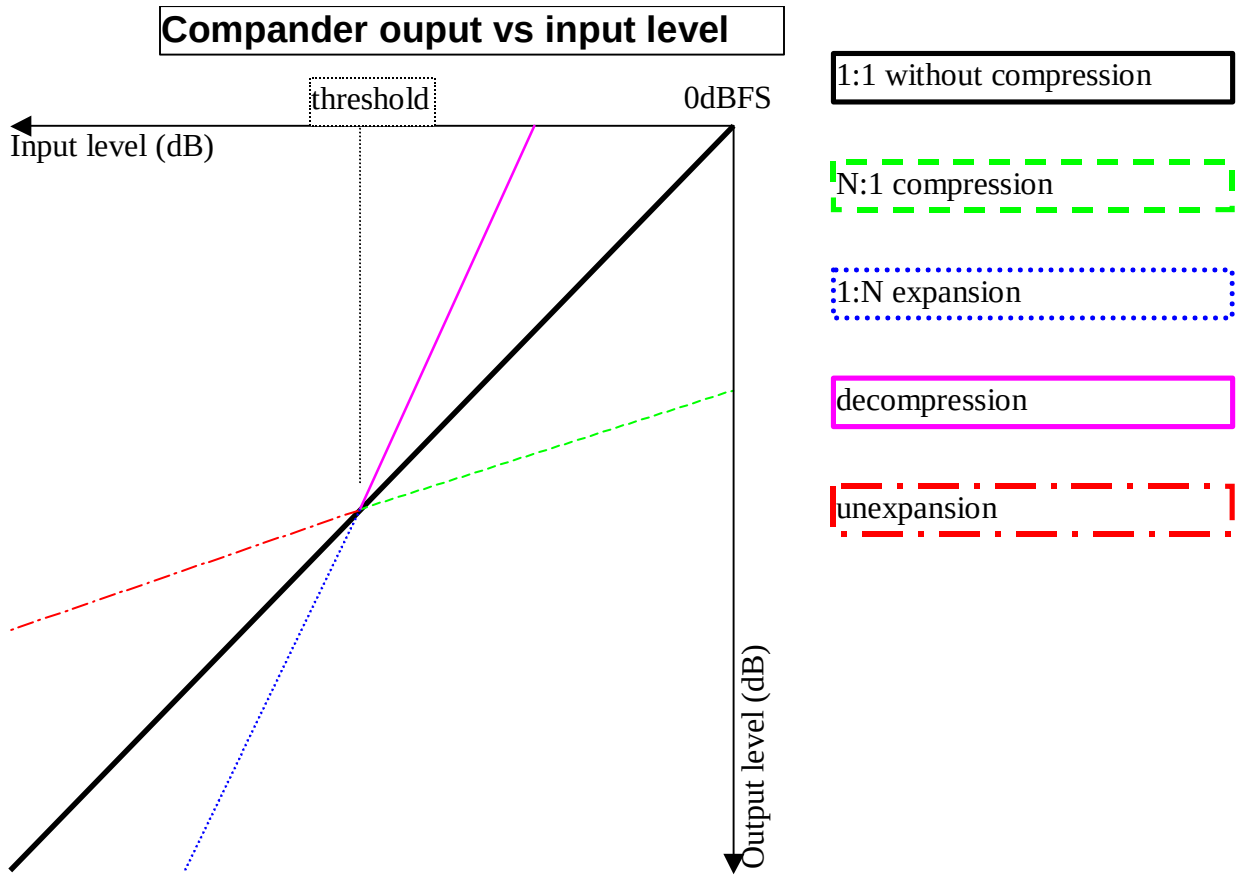
0 < R < 1		*Decompressor: Above the threshold, signal is amplified*
-1 < R < 0		*Unexpander: Below the threshold, signal is amplified*
R < -1		Expander. Below the threshold, signal is attenuated

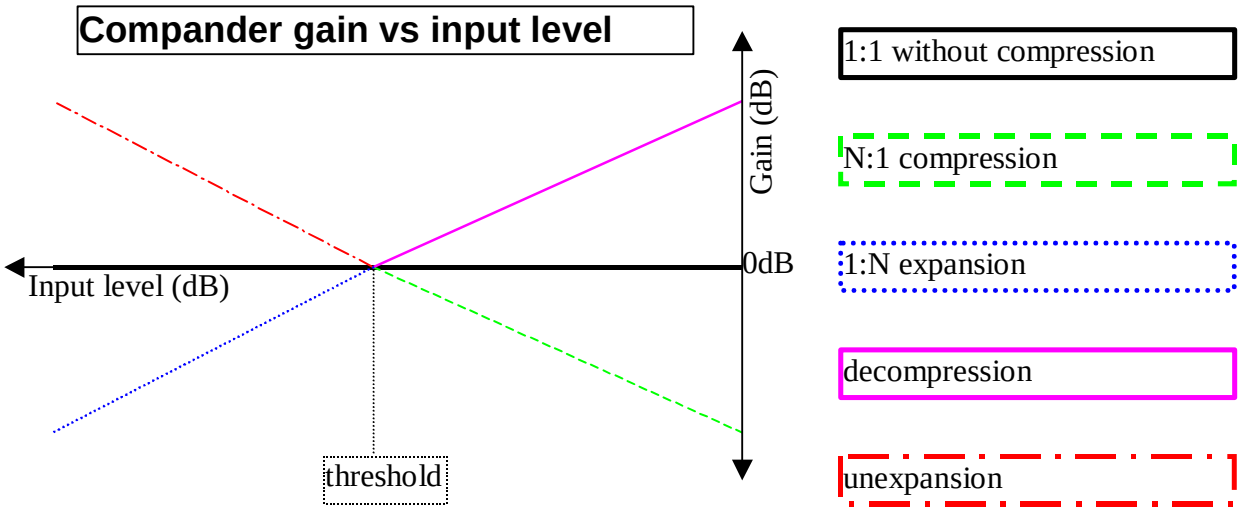
In normal use, |R| > 1, and either traditional compressor or expander functions are performed.

nThreshold0_01dB The threshold is relative to 0dBFS, so only negative values make sense.
For compressor/decompressor it is the level above which the ratio is applied.
For expander/unexpander it is the level below which the ratio is applied.

nMakeupGain0_01dB (units dB/100) is an additional fixed gain that is applied to the output of the compressor. This can be used to bring the maximum level back up to nominal after compression by setting MakeupGain=(SignalMax-Threshold)*(1-1/Ratio)

For example, with an input that has –6dB maximum, threshold at –12dB and compression ratio of 2:1, the output of the compressor will only reach –9dB. A makeup gain of +3dB will bring this back up to the nominal -6dB.





3.13.2 HPI_Compander_Get

Syntax
```
HW16 HPI_Compander_Get(
                HPI_HSUBSYS       *phSubSysHandle,
                HPI_HCONTROL      hControlHandle,
                HW16  *pwAttack;
                HW16  *pwDecay;
                short *pwRatio100;
                short *pnThreshold0_01dB;
                short *pnMakeupGain0_01dB;
                );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*pwAttack	Attack time of compander in milliseconds
HW16	*pwDecay	Decay or release time of compander in milliseconds
short	*pwRatio100	Compression/expansion level x 100
short	*pnThreshold0_01dB	Threshold level in 100ths of dB
short	*pnMakeupGain0_01dB;	Makeup gain 100ths of dB

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Get the current settings of a compander control.
It is permissible for any of the output pointers to be NULL.

3.14 Level Control

Level controls are always situated on a HPI_DESTNODE_LINEOUT (LineOut) or HPI_SOURCENODE_LINEIN (LineIn) node. They are used to control the analog input and output signal level. The level is represented as dBu (0dBu = 0.775VRMS)

For an output, the level set will be the signal level when a digital full-scale signal (0dBfs) is played through an OutStream. For an input the level set will result in a digital full-scale signal being recorded when a signal of that level is fed into the input.

Note that headroom is not taken into account. If you wish to set your output level to, say, +4dBu with 12dB of headroom then you would set the level to 16dBu. You would then need to make sure that your digital signal had an average signal level that was 12dB down from full-scale (0dBFs).

Example: Set the input level on LineIn#0to +14dBu and the output level on Lineout2 to +20dBu.

```
HW16 anGain[2];
HPI_HCONTROL hpiControl;

// get level control on line-in #0
wHE = HPI_MixerGetControl(
       phSubSys, hMixer,
       HPI_SOURCENODE_LINEIN,0,0,0,
       HPI_CONTROL_LEVEL, &hpiControl
       );

// set level to +14dBu
anGain[0]=1400;
anGain[1]=1400;
HPI_LevelSetGain( phSubSys, hpiControl, anGain );

// get level control on line-out #2
wHE = HPI_MixerGetControl(
       phSubSys, hMixer,
       0,0,HPI_DESTNODE_LINEOUT,2,
       HPI_CONTROL_LEVEL, &hpiControl
       );

// set level to +14dBu
anGain[0]=2000;
anGain[1]=2000;
HPI_LevelSetGain( phSubSys, hpiControl, anGain );
```

3.14.1 HPI_LevelSetGain

Syntax

```
HW16 HPI_LevelSetGain(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anGain0_01dB[2]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to control of type HPI_CONTROL_LEVEL
anGain0_01dB[2]	Array containing the level control gain. The gain is in units of 0.01 dBu, where 0dBu = 0.775VRMS. For example a gain of 1400 is 14dBu. Index 0 is the left channel and index 1 is the right channel

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Sets the gain of a HPI_CONTROL_LEVEL control. Note the gain is stereo. The gains will typically range between 0 and +24dBu.

3.14.2 HPI_LevelGetGain

Syntax:

```
HW16 HPI_LevelGetGain(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anGain0_01dB[2]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to control of type HPI_CONTROL_LEVEL

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
anGain0_01dB[2]	Array containing the level control gain. The gain is in units of 0.01 dBu, where 0dBu = 0.775VRMS. For example a gain of 1400 is 14dBu. Index 0 is the left channel and index 1 is the right channel

Description:

Gets the gain of a level control. Note the gain is stereo and is in units of 0.01dBu. The gains will typically range between 0.00 and +24.00dBu.

3.15 Meter Control

Every meter control is able to measure the absolute peak of the signal it is monitoring (see HPI_MeterGetPeak). In some adapters (ASI6000 series with driver v2.68+) each meter control is able to measure the RMS (see HPI_MeterGetRms). The same control handle is used for both functions.

Some adapters (ASI6000 series with driver v2.68+) implement meter controls with ballistics. This means that instead of following the signal instantaneously, meters values have finite attack and decay time constants Ta and Td. For instance when the input is removed, the meter value will decay towards zero (whether or not the meter control is read). It will decay to 37% of its original value in Td seconds. (14% @ 2xTd, 5% @ 3xTd etc)

If meter ballistics are not implemented, when the HPI_MeterGet* function is called the meter statistic for that control will be reset to zero. This means that the period over which the statistic is computed depends on the time between calls to HPI_MeterGet*. If the call is made 100 times a second, then the value returned would reperesent the peak during a 10ms interval of audio.

The lower limit of meter return value, depends on the adapter series. For all ASI4000 adapters, the minimum value returned is –100dB. For ASI6000 adapters, the minimum value returned is –192dB.

3.15.1 HPI_MeterGetPeak

Syntax

```
HW16 HPI_MeterGetPeak(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anLogPeak[HPI_MAX_CHANNELS]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to meter control

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
anLogPeak[]	Array containing the meter peak readings in units of 0.01 decibels relative to full scale digital (dBfs). This will typically range from 0 to -100dB (ASI4000) or –192dB (ASI6000)

Description:
Gets the current peak reading of a meter control. The peak is stereo and the units are in decibels relative to full-scale digital (dbFs). Playing any signal with an amplitude of +/-32767 for a 16bit PCM format will return a peak of 0dB.

If this meter has ballistics then Ta=0, (i.e it jumps up to the highest peak instantly) and Td=650ms, which approximates the decay of a PPM (Peak Program Meter) type meter.

3.15.2 HPI_MeterGetRms

Syntax

```
HW16 HPI_MeterGetRms(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anLogRms[HPI_MAX_CHANNELS]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to meter control.

Output Parameters:

Returns Error	0 upon success, HPI_ERROR_XXX code if an error occurs HPI_ERROR_INVALID_CONTROL if RMS metering is not implemented.
anLogRms[]	Array containing the meter rms readings in units of 0.01 decibels relative to full scale digital (dBfs). This will typically range from 0 to -100dB

Description:
Gets the current RMS reading of a meter control. The peak is stereo and the units are in decibels relative to full-scale digital (dbFs). The RMS measurement depends on the waveform shape. For example, playing a fullscale sinewave (with an amplitude of +/-32767 for a 16bit PCM format) will return a RMS reading of –3dB (compared with a Peak meter reading of 0dBFs), while playing a square wave will return a RMS reading of 0dB and playing an impulsive signal like solo drums will return an RMS value much lower than the peak value.

If this meter has ballistics, then Ta=Td=150ms, which simulates the ballistics of a VU meter.

If RMS metering is not implemented this function returns HPI_ERROR_INVALID_CONTROL, and anLogRms will contain {-10000,-10000} i.e –100dB

3.15.3 HPI_MeterSetPeakBallistics

Syntax `HW16 HPI_MeterSetPeakBallistics(…`

parameters and description are identical to HPI_MeterSetRmsBallistics, but apply to all peak meters on the adapter.

3.15.4 HPI_MeterSetRmsBallistics

Syntax

```
HW16 HPI_MeterSetRmsBallistics(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             nAttack,
      short             nDecay
      );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to ANY meter control
short	nAttack	Attack time constant in milliseconds – **currently not used**
short	nDecay	Decay time constant in milliseconds

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the attack and decay time constants for all the RMS meters on an adapter.

