SPECIFICATION

WavX

Windows Audio Driver Extensions

Applies to asiwavx.h in driver v3.00 and later

1. INTRODUCTION

AudioScience supplies Windows multimedia drivers for its adapters that allow applications to use the adapter's audio functionality under Windows (Win3.X,95,98 and NT). AudioScience adapters contain functionality that is not currently usable under Windows, because it does not map directly into existing multimedia API call structures. However, the additional functionality can be supported within the framework of mixer controls. This new functionality is called wavX.

AudioScience has added mixer control structures to allow access to the following wavX functionality (in no particular order):

- Adapter's non-volatile memory
- Adapter's digital input and output (not AES/EBU)
- Digital AESEBU/SPDIF controls
- AES18 messaging controls
- Auto-fade of volume controls
- VOX (Threshold activated recording)
- Adapter's watchdog timer
- Adapter's serial number

Since many of the above functions do not directly relate to audio they cannot be placed on an existing line. Instead a new destination line labeled as "Adapter" has been added. The Adapter line contains controls that allow the user to access control details through calls to mixerSetControlDetails() and mixerGetControlDetailsl().

In addition to the Mixer extensions AudioScience has added several custom waveOut( ) calls that afford application control of certain advanced functions supported by AudioScience audio adapters.

*All the extensions are referenced in the include file **asiwavx.h**, which is included in all AudioScience driver ZIPs*

2. Serial Number

The adapter serial number control is:
Type : MIXER_CONTROL_CONTROLTYPE_UNSIGNED
Shortname : "Serial"
Longname : "Serial Number"

This is a read-only control that lives on line out 1.

2.1 Locating the serial number control

This section describes the steps to determine the control ID of an ASI adapter's serial number. The description below assumes that a mixer handle, hMixer (of type HMIXER) has been opened using mixerOpen( ).

2.1.1 Search for line out 1

The first step in locating the serial number control is to search the mixer destination lines for line out 1.

```
for(nDest=0; nDest<mxcaps.cDestinations; nDest++)
{
       MIXERLINE     LineInfo;
       LineInfo.dwDestination = nDest;
       mmError=mixerGetLineInfo( hMixer, &LineInfo, MIXER_GETLINEINFOF_DESTINATION);
       if( LineInfo.dwComponentType == MIXERLINE_COMPONENTTYPE_DST_LINE )
           break;
}
```

In the above, `mxcaps` is returned from a call to `mixerGetDevCaps(nIndex, &mxcaps, sizeof(MIXERCAPS)).` On exit of the above loop nDest contains the destination line index of line out 1.

2.1.2 Finding the serial number control

Given nDest, the index of line out 1, the next step is to locate the serial number control.

```
MIXERLINE     LineInfo;
MIXERCONTROL *AllControls;
MIXERLINECONTROLS  LineControls;
DWORD dwSerialControl;

LineInfo.dwDestination = nDest;
mmError=mixerGetLineInfo( hMixer, &LineInfo, MIXER_GETLINEINFOF_DESTINATION);
AllControls = alloc(sizeof(MIXERCONTROL)*pLineInfo.cControls);

for(i=0; i<pLineInfo->cControls; i++)
    AllControls[i].cbStruct = sizeof(MIXERCONTROL);             // set struct size

// setup line controls structure
LineControls.cbStruct = sizeof(MIXERLINECONTROLS);
LineControls.dwLineID = LineInfo.dwLineID;
LineControls.cControls = pLineInfo.cControls;
LineControls.cbmxctrl = sizeof(MIXERCONTROL);
LineControls.pamxctrl = AllControls;
mmError = mixerGetLineControls(hMixer, &LineControls, MIXER_GETLINECONTROLSF_ALL);

for(i=0;i< LineControls.cControls;i++)
{
    if( (AllControls[i].dwControlType==MIXERCONTROL_CONTROLTYPE_UNSIGNED)
               && strstr(AllControls.[i].szName,”Serial”))
```

```
                dwSerialControlID = AllControls[i].dwControlID;
        }
        free(AllControls);
```

The variable dwSerialControlID contains a unique reference to the serial number control.

2.1.3 Reading the serial number

The following assumes that the control ID, dwSerialControlID is known (see above). The Serial Number control is a uniform control, so the number of channels can be set to 1.

```
        MIXERCONTROLDETAILS_UNSIGNED         mxcd_u;
        MIXERCONTROLDETAILS MixerControlDetails;
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwSerialControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = mxcd_u;
        MixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);
        mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
        dwSerialNumber = mxcd_u.dwValue;
```

The serial number has now been assigned to dwSerialNumber.

3. GPIO

Digital input and output consists of a number of bits that can be set/reset:
Type : MIXERCONTROL_CONTROLTYPE_BOOLEAN
Shortname : "Dig In"
Longname : Digital Bit Input"

Type : MIXERCONTROL_CONTROLTYPE_BOOLEAN
Shortname : "Dig Out"
Longname : Digital Bit Output"

The input is read-only.

The control field cMultiple is set to the number of bits the control has. Currently, the maximum allowable number of bits is set to be 32. The control type is therefore a list of cMultiple occurrences of a standard BOOLEAN control.

3.1 Intercepting GPIO OPTO state change messages

The fastest notifications of GPIO Opto input state changes are implemented by intercepting a BROADCAST message that is sent to all top level Windows. The broadcast message has the current GPIO Opto input values embedded in it.

The steps for intercepting the call are as follows.
1) determine the system message number by calling
`nCustomCallbackMessage = RegisterWindowMessage(ASI_MIXER_NOTIFICATION_MESSAGE_ID);`
where `ASI_MIXER_NOTIFICATION_MESSAGE_ID` is defined in asiwavx.h.

2) Assuming the top level window has message processing loop something like

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
        switch( uMsg)
        {
                case …..

                default:
                        return ( DefWindowProc( hWnd, uMsg, wParam, lParam) );
        }
        return 0;
}
```

add a

```
        case nCustomCallbackMessage:
                {
                // lParam = the dwControlID. Check it equals that controlID of the GPIO Rd
control.
                // wParam = the current bit setting of the opto input.
                }
                break;
```

section.

3.1.1 C++ Builder Specifics

In C++ Builder the top level window is the application window (which is hidden). The application window has to be hooked to get access to any broadcast messages.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
        nCustomCallbackMessage = RegisterWindowMessage(ASI_MIXER_NOTIFICATION_MESSAGE_ID);
        Application->HookMainWindow(HookedMainWindow);
}
```

where HookedMainWindow is something like

```
bool __fastcall TForm1::HookedMainWindow(Messages::TMessage &Message)
{
        bool nHandled = false;

        if(Message.Msg == nCustomCallbackMessage)
        {
                if(Message.LParam==Control_GPIO_OPTOS.dwControlID)
                {
                        char szString[256];
                        sprintf(szString,"%d:%d <- ASI_GPIO msg
",RelayState[0],Message.WParam&1);
                        OutputDebugString(szString);
                        Memo1->Lines->Add( (AnsiString)szString);
                 }
                nHandled = true;
        }
    return nHandled;
}
```

The Main Window should be unhooked as the form closes, ie

```
Application->UnhookMainWindow(HookedMainWindow);
```

Additionally, to process the control change messages the WndProc() function should be overridden as follows

```
void __fastcall TForm1::WndProc(Messages::TMessage &Message)
{
        if(Message.Msg==MM_MIXM_CONTROL_CHANGE)
```

```
               HandleMixerControlChange(Message);
       TForm::WndProc(Message);
}
```

4. AESEBU and SPDIF Controls

Two independent custom control types are defined to provide access to options for both transmit and receive.

