Version 2.51

Developer's Information Pack

Creative Technology Ltd.

CREATIVE

Creative Technology Ltd.

Contents

PART I Introduction

Limitation And Disclaimer Of Warranties

This Package

This developer's information pack is made for third party DOS and Microsoft Windows 3.1 developers who intend to develop MIDI oriented software programs for Creative's Sound Blaster AWE32. It includes easy-to-use functions and a complete interface that supports MIDI playback and Sound Blaster AWE32 DRAM downloading.

This document describes the pack's set of low level DOS and Microsoft Windows 3.1 API to program the Sound Blaster AWE32. It contains object libraries for common MIDI routines and SB AWE32 SoundFont bank downloading for DOS (real and protected modes) and Microsoft Windows 3.1. These library functions are designed with the objective of allowing you to create your own code in the following forms :

- Terminate-stay-resident MIDI drivers
- Loadable MIDI drivers
- Embedded MIDI applications

We at Creative Labs has spent much effort in creating the drivers and libraries to save your development time. We have taken great care to meet the requirements of the various types of developers and to reduce the possibilities of clashes with other TSRs or Windows system drivers. It is our hope that all the facilities provided in this information pack meet with your development needs.

Using This Manual

This document is organized into four main parts. The first details the API for DOS, the second the API for Windows 3.1 and the third on SB AWE32's MIDI non-registered-parameter-number implementation. The organization, in detail, is as follows:

PART II DOS Real/Protected Mode API, describes the SB AWE32 DOS real and protected mode API.

> **Overview**, gives a quick look at the SB AWE32 DOS real and protected mode API.
>
> **Hardware Detection And Initialization**, provides interfaces to prepare the EMU8000 subsystem for use.
>
> **MIDI Services**, provides interfaces to process MIDI events.
>
> **SoundFont Bank And Downloadable DRAM Services**, gives interfaces to load SoundFont banks and wave data.
>
> **Real and Protected Mode API Programming Guide**, gives a general description on using the DOS Real and Protected mode API.

PART III Windows Driver API, describes the SB AWE32 Windows driver API.

> **Overview**, gives a general look at the SB AWE32 Windows driver API.
>
> **AWE Manager DLL**, provides a detailed description of the AWE Manager and its functions.
>
> **Windows Programming Guide**, provides C examples to let you access and manipulate the Windows drivers.

PART IV MIDI NRPN Implementation, describes the MIDI NRPN implementation of the SB AWE32.

> **SB AWE32 MIDI NRPN List**, details the Non-Registered Parameter Number implementation of the SB AWE32 Window MIDI driver.

PART V Audio Spatialization API, describes the a low-level access to 3D audio algorithms running on the SB AWE32.

> **Overview**, gives a quick look at the Audio Spatialization API implementation on the SB AWE32.
>
> **Types and Structures**, describes the data types and structures used by the Audio Spatialization Library.
>
> **System Functions**, describes the system wide, environment functions.
>
> **Emitter Functions**, describes the operations on the emitters.
>
> **Receiver Functions**, describes the operation on the receivers.

Programming Example, gives an example of using the Audio Spatialization Library.

Document Conventions

To help you locate and identify information easily, this manual uses visual cues and standard text formats. The following typographical conventions are used throughout this document:

Example	Description
bold_letter	Bold letters indicate variable names, library functions, or commands. These are case-sensitive. Bold letters are also used for terms intended as keywords or for emphasis in certain phrases.
BOLD_CAPS	All bold capital letters indicate constants.
NORMAL_CAPS	All capital letters indicate file names or directory names.
italics	Italic letters represent actual values or values of variables that you are expected to provide.
`program`	This font is used for example codes.

`program`
.
.
`fragment`
	Vertical ellipsis in an example program indicate that part of the program has been intentionally omitted.

[ ]	Square brackets in a command line indicate that the enclosed item is optional.
< >	Angle brackets in a command line indicate that you must provide the actual value of the enclosed item.
/	A forward slash in a command line indicates an either/or choice.

In this document, "you" refers to you the developer or sometimes your application. The word "user" refers to the person who uses your application.

Getting Assistance

If you have any comments, suggestions, questions, or problems concerning this information pack, please feel free to contact us. You can reach us at:

CompuServe: GO BLASTER
Internet E-mail address: 72662.1602@compuserve.com

What You Need To Know

This manual assumes you are an experienced software developer who is familiar with using Sound Blaster cards or any of its derivatives. Thus, the focus in this document is on highly technical aspects of the cards.

This developer information pack supports the following programming tools :

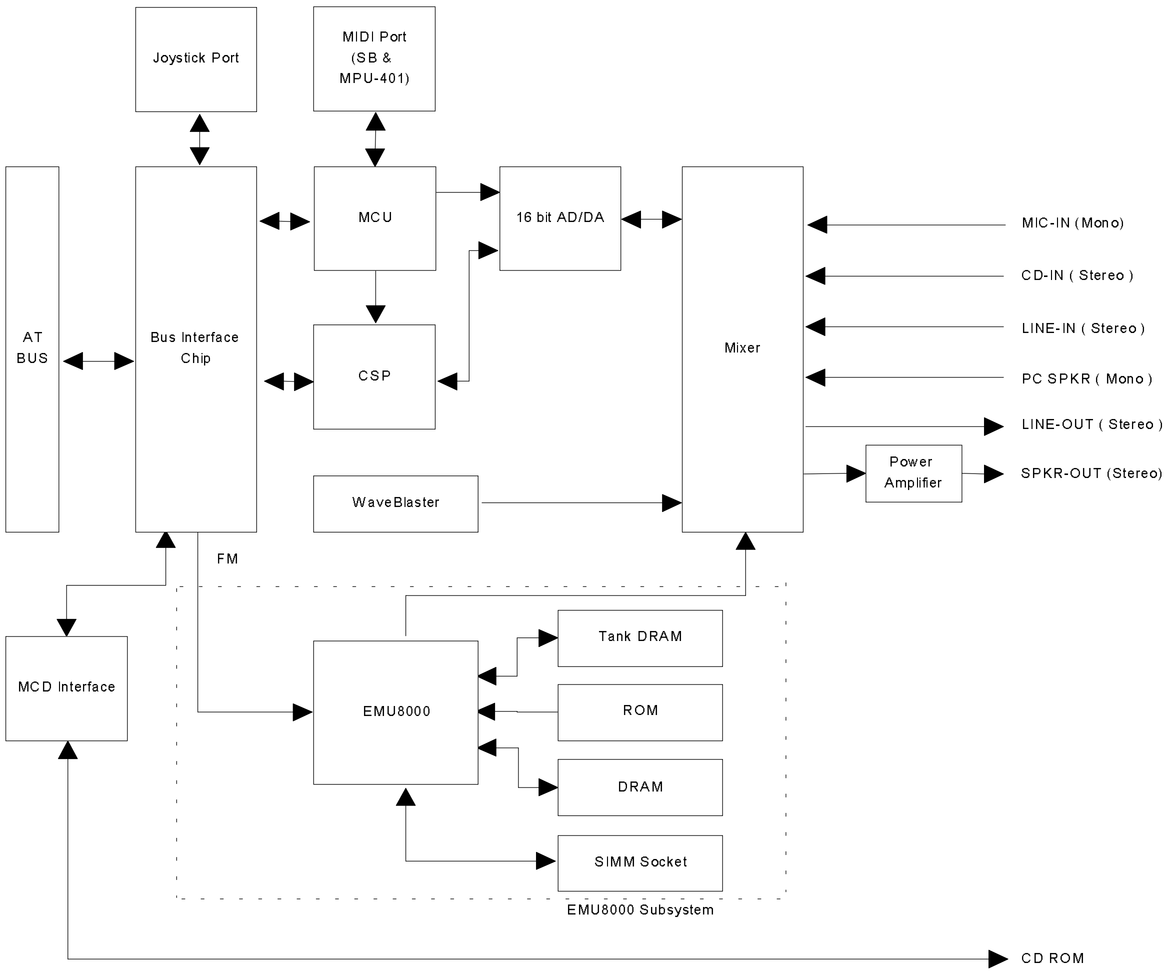
- Microsoft Visual C++ version 1.0 and 1.5
- Microsoft C version 6.0 and 7.0
- Borland C++ version 3.1 and 4.0
- Watcom C/C++^{32} version 9.5 and 10.0
- MetaWare High C/C++ version 3.2
- Symantec C/C++ version 6.1

The real mode DOS object libraries are available for the following memory models:

- Small
- Compact
- Medium
- Large

A flat model protected mode library is also provided.