The handle used can be that of any of the meter controls on the particular adapter.

The attack and decay values represent the time constants of the equivalent single pole low pass filter used to create the ballistics. With a time constant of T, if the meter is stable at full scale and the input is suddenly removed, the meter will decay. Similarly, if the meter is at zero and a full scale input is applied

The following table shows the percentage of the final meter value over time, when a constant input is suddenly removed (decay) or applied (attack)

time	meter decay	meter attack
0	100%	0%
T	37%	63%
2T	14%	86%
3T	5%	95%
4T	2%	98%
5T	0.7%	99%

$$decay = initial \times e^{\frac{-t}{T}}$$

$$attack = final \times \left(1 - e^{\frac{-t}{T}}\right)$$

Setting nAttack to 0 gives the meter instantaneous rise time.
Setting nDecay to a value smaller than a few times your meter polling interval is not advised. The meter will appear to read something approaching the instantaneous value at the time of polling rather than the maximum peak since the previous reading.

Example:

```
// set the RMS meter ballistics to 50ms attack, 200ms decay
HPI_MeterSetBallistics( phSubSys, hpiMeterControl, 50, 200  );
```

3.15.5 HPI_MeterGetPeakBallistics

Syntax `HW16 HPI_MeterGetPeakBallistics(…`

Parameters and behaviour are identical to HPI_GetRmsBallistics, but for all Peak meters.

3.15.6 HPI_MeterGetRmsBallistics

Syntax

```
HW16 HPI_MeterGetRmsBallistics(
        HPI_HSUBSYS     *phSubSysHandle,
        HPI_HCONTROL    hControlHandle,
        short           *nAttack,
        short           *nDecay
        );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to ANY meter control
short	*nAttack	Attack time constant in milliseconds
short	*nDecay	Decay time constant in milliseconds

Description:
Gets the current settings for attack and decay time constants for all RMS meters on the adapter.

3.16 Microphone Control

A Microphone control is always located on a source node of type HPI_SOURCE_NODE_MICROPHONE. This node type receives an audio signal from a microphone. If the microphone has adjustable gain, then a VOLUME control will also be present on the node. Currently the Microphone control is only used to turn on/off the microphone's phantom power.

Example Code:

```
// get the control handles
wError = HPI_MixerGetControl(phSubSys, hMixer,
                             HPI_SOURCENODE_MICROPHONE, 0,0,0, HPI_CONTROL_MICROPHONE,
                             &hMicControl);
wError = HPI_MixerGetControl(phSubSys, hMixer,
                             HPI_SOURCENODE_MICROPHONE, 0,0,0, HPI_CONTROL_VOLUME,
                             &hMicVolControl);

// set the microphone volume to 20dB
anGain[0] = 20*HPI_UNITS_PER_dB;
anGain[1] = 20*HPI_UNITS_PER_dB;
wError = HPI_VolumeSetGain( phSubSys, hMicVolControl, anGain );

// turn on the phantom power
wError = HPI_Microphone_SetPhantomPower( phSubSys, hMicControl, 1 );
```

3.16.1 HPI_Microphone_SetPhantomPower

Syntax

```
HW16 HPI_Microphone_SetPhantomPower (
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              wOnOff
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to microphone control.
wOnOff	Should be set to 1 to turn on the microphone phantom power and 0 to turn it off.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the microphone phantom power on or off.

3.16.2 HPI_Microphone_GetPhantomPower

Syntax

```
HW16 HPI_Microphone_GetPhantomPower(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              * wOnOff
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle microphone control.
wOnOff	Returns 1 if the microphone phantom power is on, 0 if off.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Gets the current microphone phantom power setting.

3.17 Multiplexer Control

3.17.1 HPI_Multiplexer_SetSource

```
HW16 HPI_Multiplexer_SetSource(
                        HPI_HSUBSYS  *phSubSysHandle,
                        HPI_HCONTROL hControlHandle,
                        HW16         wSourceNodeType,
                        HW16         wSourceNodeIndex
                        );
```

Sets the input source on a control of type **HPI_CONTROL_MULTIPLEXER**. Typically a multiplexer will have fom 2 to 4 inputs.

A common use of a multiplexer is to select which source node an InStream will record from. To select, say, the analog LineIn source you would use a wSourceNodeType of HPI_SOURCENODE_LINEIN. To select, say, the AES/EBU digital input you would use HPI_SOURCENODE_AESEBU_IN

Example 1 – find and set a linein analog/digital mux

```
// find a multiplexer control on LineIn 2
wHE = HPI_MixerGetControl(phSS, hMixer, HPI_SOURCENODE_LINEIN,2,0,0, HPI_CONTROL_MULTIPLEXER, &hControl);

// set mux to analog linein
wHE = HPI_Multiplexer_SetSource( phSS, hpiControl, HPI_SOURCENODE_LINEIN, 2 );

// set mux to AES/EBU digital in
wHE = HPI_Multiplexer_SetSource( phSS, hpiControl, HPI_SOURCENODE_AESEBU_IN, 2 );
```

3.17.2 HPI_Multiplexer_GetSource

```
HW16 HPI_Multiplexer_GetSource(
                        HPI_HSUBSYS  *phSubSysHandle,
                        HPI_HCONTROL hControlHandle,
                        HW16         &wSourceNodeType,
                        HW16         &wSourceNodeIndex
                        );
```

Returns the current `wSourceNodeType` and `wSourceNodeIndex.`

3.17.3 HPI_Multiplexer_QuerySource

```
HW16 HPI_Multiplexer_QuerySource(
                        HPI_HSUBSYS  *phSubSysHandle,
                        HPI_HCONTROL hControlHandle,
                        HW16         wIndex,
                        HW16         &wSourceNodeType,
                        HW16         &wSourceNodeIndex
                        );
```

Call with wIndex = 0..N to establish valid source node settings for this multiplexer. A non-zero return indicates that the current wIndex is invalid, so querying should be terminated.

3.18 Parametric Equalizer Control

The parametric equalizer control consists of a series of filters that are applied successively to the signal. The number of filters available is obtained by calling HPI_ParametricEQ_GetInfo(), then the characteristics of each filter are configured using HPI_ParametricEQ_SetBand.

The equalizer as a whole can be turned on and off using HPI_ParametricEQ_SetState(). Filters can still be set up when the equalizer is switched off.

Equalizers are typically located on a LineIn input node or an OutStream node.

Obtain a control handle to an equalizer like this:

```
wHE = HPI_MixerGetControl(
        phSubSys,
        hMixer,
        HPI_SOURCENODE_LINEIN, 0,
        0,0,  // No destination node
        HPI_CONTROL_PARAMETRIC_EQ,
        &hControl
        );
```

3.18.1 HPI_ParametricEQ_GetInfo

Syntax

```
HW16 HPI_Equalizer_GetInfo(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              *pwNumberOfBands,
      HW16              *pwOnOff
      );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to equalizer control

Output Parameters:

HW16	*pwNumberOfBands	The number of frequency bands in the equalizer
HW16	*pwOnOff	Enabled state of the equalizer

Return value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Find out the number of bands that an equalizer supports, and whether it is turned on or off

3.18.2 HPI_ParametricEQ_SetState

Syntax

```
HW16 HPI_Equalizer_GetInfo(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      HW16              wOnOff
      );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to equalizer control
HW16	wOnOff	State setting, either

HPI_SWITCH_ON or
HPI_SWITCH_OFF

Return value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Turn an equalizer on or off

3.18.3 HPI_ParametricEQ_SetBand

Syntax

```
HW16 HPI_ParametricEQ_SetBand(
      );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to equalizer control
HW16	wIndex	Index to particular frequency band
HW16	nType	The type of filter to be implemented. One of HPI_FILTER_TYPE_<one of the following> BYPASS, LOW_PASS,HIGH_PASS,BAND_PASS LOW_SHELF,HIGH_SHELF,BAND_EQ ALL_PASS,BANDSTOP
HW32	dwFrequencyHz	Defining frequency of band in Hz.
HW32	dwQ100	Filter Q x 100. Filter sharpness is proportional to Q
short	nGain0_01dB	Gain of band in 100ths of dB

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Set the parameters for one equalizer filter. The overall equalizer response will be a product of all its filter responses.

nType: The kind of filter that the band will implement.
In the following descriptions, low and high frequencies mean relative to dwBandFrequency.
“Elsewhere” means at frequencies different from dwBandFrequency, how different depends on Q (look at the following figures)

HPI_FILTER_TYPE_BYPASS
The filter is bypassed (turned off). Other parameters are ignored
HPI_FILTER_TYPE_LOWPASS
Has unity gain at low frequencies, and attenuation tending towards infinite at high frequencies
nGain0_01dB parameter is ignored.
See “lp” in the following diagrams.
HPI_FILTER_TYPE_HIGHPASS
Has unity gain at high frequencies, and attenuation tending towards infinite at low frequencies
nGain0_01dB parameter is ignored.
(Not illustrated, basically the opposite of lowpass)
HPI_FILTER_TYPE_BANDPASS
Has unity gain at dwFrequencyHz and tends towards infinte attenuation elsewhere
nGain0_01dB parameter is ignored.

See “bp” in the following diagrams.

HPI_FILTER_TYPE_BANDSTOP
Maximum attenuation at dwFrequencyHz, tends towards unity gain elsewhere.
nGain0_01dB parameter is ignored.
See “bs” in the following diagrams.

HPI_FILTER_TYPE_LOWSHELF
Has gain of nGain0_01dB at low frequencies and unity gain at high frequencies.
See “ls” in the following diagrams.

HPI_FILTER_TYPE_HIGHSHELF
Has gain of nGain0_01dB at high frequencies and unity gain at low frequencies.
See “hs” in the following diagrams.

HPI_FILTER_TYPE_EQ_BAND
Has gain of nGain0_01dB at dwFrequencyHz and unity gain elsewhere.
See “eq” in the following diagrams.

dwFrequencyHz is the defining frequency of the filter. It is the center frequency of types HPI_FILTER_TYPE_BANDPASS, HPI_FILTER_TYPE_BANDSTOP, HPI_FILTER_TYPE_EQ_BAND.
It is the -3dB frequency of HPI_FILTER_TYPE_LOWPASS, HPI_FILTER_TYPE_HIGHPASS when Q=1 or resonant frequency when Q>1 and it is the half gain frequency of HPI_FILTER_TYPE_LOWSHELF, HPI_FILTER_TYPE_HIGHSHELF

The maximum allowable value is half the current adapter samplerate i.e. Fs/2. When the adapter samplerate is changed, the equalizer filters will be recalculated. If this results in the band frequency being greater than Fs/2, then the filter will be turned off

dwQ100 controls filter sharpness. To allow the use of an integer parameter, Filter Q = dwQ100/100.

In the following figure, gain is 20dB (10x) (nGain0_01dB=2000) and sampling frequency is normalized to 1Hz (10^0) and nFrequency is 0.1 x sampling frequency. Q=[0.2 0.5 1 2 4 8]

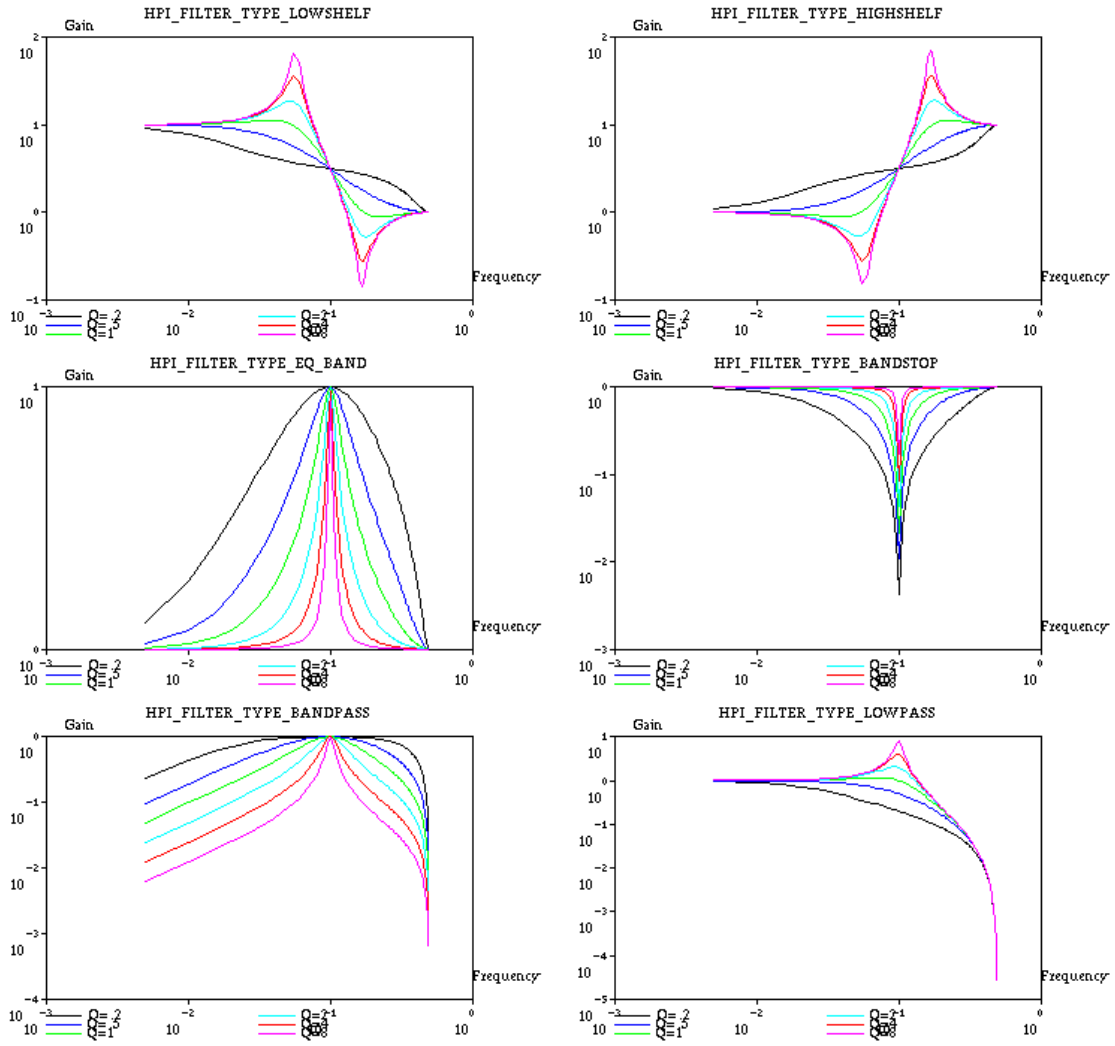
Q can also be thought of as affecting bandwidth or shelf slope of some of these filters