The following defines are used to set the format of the digital data stream.

```
#define ASI_DIGITAL_FORMAT_AESEBU (1)
#define ASI_DIGITAL_FORMAT_SPDIF (0)
```

4.1 Digital In Control

The digital in control is used to set the format of the input digital stream.
Type : ASI_MIXERCONTROL_CONTROLTYPE_DIGITAL_IN
Shortname : "Dig In Ctrl"
Longname : "Digital In Control"

This control uses a custom control details structure.

```
typedef struct {
unsigned int DigitalFormat :1; /* set to AESEBU or SPDIF */
unsigned int ErrorFlag :1; /* set when receiver is not working correctly */
// expansion allowed here
} ASI_DIGITAL_IN_BITFIELDS;

typedef struct {
DWORD dwSampleRate; /* Not currently implemented - will return 0 */
union {
DWORD dw;
ASI_DIGITAL_IN_BITFIELDS bf;
} u;
} ASI_MIXERCONTROLDETAILS_DIGITAL_IN;
```

4.2 Digital Out Control

The digital out control is used to set the format of the output digital stream.
Type : ASI_MIXERCONTROL_CONTROLTYPE_DIGITAL_OUT
Shortname : "Dig Out Ctrl"
Longname : "Digital Out Control"

This control uses a custom control details structure.

```
typedef struct {
unsigned int DigitalFormat :1; /* set to AESEBU or SPDIF */
// expansion allowed here
} ASI_DIGITAL_OUT_BITFIELDS;

typedef struct {
DWORD dwSampleRate; /* Not currently implemented - will return 0 */
union {
DWORD dw;
ASI_DIGITAL_OUT_BITFIELDS bf;
```

```
        } u;
        } ASI_MIXERCONTROLDETAILS_DIGITAL_OUT;
```

5. Autofade volume controls

ShortName: "AutoFade"
LongName: "Auto Volume Fade"

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_AUTOFADE (MIXERCONTROL_CONTROLTYPE_CUSTOM +
101)
```

The control details structure would be

```
typedef struct {
DWORD dwTargetVolume;
DWORD dwFadeMs;
} ASI_MIXERCONTROLDETAILS_AUTOFADE;
```

Operation of the control is as follows:
This is a write only control.
A call to mixerSetControlDetails() sets the target fade volume to dwTargetVolume.
If cChannels is set to 2, the two control details structures should be provided, with dwFadeMs set to be the same value in both structures. If cChannels is set to 1, the dwTargetVolume is applied to both the left and right channels.

6. Master Volume controls

ShortName: “Master Vol”
LongName: “Master Volume”

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_MASTER_VOLUME
(MIXERCONTROL_CONTROLTYPE_CUSTOM+109)
```

The control details structure is exactly the same as that used for MIXERCONTROL_CONTROLTYPE_VOLUME.
The details structure is of type
MIXERCONTROLDETAILS_SIGNED

7. Channel Mode

The Channel Mode control allows swapping and mixing of the stereo channels within an audio stream. It is implemented as a Microsoft standard SingleSelect control (dwControlType = MIXERCONTROL_CONTROLTYPE_SINGLESELECT).

Option	Description
normal	Left channel out = left channel in. Right channel out = right channel in
swap	Left channel out = right channel in. Right channel out = left channel in
left	Left channel out = (left channel in + right channel in)/2. Right channel out = mute
right	Left channel out = mute Right channel out = (right channel in + left channel in)/2.

8. Time Scaling

ASI6000 series adapters support real-time stretching and shrinking of an audio file with pitch preservation. This functionality is accessed through the following function:

`asiwaveOutSetTimeScale(HWAVEOUT hwo, DWORD dwTimeScale)`

Hwo Handle to an opened waveOut device

dwTimeScale The time stretch factor in 100ths of a percent. Eg,
`dwTimeScale` = 10000 (100%) no time duration change.
`dwTimeScale` = 9000 (90%) the file plays in 90% of the original time, ie 10 sec file will play in 9 sec
`dwTimeScale` = 11000 (110%) the file plays in 110% of the original time, ie 10 sec file will play in 11 sec

The calling sequence is critical. The call to `asiwaveOutSetTimeScale()` should before the file starts playing. A typical sequence might be,

```
waveOutOpen()
waveOutPause()
waveOutWrite()
waveOutWrite() ……
asiwaveOutSetTimeScale()
waveOutStart()
```

9. Equalizer

ShortName: “ParamEQ”
LongName: “Parametric EQ”

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_PARAMETRIC_EQ
(MIXERCONTROL_CONTROLTYPE_CUSTOM+110)
```

The multi-band parametric equalizer is implemented using the ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ structure defined in asiwavx.h.

Several different calls are supported.
Set state turns the parametric EQ on and off.

```
ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ        mxcd_eq;
MIXERCONTROLDETAILS MixerControlDetails;
```

```
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwNvMemControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = mxcd_eq;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ);
        mxcd_eq.dwMsgType = ASI_PARAMETRIC_EQ_SET_STATE;
        mxcd_eq.u.dwState = 1;        // 1 to turn on, 0 to turn off
        mmError = mixerSetControlDetails(hMixer, &MixerControlDetails, MIXER_SETCONTROLDETAILSF_VALUE);
```

Set band is used to configure a particular filter.