Before you proceed to Part II, III and IV of this manual, you need to familiarize yourself with the hardware functional blocks of the Sound Blaster AWE32 audio card. A diagram of the functional blocks is provided below.



Functional Block Diagram of Sound Blaster AWE32

The EMU8000 subsystem consists of the following parts:

- ROM

  The ROM contains 1MB of General MIDI sound data.

- DRAM

  This is the supplied 512 KB of DRAM on Sound Blaster AWE32 and Sound Blaster AWE32 Value Edition for custom sound samples and GS support.

- SIMM Socket

  2 optional SIMM sockets for DRAM expansion. You can expand the on-board DRAM a maximum of 28 MB by inserting off-the-shelf SIMM modules.

The following lists the I/O ports used by the EMU8000 subsystem :

- 0x6X0 - 0x6X3
- 0xAX0 - 0xAX3
- 0xEX0 - 0xEX3

‘X’ denotes the center digit in the base I/O address.

The library functions provided accesses all the EMU8000 subsystem’s I/O addresses.

For a detailed look at the other functional blocks in the diagram, refer to “Developer Kit for Sound Blaster Series Hardware Programming Reference” documentation from us.

PART II DOS Real/Protected Mode API

Overview

This chapter gives an overview of the DOS object modules and the sample applications. Note that this chapter does not attempt to cover programming the Sound Blaster. Please refer to “Sound Blaster Developer Kit for Sound Blaster Series, 2nd Edition” if you need more information on programming the Sound Blaster.

The object modules are provided in five memory models; SMALL, COMPACT, MEDIUM, LARGE and FLAT. The object module files are :

MIDIENG.OBJ	MIDI engine. Contains all MIDI channel message services.
SYSEX.OBJ	SysEx parser
SFHELP1.OBJ	SoundFont helper module 1
SFHELP2.OBJ	SoundFont helper module 2 shared by SBKLOAD.OBJ and WAVLOAD.OBJ
EMBED.OBJ	General MIDI preset module
MIDIVAR.OBJ	MIDI variables, used by all modules
HARDWARE.OBJ	EMU8000 hardware initialization module
SBKLOAD.OBJ	SoundFont bank loader module
WAVLOAD.OBJ	Wave PCM loader module
NRPN.OBJ	Non-Registered Parameter Number interpreter.
NRPNVAR.OBJ	Data buffers use by Non-Registered Parameter Number interpreter.

HARDWARE.OBJ, SBKLOAD.OBJ, WAVLOAD.OBJ and SFHELP2.OBJ are discardable after they have been used. For example, in a TSR program, after you have initialized the SB AWE32 hardware and loaded your SoundFont bank file, you can mark portions of your code that must stay resident (the MIDI engine, the embedded General MIDI preset data and any SoundFont preset data), and discard the hardware and the SoundFont loader module.

A sample application is included in this information package.

Hardware Detection And Initialization

This group of API consists of the following :

- awe32Detect
- awe32InitHardware
- awe32Terminate

awe32Detect

```
WORD
PASCAL
awe32Detect(WORD wBaseIOAddx)
```

Actions Detect the presence of the EMU8000 subsystem.

Parameters *wBaseIOAddx*

Specify the base I/O address of the EMU8000 subsystem. The address can be found in the BLASTER environment variable with a 'E' prefix. For example, E620.

The 'E' BLASTER environment parameter is introduced to cater for future derivatives of the SB AWE32 sound card that may have the EMU8000 at different I/O addresses.

This API in this release is backwards compatible with its previous version, i.e., it can detect for the presence of the EMU8000 subsystem using the 'A' parameter.

Return The return value is 0 if the EMU8000 is detected, and non-zero if otherwise.

awe32InitHardware

```
WORD
PASCAL
awe32InitHardware(VOID)
```

Actions Prepare the EMU8000 subsystem for MIDI playback.

Parameters None.

Return Return 0 if the EMU8000 subsystem had been properly initialized, and non-zero if otherwise.

awe32Terminate

```
WORD
PASCAL
awe32Terminate(VOID)
```

Actions Restore the EMU8000 chip to a known state.

Parameters None.

Return Return 0 if the EMU8000 subsystem had been properly terminated, and non-zero if otherwise.

Remarks The EMU8000 will be initialized to process FM audio. The FM initialization has a tight timing loop. It is recommended that all interrupts are disabled before calling `awe32Terminate`.

MIDI Services

This group of API consists of the following :

- awe32InitMIDI
- awe32InitNRPN
- awe32NoteOn
- awe32NoteOff
- awe32ProgramChange
- awe32PitchBend
- awe32Controller
- awe32ChannelPressure
- awe32PolyKeyPressure
- awe32Sysex
- __awe32NoteOff
- __awe32IsPlaying

awe32InitMIDI

```
WORD
PASCAL
awe32InitMIDI(VOID)
```

Actions Initialize MIDI engine. It resets all controller values and prepares for subsequent MIDI engine calls.

Parameters None.

Return The return value is 0 if the initialization is successful, non-zero if otherwise.

awe32InitNRPN

```
WORD
PASCAL
awe32InitNRPN(VOID)
```

Actions Initialize the data buffer used for NRPN and to link in the NRPN.OBJ and NRPNVAR.OBJ modules.

Parameters None.

Return Return 0 upon successful initialization, and non-zero if otherwise.

awe32NoteOn

```
WORD
PASCAL
awe32NoteOn(
      WORD  wMIDIChannel,
      WORD  wNoteNumber,
      WORD  wVelocity
)
```

Actions Turn on a MIDI note.

Parameters *wMIDIChannel*

Specify from which MIDI channel the note on originated. Valid range for this variable is from 0 to 15 decimal.

wNoteNumber

Specify the MIDI note number. Valid range for this variable is from 0 to 127.

wVelocity

Specify the MIDI note's velocity. Valid range for this variable is from 0 to 127.

Return The return value is 0 if the MIDI note on is successful, non-zero if otherwise.

Remarks To prevent note stealing, especially for lengthy special effects, add 16 to the channel number. Care must be taken to issue a corresponding note-off with the same channel number.

awe32NoteOff

```
WORD
PASCAL
awe32NoteOff(
      WORD  wMIDIChannel,
      WORD  wNoteNumber,
      WORD  wVelocity
)
```

Actions Turn off a MIDI note.

Parameters *wMIDIChannel*

Specify from which MIDI channel the note off originated. Valid range for this variable is from 0 to 15 decimal.

wNoteNumber

Specify the MIDI note number. Valid range for this variable is from 0 to 127.

wVelocity

Specify the MIDI note's velocity. Valid range for this variable is from 0 to 127.

Return The return value is 0 if the MIDI note off is successful, non-zero if otherwise.

awe32ProgramChange

```
WORD
PASCAL
awe32ProgramChange(
      WORD  wMIDIChannel,
      WORD  wProgram
)
```

Actions Select a program.

Parameters *wMIDIChannel*

Specify from which MIDI channel the program change originated. Valid range for this variable is from 0 to 15 decimal.

wProgram

Specify the desired program value. Valid range for this variable is from 0 to 127.

Return The return value is 0 if the MIDI program change is successful, non-zero if otherwise.

awe32PitchBend

```
WORD
PASCAL
awe32PitchBend(
      WORD  wMIDIChannel,
      WORD  wLSB,
      WORD  wMSB
)
```

Actions Pitch bend all notes sounding in a MIDI channel.