Bandwidth is measured in octaves (between -3 dB frequencies for BPF and notch or between midpoint (dBgain/2) gain frequencies for peaking EQ)

The relationship between bandwidth and Q is
1/Q = 2*sinh[ln(2)/2*bandwidth*omega/sin(omega)] (digital filter using BLT)
or 1/Q = 2*sinh[ln(2)/2*bandwidth]) (analog filter prototype)
Where omega = 2*pi*frequency/sampleRate

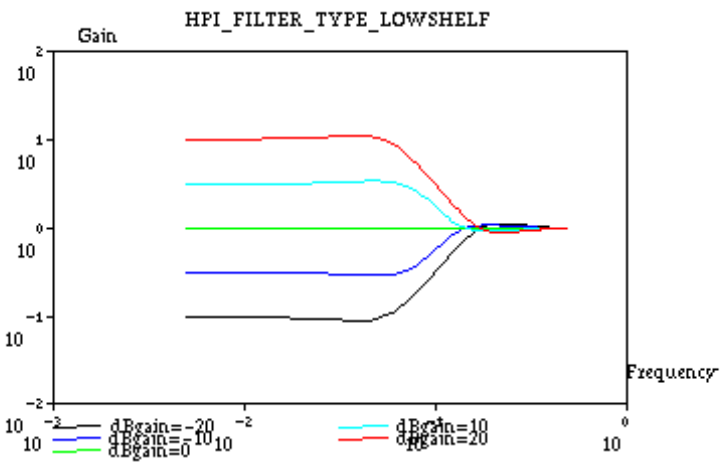
Shelf slope S, a "shelf slope" parameter (for shelving EQ only). When S = 1, the shelf slope is as steep as it can be and remain monotonically increasing or decreasing gain with frequency. The shelf slope, in dB/octave, remains proportional to S for all other values.

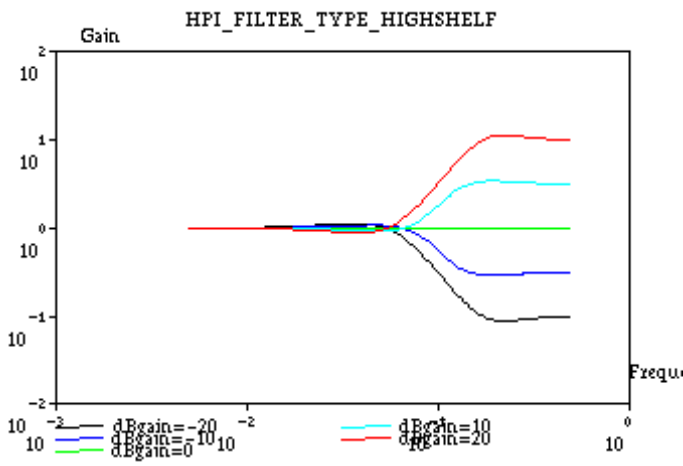
The relationship between shelf slope and Q is 1/Q = sqrt[(A + 1/A)*(1/S - 1) + 2]
where A = 10^(dBgain/40)

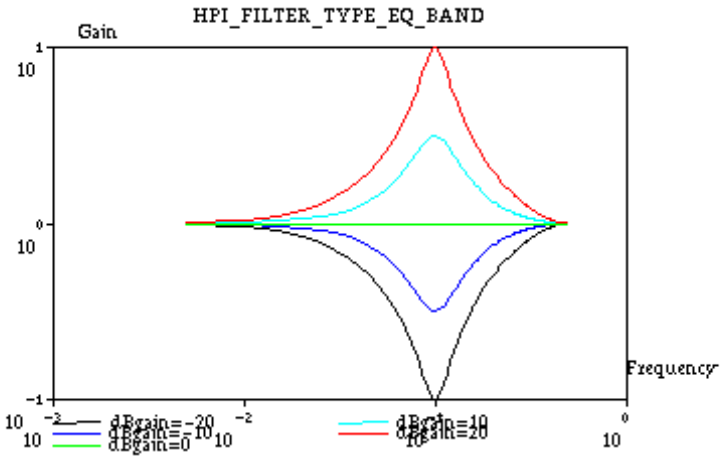


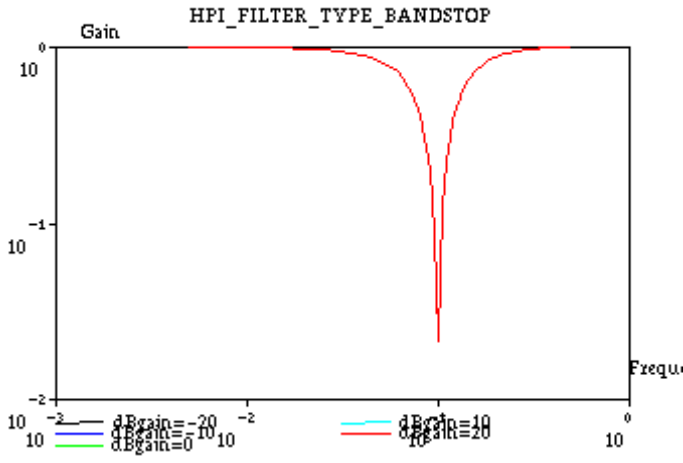
nGain0_01dB
this is expressed in milliBels (100ths of a decibel)
Allowable range is –1000 to +1000 mB. Usable range will likely be less than this.
This parameter is only applicable to the equalizer filter types HPI_FILTER_TYPE_LOWSHELF, HPI_FILTER_TYPE_HIGHSHELF and HPI_FILTER_TYPE_EQ_BAND.Other filters always have unity gain in the passband.

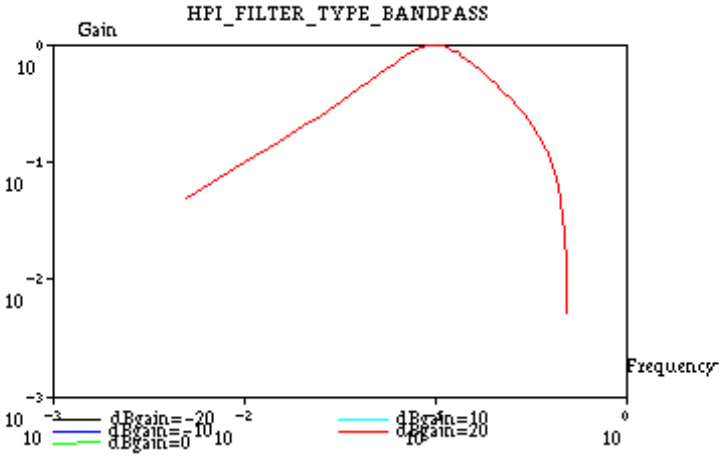
In the following figure, Q=1.0 and sampling frequency is normalized to 1Hz (10^0) and nFrequency is 0.1 x sampling frequency. dBgain=[-20 –10 0 10 20]

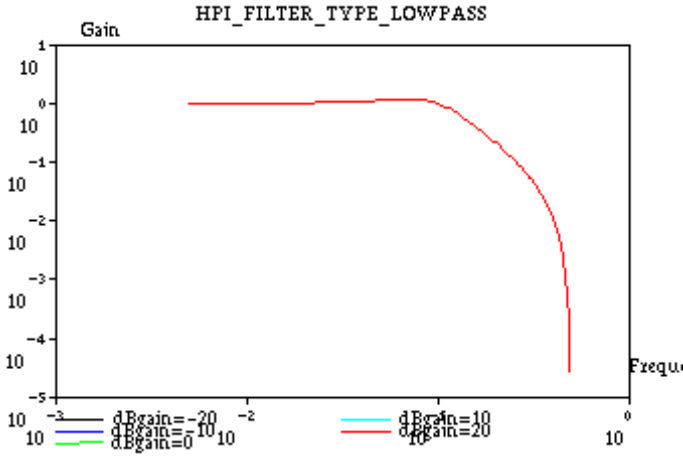












Example:

To produce the upper (red) curve in the above “Filtertype_eq_band” graph:

```
wHE= HPI_ParametricEQ_SetBand(
        phSubsys,
        hMicrophoneControl,
        HPI_FILTER_TYPE_EQ_BAND,
        4410                    // 4.41khz
        100                     // Q=1
        20*100,                 // 20dB
```

)

3.18.4 HPI_ParametricEQ_GetBand

Syntax `HW16 HPI_Equalizer_GetBand( )`

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to equalizer control
HW16	wBandIndex	Index to particular frequency band

Output Parameters:

HW16 *	nBandType	The type of filter. One of LOW_PASS,HIGH_PASS,BAND_PASS,NOTCH LOW_SHELF,HIGH_SHELF,BAND_EQ
HW32 *	dwBandFrequency	Center frequency of band in Hz. -3dB point of low/high pass Half gain point of shelf ,band_eq
HW32 *	dwQ100	Q x 100
short *	nBandGain	Gain of band in 100ths of dB

Return Value:

HW16 Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

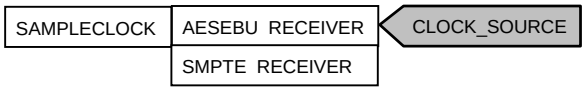
Get the current settings for the band. See HPI_ParametricEQ_SetBand for details of parameter interpretation.

3.19 SampleClock Control

The SampleClock control is used to control the clock source for the adapter. The SampleClock control is always attached to a node of type HPI_SOURCENODE_CLOCK_SOURCE. The SampleClock control provides the following attributes:

Attribute	Access	Values
HPI_SAMPLECLOCK_SOURCE	Set	HPI_SAMPLECLOCK_SOURCE_INTERNAL, HPI_SAMPLECLOCK_SOURCE_AESEBU_SYNC, HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT, HPI_SAMPLECLOCK_SOURCE_WORD, HPI_SAMPLECLOCK_SOURCE_AUTO
HPI_SAMPLECLOCK_SAMPLERATE	Set/Get	0-50000Hz

Depending on the type of clock sources available, other controls may be present on the SOURCENODE_CLOCK_SOURCE. For example if one of the SampleClock sources is AESEBU then an AESEBU_RECEIVER control will be present.



3.19.1 HPI_SampleClock_SetSource

Syntax

```
HW16 HPI_SampleClock_SetSource(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      HW16            wClockSource
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
wClockSource	HPI_SAMPLECLOCK_SOURCE_ADAPTER HPI_SAMPLECLOCK_SOURCE_AESEBU_SYNC HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT HPI_SAMPLECLOCK_SOURCE_WORD HPI_SAMPLECLOCK_SOURCE_AUTO

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the adapter clock source for the samplerate generators.

HPI_SAMPLECLOCK_SOURCE_ADAPTER – the adapter is clocked from its on-board samplerate generator. When this source is set, the adapter samplerate may be set using HPI_SampleClock_SetSampleRate().

HPI_SAMPLECLOCK_SOURCE_AESEBU_SYNC – the adapter is clocked from a dedicated AES/EBU SampleClock input. (NOTE – this define is equivalent to the HPI_SAMPLECLOCK_SOURCE_AESEBU in the pre v2.94 HPI)

HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT – the adapter is clocked from one of the AES/EBU inputs.

HPI_SAMPLECLOCK_SOURCE_WORD – the adapter is clocked from a Word clock (i.e a clock at the frequency of the desired sample rate).

HPI_SAMPLECLOCK_SOURCE_AUTO – the adapter will automatically switch sample clock sources depending on which inputs have a valid clock available on them. More details of this operation are available on the adapter's datasheet.

3.19.2 HPI_SampleClock_SetSourceIndex

Syntax

```
HW16 HPI_SampleClock_SetSourceIndex(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      HW16            wClockSourceIndex
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
wClockSourceIndex	A number ranging from 0 to N-1, where N is the number of AESEBU inputs on the adapter.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the adapter clock source for the samplerate generators to one of the AESUBU inputs. Note, to use this function the source must already be set to HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT.