```
        ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ        mxcd_eq;
        MIXERCONTROLDETAILS MixerControlDetails;
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwEQControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = &mxcd_eq;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ);
        mxcd_eq.dwMsgType = ASI_PARAMETRIC_EQ_SET_BAND;
        mxcd_eq.u.Band.dwIndex = 0;          // filter number (0-4 on an ASI5111)
        mxcd_eq.u.Band.dwType = ASI_FILTER_TYPE_LOWSHELF;
        mxcd_eq.u.Band.dwFrequencyHz = 2000;
        mxcd_eq.u.Band.dwQ100 = 100;
        mxcd_eq.u.Band.nGain0_01dB = 0;
        mmError = mixerSetControlDetails(hMixer, &MixerControlDetails, MIXER_SETCONTROLDETAILSF_VALUE);
```

In the following descriptions, low and high frequencies mean relative to dwFrequencyHz.
“Elsewhere” means at frequencies different from dwFrequencyHz, how different depends on Q (look at the following figures). In the figures that follow below, mentally replace the HPI_ part of the label with ASI_.

ASI_FILTER_TYPE_BYPASS
: The filter is bypassed (turned off). Other parameters are ignored

ASI_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.

ASI_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)

ASI_FILTER_TYPE_BANDPASS
: Has unity gain at dwFrequencyHz and tends towards infinite attenuation elsewhere
: nGain0_01dB parameter is ignored.
: See “bp” in the following diagrams.

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

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

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

ASI_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 ASI_FILTER_TYPE_BANDPASS, ASI_FILTER_TYPE_BANDSTOP, ASI_FILTER_TYPE_EQ_BAND.
It is the -3dB frequency of ASI_FILTER_TYPE_LOWPASS, ASI_FILTER_TYPE_HIGHPASS when Q=1 or resonant frequency when Q>1 and it is the half gain frequency of ASI_FILTER_TYPE_LOWSHELF, ASI_FILTER_TYPE_HIGHSHELF.

The maximum allowable value is half the current adapter sample rate i.e. Fs/2. When the adapter sample rate 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 or “Q”. 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]

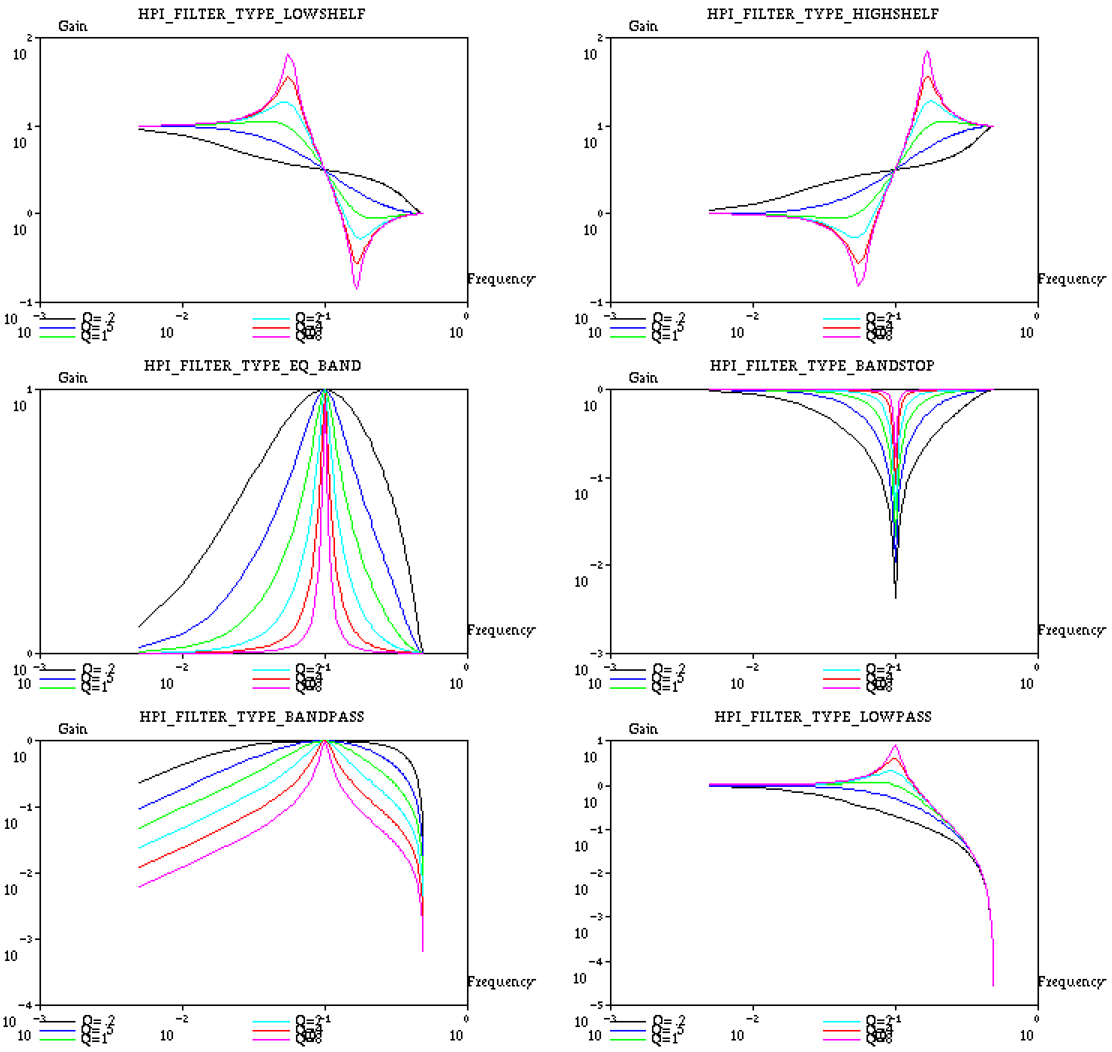
Q can also be thought of as affecting bandwidth or shelf slope of some of these filters

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

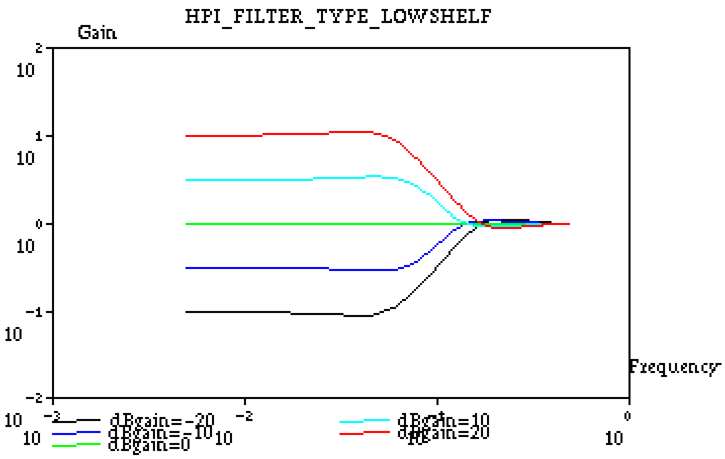
Shelf slope S, a "shelf slope" parameter (for shelving EQ only). When S = 1, the shelf slope is as steep as it can be and remain monotonically increasing or decreasing gain with frequency. The shelf slope, in dB/octave, remains proportional to S for all other values.

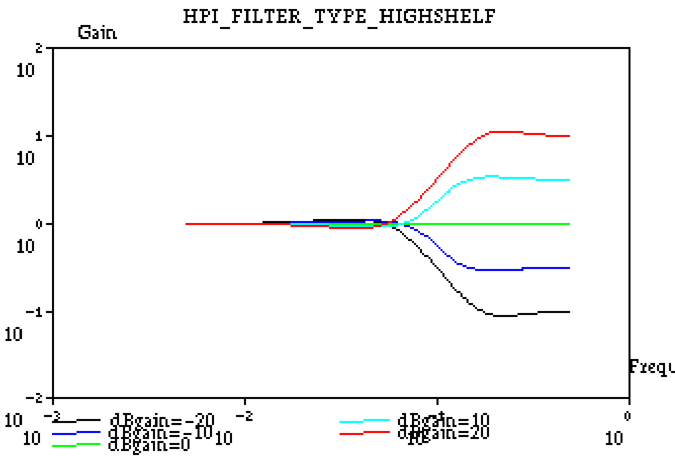
The relationship between shelf slope and Q is 1/Q = sqrt[(A + 1/A)*(1/S - 1) + 2]
where A = 10^(dBgain/40)

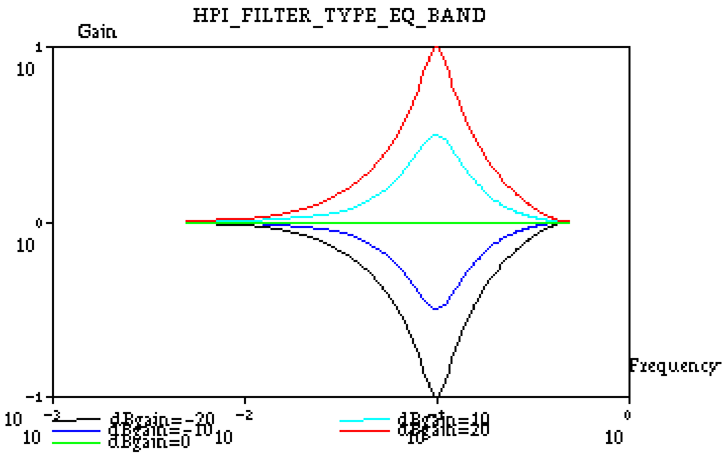


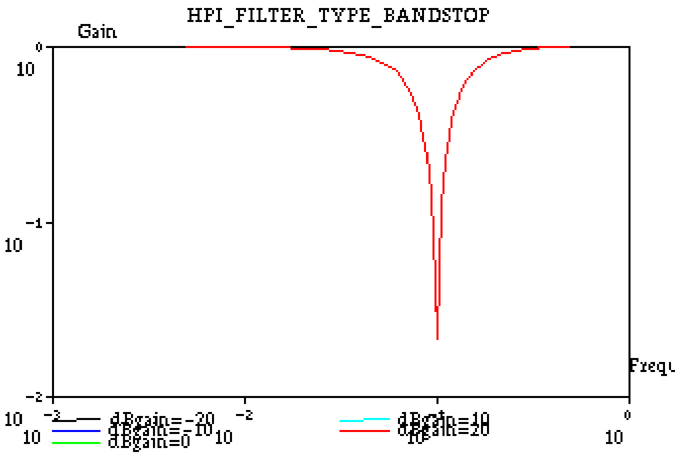
nGain0_01dB
this is expressed in milliBels (100ths of a decibel)
Allowable range is –1000 to +1000 mB. Usable range will likely be less than this.
This parameter is only applicable to the equalizer filter types ASI_FILTER_TYPE_LOWSHELF, ASI_FILTER_TYPE_HIGHSHELF and ASI_FILTER_TYPE_EQ_BAND.Other filters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

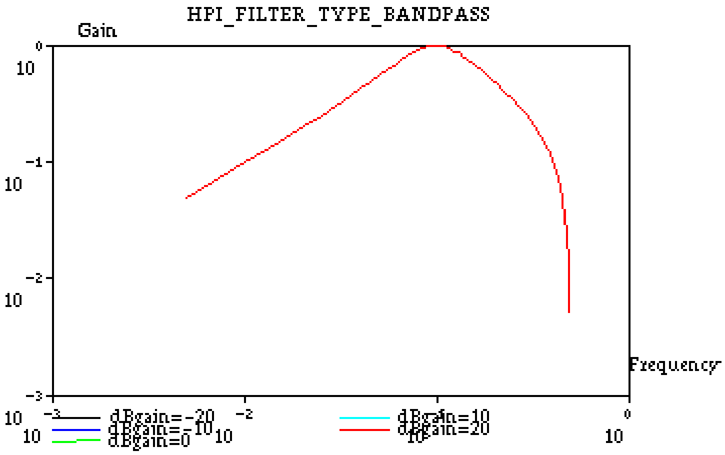
In the following figure, Q=1.0 and sampling frequency is normalized to 1Hz (10^0) and nFrequency is 0.1 x sampling frequency. dBgain=[-20 –10 0 10 20]

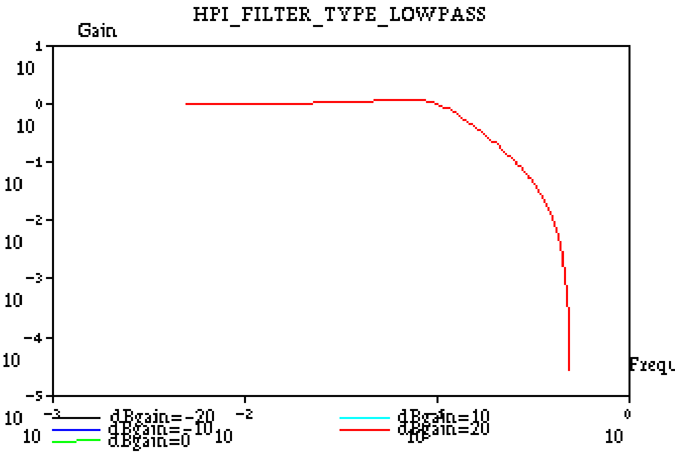












There are several functions available to query the EQ control.

The state of the EQ returns whether the EQ is on or off.