Parameters *wMIDIChannel*

Specify from which MIDI channel the pitch bend originated. Valid range for this variable is from 0 to 15 decimal.

wLSB

Specify the MIDI pitch bend LSB data value. Valid range for this variable is from 0 to 127.

wMSB

Specify the MIDI pitch bend MSB data value. Valid range for this variable is from 0 to 127.

Return The return value is 0 if the MIDI pitch bend is successful, non-zero otherwise.

awe32Controller

```
WORD
PASCAL
awe32Controller(
      WORD  wMIDIChannel,
      WORD  wControlNumber,
      WORD  wControlData
)
```

Actions Controller change on a MIDI channel.

Parameters *wMIDIChannel*

Specify from which MIDI channel the note on originated. Valid range for this variable is from 0 to 15 decimal.

wControlNumber

Specify the MIDI controller number. Valid range for this variable is from 0 to 127.

wControlData

Specify the MIDI controller's data value. Valid range for this variable is from 0 to 127.

Return The return value is 0 if the MIDI control change is successful, non-zero if otherwise.

Remarks Supported controllers are:

• CC0	Bank Select
• CC1	Modulation Wheel
• CC6	Data Entry MSB
• CC7	Master Volume
• CC10	Pan Position
• CC11	Expression
• CC38	Data Entry LSB
• CC64	Sustain Pedal
• CC91	Effects Depth (Reverb)
• CC93	Chorus Depth
• CC98	Non-Registered Parameter Number LSB
• CC99	Non-Registered Parameter Number MSB
• CC100	Registered Parameter Number LSB
• CC101	Registered Parameter Number MSB
• CC120	All Sound Off
• CC121	Reset All Controllers
• CC123	All Notes Off

RPN recognizes controller value 0 (pitch-bend sensitivity). Reset All Controllers resets the following:

- Pitch Wheel
- Modulation Wheel
- Expression (CC11)
- Sustain Pedal (CC64)
- Channel Pressure
- Non-Registered Parameter Numbers

awe32ChannelPressure

```
WORD
PASCAL
awe32ChannelPressure(
      WORD  wMIDIChannel,
      WORD  wData
)
```

Actions MIDI Channel Pressure.

Parameters *wMIDIChannel*

Specify from which MIDI channel the channel pressure originated. Valid range for this variable is from 0 to 15 decimal.

wData

Specify the channel pressure data value. Valid range for this variable is from 0 to 127.

Return This function will return 0 if successful, and non-zero if otherwise.

awe32PolyKeyPressure

```
WORD
PASCAL
awe32PolyKeyPressure(
      WORD  wMIDIChannel,
      WORD  wNoteNumber,
      WORD  wData
)
```

Actions None.

Parameters *wMIDIChannel*

Specify from which MIDI channel the polyphonic key pressure originated. Valid range for this variable is from 0 to 15 decimal.

wNoteNumber

Specify the MIDI note number. Valid range for this variable is from 0 to 127.

wData

Specify the polyphonic key pressure data value. Valid range for this variable is from 0 to 127.

Return This function will always return 0.

Remarks This function is a dummy place holder and does not do anything in its main body. It is provided for MIDI compatibility purposes.

awe32Sysex

```
WORD
PASCAL
awe32Sysex(
      WORD        wMIDIChannel,
      LPBYTE      lpSysexBuffer,
      WORD        wBufferSize
)
```

Actions MIDI Sysex command

Parameters *wMIDIChannel*

Specify from which MIDI channel the Sysex originated. Valid range for this variable is from 0 to 15 decimal.

lpSysexBuffer

Specify a far byte pointer to a string of Sysex data.

wBufferSize

Specify the size of the Sysex buffer.

Return The return value is 0 if the MIDI Sysex command is successful, non-zero if otherwise.

Remarks This current API recognizes the Sysex message switching Reverb/Chorus effects variation.

Reverb Sysex macro :

F0 41 10 42 12 40 01 30 XX CS F7

where XX denotes the Reverb variation to be selected, and CS denotes a checksum value that is not verified. The valid values for XX are

0 - Room 1

1 - Room 2

2 - Room 3

3 - Hall 1

4 - Hall 2

5 - Plate

6 - Delay

7 - Panning Delay

Chorus Sysex macro

F0 41 10 42 12 40 01 38 XX CS F7

again, XX denotes the chorus variation to be selected, and CS notes a checksum value that is not verified. The valid values for XX are :

0 - Chorus 1

1 - Chorus 2

2 - Chorus 3

3 - Chorus 4

4 - Feedback chorus

5 - Flanger

6 - Short Delay

7 - Short delay (FB)

__awe32NoteOff

```
WORD
PASCAL
__awe32NoteOff(
      WORD  wMIDIChannel,
      WORD  wBank,
      WORD  wPreset,
      WORD  wNote
)
```

Actions

Turn off a MIDI note. You would use this function to turn off a specific user bank note.

Parameters

wMIDIChannel

Specify from which MIDI channel the note off originated. Valid range for this variable is from 0 to 15 decimal.

wBank

Specify the user bank the note originated from. Valid range for this variable is from 0 to 127.

wPreset

Specify the note's preset number. Valid range for this variable is from 0 to 127.

wNote

Specify the note's note number. Valid range for this variable is from 0 to 127.

Return

The return value is 0 if the MIDI note off is successful, non-zero if otherwise.

__awe32IsPlaying

```
BOOL
PASCAL
__awe32IsPlaying(
      WORD  wMIDIChannel,
      WORD  wBank,
      WORD  wPreset,
      WORD  wNote
)
```

Actions Check if a MIDI note is still playing.

Parameters *wMIDIChannel*

Specify from which MIDI channel the note off originated. Valid range for this variable is from 0 to 15 decimal.

wBank

Specify the user bank the note originated from. Valid range for this variable is from 0 to 127.

wPreset

Specify the note's preset number. Valid range for this variable is from 0 to 127.

wNote

Specify the note's note number. Valid range for this variable is from 0 to 127.

Return The return value is TRUE if the note is still playing, FALSE otherwise.

SoundFont Bank And Downloadable DRAM Services

This chapter gives an overview of the SoundFont Bank and WAVE file loading API. You will typically use these API when you want to customize the MIDI instruments and sound effects in your application.

These API use the following data structures :

```
typedef struct {
    SHORT bank_no;           /* Slot number being used */
    SHORT total_banks;       /* Total number of banks */
    LONG FAR* banksizes;     /* Pointer to a list of bank sizes */
    LONG reserved;           /* Unused */
    char FAR* data;          /* Address of buffer of size PACKETSIZE */
    char FAR* presets;       /* Allocated memory for preset data */

    LONG total_patch_ram;    /* Total patch ram available */
    SHORT no_sample_packets;/* No. of packets of samples to stream */
    LONG sample_seek;        /* Start file location of sound sample */
    LONG preset_seek;        /* Address of preset_seek location */
    LONG preset_read_size;   /* No. of bytes from preset_seek to */
                             /* read into a buffer */
    LONG preset_size;        /* Preset actual size */
} SOUND_PACKET;

typedef struct {
    SHORT tag;               /* Must be 0x100 or 0x101 */
    SHORT preset_size;       /* Preset table of this size is required */
    SHORT no_wave_packets;   /* No. of packets of Wave sample */
    LONG reserved;
```

```
        SHORT bank_no;          /* Bank number */
        char FAR* data;         /* Address of packet of size PACKETSIZE */
        char FAR* presets;      /* Allocated memory for preset data */
        LONG sample_size;       /* Sample size, i.e. number of samples */
        LONG samples_per_sec;   /* Samples per second */
        SHORT bits_per_sample;  /* Bits per sample, 8 or 16 */
        SHORT no_channels;      /* Number of channels, 1=mono, 2=stereo */
        SHORT looping;          /* Looping? 0=no, 1=yes */
        LONG startloop;         /* If looping, these addresses */
        LONG endloop;
        SHORT release;          /* Release time, 0=24ms, 8191=23.78s */
    } WAVE_PACKET;
```

SOUND_PACKET data structure is used in API that involve loading and unloading of SoundFont bank data objects.