3.19.3 HPI_SampleClock_GetSource

Syntax

```
HW16 HPI_SampleClock_GetSource(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      HW16            *pwClockSource
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
*pwClockSource	pointer to a var to receive the clock source, which will be one of: HPI_SAMPLECLOCK_SOURCE_ADAPTER HPI_SAMPLECLOCK_SOURCE_AESEBU_SYNC HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT HPI_SAMPLECLOCK_SOURCE_WORD HPI_SAMPLECLOCK_SOURCE_AUTO

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Gets the current adapter samplerate clock source.

3.19.4 HPI_SampleClock_GetSourceIndex

Syntax

```
HW16 HPI_SampleClock_GetSourceIndex(
        HPI_HSUBSYS     *phSubSysHandle,
        HPI_HCONTROL    hControlHandle,
        HW16            *pwClockSourceIndex
        );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
*pwClockSourceIndex	pointer to a var to receive the clock source index

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Gets the current adapter samplerate clock source index. Note, to use this function the source must already be set to HPI_SAMPLECLOCK_SOURCE_AESEBU_INPUT.

3.19.5 HPI_SampleClock_SetSampleRate

Syntax

```
HW16 HPI_SampleClock_SetSampleRate(
        HPI_HSUBSYS     *phSubSysHandle,
        HPI_HCONTROL    hControlHandle,
        HW32            dwSampleRate
        );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
dwSampleRate	0..50000 Hz.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the adapter samplerate when the SampleClock source is HPI_SAMPLECLOCK_SOURCE_ADAPTER.

3.19.6 HPI_SampleClock_GetSampleRate

Syntax

```
HW16 HPI_SampleClock_GetSampleRate(
        HPI_HSUBSYS     *phSubSysHandle,
        HPI_HCONTROL    hControlHandle,
        HW32            *pdwSampleRate
        );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to bitstream control
*pdwSampleRate	pointer to a var to receive the sample rate

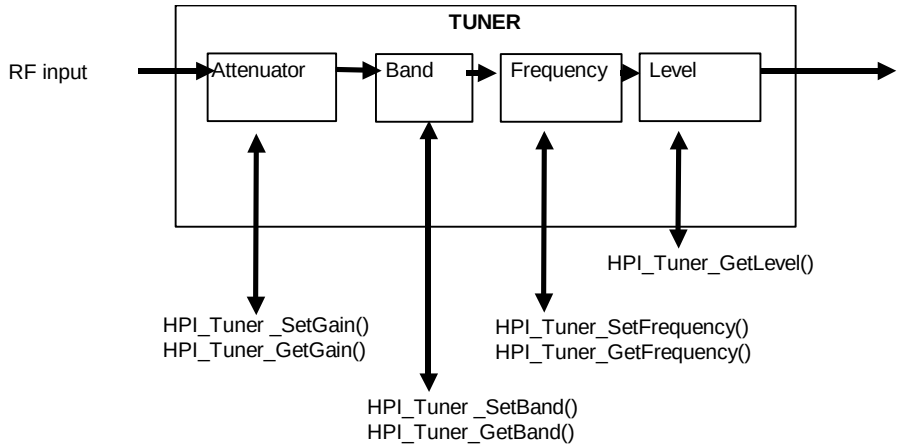
Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Gets the current adapter samplerate.

3.20 Tuner Control

The tuner control sets the band and frequency of a tuner, and measures the RF level



3.20.1 HPI_Tuner_SetBand

Syntax HW16 HPI_Tuner_SetBand(
HPI_HSUBSYS *phSubSysHandle,
HPI_HCONTROL hControlHandle,
HW16 wBand
);

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to tuner control
HW16	wBand	Set the band that the receiver tunes across. Valid values are: HPI_TUNER_BAND_AM HPI_TUNER_BAND_FM HPI_TUNER_BAND_TV HPI_TUNER_BAND_FM_STEREO

Return Value:

Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Set the band that the receiver tunes across. Not all tuners support all bands e.g. AM+FM or TV+FM

NOTE: Please see HPI_ControlQuery() to determine how to find the Bands supported by a particular tuner

3.20.2 HPI_Tuner_GetBand

Syntax HW16 HPI_Tuner_GetBand(
HPI_HSUBSYS *phSubSysHandle,
HPI_HCONTROL hControlHandle,
HW16 *pwBand

```
);
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to tuner control

Output Parameters:

HW16	*pwBand	Band that tuner tunes across

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the current band of a tuner.
It is permissible for any of the output pointers to be NULL.

3.20.3 HPI_Tuner_SetFrequency

Syntax HW16 HPI_Tuner_SetFrequency(

```
HPI_HSUBSYS     *phSubSysHandle,
HPI_HCONTROL    hControlHandle,
HW32            dwFrequency
);
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control
HW16	dwFrequency	Tuner frequency in kHz. Valid values depend on the tuner band setting

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Set the frequency that the tuner is tuned to.

NOTE: Please see HPI_ControlQuery() to determine how to find the frequency supported by a particular tuner band

3.20.4 HPI_Tuner_GetFrequency

Syntax HW16 HPI_Tuner_GetFrequency(

```
HPI_HSUBSYS     *phSubSysHandle,
HPI_HCONTROL    hControlHandle,
HW32            *pdwFrequency
);
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*pdwFrequency	Tuner frequency in kHz.

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the current frequency of a tuner.
It is permissible for any of the output pointers to be NULL.

3.20.5 HPI_Tuner_GetRFLevel

Syntax
```
HW16 HPI_Tuner_GetRFLevel(
    HPI_HSUBSYS     *phSubSysHandle,
    HPI_HCONTROL    hControlHandle,
    HW16            *pwLevel
    );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*pwLevel	The units of this are mBuV (mB micro volts). Range is +/- 100 dBuV

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the RF level of a tuner input in millibel microvolts. Divide the return value by HPI_UNITS_PER_dB to get the level in dBuV.
This only applies to certain bands on certain tuners.
It is permissible for any of the output pointers to be NULL.

3.20.6 HPI_Tuner_GetRawRFLevel

Syntax
```
HW16 HPI_Tuner_GetRawRFLevel(
    HPI_HSUBSYS     *phSubSysHandle,
    HPI_HCONTROL    hControlHandle,
    short           *pwLevel
    );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*short	The units of this depend on the tuner type being access. This is the raw level reading that the tuner returns.

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the RF raw level of a tuner.This is a “raw” value and it will depend on the type of tuner being accessed. This only applies to certain bands on certain tuners.

Tuner Type	Raw RF Level values	Comments
MT4039 (TV/FM)	1..4	Only present in FM mode
MT1384 (AM/FM)	0..255	

It is permissible for any of the output pointers to be NULL.

3.20.7 HPI_Tuner_SetMode

Syntax
```
HW16 HPI_Tuner_SetMode(
                HPI_HSUBSYS       *phSubSysHandle,
                HPI_HCONTROL      hControlHandle,
                HW32              dwMode,
                HW32              dwSetting
                );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control
HW32	dwMode	Should be set to HPI_TUNER_MODE_RSS.
HW32	dwSetting	Should be set to either HPI_TUNER_MODE_RSS_DISABLE or HPI_TUNER_MODE_RSS_ENABLE.

Output Parameters:

HW16	*short	The units of this depend on the tuner type being access. This is the raw level reading that the tuner returns.

Return Value:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
This function turns off the RSS (FM FR level reading) capability for the specified tuner.This only applies to certain bands on certain tuners.

3.20.8 HPI_Tuner_GetVideoStatus

This function has been superceded by HPI_Tuner_GetStatus()

Syntax
```
HW16 HPI_Tuner_GetVideoStatus(
                HPI_HSUBSYS       *phSubSysHandle,
                HPI_HCONTROL      hControlHandle,
                HW16              *pwVideoStatus
                );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*pwVideoStatus	A bitfield with the following bits HPI_TUNER_VIDEO_COLOR_PRESENT HPI_TUNER_VIDEO_HORZ_SYNC_MISSING

HPI_TUNER_VIDEO_IS_60HZ

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the video status of a tuner input
It is permissible for any of the output pointers to be NULL.

3.20.9 HPI_Tuner_GetStatus

Syntax
```
HW16 HPI_Tuner_GetStatus(
                HPI_HSUBSYS       *phSubSysHandle,
                HPI_HCONTROL      hControlHandle,
                HW16              *pwStatusMask
                HW16              *pwStatus
                );
```

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to compander control

Output Parameters:

HW16	*pwStatusMask	A bitfield showing which bits in wStatus are valid. HPI_TUNER_VIDEO_COLOR_PRESENT HPI_TUNER_VIDEO_HORZ_SYNC_MISSING HPI_TUNER_VIDEO_IS_60HZ HPI_TUNER_PLL_LOCKED HPI_TUNER_FM_STEREO
HW16	*pwStatus	A bitfield with the following bits HPI_TUNER_VIDEO_COLOR_PRESENT HPI_TUNER_VIDEO_HORZ_SYNC_MISSING HPI_TUNER_VIDEO_IS_60HZ HPI_TUNER_PLL_LOCKED HPI_TUNER_FM_STEREO

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Get the status of various boolean attributes of a tuner control. The wStatusMask tells you which bits in wStatus are valid, as not all tuners support all the attributes.

It is permissible for any of the output pointers to be NULL.

The definition of the various bits is:

HPI_TUNER_VIDEO_COLOR_PRESENT – Only supported on a tuner capable of tuning to television audio. Tells you that the video signal has color information in it. This is one indication that you have sucessfully tuned to television station.

HPI_TUNER_VIDEO_HORZ_SYNC_MISSING - Only supported on a tuner capable of tuning to television audio. Tells you that the video signal has no horizontal sync. This is is an indication that there is no TV signal present or the signal strength is very low.

HPI_TUNER_VIDEO_IS_60HZ - Only supported on a tuner capable of tuning to television audio. Tells you that the frame rate of the TV signal is 60Hz (vs 50Hz)

HPI_TUNER_PLL_LOCKED – Tells you that the tuner has locked onto a signal and is demodulating it.

HPI_TUNER_FM_STEREO - Only supported on a tuner capable of tuning to FM stereo audio. Tells you that that the tuner has found an FM stereo MPX signal and is decoding it.

3.20.10 HPI_Tuner_SetGain

Syntax HW16 HPI_Tuner_SetBand(
HPI_HSUBSYS *phSubSysHandle,
HPI_HCONTROL hControlHandle,
HW16 wGain
);

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to tuner control
HW16	wGain	Set the gain of the RF attenuator Valid values depend on the adaptor type ASI8700: 0dB or –20 * HPI_UNITS_PER_dB

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Set the RF attenuator gain

3.20.11 HPI_Tuner_GetGain

Syntax HW16 HPI_Tuner_GetGain(
HPI_HSUBSYS *phSubSysHandle,
HPI_HCONTROL hControlHandle,
HW16 *pwGain
);

Input Parameters:

HPI_HSUBSYS	*phSubSysHandle	Pointer to HPI subsystem handle.
HPI_HCONTROL	hControlHandle	Handle to tuner control

Output Parameters:

HW16	*pwBand	Current RF attenuator gain

Return Value:
Error 0 upon success, HPI_ERROR_XXX code if an error occurs

Description:

Get the current RF attenuator gain
It is permissible for any of the output pointers to be NULL.

3.21 Volume Control

Volume controls are usually situated on a "connection" between a source node and a destination node. They can also be present on source nodes, such as Ostream or LineIn. They control the gain/attenuation of the signal passing through them.

For example if you have a -10dB (relative to digital full-scale) signal passing through a volume control with a gain set to -6dB, then the output signal would be -16dB.

The units of gain parameters are milliBels (mB), equivalent to 1/1000 of a Bel, or 1/100 of a decibel. For example a gain of -14.00dB is represented as -1400.

Gain parameters are stereo, stored in a 2 element array, element 0 is the left channel and element 1 is the right channel

A gain value of HPI_GAIN_OFF will set the gain to its maximum attenuation or mute if the adapter supports it.

While most volume controls are attenuation only, some volume controls support gain as well. This is adapter and control dependant. Use the HPI_VolumeGetRange() function to determine if the control supports gain.

3.21.1 HPI_VolumeSetGain

Syntax

```
HW16 HPI_VolumeSetGain(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anGain0_01dB[2]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to volume control
anGain0_01dB[2]	Control gain.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Sets the gain of a volume control.