```
ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ        mxcd_eq;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID    = dwEQControlID;
MixerControlDetails.cChannels = 1;
MixerControlDetails.cMultipleItems = 0;
```

```
        MixerControlDetails.paDetails = &mxcd_eq;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ);
        mxcd_eq.dwMsgType = ASI_PARAMETRIC_EQ_GET_STATE;
        mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
```

On return `mxcd_eq.u.dwState` is set to 1 if the equalizer is enabled and 0 if it is off.

The settings for a particular filter may be retrieved by doing a GET_BAND call.

```
        ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ        mxcd_eq;
        MIXERCONTROLDETAILS MixerControlDetails;
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwEQControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = &mxcd_eq;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ);
        mxcd_eq.dwMsgType = ASI_PARAMETRIC_EQ_GET_BAND;
        mxcd_eq.u.dwIndex = 0;        // filter number (0-4 on an ASI5111)
        mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
```

Returned values are, `mxcd_eq.u.Band.dwType, mxcd_eq.u.Band.dwFrequencyHz, mxcd_eq.u.Band.dwQ100 and mxcd_eq.u.Band.nGain0_01dB.` The same units apply as described in the Set code snippet above.

The EQ control also returns the number of filters, or bands, that it supports.

```
        ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ        mxcd_eq;
        MIXERCONTROLDETAILS MixerControlDetails;
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwEQControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = &mxcd_eq;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_PARAMETRIC_EQ);
        mxcd_eq.dwMsgType = ASI_PARAMETRIC_EQ_GET_INFO;
        mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
```

and the number of bands is returned in `mxcd_eq.u.dwNumberOfBands.`

10. Compander

ShortName: “Compander”
LongName: “Compander”

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_COMPANDER
(MIXERCONTROL_CONTROLTYPE_CUSTOM+111)
```

The compander control implements an adjustable compressor-expander function on the input audio.

To set the control use

```
        ASI_MIXERCONTROLDETAILS_COMPANDER    mxcd_compander;
        MIXERCONTROLDETAILS MixerControlDetails;
        MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
        MixerControlDetails.dwControlID    = dwCompanderControlID;
        MixerControlDetails.cChannels = 1;
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = &mxcd_compander;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_ COMPANDER);
        mxcd_compander.dwAttack_ms = 0;
        mxcd_compander.dwDecay_ms = 500;
        mxcd_compander.nCompressionRatio100 = 200;
        mxcd_compander.nThreshold0_01dB = -2000
        mxcd_compander.nMakeupGain_01dB = 1000
        mmError = mixerSetControlDetails(hMixer, &MixerControlDetails, MIXER_SETCONTROLDETAILSF_VALUE);
```

dwAttack_ms, dwDecay_ms The timeconstants of the level measurement process that controls the compander. The attack and decay values represent the time constants of the equivalent single pole low pass filter used to create the RMS . With a time constant of T, if the RMS is stable at full scale and the input is suddenly removed, the meter will decay.

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

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

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

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

The output of an RMS detector with these timeconstants is compared with `nThreshold0_01dB`

nCompressionRatio100: Actual compression ratio x 100. In the following table R= `nCompressionRatio100` /100

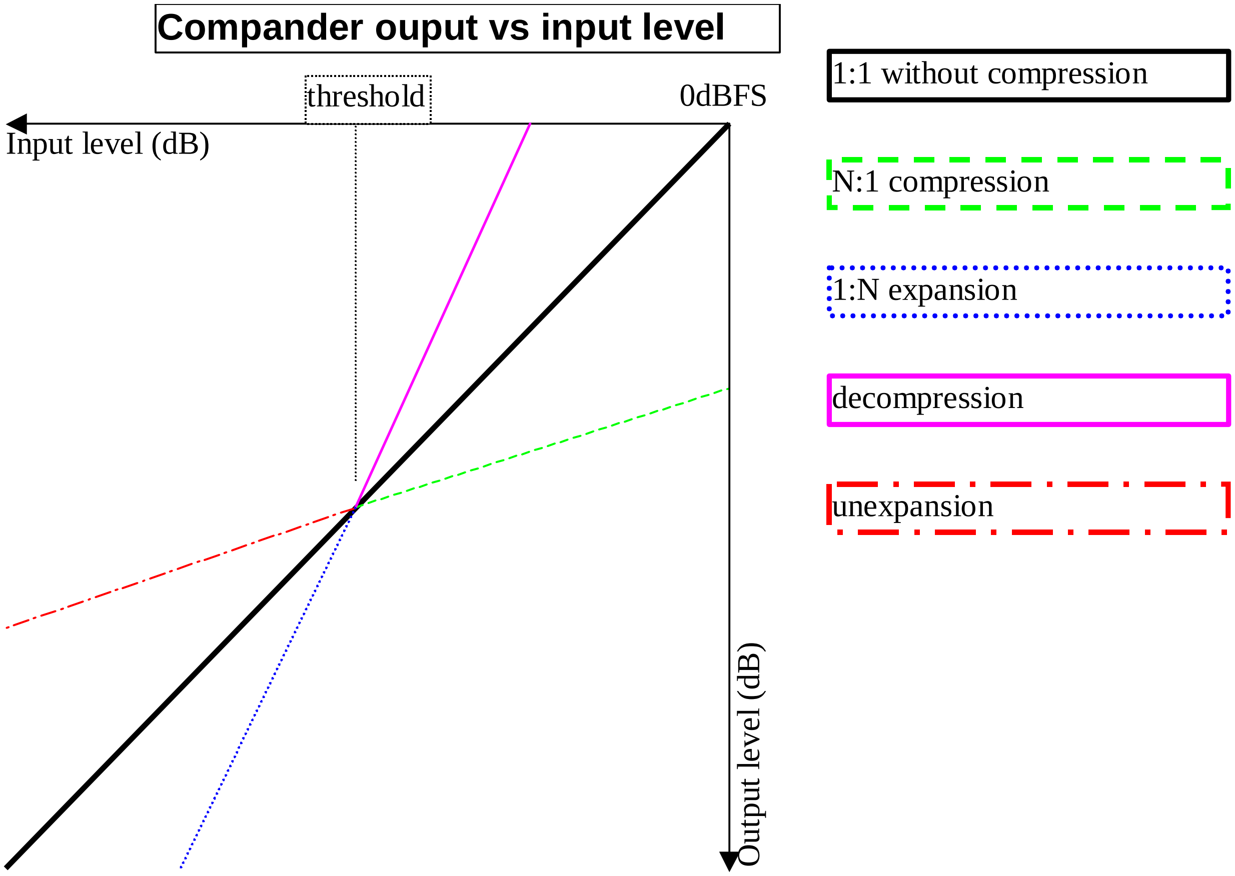
R	ratio	
R>0	R:1	For every R dB above the threshold, the output increases by 1dB
R< 0	1:R	For every 1 dB below the threshold, the output decreases by R dB
R=0		Compander is disabled
1 < R		Compressor: Above the threshold, signal is attenuated
0 < R < 1		*Decompressor: Above the threshold, signal is amplified*
-1 < R < 0		*Unexpander: Below the threshold, signal is amplified*
R < -1		Expander. Below the threshold, signal is attenuated