WAVE_PACKET data structure provides WAVE loading functionality on top of SOUND_PACKET data structure.

awe32TotalPatchRam

```
WORD
PASCAL
awe32TotalPatchRam(SOUND_PACKET FAR* SP)
```

Actions Determine the total amount of RAM on the AWE32. This is not the amount of "unused" RAM but the total amount of RAM. **awe32TotalPatchRam** assumes the AWE32 card has been detected and initialized.

Parameters *SP*

Points to the SOUND_PACKET. **awe32TotalPatchRam** assumes that **SP** is not **NULL**.

Return If successful, **awe32TotalPatchRam** returns zero; otherwise, it returns non-zero. The following fields of **SP** will filled upon successful return from this API.

Member	Remarks
total_patch_ram	The total amount of RAM on the SB AWE32 card.

awe32DefineBankSizes

```
WORD
PASCAL
awe32DefineBankSizes(SOUND_PACKET FAR* SP)
```

Action Divide the AWE32's RAM into banks. All previous bank divisions are forgotten. You would normally invoke this API at the beginning of your application.

Parameters *SP*

Points to the SOUND_PACKET data object in which the size of each bank and the number of banks is specified. **awe32DefineBankSizes** assumes **SP** is not **NULL**.

Member	Remarks
total_banks	The total number of banks that will be used. This is also the number of elements in **SP->banks[]**. The number of banks may range from 1 to 64.
banksizes[]	The sizes of the banks are in **SP->banksizes[]**; the number of banks is in **SP->total_banks**. **SP->banksizes[0]** through **SP->banksizes[SP->total_banks-1]** are used to define the bank sizes. The sum of all the bank sizes must be less than or equal to the total amount of patch RAM (which may be obtained by calling **awe32TotalPatchRam**).

The AWE32 uses only 16-bit samples internally; all 8-bit sample data are converted to 16-bit by the library. Also, each bank requires 160 bytes of overhead for internal storage. So the bank memory required for a single sound is 160 bytes plus two bytes for every sample in the sound, regardless of whether the sound is composed of 8-bit samples or 16-bit samples.

That is, the bank memory required is

(2 bytes) * number_of_samples + 160 bytes

For example, the bank memory required for a WAV file of 10000 bytes of 16-bit samples is (10000 + 160 = 10160 bytes). The bank memory required by a WAV file of 15000 bytes of 8-bit samples is (2 bytes * 15000 + 160 = 30160 bytes).

The bank memory required for a SoundFont bank file that does not contain samples is simply 0 byte.

Return If successful, **awe32DefineBankSizes** returns zero; otherwise, it returns non-zero.

awe32SFontLoadRequest

```
WORD
PASCAL
awe32SFontLoadRequest(SOUND_PACKET FAR* SP)
```

Actions Parse the SoundFont bank header and prepares the specified SOUND_PACKET to download a SoundFont bank.

Parameters *SP*

Points to the SOUND_PACKET data object in which the bank number, the SoundFont file size, and the memory buffer are specified. **awe32SFontLoadRequest** assumes **SP** is not **NULL**.

Member	Remarks
bank_no	The bank number into which a SoundFont file will be loaded.
data	Points to the buffer where the first **PACKETSIZE** bytes of SoundFont file are read.

Return If successful, **awe32SFontLoadRequest** returns zero; otherwise, it returns non-zero. The following fields of **SP** will filled upon successful return from this API.

Member	Remarks
sample_seek	The offset in the SoundFont bank file where the instrument samples are stored. The application must seek to this position before it begins a read and **awe32StreamSample** loop.
no_sample_packets	The number of packets of **PACKETSIZE** bytes, each, that must be read and passed to **awe32StreamSample**.
preset_seek	The offset in the SoundFont bank file where the preset data are stored. The application must seek to this position before reading the preset data.
preset_read_size	This number of bytes the client should read for the presets.

Remarks The function requires that the entire SoundFont bank header be read into the buffer. In Real mode libraries, the size of the buffer is 512 bytes. If the size of the header is larger than 512 bytes, **awe32SFontLoadRequest** will fail. This could happen with SoundFont banks created with Vienna SF Studio. It is because Vienna allows users to insert a comment field in the header. If the field is long, the could be larger than 512 bytes. Please restrict the comment field to less than 128 characters long.

awe32StreamSample

```
WORD
PASCAL
awe32StreamSample(SOUND_PACKET FAR* SP)
```

Action For each call, **awe32StreamSample** loads one packet of SoundFont bank instrument samples into a bank. **awe32SFontLoadRequest** must be used before calling **awe32StreamSample**.

Parameters *SP*

Points to the SOUND_PACKET in which the bank number, the size of the SoundFont file, and the memory buffer are specified. **awe32StreamSample** assumes **SP** is not **NULL**.

Member	Remarks
bank_no	The bank number into which a SoundFont file will be loaded.
data	Points to the buffer that contains **PACKETSIZE** bytes of SoundFont data.

Return If successful, **awe32StreamSample** returns zero; otherwise, it returns non-zero.

awe32SetPresets

```
WORD
PASCAL
awe32SetPresets(SOUND_PACKET FAR* SP)
```

Action Use the specified presets (in **SP->presets**) for the specified SoundFont bank (in **SP->bank**_no). Until the **awe32Terminate** function is called or the presets are reset with **awe32ReleaseBank**, the library will continue to use the memory block pointed to by **SP->presets**.

Parameters *SP*

Points to the SOUND_PACKET in which the bank number, and the memory buffer are specified. **awe32SetPresets** assumes **SP** is not **NULL**.

Member	Remarks
bank_no	The bank number for which these presets will be set.
presets	Points to the buffer in which the presets are stored. The buffer should be at least **SP->presets_read_size** bytes in length. (**SP->presets_read_size** is set by calling **awe32SFontLoadRequest**). The library will continue to use this memory for the SoundFont bank, so you must not free it unless the library is terminated with **awe32Terminate** or the presets are reset with a call to **awe32ReleaseBank**.

Return If successful, **awe32SetPresets** returns zero; otherwise, it returns non-zero. The following fields of **SP** will filled upon successful return from this API.

Member	Remarks
preset_size	The actual size required in **SP->presets**. This will not be more than the value in **SP->preset_read_size**; typically, it will be about 30 percent smaller. The client may resize the presets memory block if the location of the block does not change. (The Standard C library function, **realloc**, may move a block to resize it, so **realloc** is not suitable for resizing the presets block.)

awe32ReleaseBank

```
WORD
PASCAL
awe32ReleaseBank(SOUND_PACKET FAR* SP)
```

Action Mark the memory being used for presets by the bank as free and makes the bank unusable. Applications are responsible for freeing any allocated memory buffers. Attempting to play a patch from the released bank results in undefined behavior.

Parameters *SP*

Points to the SOUND_PACKET in which the bank number is specified. **awe32ReleaseBank** assumes **SP** is not **NULL**.

Member	Remarks
bank_no	The bank number to be released.

Return If successful, **awe32ReleaseBank** returns zero, otherwise, it returns non-zero.

awe32ReleaseAllBanks

```
WORD
PASCAL
awe32ReleaseAllBanks(SOUND_PACKET FAR* SP)
```

Action Call the **awe32ReleaseBank** for each bank..

Parameters *SP*

Points to the SOUND_PACKET data object. **SP** is not used and is meant as a place-holder.

Return If successful, **awe32ReleaseAllBanks** returns zero; otherwise, it returns non-zero.

awe32WPLoadRequest

```
WORD
PASCAL
awe32WPLoadRequest(WAVE_PACKET FAR* WP)
```

Action Prepare the specified WAVE_PACKET to load wave data (PCM samples) into a specified bank. The wave data is later load into the bank by calling **awe32WPStreamWave**, or **awe32WPLoadWave**. So the wave data may reside in a file, memory, or any other place; the client has the responsibility of retrieving the data.