3.21.2 HPI_VolumeGetGain

Syntax:

```
HW16 HPI_VolumeGetGain(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anGain0_01dB[2]
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to volume control

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
anGain0_01dB[2]	Current value of control gain. If an autofade is in progress, it will be reflected here.

Description:
Gets the current gain setting of a volume control.

3.21.3 HPI_VolumeAutoFadeProfile

Syntax:
```
HW16 HPI_VolumeAutoFadeProfile(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anStopGain0_01dB[HPI_MAX_CHANNELS],
      HW32              dwDurationMs,
      HW16              wProfile
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to volume control
anStopGain0_01dB[2]	The endpoint of the fade.
dwDurationMs	Duration of fade in milliseconds. Minimum duration is 20 ms. Maximum duration is 100 seconds or 100 000 ms. Durations outside this range will be converted to the nearest limit.
wProfile	The profile, or shape of the autofade curve. Allowed values are HPI_VOLUME_AUTOFADE_LOG or HPI_VOLUME_AUTOFADE_LINEAR

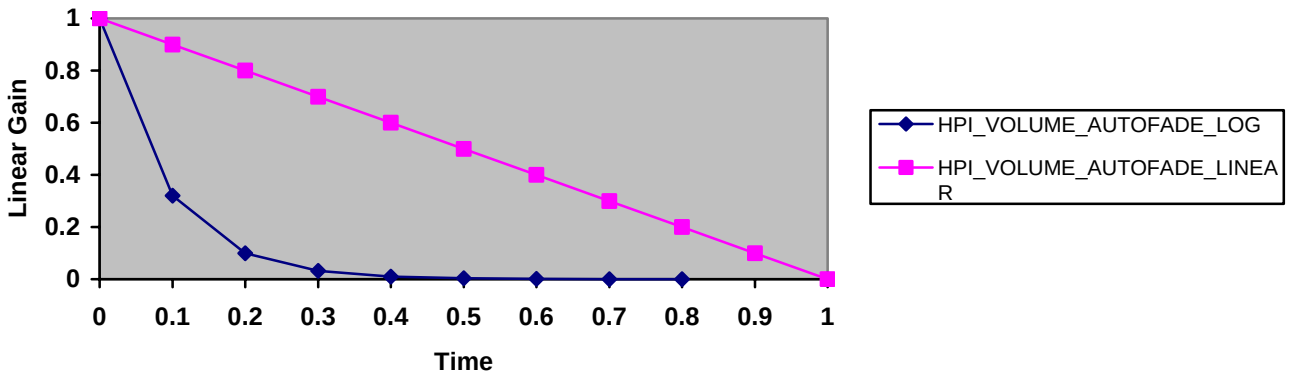
Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

Description:
Starts an auto-fade operation on the specified volume control.
The gain starts at the current gain value and fades up/down to `anStopGain0_01dB[]` over the specified duration.

When wProfile==HPI_VOLUME_AUTOFADE_LOG the gain in dB changes linearly over time.

When wProfile==HPI_VOLUME_AUTOFADE_LINEAR the gain multiplier changes linearly over time. For example half way through the fade time of a fade from 0dB (100%) to –100dB (approx 0%) the the gain will be –6dB (50%)



3.21.4 HPI_VolumeAutoFade

Syntax:
```
HW16 HPI_VolumeAutoFade(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anStopGain0_01dB[HPI_MAX_CHANNELS],
      HW32              dwDurationMs
      );
```

Description:
This is equivalent to calling HPI_VolumeAutoFadeProfile with wProfile= HPI_VOLUME_AUTOFADE_LOG

3.21.5 HPI_VolumeQueryRange (was HPI_VolumeGetRange)

Syntax:
```
HW16 HPI_VolumeQueryRange(
      HPI_HSUBSYS     *phSubSysHandle,
      HPI_HCONTROL    hControlHandle,
      short           *nMinGain_01dB,
      short           *nMaxGain_01dB,
      short           *nStepGain_01dB
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to volume control

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
*nMinGain_01dB	Minimum gain setting in 100ths of a dB
*nMaxGain_01dB	Maximum gain setting in 100ths of a dB
*nStepGain_01dB	Step size in 100ths of a dB

Description:
Gets the max,min and step of the specified volume control.

3.22 VOX Control

VOX controls are always situated on a destination node of type HPI_DESTNODE_ISTREAM. The VOX control specifies the signal level required for recording to begin. Until the signal exceeds the VOX level no samples are recorded to the record stream. After the VOX level has been exceeded, recording continues until the stream is stopped or reset.

A trigger level of -100 dB indicates that recording begins with any level audio signal. A single threshold level is used for both left and right channels. If either channel exceeds the threshold, recording will proceed.

3.22.1 HPI_VoxSetThreshold

Syntax

```
HW16 HPI_VoxSetThreshold(
      HPI_HSUBSYS       *phSubSysHandle,
      HPI_HCONTROL      hControlHandle,
      short             anGain0_01dB
      );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hControlHandle	Handle to VOX control
anGain0_01dB	Array containing the control gain. The gain is always negative (attenuation) and is in units of 0.01 dB. For example a gain of -1400 is -14.00dB.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs

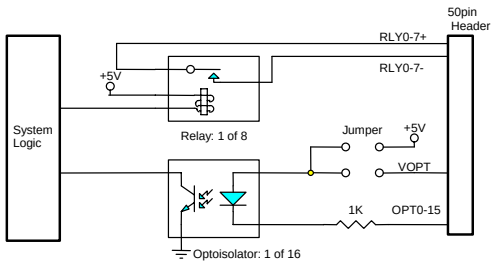
Description:

Sets the threshold of a VOX control. Note the threshold is in units of 0.01dB. The threshold will typically range between 0 and -100dB. A gain value of HPI_GAIN_OFF will set the gain to its maximum attenuation. On startup the VOX control is set to -100 dB.

3.23 General Purpose I/O (GPIO)

AudioScience's implementaion of GPIO provides a robust environment for detecting changes in logic input signals and for controlling external equipment.

On an adapter such as an ASI4336, the bit outputs control relay closurers. HPI_GpioWriteBit() can be used to set the state of each of the relays. Similarly, the inputs on the ASI4336 are mapped 1 to 1 to opto isolated inputs.



3.23.1 HPI_GpioOpen

```
HW16 HPI_GpioOpen(
            HPI_HSUBSYS         *phSubSys,
            HW16                wAdapterIndex,
            HPI_HGPIO           *phGpio,
            HW16                *pwNumberInputBits,
            HW16                *pwNumberOutputBits
            );
```

Opens the GPIO on a particular adapter for reading and writing. It takes as input the handle to the subsytem (**phSubSys**) and the adapter index (**wAdapterIndex**) and returns a handle to the digital i/o object (**hGpio**) and the number of input and output bits (***pwNumberInputBits,*pwNumberOutputBits**). If the adapter does not have any digital I/O functionality, the function will return an error.

3.23.2 HPI_GpioReadBit

```
HW16 HPI_GpioReadBit(
            HPI_HSUBSYS         *phSubSys,
            HPI_HGPIO           hGpio,
            HW16                wBitIndex,
            HW16                *pwBitData
            );
```

Reads the state (0 or 1) from a particular input bit. The input is a handle to the digital i/o object (**hGpio** - returned from HPI_GpioOpen()) and an index which addresses one of the input bits (**wBitIndex**). The index may range from 0 to NumberInputBits-1 (returned by HPI_GpioOpen()). The bit state is returned in ***pwBitData**.

3.23.3 HPI_GpioReadAllBits

```
HW16 HPI_GpioReadAllBits(
```

```
            HPI_HSUBSYS       *phSubSys,
            HPI_HGPIO         hGpio,
            HW16              *pwBitData
            );
```

Reads all input bits at once. The input is a handle to the digital i/o object (**hGpio** - returned from HPI_GpioOpen()). The bit states are returned in ***pwBitData**.

3.23.4 HPI_GpioWriteBit

```
HW16 HPI_GpioWriteBit(
            HPI_HSUBSYS       *phSubSys,
            HPI_HGPIO         hGpio,
            HW16              wBitIndex,
            HW16              wBitData
            );
```

Sets the state of a digital output bit to either 0 or 1 . The input is a handle to the digital i/o object (**hGpio** - returned from HPI_GpioOpen()), an index which addresses one of the output bits (**wBitIndex**) and the value (0 or 1) to write (**wBitData**). The index may range from 0 to NumOutputBits-1 (returned by HPI_GpioOpen());

3.24 Nonvolatile Memory

3.24.1 HPI_NvMemoryOpen

```
HW16 HPI_NvMemoryOpen(
            HPI_HSUBSYS       *phSubSys,
            HW16              wAdapterIndex,
            HPI_HNVMEMORY     *phNvMemory,
            HW16              *pwSizeInBytes
            );
```

Opens the non-volatile memory on a particular adapter for reading and writing. It takes as input the handle to the subsytem (**phSubSys**) and the adapter index (**wAdapterIndex**) and returns a handle to the non-volatile memory (**hNvMemory**) and the size of the memory in bytes (**wSizeInBytes**). If the adapter does not have any non-volatile memory, the function will return an error.

3.24.2 HPI_NvMemoryReadByte

```
HW16 HPI_NvMemoryReadByte(
            HPI_HSUBSYS       *phSubSys,
            HPI_HNVMEMORY     hNvMemory,
```

```
            HW16            wIndex,
            HW16            *pwData
            );
```

Reads a byte from an adapters non-volatile memory. The input is a handle to the non-volatile memory (**hNvMemory** - returned from HPI_NvMemoryOpen()) and an index which addresses one of the bytes in the memory (**wIndex**). The index may range from 0 to SizeInBytes-1 (returned by HPI_NvMemoryOpen()). The byte is returned in ***pwData**.). An error return of HPI_ERROR_NVMEM_BUSY indicates that an attempt to access the NvMem was made before the previous operation has completed. The call should be re-tried.

3.24.3 HPI_NvMemoryWriteByte

```
HW16 HPI_NvMemoryWriteByte(
            HPI_HSUBSYS       *phSubSys,
            HPI_HNVMEMORY     hNvMemory,
            HW16              wIndex,
            HW16              wData
            );
```

Writes a byte to an adapters non-volatile memory. The input is a handle to the non-volatile memory (**hNvMemory** - returned from HPI_NvMemoryOpen()), an index which addresses one of the bytes in the memory (**wIndex**) and the data to write (**wData**). The index may range from 0 to SizeInBytes-1 (returned by HPI_NvMemoryOpen()). An error return of HPI_ERROR_NVMEM_BUSY indicates that an attempt to access the NvMem was made before the previous operation has completed. The call should be re-tried.

3.25 Profile

The Profile object supports profiling of the DSP code. It should be used as a development tool for measuring DSP code operation. Comments in AXPROF.H describe the DSP side of the profiling operation.

3.25.1 HPI_ProfileOpenAll

```
HW16 HPI_ProfileOpenAll(
            HPI_HSUBSYS       *phSubSys,
            HW16              wAdapterIndex,
            HPI_HPROFILE      *phProfiles,
            HW16              *pwMaxProfiles
            );
```

Opens all the profiles on a particular adapter. It takes as input the handle to the subsytem (**phSubSys**) and the adapter index (**wAdapterIndex**) and returns a handle to the profile (**hProfiles**) and the number of profiles available (**wMaxProfiles**). If the adapter does not have profiling enabled, the function will return an error.

The complete profile set of all profiles can be thought of as any array of execution timings/profiles. Each indexed profile corresponds to the execution of a particular segment of DSP code.

Note that HPI_ProfileAllStart() must be called after HPI_ProfileOpenAll() to starting the profiling operation on the DSP.

3.25.2 HPI_ProfileGet

```
HW16 HPI_ProfileGet(
            HPI_HSUBSYS       *phSubSys,
            HPI_HPROFILE      hProfiles,
            HW16              wIndex,
            HW16              *pwSeconds,
            HW32              *pdwMicroSeconds,
            HW32              *pdwCallCount,
            HW32              *pdwMaxMicroSeconds,
            HW32              *pdwMinMicroSeconds
            );
```

Reads a single profile from the DSP's profile store. The input is a handle to the profiles (**hProfiles** - returned from HPI_ProfileOpenAll()) and an index which addresses one of the profiles (**wIndex**). The index may range from 0 to wMaxProfiles (returned by HPI_ProfileOpenAll()). The following returns describe the execution of the profiled segment of DSP code:
wSeconds is the number of seconds spent executing the profiled code.
dwMicroseconds is the fractional seconds spent exectuing the profiled code (measured in μs, range 0-999,999).
dwCallCount is the number of times the profiled code was executed.
dwMaxMicroSeconds is the maximum μs spent executing the profiled code (range 0-8,388,608, or 8 s).
dwMinMicroSeconds is the minimum μs spent executing the profiled code (range 0-8,388,608, or 8 s).

3.25.3 HPI_ProfileStartAll

```
HW16 HPI_ProfileStartAll(
            HPI_HSUBSYS       *phSubSys,
            HPI_HPROFILE      hProfiles
            );
```

Starts the DSP's profile operation. The input is a handle to the profiles (**hProfiles** - returned from HPI_ProfileOpenAll()).

3.25.4 HPI_ProfileStopAll

```
HW16 HPI_ProfileStopAll(
            HPI_HSUBSYS       *phSubSys,
            HPI_HPROFILE      hProfiles
            );
```

Stops the DSP's profile operation. The input is a handle to the profiles (**hProfiles** - returned from HPI_ProfileOpenAll()).

3.25.5 HPI_ProfileGetUtilization

```
HW16  HPI_ProfileGetUtilization(
            HPI_HSUBSYS       *phSubSys,
            HPI_HPROFILE      hProfiles
            HW32              *pdwUtilization
            );
```

Input Parameters:

*phSubSysHandle	Pointer to HPI subsystem handle.
hProfiles	Handle to a profile object.