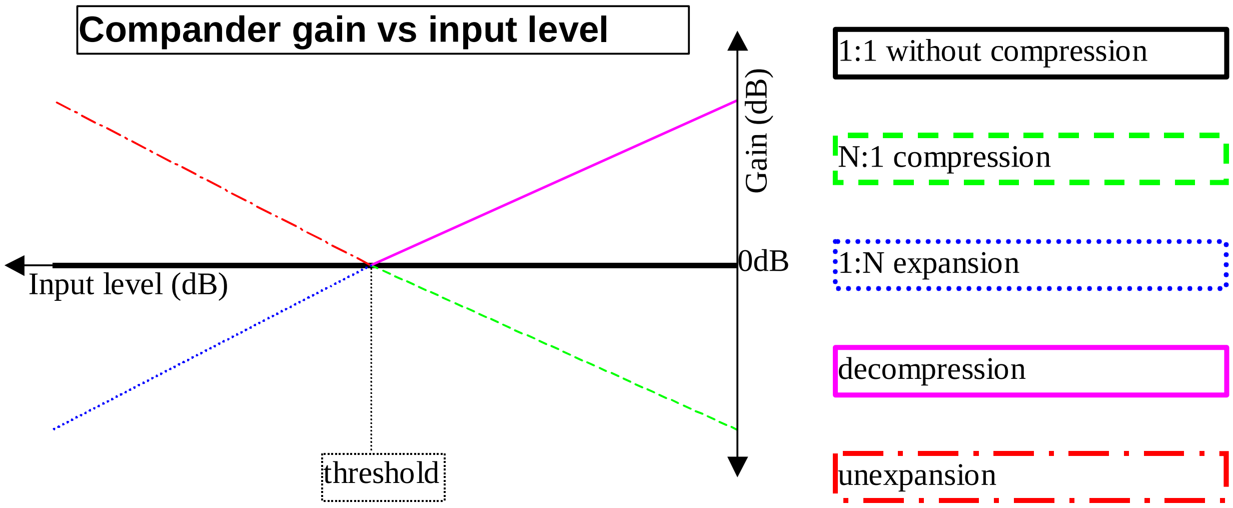
In normal use, |R| > 1, and either traditional compressor or expander functions are performed.

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- nThreshold0_01dB)*(1-1/ nCompressionRatio100)

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.





To get the current compander settings use

```
ASI_MIXERCONTROLDETAILS_COMPANDER     mxcd_compander;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID    = dwCompanderControlID;
MixerControlDetails.cChannels = 1;
```

```
        MixerControlDetails.cMultipleItems = 0;
        MixerControlDetails.paDetails = &mxcd_compander;
        MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_ COMPANDER);
        mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
```

Returns `mxcd_compander.dwAttack_ms, mxcd_compander.dwDecay_ms, mxcd_compander.nCompressionRatio100, mxcd_compander.nThreshold0_01dB and mxcd_compander.nMakeupGain_01dB.`

11. VOX

Shortname:"Vox"
Longname: "Vox"
Type:MIXERCONTROL_CONTROLTYPE_FADER
Vox is a trigger level that controls the start of recording. In the AudioScience implementation, the trigger level is a "peak level" that the record input stream must exceed before the DSP starts recording data. The fader should be set to the desired record trigger level. Recording will not proceed until the trigger level has been exceeded. If the fade level is set to zero, then Vox is disabled. As described in Microsoft documentation, the range of values is 0..65535.

12. Nonvolatile Memory

Shortname : "NvMem"
Longname : "Nonvolatile Memory"

This is a custom control type.

#define ASI_MIXERCONTROL_CONTROLTYPE_NVMEM (MIXERCONTROL_CONTROLTYPE_CUSTOM + 100)

The control details structure is

```
        typedef struct {
        DWORD dwIndex;
        DWORD dwData;
        DWORD dwLocations;
        } ASI_MIXERCONTROLDETAILS_NVMEM;
```

Operation of the control is as follows:
A call to mixerGetControlDetails() returns the data for location wIndex and also sets dwLocations to reflect the number of locations available in the NvMem storage.
A call to mixerSetControlDetails() writes the data from dwData to location dwIndex of the NvMem.

12.1 Locating the NvMem control

This section describes the steps to determine the control ID of an ASI adapter’s NvMem. The description below assumes that a mixer handle, hMixer (of type HMIXER) has been opened using mixerOpen( ).

12.1.1 Search for line out 1

The first step in locating the serial number control is to search the mixer destination lines for line out 1.

```
for(nDest=0; nDest<mxcaps.cDestinations; nDest++)
{
       MIXERLINE     LineInfo;
       LineInfo.dwDestination = nDest;
       mmError=mixerGetLineInfo( hMixer, &LineInfo, MIXER_GETLINEINFOF_DESTINATION);
       if( LineInfo.dwComponentType == MIXERLINE_COMPONENTTYPE_DST_LINE )
           break;
}
```

In the above it is assumed that `mxcaps` is returned from a call to `mixerGetDevCaps(nIndex, &mxcaps, sizeof(MIXERCAPS)).` On exit of the above loop nDest contains the destination line index of the line out 1 line.

12.1.2 Finding theNvMem Control

Given nDest, the index of line out 1, the next step is to locate the NvMem control.

```
MIXERLINE     LineInfo;
MIXERCONTROL *AllControls;
MIXERLINECONTROLS  LineControls;

LineInfo.dwDestination = nDest;
mmError=mixerGetLineInfo( hMixer, &LineInfo, MIXER_GETLINEINFOF_DESTINATION);
AllControls = alloc(sizeof(MIXERCONTROL)*pLineInfo.cControls);

for(i=0; i<pLineInfo->cControls; i++)
    AllControls[i].cbStruct = sizeof(MIXERCONTROL);             // set struct size

// setup line controls structure
LineControls.cbStruct = sizeof(MIXERLINECONTROLS);
LineControls.dwLineID = LineInfo.dwLineID;
LineControls.cControls = pLineInfo.cControls;
LineControls.cbmxctrl = sizeof(MIXERCONTROL);
LineControls.pamxctrl = AllControls;
mmError = mixerGetLineControls(hMixer, &LineControls, MIXER_GETLINECONTROLSF_ALL);

for(i=0;i< LineControls.cControls;i++)
{
    if(AllControls[i].dwControlType==ASI_MIXERCONTROL_CONTROLTYPE_NVMEM)
            dwNvMemControlID = AllControls[i].dwControlID;
}
free(AllControls);
```

The variable dwNvMemControlID contains a unique reference to the NvMem control.

12.1.3 Finding NvMem size

The following assumes that the control ID, dwNvMemControlID is known (see above). The NvMem control is a uniform control, so the number of channels can be set to 1.

```
ASI_MIXERCONTROLDETAILS_NVMEM         mxcd_nv;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID    = dwNvMemControlID;
MixerControlDetails.cChannels = 1;
MixerControlDetails.cMultipleItems = 0;
MixerControlDetails.paDetails = mxcd_nv;
MixerControlDetails.cbDetails = sizeof(AS_MIXERCONTROLDETAILS_NVMEM);
mxcd_nv.dwIndex=0;
mxcd_nv.dwData=0;
mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);

dwNvMemSize = mxcd_nv.dwLocations;
```

The dwNvMemSize has now been assigned the number of bytes available in non-volatile memory.