Parameters *WP*

Points to the WAVE_PACKET. **awe32WPLoadRequest** assumes **WP** is not **NULL**.

Member	Remarks
bank_no	The bank number into which the wave data will be loaded.
sample_size	Size of wave data in number of samples.
no_channels	The number of channels in the wave data. 1 is mono and 2 is stereo. Only mono (1) is supported.
bits_per_sample	The number of bits per sample. 8 and 16 bits samples are supported.
tag	Current version number of WAVE_PACKET. Must be *0x101*.

Return If successful, **awe32WPLoadRequest** returns zero; otherwise, it returns non-zero. In addition, values are returned in some fields of WAVE_PACKET object.

Member	Remarks
no_wave_packets	The number of wave packets of **PACKETSIZE** bytes, each, that must be passed to **awe32WPStreamWave**; if it is used.
preset_size	The number of bytes required by the library for presets for the bank. Allocate this much memory for this purpose.

awe32WPLoadWave

```
WORD
PASCAL
awe32WPLoadWave(WAVE_PACKET FAR* WP)
```

Action Load all of the wave data into a bank. Use this function instead of repeatedly calling **awe32WPStreamWave** when all the sound data can fit in a single block of memory. (In Real mode, the sound data must be less than 64 KB long.)

Parameters *WP*

Points to the WAVE_PACKET. **awe32WPLoadWave** assumes **WP** is not **NULL**.

Member	Remarks
data	Points to the sample data.
bits_per_sample	The wave data resolution (number of bits per sample) of the sample object. This should have the same value as it did when **awe32WPLoadRequest** was called.

Return If successful, **awe32WPLoadWave** returns zero; otherwise, it returns non-zero.

awe32WPStreamWave

```
WORD
PASCAL
awe32WPStreamWave(WAVE_PACKET FAR* WP)
```

Action Load one packet of wave data into the specified bank.

Parameters *WP*

Points to the WAVE_PACKET. **awe32WPStreamWave** assumes **WP** is not **NULL**. The fields in the WAVE_PACKET should have the same values they did when previous call to **awe32WPLoadRequest** returned.

Member	Remarks
bank_no	The bank number to which the wave data will be loaded.
data	Points to the wave data buffer of **PACKETSIZE** bytes.
bits_per_sample	The wave data resolution (number of bits per sample) of the sample object. This should have the same value as it did when **awe32WPLoadRequest** was called.

Return If successful, **awe32WPStreamWave** returns zero; otherwise, it returns non-zero.

awe32WPBuildSFont

```
WORD
PASCAL
awe32WBuildSFont(WAVE_PACKET FAR* WP)
```

Action Construct a SoundFont preset object for the download wave data. In effect the wave data becomes an instrument in the specified bank.

Parameters *WP*

Points to the WAVE_PACKET. **awe32WPBuildSFont** assumes **WP** is not **NULL**.

Member	Remarks
bank_no	The bank number into which the wave data will be loaded.
tag	Current version number of WAVE_PACKET. This value must be the same value that was present when **awe32WPLoadRequest** and **awe32WPStreamWave** were called.
sample_size	Holds the number of samples in the sample object. This must be the same value that was present when **awe32WPLoadRequest** and **awe32WPStreamWave** were called.
samples_per_sec	Holds the frequency of the wave data. Supported frequencies are *8000Hz*, *11025Hz*, *22050Hz*, and *44100Hz*.
looping	Holds zero if there is no loop in the sample. Otherwise, holds non-zero and the start and end of the loop as offsets into the WAVE data are specified in **WP->startloop** and **WP->endloop**, respectively.
release	Specify the duration of the release section of the sample. Zero causes a release duration of 24 milliseconds; 5940, 23.78 seconds. Values higher than 5940 are treated as 5940.
presets	Points to a block of memory the library may use for holding presets. The block of memory must be at least **WP->preset_size** bytes in length. (This length was obtained by calling **awe32WPLoadRequest**).

Return If successful, **awe32WPBuildSFont** returns zero; otherwise, it returns non-zero.

Real and Protected Mode API Programming Guide

This chapter gives a general guide in using the DOS Real and Protected mode API. This chapter consists of 10 sections:

- Using the libraries
- Initialization
- Termination
- Using embedded GM presets
- Loading a SoundFont Bank
- Loading wave data as an instrument
- Using **awe32DefineBankSizes**
- Starting and ending addresses
- Significance of **awe32NumG** variable
- Enable real-time panning using **awe32RTimePan** variable

Using the libraries

A header file CTAWEAPI.H for C/C++ compilers is included in this package. The libraries are provided in 2 formats: OBJ and LIB files. Libraries of 5 memory models are provided:

- Small memory model
- Compact memory model
- Medium memory model
- Large memory model
- Flat memory model (protected mode)

Please note that all libraries are compiled using structure members alignment of 1. For flat memory model library, the **FAR** keyword has been defined away, **LPBYTE** is the same as **PBYTE**.

Initialization

The SB AWE32 EMU8000 subsystem must be properly initialized prior to any MIDI playback. Two steps are required for initialization as shown be below. The example assumes that the base I/O addresses of the EMU8000 subsystem is at 0x620.

```
wEmuBase = 0x620;
if ( awe32Detect(wEmuBase) ) {
        // Error, EMU8000 not found
}
else {
        if ( awe32InitHardware() ) {
                // Error, initialising EMU8000 failed
        }
}
```

Termination

The SB AWE32 EMU8000 subsystem must be properly terminated when your application wishes to quit. Failure to do so may render any subsequent playing of FM music inaudible.

```
if ( awe32Terminate() ) {
        // Error, termination failed
}
```

Using embedded GM presets

General MIDI presets are included in this package, EMBED.OBJ. In order to embed the presets, you need to initialize the **awe32SoundPad** structure as shown below :

```
awe32SoundPad.SPad1 = awe32SPad1Obj;
awe32SoundPad.SPad2 = awe32SPad2Obj;
awe32SoundPad.SPad3 = awe32SPad3Obj;
awe32SoundPad.SPad4 = awe32SPad4Obj;
awe32SoundPad.SPad5 = awe32SPad5Obj;
awe32SoundPad.SPad6 = awe32SPad6Obj;
awe32SoundPad.SPad7 = awe32SPad7Obj;

if ( awe32InitMIDI() ) {
```

```
                // Error, MIDI engine initialisation failed
        }
```

The **awe32SoundPad** is of type SOUNDPAD and is defined in MIDIVAR.OBJ, and the **awe32SPadXObj** variables are defined in EMBED.OBJ.

awe32SoundPad has to be initialized before calling **awe32InitMIDI**. The General MIDI presets will be setup as Bank 0 and uses the ROM samples.

Loading a SoundFont Bank

There are several steps involved. For example, to load a User Bank.

```
        #include "ctaweapi.h"

        int i;
        FILE *fp;
        LONG banks[1];
        SOUND_PACKET sp;
        char buffer[PACKETSIZE];

        // Determine available patch DRAM
        awe32TotalPatchRam(&sp);
        if ( sp.total_patch_ram < 512*1024 ) {
                // Error, not enough patch DRAM
        }

        // Setup bank sizes
        banks[0] = sp.total_patch_ram;        // Use all available ram,
        sp.banksizes = banks;                 // could be less
        sp.total_banks = 1;                   // Total no. of banks
        if ( awe32DefineBankSizes(&sp) ) {
                // Error, invalid sizes
        }

        // Open SoundFont Bank
        fp = fopen("USER.SBK", "rb");
        fread(buffer, 1, PACKETSIZE, fp);     // Read SoundFont header

        // Prepare to load SoundFont Bank
        sp.bank_no = 0;                       // Load into bank 0
        sp.data = buffer;                     // Packet buffer
        if ( awe32SFontLoadRequest(&sp) ) {
                // Error, invalid SoundFont bank
        }

        // To stream sound sample
        if ( sp.no_sample_packets > 0 ) {
                fseek(fp, sp.sample_seek, SEEK_SET);
                for (i=0; i<sp.no_sample_packets; i++) {
                        fread(buffer, 1, PACKETSIZE, fp);
                        awe32StreamSample(&sp);
                }
        }

        // To load presets
        sp.presets = (char *) malloc(sp.preset_read_size);
        fseek(fp, sp.preset_seek, SEEK_SET);
        fread(sp.presets, 1, sp.preset_read_size, fp);
        if ( awe32SetPresets(&sp) ) {
                // Error, invalid SoundFont bank
        }

        if ( awe32InitMIDI() ) {
                // Error, MIDI engine initialization failed
        }
```

It is recommended that synthesizer SoundFont bank such as SYNTHGM.SBK loads as Bank 0.