Output Parameters:

Error	0 upon success, HPI_ERROR_XXX code if an error occurs
pdwUtilization	Returned utilization in 100ths of a percent.

Description:
This function returns the DSP utilization in 100ths of a percent.

3.25.6 HPI_ProfileGetIdleCount

```
HW16  HPI_ProfileGetIdleCount(
            HPI_HSUBSYS       *phSubSys,
            HPI_HPROFILE      hProfiles
            HW16              wIndex,
            HW32              *pdwIdleCycles,
            HW32              *pdwCount
            );
```

Note this function is no longer supported. Please use HPI_ProfileGetUtilization() instead.

Reads the idle count profile from the DSP's profile store. The input is a handle to the profiles (**hProfiles** - returned from HPI_ProfileOpenAll()) and an index which should be set to 0.
pdwIdleCycles is the number of cycles required to execute the IDLE task once.
pdwCount is the number of times the IDLE task executed.

Note: This function is currently only supported on ASI adapters that use TI's C6711 DSP.

Given the above information it is possible to compute the DSP utilization as a percentage, i.e.

```
if(hProfile) {
    static HW32 dwProfilePreviousMs;
    static HW32 dwProfileCurrentMs;
    float  fTimeDiff;
    HW32   dwIdleCycles=0, dwCount=0;
    dwProfileCurrentMs = GetTickCount();
    if((dwProfileCurrentMs - dwProfilePreviousMs) > 1000)
    {
        // more than 1000 ms elapsed since last call ?
        wHE =  HPI_ProfileGetIdleCount( phSubSys, hProfile, 0, &dwIdleCycles, &dwCount);
        dwProfileCurrentMs = GetTickCount();
        fTimeDiff = (float)(dwProfileCurrentMs - dwProfilePreviousMs);
        dwProfilePreviousMs = dwProfileCurrentMs;
        if(!wHE) {
            float fDSPUtilization=0;
            char  szText[64];
            float fTotalIdleCycles=0;
            fTotalIdleCycles = (float)dwIdleCycles * (float)dwCount;
            fDSPUtilization = (1.0 - fTotalIdleCycles/((float)147000*fTimeDiff) ) * 100.0;
            sprintf(szText, "DSP  %4.01f%%",fDSPUtilization);
            SetWindowText( ghstProfileMIPs, szText );
        }
    }
}
```

The above code is specific to a DSP with a clock rate of 147 MHz. The principle is to work out how many cycles in a known time period have been used for Idle processing and then subtract that from the total available. The computation increases in accuracy as DSP utilization approaches 100%.

3.26 Clock

The clock object is a 24 hour clock that runs on the DSP. It keeps time in terms of hours,minutes,seconds and milliseconds.

3.26.1 HPI_ClockOpen

```
HW16 HPI_ClockOpen(
            HPI_HSUBSYS       *phSubSys,
            HW16              wAdapterIndex,
            HPI_HCLOCK        *pClock
            );
```

Opens the clock on a particular adapter. It takes as input the handle to the subsytem (**phSubSys**) and the adapter index (**wAdapterIndex**) and returns a handle to the clock (**hClock**).

3.26.2 HPI_ClockSetTime

```
HW16 HPI_ClockSetTime(
            HPI_HSUBSYS       *phSubSys,
            HPI_HCLOCK        hClock,
            HW16              wHour,
            HW16              wMinute,
            HW16              wSecond,
            HW16              wMilliSecond
            );
```

Sets the time on the clock. The input is a handle to the clock (**hClock** - returned from HPI_ClockOpen()) and the time paratmeters which are as follows:
wHour - the hour in the range of 0-23 (this is a 24 hour clock).
wMinute - the minute in the range of 0-59.
wSecond - the second in the range 0-59.
wMillisecond - the millisecond in the range of 0-1000.

3.26.3 HW16 HPI_ClockGetTime

```
HW16 HPI_ClockGetTime(
            HPI_HSUBSYS       *phSubSys,
            HPI_HCLOCK        hClock,
            HW16              *pwHour,
            HW16              *pwMinute,
            HW16              *pwSecond,
            HW16              *pwMilliSecond
            );
```

Gets the time on the clock. The input is a handle to the clock (**hClock** - returned from HPI_ClockOpen()) and the time parameters returned are as follows:
wHour - the hour in the range of 0-23 (this is a 24 hour clock).
wMinute - the minute in the range of 0-59.
wSecond - the second in the range 0-59.
wMillisecond - the millisecond in the range of 0-1000.

3.27 Structures

3.27.1 HPI_RESOURCE

Syntax:

```
typedef struct
{
      HW16        wBusType;         // HPI_BUS_PNPISA, _PCI, _USB etc
      union
      {
            // PNP ISA bus type resource
            struct
            {
                  HW16  wVendorId;
                  HW16  wDeviceId;
                  HW16  wIoBase;
                  HW16  wIoLength;
                  HW16  wInterrupt;
            } IsaPnp;

            // PCI bus type resource
            struct
            {
                  HW16  wVendorId;
                  HW16  wDeviceId;
                  HW16  wIoBase;
                  HW16  wIoLength;
                  HW32  dwMemBase;
                  HW32  dwMemLength;
                  HW16  wInterrupt;
            } Pci;
      } r;
} HPI_RESOURCE;
```

Description:
This structure holds the definition of an adapter hardware bus resources. It is a union of a different types of bus resources. The particular type of bus is specified with the wBusType member.

You would normally fill in this structure using the HPI_CreateResource function.

3.27.2 HPI_FORMAT

Syntax:

```
typedef struct
{
      HW16  wChannels;        // 1,2...
      HW16  wFormat;          // HPI_FORMAT_PCM16, _AC2, _MPEG ...
      HW32  dwSampleRate;     // 11025, 32000, 44100 ...
      HW32  dwBitRate;        // for MPEG
      HW16  wMode;            // Stereo/JointStereo/Mono etc
      HW32  dwAttributes;     // special stuff like CRC/Copyright on/off
} HPI_FORMAT;
```

3.27.3 HPI_DATA

Syntax:

```
typedef struct
{
      HW32        dwSize;   // length of this structure
      HPI_FORMAT  Format;
      HW8         *pbData;
      HW32        dwDataSize;
} HPI_DATA;
```

3.28 Utility Functions

3.28.1 HPI_GetLastErrorDetail

Syntax:

```
void HPI_GetLastErrorDetail(
      HPI_HSUBSYS *phSubSys,
      char *pszErrorText,
      HW32 **padwError);
```

Description:
Gets h/w specific errors related to the last error generated by the HPI (i.e an error from the last HPI message).

*`pszErrorText` is a pointer to a string containing a textual description of the error (the same as returned by HPI_GetErrorText)

`**padwError` is a pointer to an array of 32bit words that is used to return upto 4 32bit words of hardware specific information, for instance in HPI6000, if FindAdapters fails with error #933, then padError[] has the following info:

padError[0] = DSP address
padError[1] = DSP data writen
padError[2] = DSP data read
padError[3] = DSP index

For example:

```
char szError[256];
HW32 *pdwSpecificError=NULL;

// an error occurs here..

HPI_GetLastErrorDetail( phSubSys, szError,  &pdwSpecificError);
sprintf(gsz, "ERROR %d - %s, %08lX %08lX %08lX %08lX\n", wHE, szError, pdwSpecificError[0],
pdwSpecificError[1], pdwSpecificError[2], pdwSpecificError[3]);
```

3.28.2 HPI_GetErrorText

Syntax:

```
void HPI_GetErrorText(
      HW16 wError,
      char *pszErrorText);
```

Description:
Returns the text for a an error message. The error code is specified by wError and the text is returned in the array specified by *pszErrorText. The array must be at least 100 characters long.

3.28.3 HPI_WaveFormatToHpiFormat

Syntax:
```
HW16 HPI_WaveFormatToHpiFormat(
                    const WAVEFORMATEX *lpFormatEx,
                    HPI_FORMAT *pHpiFormat
                    );
```

Description:
Initializes the HPI_FORMAT structure pointed to by pHpiFormat to match the format in the WAVEFORMATEX structure pointed to by lpFormatEx. Returns 0 if successful, otherwise returns an HPI error code.

3.28.4 HPI_HpiFormatToWaveFormat

Syntax:
```
HW16 HPI_HpiFormatToWaveFormat(
                    const HPI_FORMAT *pHpiFormat,
                    WAVEFORMATEX *lpFormatEx
                    );
```

Description:
Initializes the WAVEFORMATEX structure pointed to by lpFormatEx to match the format in the HPI_FORMAT structure pointed to by pHpiFormat. Returns 0 if successful, otherwise returns an HPI error code.

3.29 Structure Create Functions

3.29.1 HPI_CreateResourceIsaPnp

Syntax:
```
HW16 HPI_ResourceCreateIsaPnp(
      HPI_RESOURCE *pResource,
      HW16  wIsaPnpVendorId,
      HW16  wIsaPnpDeviceId
      HW16  wIoPortBase,
      HW16  wInterrupt
      );
```

Description:
Fills in the HPI_RESOURCE structure pointed to by pResource with the following items:
wIsaPnpVendorId - 3 letter compressed id. The AudioScience Id is “ASI”=0x0669
wIsaPnpDeviceId - device or adapter Id i.e A120 has Id of 0x1200.
wIoPortBase - I/O port base address used by the adapter
wInterrupt - Interrupt used by the adapter. If none used then set to zero.

3.29.2 HPI_FormatCreate

Syntax:
```
HW16 HPI_FormatCreate(
      HPI_FORMAT *pFormat,
      HW16 wChannels,
      HW16 wFormat,
      HW32 dwSampleRate,
      HW32 dwBitRate,
      HW32 dwAttributes
      );
```

Description:
Fills in the HPI_FORMAT structure pointed to by pFormat with the following items:
wChannels - number of channels (1=mono, 2=stereo)
wFormat - audio format i.e 16bit PCM is HPI_FORMAT_PCM16_SIGNED and MPEG Layer II is HPI_FORMAT_MPEG_L2.
dwSampleRate - sample rate in Hertz
dwBitRate - bit rate in bits/second - only used for MPEG formats
dwAttributes - format dependant attributes. Currently only defined for MPEG-1 Layer II on an ASI6xxx. Valid settings are HPI_MPEG_MODE_DEFAULT, HPI_MPEG_MODE_STEREO, HPI_MPEG_MODE_JOINTSTEREO, HPI_MPEG_MODE_DUALCHANNEL.

3.29.3 HPI_DataCreate

Syntax:

```
HW16 HPI_DataCreate(
      HPI_DATA    *pData,
      HPI_FORMAT  *pFormat,
      HW8         *pbData,
      HW32        dwDataSize
      );
```

Description:
Fills in the HPI_DATA structure pointed to by pData with the following items:
pFormat - pointer to a HPI_FORMAT structure containing information about the audio format
pbData - pointer to a buffer of audio data
dwDataSize - size of the buffer pointed to by pData (in bytes)

4. AUDIO FORMATS

4.1 8bit Unsigned PCM

HPI: HPI_FORMAT_PCM8_UNSIGNED

Windows: WAVE_FORMAT_PCM

Sample format

4.2 16bit Signed PCM

HPI: HPI_FORMAT_PCM16_SIGNED

Windows: WAVE_FORMAT_PCM

Sample format

4.3 24bit Signed PCM

HPI: HPI_FORMAT_PC24_SIGNED

Windows: WAVE_FORMAT_PCM

Sample format

4.4 32bit Floating Point PCM (+/-1.0)

HPI: HPI_FORMAT_PCM32_FLOAT

Windows: WAVE_FORMAT_IEEE_FLOAT

Sample format
Each sample is a 32bit word in IEEE754 floating point format. The range is +1.0 to –1.0, which corresponds to digital fullscale.

4.5 32bit Fixed Point PCM

HPI: HPI_FORMAT_PCM32_SIGNED

Windows: ?

Sample format

Each sample is a 32bit word. The most significant 24 bits contain a 24-bit sample and the least significant 8 bits are set to 0.