12.1.4 Reading NvMem

The following assumes that the control ID, dwNvMemControlID is known (see above). The NvMem control is a uniform control, so the number of channels can be set to 1.

dwIndex contains the byte number to read (range is 0 to dwNvMemSize).

```
ASI_MIXERCONTROLDETAILS_NVMEM         mxcd_nv;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID    = dwNvMemControlID;
MixerControlDetails.cChannels = 1;
MixerControlDetails.cMultipleItems = 0;
MixerControlDetails.paDetails = mxcd_nv;
MixerControlDetails.cbDetails = sizeof(AS_MIXERCONTROLDETAILS_NVMEM);
mxcd_nv.dwIndex=dwIndex;
mxcd_nv.dwData=0;
mmError = mixerGetControlDetails(hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
```

mxcd_nv.dwData contains the value of NvMem location dwIndex.

12.1.5 Writing NvMem

The following assumes that the control ID, dwNvMemControlID is known (see above). The NvMem control is a uniform control, so the number of channels can be set to 1.

dwIndex contains the byte number to write (range is 0 to dwNvMemSize).
dwData contains the byte value to write.

```
ASI_MIXERCONTROLDETAILS_NVMEM         mxcd_nv;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct       = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID    = dwNvMemControlID;
MixerControlDetails.cChannels = 1;
MixerControlDetails.cMultipleItems = 0;
MixerControlDetails.paDetails = mxcd_nv;
MixerControlDetails.cbDetails = sizeof(AS_MIXERCONTROLDETAILS_NVMEM);
mxcd_nv.dwIndex=dwIndex;
mxcd_nv.dwData=dwData;
mmError = mixerSetControlDetails(hMixer, &MixerControlDetails, MIXER_SETCONTROLDETAILSF_VALUE);
```

13. AES-18 Controls

The AES-18 messaging subsystem utilizes user bits in an AESEBU data stream to pass messages between an AESEBU transmitter and an AESEBU receiver. Messages can have differing priorities and may be sent on either the left or right channel. The following sections detail the interface to the AES-18 transmitter and receiver controls.

The AES-18 messaging subsystem on an AudioScience AES-18 capable adapter contains the following three controls. A transmitter, a receiver and a block generator. The transmitter and receiver controls work as one would expect. The transmitter "sends" messages and the receiver "receives" messages. The role of the block generator is to provide data blocks to the transmitter that can then have data inserted into them before transmission. When the block generator is in master mode it generates empty blocks. In slave mode the block generator passes blocks from the receiver to the transmitter. It is "slaved" to the block generator of another AES-18 data source. These blocks may be partially full of data. The transmitter will "add" its data to the block before transmitting the block.

New defines have been specified that pertain to the AES-18 subsystem:

```
#define ASI_AES18_MAX_CHANNELS 2
#define ASI_AES18_MAX_MSG_PRIORITES 4
#define ASI_AES18BG_MASTER 1
#define ASI_AES18BG_SLAVE 2
```

13.1.1 AES-18 Block Generator Control

ShortName: "AES18 BG"
LongName: "AES18 Block Generator"

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_AES18BG (MIXERCONTROL_CONTROLTYPE_CUSTOM + 102)
```

The block generator is used to provide blocks to the AES18 transmitter. When in master mode the block generator creates empty blocks without any messages inserted. The transmitter may then insert messages into these blocks. In slave mode the block generator passes blocks from the AES18 receiver to the transmitter. This control responds to both Set/Get details calls.

The Microsoft mixer control details structure is as follows

```
typedef struct {
DWORD dwMasterSlave[ASI_AES18_MAX_CHANNELS];
DWORD dwMasterBlocksPerSec[ASI_AES18_MAX_CHANNELS];
DWORD dwMasterPriorityEnable[ASI_AES18_MAX_CHANNELS][ASI_AES18_MAX_PRIORITIES];
} ASI_MIXERCONTROLDETAILS_AES18BG;
```

Parameters:
dwMasterSlave Specifies master or slave mode. Valid values are ASI_AES18BG_MASTER and ASI_AES18BG_SLAVE.
dwMasterBlocksPerSec This specifies the number of blocks transmitted per second. It is not used in slave mode. Valid values are 2,5,24,25,33 and 100.
dwMasterPriorityEnable This array dictates whether a specific priority is enabled on a specific channel. A value of 1 indicates that the channel and priority is enabled, while 0 indicates that it is disabled.

13.1.2 AES-18 Receiver Control

ShortName: "AES18Rx"
LongName: "AES18 Receiver"

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_AES18RX (MIXERCONTROL_CONTROLTYPE_CUSTOM + 103)
```

The receiver object collects and buffers incoming AES-18 messages. It handles 4 different priority queues per channel and messages that may received on either the left or right channels.

The general approach taken here is to create a single control details structure that is a union of the different types of messages that may be used to interact with the AES18 receiver. Most of the structures are limited to either a SetDetails() or GetDetails() call. The type of operation supported is indicated by SET/GET in the #defines below.

The Microsoft mixer control details structure is as follows

```
#define ASI_AES18RX_SET_ADDRESS (1)
#define ASI_AES18RX_GET_STATE (2)
#define ASI_AES18RX_GET_MSG_SIZE (3)
#define ASI_AES18RX_GET_MSG (4)
#define ASI_AES18RX_GET_ADDRESS (5)

typedef struct {
DWORD dwAddress[ASI_AES18_MAX_CHANNELS];
} ASI_MIXERCONTROLDETAILS_AES18RX_ADDRESS;
```

Parameters:
`dwAddress` Specifies the address of the receiver.

```
typedef struct {
DWORD dwFrameError[ASI_AES18_MAX_CHANNELS];
DWORD dwMessageWaiting[ASI_AES18_MAX_CHANNELS][ASI_AES18_MAX_PRIORITIES];
DWORD dwQueueOverFlow[ASI_AES18_MAX_CHANNELS][ASI_AES18_MAX_PRIORITIES];
DWORD dwMissedMessage[ASI_AES18_MAX_CHANNELS][ASI_AES18_MAX_PRIORITIES];
} ASI_MIXERCONTROLDETAILS_AES18RX_STATE;
```

Parameters:
`dwFrameError` Set to one if a frame check sequence indicates an error.
`dwMessageWaiting` Set to one if a message is waiting.
`dwQueueOverflow` Set to one if a queue has overflowed. This occurs if an incoming message exceeds the available buffering on the DSP.
`dwMissedMessage` Set to one if an incoming message was missed due to a message already being queued in the queue required by the incoming message.

```
typedef struct {
DWORD dwBytesPerQueue[ASI_AES18_MAX_PRIORITIES];
} ASI_MIXERCONTROLDETAILS_AES18RX_MSG_SIZE;
```

Parameters:
`dwBytesPerQueue` Returns the maximum queue size (in bytes) that the receiver supports. This can then be used by the application to allocate buffers for receiving AES-18 messages.

```
typedef struct {
DWORD dwChannel;
DWORD dwPriority;
DWORD dwHostMessage[ASI_AES18_MESSAGE_BUFFER_SIZE];
DWORD dwReturnedMessageSize;
} ASI_MIXERCONTROLDETAILS_AES18RX_MSG;
```

Parameters:
`dwChannel` Specifies the channel to retrieve the message from.
`dwPriority` Specifies the priority queue to retrieve the message from.
`dwHostMessage` The destination buffer for the retrieved message.
`dwReturnedMessageSize` The size (in bytes) of the retrieved message.

```
        typedef struct {
        DWORD dwAES18MsgType;
        union {
        ASI_MIXERCONTROLDETAILS_AES18RX_MSG Msg;
        ASI_MIXERCONTROLDETAILS_AES18RX_MSG_SIZE Size;
        ASI_MIXERCONTROLDETAILS_AES18RX_STATE State;
        ASI_MIXERCONTROLDETAILS_AES18RX_ADDRESS Adr;
        } u;
        } ASI_MIXERCONTROLDETAILS_AES18RX;
```

Parameters:
dwAES18RxMsgType Should be set to one of ASI_AES18RX_SET_ADDRESS, ASI_AES18RX_GET_STATE, ASI_AES18RX_GET_MSG_SIZE, ASI_AES18RX_GET_MSG.
The union statement should be self explanatory.