Loading wave data as an instrument

The steps are similar to those of loading a SoundFont bank except that WAVE_PACKET functions are used.

```
#include "ctaweapi.h"

int i;
FILE *fp;
LONG banks[2];
WAVE_PACKET wp;
SOUND_PACKET sp;
char buffer[PACKETSIZE];

// Embed GM presets
awe32SoundPad.SPad1 = awe32SPad1Obj;
awe32SoundPad.SPad2 = awe32SPad2Obj;
awe32SoundPad.SPad3 = awe32SPad3Obj;
awe32SoundPad.SPad4 = awe32SPad4Obj;
awe32SoundPad.SPad5 = awe32SPad5Obj;
awe32SoundPad.SPad6 = awe32SPad6Obj;
awe32SoundPad.SPad7 = awe32SPad7Obj;

// Determine available patch DRAM
awe32TotalPatchRam(&sp);
if ( sp.total_patch_ram < 512*1024 ) {
        // Error, not enough patch DRAM
}

// Setup bank sizes
banks[0] = 0;                           // Embeded GM presets
banks[1] = sp.total_patch_ram;          // Use all available ram,
sp.banksizes = banks;                   // could be less
sp.total_banks = 2;                     // Total no. of banks
if ( awe32DefineBankSizes(&sp) ) {
        // Error, invalid sizes
}

// Open wave data
fp = fopen("WAVE.PCM", "rb");

// Prepare to load wave data
wp.tag = 0x101;                         // Tag
wp.bank_no = 1;                         // Load into bank 1
wp.data = buffer;                       // Packet buffer
wp.sample_size = 10240;                 // 10240 samples
wp.samples_per_sec = 22050;             // 22050 Hz
wp.bits_per_sample = 8;                 // 8-bit sample
wp.no_channels = 1;                     // Mono sample
wp.looping = 1;                         // Looping on
wp.startloop = 0;                       // Loop from beginning
wp.endloop = 10240;                     // To the end
wp.release = 0;                         // Immediate release
if ( awe32WPLoadRequest(&wp) ) {
        // Error, cannot use specified wave data
}

// To stream wave data
// See also awe32WPLoadWave
for (i=0; i<wp.no_wave_packets; i++) {
        fread(buffer, 1, PACKETSIZE, fp);
        awe32WPStreamWave(&wp);
}

// To build SoundFont presets
wp.presets = (char *) malloc(wp.preset_size);
if ( awe32WPBuildSFont(&wp) ) {
        // Error, cannot build SoundFont presets
}
```

```
if ( awe32InitMIDI() ) {
        // Error, MIDI engine initialization failed
}
```

Using **awe32DefineBankSizes**

Beside defining bank sizes, **awe32DefineBankSizes** can in the following ways:

- To resize an existing bank. However, the bank to be resized has to be the last bank.
- To define additional banks. The new banks have to be added beyond the last bank.

Assume that the RAM on the SB AWE32 is divided into 3 banks of 128 kilobytes each.

```
LONG banks[4];       /* reserve for 4 banks */
SOUND_PACKET sp;
.
.
banks[0] = 128000;
banks[1] = 128000;
banks[2] = 128000;
sp.total_banks = 3;
sp.banksizes = banks;
if ( awe32DefineBankSizes(&sp) ) {
        // Error, invalid sizes
}
```

The last bank can be resized by calling **awe32DefineBankSizes**.

```
banks[2] = 150000;     /* new size */
sp.total_banks = 3;
sp.banksizes = banks;
if ( awe32DefineBankSizes(&sp) ) {
        // Error, invalid sizes
}
```

An additional bank can be added beyond the last bank.

```
banks[3] = 64000;      /* new bank */
sp.total_banks = 4;
sp.banksizes = banks;
if ( awe32DefineBankSizes(&sp) ) {
        // Error, invalid sizes
}
```

In order to resize other banks, the bank(s) beyond the bank to be resized must be freed first by calling **awe32ReleaseBank**.

Starting and ending addresses

The starting and ending addresses for code and data segments of all the module files are marked.

Module	Start / End	Symbols
MIDIENG.OBJ	CODE Start	`__midieng_code()`
	CODE End	`__midieng_ecode()`

	DATA Start	`int* __midieng_code()`
	DATA End	`int* __midieng_ecode()`
SYSEX.OBJ	CODE Start	`__sysex_code()`
	CODE End	`__sysex_ecode()`
	DATA Start	`int* __sysex_code()`
	DATA End	`int* __sysex_ecode()`
HARDWARE.OBJ	CODE Start	`__hardware_code()`
	CODE End	`__hardware_ecode()`
	DATA Start	`int* __hardware_code()`
	DATA End	`int* __hardware_ecode()`
SFHELP1.OBJ	CODE Start	`__sfhelp1_code()`
	CODE End	`__sfhelp1_ecode()`
	DATA Start	`int* __sfhelp1_code()`
	DATA End	`int* __sfhelp1_ecode()`
SFHELP2.OBJ	CODE Start	`__sfhelp2_code()`
	CODE End	`__sfhelp2_ecode()`
	DATA Start	`int* __sfhelp2_code()`
	DATA End	`int* __sfhelp2_ecode()`
SBKLOAD.OBJ	CODE Start	`__sbkload_code()`
	CODE End	`__sbkload_ecode()`
	DATA Start	`int* __sbkload_code()`
	DATA End	`int* __sbkload_ecode()`
WAVLOAD.OBJ	CODE Start	`__wavload_code()`
	CODE End	`__wavload_ecode()`
	DATA Start	`int* __wavload_code()`
	DATA End	`int* __wavload_ecode()`
NRPN.OBJ	CODE Start	`__nrpn_code()`
	CODE End	`__nrpn_ecode()`
	DATA Start	`int* __nrpn_code()`
	DATA End	`int*  __nrpn_ecode()`
MIDIVAR.OBJ	DATA Start	`__midivar_data`
	DATA End	`__midivar_edata`
NRPNVAR.OBJ	DATA Start	`__nrpnvar_data`
	DATA End	`__nrpnvar_edata`
EMBED.OBJ	DATA Start	`__embed_data`
	DATA End	`__embed_edata`

For applications that require the starting and ending addresses of each file module, they can be obtained as follows:

```
// midieng.obj
void* code_start = (void*) __midieng_code;
void* code_end   = (void*) __midieng_ecode;
void* data_start = (void*) __midieng_code();
void* data_end   = (void*) __midieng_ecode();
```

For users who are developing drivers that make use of hardware interrupts in protected mode, all modules except HARDWARE.OBJ and SBKLOAD.OBJ and any additional preset buffers must be page locked. If NRPN is not used, NRPN.OBJ and NRPNVAR.OBJ can be omitted.

Significance of awe32NumG variable

If you intend to use only the ROM (i.e., General MIDI) instruments, you can set **awe32NumG** to 32 and you will be using the full 32 oscillators on the EMU8000 subsystem.

If you plan to use both ROM sound and DRAM instruments/sound effects, or instruments/sound effects from DRAM only, you have to set **awe32NumG** to 30, i.e., using only 30 oscillators on the EMU8000 subsystem. The remaining 2 oscillators have to be used for DRAM memory refresh. Failure to do so may result in the DRAM sound being playback incorrectly.