4.6 MPEG Layer 2

HPI: HPI_FORMAT_MPEG_L2

Windows: WAVE_FORMAT_MPEG

Sample format

The following table shows what combinations of mode and bitrate are possible:

	Sample rates = 32, 44.1 and 48kHz	
Bitrate (kbs)	**Mono**	**Stereo, Joint Stereo or Dual Channel**
32	X	
40		
48	X	
56	X	
64	X	X
80	X	
96	X	X
112	X	X
128	X	X
160	X	X
192	X	X
224		X
256		X
320		X
384		X

4.7 MPEG Layer 3

HPI: HPI_FORMAT_MPEG_L3

Windows: WAVE_FORMAT_MPEG

Sample format

The following table shows what combinations of mode and bitrate are possible:

Bitrate (kbs)	**Mono,Stereo @ 8, 11.025 and 12kHz***	**Mono,Stereo @ 16, 22.05 and 24kHz***	**Mono,Stereo @ 32, 44.1 and 48kHz**
8	X (mono only)	X	
16	X	X	
24	X	X	
32	X	X	X
40	X	X	X
48	X	X	X

56	X	X	X
64	X	X	X
80		X	X
96		X	X
112		X	X
128		X	X
144		X	
160		X	X
192			X
224			X
256			X
320			X

*Available on the ASI6000 series only

4.8 Dolby AC2

HPI: HPI_FORMAT_DOLBY_AC2

Windows: WAVE_FORMAT_DOLBY_AC2

Sample format

The following table shows what combinations of mode and bitrate are possible:

Sample rate (kHz)	Mono bitrate	Stereo bitrate
32	96	192
44.1	128	256
48	128	256

5. EXAMPLE CODE

```
//////////////////////////////////////////////////////////////////////////
//  THPIF.C
//  Test Hardware Programming Interface (HPI)  using HPI functions
//
//
//////////////////////////////////////////////////////////////////////////

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#pragma pack(1)       // we need this for Visual C Win32  //bytes->words in struct??
#include <hpi.h>

// local protos
void THPI_Message( HPI_MESSAGE *phm, HPI_RESPONSE *phr);
void THPI_PrintMsg( HPI_MESSAGE *phm );
void THPI_PrintResult( HPI_RESPONSE *phr );
void HandleError( HW16 wHE );

// file read/write
void THPI_WavFileOpen( short nIndex, char *pszFile);
short THPI_WavFileRead( short nIndex, char *pbData, long lLength);
void THPI_WavFileClose( short nIndex );


void main()
{
       //HPI_MESSAGE hm;
       //HPI_RESPONSE hr;

       HW16                  wHE=0;  // HPI error
       HW32                  dwVersion=0;
       HPI_HSUBSYS           hSubSys;       // handle to audio subsystem
       HPI_RESOURCE          Resource;
       HPI_DATA              Data;
       HPI_FORMAT            Format;
       HW16                  wAdapterIndex=0;
       HW16                  wNumAdapters=0;
       HW16                  awAdapterList[20];
       HW16                  wListLength=20;
       HPI_HOSTREAM          hOutStream0=0;
       HPI_HOSTREAM          hOutStream1=0;
       HW32                  dwStrBufferSize=0;
       HW32                  dwDataToPlay=0;
       HW16                  wState=0;
       HPI_HMIXER            hMixer=0;
       HPI_HCONTROL          hControl=0;

       #define BLOCK_SIZE 8192
       char   abBuffer[BLOCK_SIZE];
       HW32   i=0;
       short  nPeakLeft=0, nPeakRight=0;

       printf("\n** Test HPI using Functions **\n");

       ////////////////////////////////////////////////////////////////
       // open subsystem and find adapters
       hSubSys = HPI_SubSysCreate();
       HandleError( wHE );
       printf("HPI_SubSys_Create\n");

       wHE = HPI_SubSysGetVersion( &hSubSys, &dwVersion );
       HandleError( wHE );
       printf("HPI_SubSys_Version=%lx\n", dwVersion );

       wHE = HPI_SubSysFindAdapters(
```

```
        hSubSys,
        &wNumAdapters,
        awAdapterList,
        wListLength
        );
HandleError( wHE );
printf("HPI_SubSysFindAdapters NumberAdapters=%d ", wNumAdapters);
for(i=0; i<4; i++)
        printf("%ld=%X ", i, awAdapterList[i] );
printf("\n\n");

////////////////////////////////////////////////////////////////
// open adapter
{
HW16    wNumOutStreams;
HW16    wNumInStreams;

wHE = HPI_AdapterOpen(
        &hSubSys,
        wAdapterIndex
        );
HandleError( wHE );
printf("HPI Open_Adapter\n");

wHE = HPI_AdapterGetInfo(
        &hSubSys,
        wAdapterIndex,
        &wNumOutStreams,
        &wNumInStreams
        );
HandleError( wHE );
printf("HPI Get_Adapter_Info\n");
printf("Adapter Index=%d NumOut/Streams=%d NumInStreams=%d\n",
        wAdapterIndex, wNumOutStreams, wNumInStreams );
}

////////////////////////////////////////////////////////////////
// open the out_streams 0 and 1

wHE = HPI_OutStreamOpen(
        &hSubSys,
        wAdapterIndex,
        0,
        &hOutStream0
        );
printf("HPI Open_OStream 0: handle=%lX\n", hOutStream0);
HandleError( wHE );

wHE = HPI_OutStreamOpen(
        &hSubSys,
        wAdapterIndex,
        1,
        &hOutStream1
        );
printf("HPI Open_OStream 1: handle=%lX\n", hOutStream1);
HandleError( wHE );

// find out some info about the device
wHE = HPI_OutStreamGetInfo(
        &hSubSys,
        hOutStream0,
        NULL,
        &dwStrBufferSize,
        NULL
        );
printf("HPI Get_OStream_Info 0: BufferSize=%ld\n", dwStrBufferSize);
HandleError( wHE );

// open the mixer of this adapter
wHE = HPI_MixerOpen(
        &hSubSys,
        wAdapterIndex,
        &hMixer
        );
printf("HPI OpenMixer: handle=%lX\n", hMixer);
```

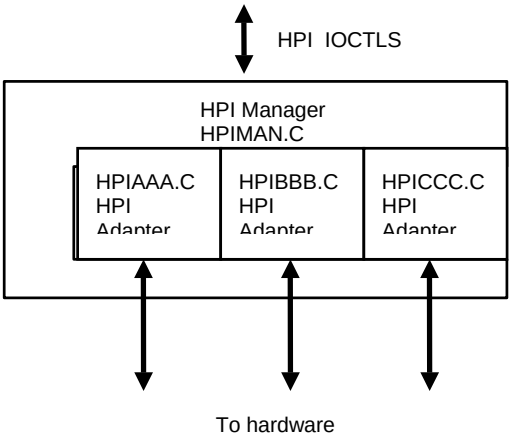
```
HandleError( wHE );


// open some audio files to play
THPI_WavFileOpen( 0, "c:\\audio\\mpeg\\MJACK42.WAV" );
THPI_WavFileOpen( 1, "c:\\audio\\mpeg\\SADE31.WAV" );

// setup format and size of data block
// we will use to send audio data to out_stream
// we would normally get the format directly from the file
// or audio data
wHE = HPI_FormatCreate(
        &Format,
        1,          // mono channel
        HPI_FORMAT_MPEG_L2,
        44100,         //sample rate
        128000,     //128k bits/sec
        0                     // no attributes
        );
HandleError( wHE );
wHE = HPI_DataCreate(
        &Data,
        &Format,
        abBuffer,
        BLOCK_SIZE
        );
HandleError( wHE );
printf("HPI Create_Data: Size=%ld Ch=%d Format=%d SR=%ld BR=%ld Attrib=%08lX\n",
        Data.dwDataSize,
        Data.Format.wChannels,
        Data.Format.wFormat,
        Data.Format.dwSampleRate,
        Data.Format.dwBitRate,
        Data.Format.dwAttributes);

// preload buffer of device
printf("Preload ");
for(i=0; i<(dwStrBufferSize/Data.dwDataSize); i++)
{
        // out_stream #1
        THPI_WavFileRead( 0, Data.pbData, Data.dwDataSize );

        wHE = HPI_OutStreamWrite(
                &hSubSys,
                hOutStream0,
                &Data
                );

        printf(".");

}
printf("\n");

// get mixer meter control on the connection
// between out_stream0 and lineOut0
wHE = HPI_MixerGetControl(
                &hSubSys,
                hMixer,
                HPI_SOURCENODE_OSTREAM,
                0,
                HPI_DESTNODE_LINEOUT,
                0,
                HPI_CONTROL_METER,
                &hControl
                );

// start play back
wHE = HPI_OutStreamStart(
                      &hSubSys,
                      hOutStream0
                      );
HandleError( wHE );
printf("OutStream - start\n");

// keep buffers full of data
```

```
	// while playing
	while(1)
	{
		char szPeakL[40];
		char szPeakR[40];
		short anPeakLog[2] = {0,0};
		short j=0;

		// get the meter level value
		wHE = HPI_MeterControlGetPeak(
			&hSubSys,
			hControl,
			anPeakLog
		);

		// create level bar
		#define BARLEN 10
		nPeakLeft = anPeakLog[0]/3;
		if(nPeakLeft < -BARLEN) nPeakLeft = -BARLEN;
		if(nPeakLeft > 0 ) nPeakLeft = 0;
		nPeakLeft = BARLEN+nPeakLeft;
		strcpy(szPeakL,"");
		for(j=0; j<nPeakLeft; j++)
			strcat(szPeakL, "*");
		nPeakRight = anPeakLog[1]/3;
		if(nPeakRight < -BARLEN) nPeakRight = -BARLEN;
		if(nPeakRight > 0 ) nPeakRight = 0;
		nPeakRight = BARLEN+nPeakRight;
		strcpy(szPeakR,"");
		for(j=0; j<nPeakRight; j++)
			strcat(szPeakR, "*");

		//get state of device
		wHE = HPI_OutStreamGetInfo(
			&hSubSys,
			hOutStream0,
			&wState,
			NULL,
			&dwDataToPlay
		);

		printf("DataToPlay=%06ld State=%d PeakMeter:L=%11s R=%11s\r", dwDataToPlay, wState,
szPeakL, szPeakR );

		if( wState != HPI_STATE_PLAYING )
			break;

		if( dwDataToPlay < dwStrBufferSize/2)
		{
			// read a block of audio data from the file
			if( ! THPI_WavFileRead( 0, Data.pbData, Data.dwDataSize ))
			{
				// write it to the device
				wHE = HPI_OutStreamWrite(
						&hSubSys,
						hOutStream0,
						&Data
						);
			}
		}

		// stop if the user presses a key
		if(kbhit())
			break;
	}
	printf("\n");

	wHE = HPI_OutStreamStop(
			&hSubSys,
			hOutStream0 );
	printf("HPI OStream_Stop\n" );
	HandleError( wHE );

	wHE = HPI_OutStreamClose(
			&hSubSys,
```

```
                        hOutStream0 );
        printf("HPI OStream_Close_0\n" );
        HandleError( wHE );

        wHE = HPI_OutStreamClose(
                        &hSubSys,
                        hOutStream1 );
        printf("HPI OStream_Close_1 \n" );
        HandleError( wHE );

        wHE = HPI_MixerClose(
                        &hSubSys,
                        hMixer );
        printf("HPI Mixer_Close\n" );
        HandleError( wHE );

        wHE = HPI_AdapterClose(
                        &hSubSys,
                        wAdapterIndex );
        printf("HPI Adapter_Close\n" );
        HandleError( wHE );

        THPI_WavFileClose(0);

        getch();
}

FILE *gpFile[4];

void THPI_WavFileOpen( short nIndex, char *pszFile)
{
        gpFile[nIndex] = fopen(pszFile,"rb");
        if(!gpFile[nIndex])
        {
                printf("****ERROR**** - can't open file\n");
                getch();
                exit(0);
        }
        fseek(gpFile[nIndex], 0x50, SEEK_SET);

}

short THPI_WavFileRead( short nIndex, char *pbData, long lLength)
{
        long lNumRead;
        lNumRead = fread( pbData, 1, lLength, gpFile[nIndex] );                    //read WAV file
        if( lNumRead != lLength)
                return(1);
        else
                return(0);
}

void THPI_WavFileClose(short nIndex )
{
        fclose(gpFile[nIndex]);
}


void    HandleError( HW16 wHE )
{
        char szError[256];

        if(wHE)
        {
                HPI_GetErrorText( wHE, szError );
                printf("ERROR %d %s\n", wHE, szError);
                printf("press a key...\n");
                getch();
        }
}
```

6. HPI SOFTWARE ARCHITECTURE

The HPI is written in C. The HPI consists of several modules/source files that are linked into the driver.. The following diagram shows the components that make up the HPI.

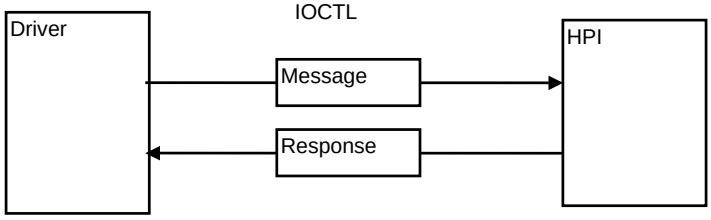


HPIMAN.C is the HPI Manager module. This provides the entry point for the HPI IOCTLs (from the driver) as well as any operating system specific functions that the HPI may need (for example hooking and interrupt or setting up a DMA channel. Each adapter type has its own HPI module, for example the A120 adapter has an HPI module called HPIA120.C

6.1 HPI Driver IOCTL Interface

The driver comunicates with the HPI using (input/output controls) IOCTL function calls. The HPI IOCTL is similar to the Win32 IOCTL - in fact an HPI IOCTL can easily be embedded in a Win32 IOCTL.

The IOCTL consists of a **message** sent from the driver to the HPI, with a **response** sent back. The responses are SampleClockhronous i.e the driver gets the reponse back at the time the IOCTL call is made.



Each message consists of a series of parameters in a structure. The IOCTLs are grouped according to the HPI object they reference. There are 4 groups: SubSystem, Adapter, Stream and Mixer. Each group has a has a particular message format and a series of functions associated with it.

The following example shows the format of the SubSystem IOCTL message and response.