13.1.2.1 Example C++ Builder code for using ASI_AES18RX_GET_MSG_SIZE

```
// get the size of the various priority queues
ASI_MIXERCONTROLDETAILS_AES18RX aes18rx;

aes18rx.dwAES18MsgType = ASI_AES18RX_GET_MSG_SIZE;
MIXERCONTROLDETAILS MixerControlDetails;
MixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID = mxc.dwControlID;
MixerControlDetails.cChannels = 0;
MixerControlDetails.cMultipleItems = mxc.cMultipleItems;

MixerControlDetails.paDetails = &aes18rx;
MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_AES18RX);
MMRESULT mmError = mixerGetControlDetails(MainForm->hMixer, &MixerControlDetails, MIXER_GETCONTROLDETAILSF_VALUE);
if(mmError)
MessageBox(NULL,"Error getting mixer control details", "mixerGetControlDetails()", MB_ICONSTOP | MB_OK);

// results are in the structure aes18rx.u.Size.
```

13.1.2.2 Example C++ Builder code for using ASI_AES18RX_GET_MSG

```
// read the message
ASI_MIXERCONTROLDETAILS_AES18RX *aes18rxmsg = new ASI_MIXERCONTROLDETAILS_AES18RX;

// get the message
aes18rxmsg->dwAES18MsgType = ASI_AES18RX_GET_MSG;
MixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);
MixerControlDetails.dwControlID = mxc.dwControlID;
MixerControlDetails.cChannels = 0;
MixerControlDetails.cMultipleItems = mxc.cMultipleItems;

MixerControlDetails.paDetails = aes18rxmsg;
MixerControlDetails.cbDetails = sizeof(ASI_MIXERCONTROLDETAILS_AES18RX);

aes18rxmsg->u.Msg.dwChannel=i;
aes18rxmsg->u.Msg.dwPriority=j;

MMRESULT mmError = mixerGetControlDetails(MainForm->hMixer, &MixerControlDetails,
MIXER_GETCONTROLDETAILSF_VALUE);
if(!mmError)
memcpy(szMessage, aes18rxmsg->u.Msg.dwHostMessage, aes18rxmsg->u.Msg.dwReturnedMessageSize);

ActivityLog->Lines->Add( (AnsiString)szMessage);
delete aes18rxmsg;
```

13.1.3 AES-18 Transmitter Control

ShortName: "AES18Tx"
LongName: "AES18 Transmitter"

This is a custom control type:

```
#define ASI_MIXERCONTROL_CONTROLTYPE_AES18TX (MIXERCONTROL_CONTROLTYPE_CUSTOM + 104)
```

The transmitter object supports the sending of AES-18 messages with one of 4 different priorities on either the left or right channels.

The general approach taken here is to create a single control details structure that is a union of the different types of messages that may be used to interact with the AES18 transmitter. Most of the structures are limited to either a SetDetails() or GetDetails() call. The type of operation supported is indicated by SET/GET in the #defines below.

The Microsoft mixer control details structure is as follows

```
#define ASI_AES18RX_SET_MSG (1)
#define ASI_AES18RX_GET_STATE (2)
#define ASI_AES18RX_GET_MSG_SIZE (3)

typedef struct {
DWORD dwChannel;
DWORD dwPriority;
DWORD dwRepetitionIndex;
DWORD dwDestinationAddress;
DWORD dwHostMessage[ASI_AES18_MESSAGE_BUFFER_SIZE];
DWORD dwHostMessageSize;
} ASI_MIXERCONTROLDETAILS_AES18TX_MSG;
```

Parameters:
dwChannel Specifies the channel to send the message on.
dwPriority Specifies the priority of the message.
dwRepetitionIndex The number of times this message should be repeated.
dwDestinationAddress The AES-18 address of the device to receive this message.
pbHostMessage A pointer to the message buffer.
dwHostMessageSize The size (in bytes) of the host message buffer.

```
typedef struct {
DWORD dwFrameError[ASI_AES18_MAX_CHANNELS];
DWORD dwPriorityChannelBusy[ASI_AES18_MAX_CHANNELS][ASI_AES18_MAX_PRIORITIES];
} ASI_MIXERCONTROLDETAILS_AES18TX_STATE;
```

Parameters:
dwPriorityChannelBusy Set to one if a particular priority channel is busy. The state of the transmitter should be checked before sending an AES-18 message to verify that the desired channel and priority combination is available.

```
typedef struct {
DWORD dwMaxBytesPerMessage[ASI_AES18_MAX_PRIORITIES];
} ASI_MIXERCONTROLDETAILS_AES18TX_MSG_SIZE;
```

Parameters:
dwMaxBytesPerMessage Specifies the maximum message size (in bytes) that the transmitter supports.

```
typedef struct {
DWORD dwAES18MsgType;
union {
ASI_MIXERCONTROLDETAILS_AES18TX_MSG Msg;
ASI_MIXERCONTROLDETAILS_AES18TX_MSG_SIZE Size;
ASI_MIXERCONTROLDETAILS_AES18TX_STATE State;
} u;
} ASI_MIXERCONTROLDETAILS_AES18TX;
```

Parameters:
dwAES18MsgType Should be set to one of ASI_AES18TX_GET_STATE, ASI_AES18TX_GET_MSG_SIZE, ASI_AES18TX_SET_MSG.

The union statement should be self explanatory.

14. Watchdog

The watch dog control is used to set the timeout for the watchdog timer (Currently not implemented).
Type : MIXERCONTROL_CONTROLTYPE_MILLITIME
Shortname : "Watchdog"
Longname : "Watchdog timer"

15. waveOutSetVolume operation

This functionality is supported in drivers 2.83 and later.

The ASI wav driver supports 2 modes of waveOutSetVolume( ) operation. The 2 modes are selected via the asiwav.ini file. The settings are

Wave Out Volume Mode = Summing [default]
Wave Out Volume Mode = One2one

When the “Summing” mode is used, calls to waveOutSetVolume( ) adjust the volume between the waveOut device and LineOut 1. That is, all calls manipulate the audio output level for Line Out 1. The “One2one” mode adjusts volumes between waveOut N and line out N.

16. Tuner controls

Certain AudioScience adapters support AM/FM/TV tuners. This section describes how to set the tuner band and frequency using standard Windows controls

The tuner controls live on a source line of type ASI_MIXERLINE_COMPONENTTYPE_SRC_TUNER, as defined in asiwavx.h. This means that the dwComponent field of the MIXERLINE structure (defined in mmsystem.h) is set to ASI_MIXERLINE_COMPONENTTYPE_SRC_TUNER.

16.1 Setting the tuner band

The tuner band is set using the standard Windows control of type MIXERCONTROL_CONTROLTYPE_SINGLESELECT. The following code snippet can be used to get the names of the bands supported.

Given

```
MIXERCONTROL *pmxc
```

and

```
MIXERCONTROLDETAILS mxcd;

MIXERCONTROLDETAILS_LISTTEXT *mxcd_l = new MIXERCONTROLDETAILS_LISTTEXT [pmxc-
>cMultipleItems];
mxcd.cbStruct       = sizeof(MIXERCONTROLDETAILS);
mxcd.dwControlID    = pmxc->dwControlID;
mxcd.cChannels      = nChannels;
mxcd.cMultipleItems = pmxc->cMultipleItems;
mxcd.cbDetails      = sizeof(MIXERCONTROLDETAILS_LISTTEXT);
mxcd.paDetails      = mxcd_l;
mxcd.cMultipleItems=pmxc->cMultipleItems;
MMRESULT mmerror = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_LISTTEXT);

 for(int i=0; i<pmxc->cMultipleItems; i++)
             printf("Tuner band names %s\n",mxcd_l[i].szName );
```

The current band is read using the following code

Given
`MIXERCONTROL *pmxc`
and
`MIXERCONTROLDETAILS mxcd;`

```
MIXERCONTROLDETAILS_BOOLEAN      *mxcd_b = new MIXERCONTROLDETAILS_BOOLEAN
[mxc.cMultipleItems];
mxcd.cbStruct       = sizeof(MIXERCONTROLDETAILS);
mxcd.dwControlID    = pmxc->dwControlID;
mxcd.cChannels      = nChannels;
mxcd.cMultipleItems = pmxc->cMultipleItems;
mxcd.cbDetails      = sizeof(MIXERCONTROLDETAILS_BOOLEAN_LISTTEXT);
mxcd.paDetails      = mxcd_b;
mxcd.cMultipleItems=pmxc->cMultipleItems;
MMRESULT mmerror = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
```

The index of `mxcd_b[i].fvalue` having a non-zero value is the current band selection.