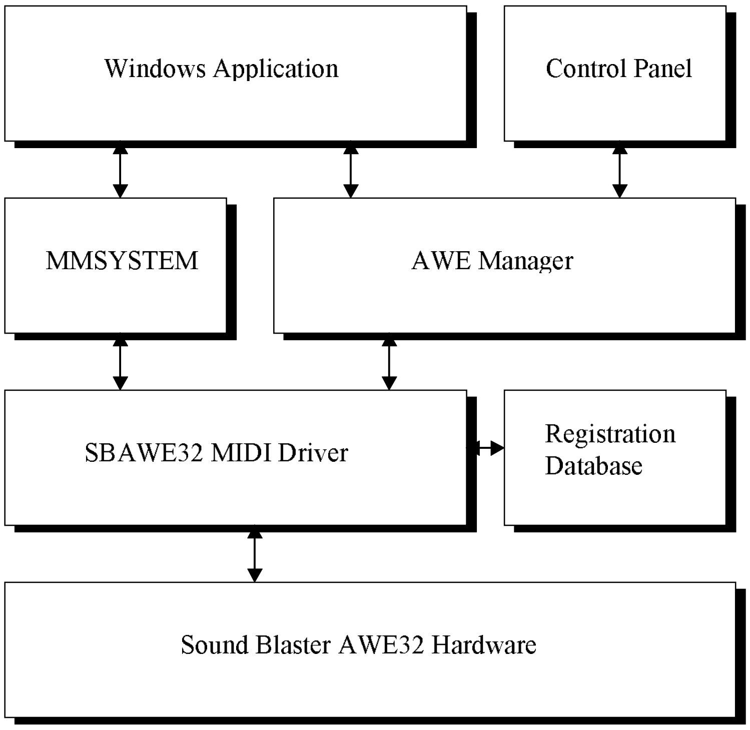
Enable real-time panning using awe32RTimePan variable

If you intend to use real-time panning, you could now enable it by setting **awe32RTimePan** to 1. The option is disabled by default because of the limitation in the EMU8000 hardware. The left and right volumes are not interpolated in real-time. As a result, if the pan positions are updated in real-time, zippering noise could be heard. However, you can reduce the zippering noise by update the pan positions in smaller steps.

PART III Windows Driver API

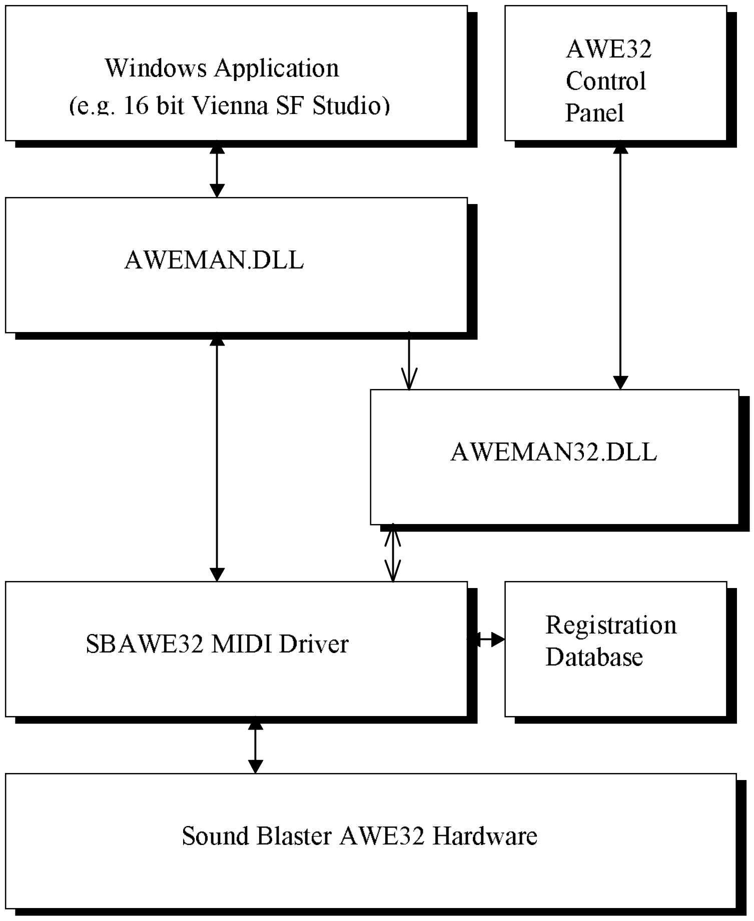
Overview

AWEMAN.DLL is a resource (effects microcode and user samples) manager. There are two types of resources, effects microcode and user samples. Download requests for effects microcode or user samples can be initiated by the user application that uses AWE Manager's API. When an effect is being selected by the user, AWE Manager will relay the request to the SBAWE32.DRV driver. This driver will download the required effect microcodes into the AWE32. The diagram below depicts the relationships of the libraries and drivers for the Sound Blaster AWE32.



There are 16bit and 32bit (Windows NT) versions of SBAWE32.DRV to accomodate different applications in different platform. AWE Manager also supports both versions: AWEMAN.DLL for Win16 and AWEMAN32.DLL for Win32. The previous diagram shows the architecture which is common to both platform.

AWEMAN.DLL contains API services which can be used by any 16bit applications running in Windows 3.1x or Windows 95 environment. AWEMAN32.DLL, on the other hand, supports 32bit applications running in Windows 95 or Windows NT 3.5x. The following diagram shows the relationship between AWEMAN.DLL and AWEMAN32.DLL when they are used in Windows 95.



AWEMAN32.DLL thunks from 32 bit to 16 bit in order to relay requests to SBAWE32.DRV. AWEMAN.DLL, however communicates with the driver directly. It also thunks to AWEMAN32.DLL in order to exchange hardware information.

Application Programmer Interface

The current implementation of API is via message-based system (different from Windows' WM_USER). The API provides parent applications with a set of messages, accessing the features of AWE32. The API exports one function call AWEManager. Applications communicate to AWEMAN.DLL through this function using the pre-defined messages. Each message will trigger different functions in AWEMAN.DLL.

AWE Manager DLL

This section describes the specifications of the Windows API. Applications should use messages describe in this chapter to interact with the SBAWE32.DRV.

This section is divided into four areas:

- AWE Manager Message Function
- AWE Manager Messages
- AWE Manager Message Reference
- AWE Manager Error Messages

The first section describes how to gain access to the AWE Manager entry point function. The second section gives a quick and brief description of what the AWE Manager DLL offers. The third section takes a deeper view into the message functions. The fourth section explains anticipated error messages.

In this chapter, AWE Manager DLL is also referenced as DLL or AWE Manager. And it could refer to either AWEMAN.DLL or AWEMAN32.DLL.

AWE Manager message function

AWEMAN.DLL provides a single message-based function entry point. This function has the following prototype:

```
LRESULT
FAR PASCAL AWEManager(
            AWEHANDLE   hUserID,
            UINT        unMsg,
            LPARAM      lParam1,
            LPARAM      lParam2
)
```

AWEMAN32.DLL provides an entry point which differ from the 16bit version.

```
LRESULT
WINAPI AWEManager(
            AWEHANDLE   hUserID,
            UINT        unMsg,
            LPARAM      lParam1,
            LPARAM      lParam2
)
```

The parameters' descriptions for both AWEMAN and AWEMAN32.DLL are as follows:

hUserID	Specify a user ID, issued by AWE Manager.
unMsg	Identifies a message that AWE Manager must process.
lParam1	
lParam2	Specify message dependent parameters.

The **AWEHANDLE** is actually a handle to a data structure, classed as CAWEObject, describing the settings and relationship between the application and the DLL. It is recommended that the AWE Manager DLL be dynamically linked with the application. In the application, the pointer to function declaration should bear the same parameter declaration as stated above. A copy of the declaration appears in the header file provided.

After making such declaration in the code and loading the DLL using the `LoadLibrary()` function call in Windows, the manager's message function can be accessed using a pointer to a function. It is important to realize that AWE Manager will only support one user application accessing the AWE32 per hardware device. It is the task of user application to 'release' the device before it can be accessed by other user application.