Message

HW16 wLength	= 5
HW8 bType	= HPI_MESSAGE
HW8 bObject	= HPI_OBJ_SUBSYSTEM
HW8 bFunction	= HPI_SUBSYS_FIND_ADAPTERS

Response

HW16 wLength	= 39
HW8 bType	= HPI_RESPONSE
HW8 bObject	= HPI_OBJ_SUBSYSTEM
HW8 bFunction	= HPI_SUBSYS_FIND_ADAPTERS
HW16 wNumAdapters	
HW16 awAdapter[0]	
⋮	
HW16 awAdapter[15]	

6.1.1 IOCTL Entry Point

All IOCTLs enter the HPI through the following entry point in HPIMAN.C

```
void HPI_Message( HPI_MESSAGE *phm, HPI_RESPONSE *phr )
```

phm is a pointer to a HPI_MESSAGE structure, which is a union of the four different structures that support the 4 different object types:

```
typedef struct
{
	HW16		wSize;
	HW8		bType;		// HPI_MSG_MESSAGE, HPI_MSG_RESPONSE
	HW8		bObject;	// HPI_OBJ_SUBSYS, _OBJ_ADAPTER, _OBJ_STREAM, _OBJ_MIXER
	HW16		wFunction;	// HPI_SUBSYS_xxx, HPI_ADAPTER_xxx
	union
	{
		HPI_SUBSYS_MSG		s; //SubSys;
		HPI_ADAPTER_MSG		a; //Adapter;
		HPI_STREAM_MSG		d; //Stream;
		HPI_MIXER_MSG		m; //Mixer;
	}u;
} HPI_MESSAGE;

typedef struct
{
	HW16		wSize;
	HW8		bType;		// HPI_MSG_MESSAGE, HPI_MSG_RESPONSE
	HW8		bObject;	// HPI_OBJ_SUBSYS, _OBJ_ADAPTER, _OBJ_STREAM, _OBJ_MIXER
	HW16		wFunction;	// HPI_SUBSYS_xxx, HPI_ADAPTER_xxx
	HW16		wError;		// HPI_ERROR_xxx
	HW16		wSpecificError; // Adapter specific error
	union
	{
		HPI_SUBSYS_RES		s; //SubSys;
```

```
            HPI_ADAPTER_RES       a; //Adapter;
            HPI_STREAM_RES        d; //Stream;
            HPI_MIXER_RES         m; //Mixer;
        }u;
} HPI_RESPONSE;
```

The individual object message/response structures are as follows:

```
        typedef struct
        {
            HPI_RESOURCE Resource;
        }HPI_SUBSYS_MSG;

        typedef struct
        {
            HW16           wNumAdapters;
            HW16           awAdapter[HPI_MAX_ADAPTERS];
        }HPI_SUBSYS_RES;



        typedef struct
        {
            HW16           wAdapterIndex;
        } HPI_ADAPTER_MSG;

        typedef struct
        {
            HW16           wAdapterType;
            HW16           wAdapterIndex;
            HW16           wNumInDevs;
            HW16           wNumOutStreams;
            HW16           wNumMixers;
            HW8            szAdapterName[16];
        } HPI_ADAPTER_RES;



        typedef struct
        {
            HW16           wAdapterIndex;
            HW16           wOutStreamIndex;
            HW16           wInDevIndex;
            HPI_DATA       Data;
        } HPI_STREAM_MSG;

        typedef struct
        {
            HW16    wOutStreamIndex;
            HW16    wInDevIndex;
        } HPI_STREAM_RES;
```

An example of the HPI_Message function being used is the following for code for SUBSYS_FIND_ADAPTERS:

```
        HPI_MESSAGE hm;
        HPI_RESPONSE hr;

        hm.bType = HPI_TYPE_MESSAGE;
        hm.wSize = sizeof(HPI_MESSAGE);
        hm.bObject = HPI_OBJ_SUBSYSTEM;
        hm.wFunction = HPI_SUBSYS_FIND_ADAPTERS;
        HPI_Message( &hm, &hr);
```

6.2 HPI File Structure

The following table describes the HPI software modules from top level to lowest level.

Module Implementation	Header file
HPIFUNC.C This module implements top level function to message translation and then calls: HPI_Message(phSubSysHandle, &hm, &hr); Also, it has routines for packing message structures. Usage is across all adapters.	HPI.H Contains declarations of all HPI functions available at the top level. Defines error codes. Defines object codes. Defines message structures. Declares entry points for various adapters
HPIODOS.C / HPIOCT32.C IOCTL layer for passing messages from HPIFUNC down into the kernel/VxD layer (for OSes that have a user/kernel architecture). For DOS this layer is not required (at present it is not used).	HPIOCT32.H Header file. Maybe we could use the same one for DOS ?
HPIMAN.C Main role is to pass HPI_Message(phSubSysHandle, &hm, &hr); on to HPI_4000(phm, phr); or HPI_6000(phm, phr); or HPI_6205(phm, phr); or HPI_Usb(phm, phr); It also does some error checking along the way. It is split from HPIFUNC.C so that an IOCTL layer can be inserted. Usage is across all adapters ??	None. Uses HPI.H and HPIOS.H
HPI4000.C Implements code for HPI_4000(phSubSysHandle, &hm, &hr) Mostly passes messages down to the DSP. In an ideal world this module would be very simple and just pass the entire message to the DSP.	None. Uses hpi.h hpipci.h hpi56301.h.
HPI56301.C Handles all message communication with DSP, including bootloading.	HPI56301.H Declares HPI56301_xxxx functions. Declares DPI_ERROR. WHY ? Should be in DPI_CMD.H ? DPI_CMD.H has 56301 specific stuff in it.....
HPI6000.C Handles all message communication with the DSP for the ASI6xxx family of adapters and the ASI5111.	
HPI6205.c Handles all message communication with the DSP for the ASI50xx family of adapters and the ASI87xx family.	
HPIDSPCD.C Utility functions to read DSP code from an asidsp.bin file. Code for specific adapter types	HPIDSPCD.H

can be extracted from the larger monolithic asidsp.bin file.	
HPIOS.C	HPIOS.H
Low-level read write routines. Aim is to isolate code above this layer so that it is portable across multiple OSes. Usage is across all adapters.	Macros for OS specific I/O.
HPIPCI.C	HPIPCI.H
Retrieves PCI configuration information.	

7. OS SPECIFIC IMPLEMENTATIONS

7.1 DOS

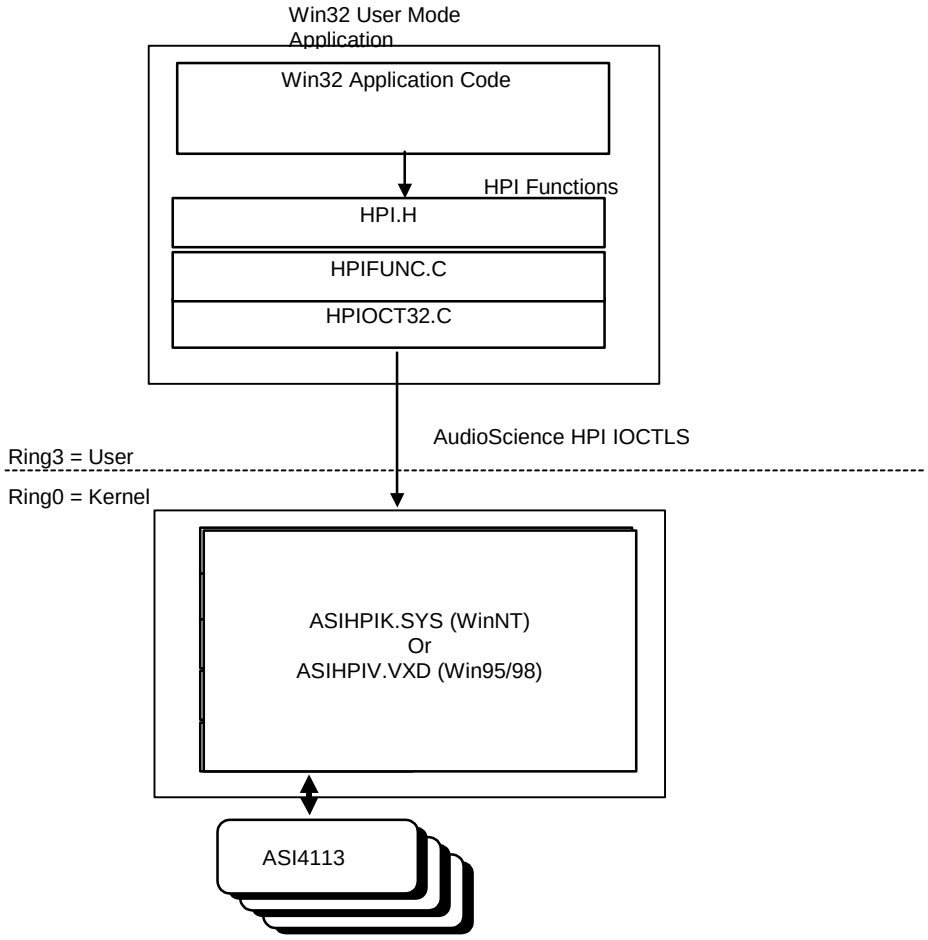
The HPI can be run under real mode DOS. The HPI is compiled/linked into the audio application. Because of the lack of memory in real mode DOS, the DSP files required to run the audio adapters are kept as disk files. These are named HEXXXXY.BIN, where XXXX is the adapter series i.e 4100,4200 and Y is A,B,C.

7.2 Win16

The HPI can be run as a Win16 program under Windows 3.1, Windows95 or 98. The HPI, including any required DSP code, is compiled/linked into the audio application.

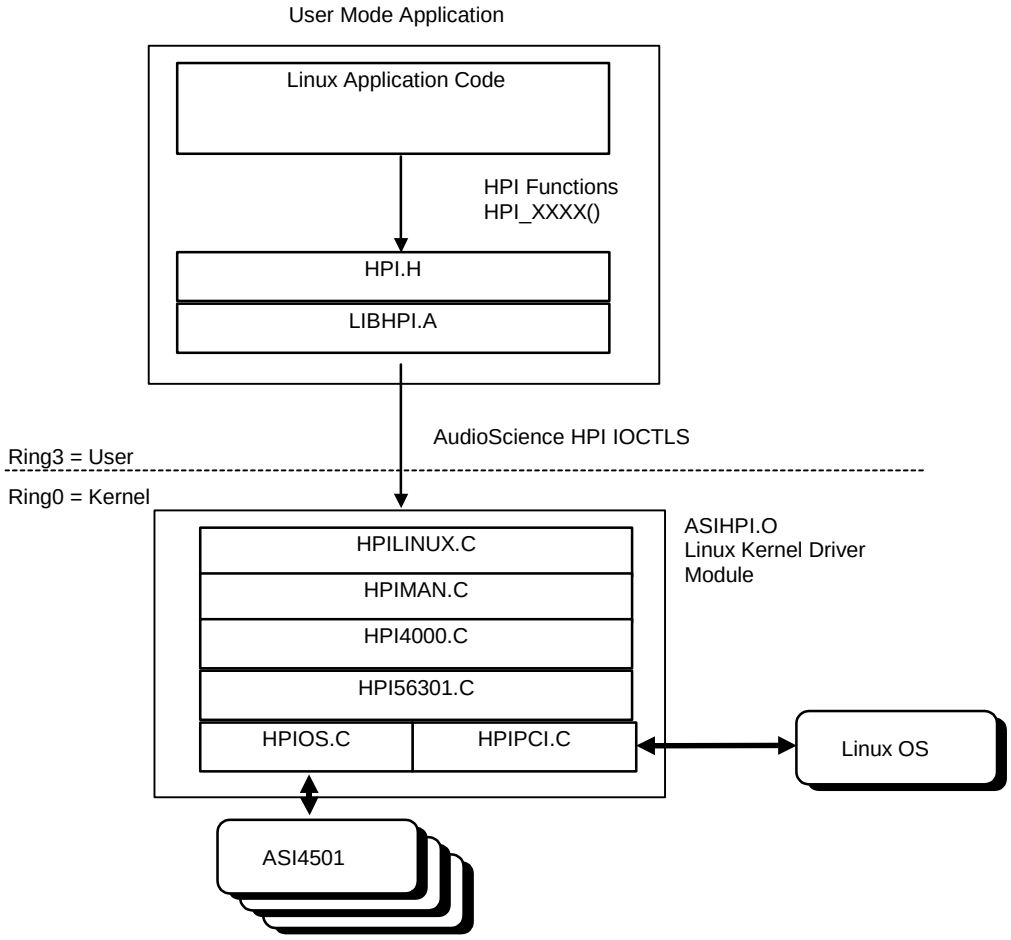
7.3 Win32

The HPI can be run as a Win32 program under Windows 95, 98 and Windows NT/2000. The HPI runs as a kenel mode driver and links to the Win32 audio application via an IOCTL interface. Under Win95/98 the HPI driver is a VXD called **asihpiv.vxd**, under WinNT the driver is a kernel driver called **asihpik.sys** and under Windows2000 it is a WDM driver called **asihpiwn.sys**



7.4 Linux

The HPI can be run under Linux with kernels of version 2.2 or greater. The HPI takes the form of a loadable kernel module called **asihpi.o**.


User Mode Application
Linux Application Code
HPI Functions
HPI_XXXX()
HPI.H
LIBHPI.A
AudioScience HPI IOCTLS
Ring3 = User
Ring0 = Kernel
ASIHPI.O
Linux Kernel Driver
Module
HPILINUX.C
HPIMAN.C
HPI4000.C
HPI56301.C
HPIOS.C
HPIPCI.C
Linux OS
ASI4501

[end]