The tuner band is set using a sequence similar to the read code above. A single index of the `mxcd_b[i].fvalue` array should be set to a value of 1. The remaining indices must be zero. A call to mixerSetControlDetails() is then made, i.e.,
` MMRESULT mmerror = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);`
with `mxcd` set up the same as the previous example.

Note that when changing the tuner band the frequency will also be updated by the adapter so that it is a valid frequency for the band selected. This means that when updating both band and frequency, the band should be changed prior to the frequency being set.

16.2 Setting the tuner frequency

The tuner frequency is adjusted via a standard Windows mixer control of type MIXERCONTROL_CONTROLTYPE_UNSIGNED. To set the frequency use code similar to:

Given
`MIXERCONTROL *pmxc`
and
`MIXERCONTROLDETAILS mxcd;`

```
MIXERCONTROLDETAILS_UNSIGNED     mxcd_u;
mxcd.cbStruct       = sizeof(MIXERCONTROLDETAILS);
mxcd.dwControlID    = pmxc->dwControlID;
mxcd.cChannels      = nChannels;
mxcd.cMultipleItems = pmxc->cMultipleItems;
mxcd.cbDetails      = sizeof(MIXERCONTROLDETAILS_UNSIGNED);
mxcd.paDetails      = mxcd_u;
mxcd.cMultipleItems=pmxc->cMultipleItems;
mxcd_u.dwValue = dwFrequencyInkHz;
MMRESULT mmerror = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

Reading the frequency is identical, except that the current frequency is returned in `mxcd_u.dwValue.`

16.3 Setting the tuner gain

The tuner gain is set via a standard Windows control of type MIXERCONTROL_CONTROLTYPE_DECIBELS. The following code can be used to set the tuner gain.

Given
```
MIXERCONTROL *pmxc
```
and
```
MIXERCONTROLDETAILS mxcd;

MIXERCONTROLDETAILS_SIGNED  mxcd_d[2];
mxcd.cbStruct       = sizeof(MIXERCONTROLDETAILS);
mxcd.dwControlID    = pmxc->dwControlID;
mxcd.cChannels      = nChannels;
mxcd.cMultipleItems = pmxc->cMultipleItems;
mxcd.cbDetails      = sizeof(MIXERCONTROLDETAILS_SIGNED);
mxcd.paDetails      = mxcd_d;
mxcd.cMultipleItems=pmxc->cMultipleItems;
mxcd_d[0].lValue =  lGainIn10thsOfdB;
mxcd_d[1].lValue =  mxcd_d[0].lValue;
MMRESULT mmerror = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

The `MIXERCONTROL` structure fields contain information about the maximum, minimum and the number of steps the control has.
pmxc->Bounds.lMaximum is the maximum value.
pmxc->Bounds.lMinimum is the minimum value.
pmxc->Metrics.cSteps is the number of steps available between the min and max values.

An ASI8702, for example, only has a single step between the minimum and maximum values.

16.4 Reading the RF level

The RF level is read by reading a control of type MIXERCONTROL_CONTROLTYPE_SIGNED on the tuner line.

Given
```
MIXERCONTROL *pmxc
```
and
```
MIXERCONTROLDETAILS mxcd;

MIXERCONTROLDETAILS_SIGNED       mxcd_s;
mxcd.cbStruct       = sizeof(MIXERCONTROLDETAILS);
mxcd.dwControlID    = pmxc->dwControlID;
mxcd.cChannels      = nChannels;
mxcd.cMultipleItems = pmxc->cMultipleItems;
mxcd.cbDetails      = sizeof(MIXERCONTROLDETAILS_SIGNED);
mxcd.paDetails      = mxcd_s;
mxcd.cMultipleItems=pmxc->cMultipleItems;
MMRESULT mmerror = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
```

The level is returned in mxcd_s.lValue and has units dBuV.

17. Tone Detector

17.1 Introduction

The ASI2416 implements a tone detector control on all line ins. The tone detector can detect the presence of both 25 Hz and 35 Hz tones on both the left and right channels independently.

17.2 Enable

17.2.1 Set

The code to set the current enable state is as follows:

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_SET_ENABLE;
mxcd_t.u.dwEnable = nEnable;
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

17.2.2 Get

The code to get the current enable state is as follows:

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_GET_ENABLE;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
```

17.3 Event Enable

The event enable attribute controls whether state changes are reported via the asynchronous event mechanism. This can be left disabled if the state is being polled.

The asynchronous event reporting mechanism works with the Windows mixer as follows…..

17.3.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_SET_ENABLE;
mxcd_t.u.dwEnable = nEnable;
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

17.3.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_GET_ENABLE_EVENTS;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
```

17.4 Threshold

The threshold in dbFS specifies the amplitude of that tone that is required before the detector will trigger.

17.4.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_SET_THRESHOLD;
mxcd_t.u.nThreshold = (int)(fThreshold*100.0f);      // units are 100th of a dB
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

17.4.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_GET_THRESHOLD;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*fThreshold = (float)mxcd_t.u.nThreshold/100.0f;     // units are 100ths of a dB
```

17.5 State

Read the state of the tone detector. Bits return in dwState are organized as follows:

17.5.1 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_GET_STATE;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nState = mxcd_t.u.dwState;
```

17.6 Frequency

17.6.1 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_TONEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_TONEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_TONEDETETECTOR_GET_FREQUENCY;
mxcd_t.u.Freq.dwIndex = nIndex;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nState = mxcd_t.u.Freq.dwFrequency;
```

18. Silence Detector

18.1 Introduction

The ASI2416 implements a silence detector on all line outs. Left and right channels are reported independently and the duration and threshold for silence detection are settable.

18.2 Enable

18.2.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_SET_ENABLE;
mxcd_t.u.dwEnable = nEnable;
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

18.2.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_GET_ENABLE;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nEnable = mxcd_t.u.dwEnable;
```

18.3 Event Enable

18.3.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_SET_ENABLE_EVENTS;
mxcd_t.u.dwEnableEvents = nEnable;
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

18.3.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_GET_ENABLE_EVENTS;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nEnable = mxcd_t.u.dwEnableEvents;
```

18.4 Delay

18.4.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;
```

```
c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_SET_DELAY;
mxcd_t.u.dwDelay = nDelay;
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

18.4.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_GET_DELAY;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nDelay = mxcd_t.u.dwDelay;
```

18.5 Threshold

18.5.1 Set

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_SET_THRESHOLD;
mxcd_t.u.nThreshold = (int)(fThreshold*100.0f);      // units are 100th of a dB
mmError = mixerSetControlDetails(hMixer, &mxcd, MIXER_SETCONTROLDETAILSF_VALUE);
```

18.5.2 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_GET_THRESHOLD;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*fThreshold = (float)mxcd_t.u.nThreshold/100.0f;     // units are 100ths of a dB
```

18.6 State

18.6.1 Get

```
MMRESULT mmError;
MIXERCONTROLDETAILS mxcd;
ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR  mxcd_t;

c->initDetails(&mxcd,sizeof(ASI_MIXERCONTROLDETAILS_SILENCEDETECTOR), &mxcd_t);
mxcd_t.dwMsgType = ASI_SILENCEDETETECTOR_GET_STATE;
mmError = mixerGetControlDetails(hMixer, &mxcd, MIXER_GETCONTROLDETAILSF_VALUE);
*nState = mxcd_t.u.dwState;
```

18.6.2

[end]