The parameters to the message function take the form of pre-defined structures. The user applications will have to allocate these structures before passing in as parameters. These structures have the following forms:

```
typedef struct {
   enum SBANK         m_SBankIndex;
   WORD               m_UBankIndex;
   WORD               m_InstrIndex;
   enum TYPEINDEX     m_TypeIndex;
   WORD               m_SubIndex;
   WORD               m_VariIndex[6];
} CParamObject;

typedef struct {
   DWORD              m_Size;
   LPSTR              m_Buffer;
   DWORD              m_SizeUsed;
   WORD               m_Flag;
} CBufferObject;
```

Not all of the fields within the structure will be used every time. Used members must be set to zeroes explicitly. It is the duty of both application and DLL to retrieve and set the values from the appropriate parameters. The long pointer version of the two structures is defined as *LPPARAMOBJECT* and *LPBUFFEROBJECT*.

```
typedef struct {
   WORD      m_SizeOf;
   DWORD      m_BaseAddr;
   DWORD      m_DevNode;
   DWORD      m_RomId;
   DWORD      m_RomVer;
   DWORD      m_hTask;
   DWORD      m_DevCaps;
   char       m_DevName[32];
   char       m_SndEngine[16];
   char       n_RegKey[256];
} CDevObject;
```

The CDevObject structure can be used to identify the capabilities of available SB AWE32 devices on the system. Some of the fields are only meaningful in certain platform.

All the above mentioned structures are already provided in the AWE_DLL.H header file. Hence, by including this header file in your application, the AWE Manager will be ready for accessing.

Manager Messages

The following messages are implemented in AWEMAN.DLL:

AWE_OPEN	Acquire AWE Manager to control and configure the hardware.
AWE_CLOSE	Releases control of AWE Manager to other applications.
AWE_GET_NUM_DEVS	Determine the total number of SBAWE32 devices available in the system.
AWE_GET_DEVICE_CAPS	Gets the device capabilities.
AWE_QUERY_EFXT_SUPPORT	Retrieve a list of available or supported Effect Types.
AWE_QUERY_EFXV_SUPPORT	Retrieve a list of available or supported Type Variations.
AWE_GET_EFX	Get the current Effect Types and/or Type Variations in use.
AWE_SELECT_EFX	Select an Effect Type and its variations, if any. This selection will be downloaded immediately into the hardware.
AWE_SELECT_EFX_EX	Extended version of AWE_SELECT_EFX.
AWE_QUERY_SYN_SUPPORT	Retrieve a list of available or supported Synthesizer Emulation.
AWE_GET_SYN_BANK	Get the bank descriptor of the current Emulation used in Synthesizer Bank.
AWE_SELECT_SYN_BANK	Select a Synthesizer Emulation for the Synthesizer Bank.
AWE_LOAD_USER_BANK	An application wishes to load a bank of instruments into the User Bank area.
AWE_GET_USER_BANK	An application wishes to retrieve the descriptor of a User Bank.
AWE_CLEAR_USER_BANK	Notifies AWE Manager to unload (remove) entire bank of instruments from the User Bank area.
AWE_LOAD_USER_INSTR	Load instrument presets into a User Bank.
AWE_GET_USER_INSTR	Retrieve the descriptor attached to an instrument in a bank. This bank can be either Synthesizer Bank or User Bank.
AWE_CLEAR_USER_INSTR	Remove instrument presets from a User Bank.
AWE_GET_UBANK_PARENT	To identify if a user bank is loaded by the Synthesizer Bank or loaded separately on its own.
AWE_QUERY_DRAM_SIZE	Retrieve current available and maximum memory on the AWE32 hardware.
AWE_GET_VERSION	Return the current AWE Manager version number.
AWE_SEND_MIDI	Sends a MIDI message directly to the driver.
AWE_ISHANDLE	To identify if the handle returned by the Manager is valid.
AWE_IS_DEVICE_FREE	Determine if a device is free.

Message Reference

Explanations of messages from the manager are documented below.

AWE_OPEN

Actions

An application sends this message to AWE Manager when it wishes to acquire the manager.

Parameters

lParam1

Specify a far pointer to a user declared handle type, *AWEHANDLE*. The DLL fills this location with a `hUserID` value if the initialization is successful.

lParam2

Specify a device number starting from 0 to (max. device - 1).

hUserID

Unused.

Remarks

This message can only be called once for each hardware device during initialization.

The manager will check for the existence of the MIDI driver and in-turn register with it. If the driver fails because of hardware failure or contention, this function will also fail. Upon successful initialization, the manager will return a unique user identification (handle) to the application. All subsequent calls to the manager should have this ID associated with

Following files must be present in Windows,

`SBAWE32.DRV` - AWE MIDI Driver v4.0 or above

Driver version 4.0 will have file time-stamp 4.00a or later. Opening more than once will cause the manager to return a busy message. If the MIDI driver is not available at the time of Open, the manager will still fail and return the access not permitted message. Such cases happen, for example, when Media Player has already acquire the MIDI device before the user application opens the manager

Return

The return value would be AWE_NO_ERR if the operation is successful and an error code otherwise. Possible error messages are:

AWE_ERR_DEVICE_DRV_INVALID

AWE_ERR_DLL_BUSY

AWE_ERR_SYSMEM_INSUFFICIENT

AWE_ERR_ACCESS_NOT_PERMITTED

AWE_ERR_VERSION_INVALID

See Also

AWE_CLOSE

AWE_CLOSE

Actions An application sends this message to AWE Manager when it wishes to release control of the acquired AWE32 device to other applications.

Parameters *lParam1*

Unused.

lParam2

Unused.

hUserID

Current ID assigned to the application by AWE Manager during initialization.

Remarks The following are some important points to note when using this function.

The close function will not cause Windows to unload the AWE Manager from memory. Unloading depends on whether anymore application is still accessing it. Only when the last application issues a close message, then the manager will be unloaded. This is the 'last one off the lights' metaphor. When the manager is unloaded by Windows, all unused handles will be 'clean' up by the manager.

The application mentioned above can only be either user applications or control panel application and not two or more of the same kind

Return The return value is AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error message is:

AWE_ERR_USERID_INVALID

See Also AWE_OPEN

AWE_GET_NUM_DEVS

Actions An application sends this message to AWE Manager when it wishes to determine how many AWE devices are available on the system.

Parameters *lParam1*

Specify a far pointer to a WORD data type. The DLL fills this location with the number of AWE devices available on the system.

lParam2

Unused.

hUserID

Unused.

Remarks The returned number gives an indication of how many AWE32 devices are actually supported by the MIDI driver. This is dependent on the different Windows platforms. Currently, only Windows 95 supports multiple devices. For other platforms, the number returned will always be 1 if the driver is successfully initialized.

Return The return value is AWE_NO_ERR if the operation is successful, and an error code otherwise.

See Also AWE_GET_DEVICE_NODE, AWE_GET_DEVICE_CAPS

AWE_GET_DEVICE_CAPS

Actions An application sends this message to AWE Manager when it wishes to know the capabilities of a device.

Parameters *lParam1*

Specify a DWORD data type. This parameter can either contain a device id or device node. Device Id simply ranges from 0 to the maximum number of devices supported by the driver. Device node can be retrieved using AWE_GET_DEVICE_NODE.

lParam2

Specify a far pointer to CDevObject (LPDEVOBJECT) data type. The DLL will fill this structure with the capabilities of the device.

hUserID

Unused.

Remarks The m_SizeOf field of CDevObject structure must be initialized first before calling this API. This is to ensure that future version of API will not have problem filling up the entries of the structures. The members of this structure are:

Member	Remarks
m_SizeOf	Contains the size of the CDevObject.
m_BaseAddr	Contains the base I/O address of the device in question. e.g 0x620, 0x640 etc.
m_DevNode	Contains the device node of the device (Windows 95).
m_RomId	Contains the ROM ID of the device.
m_RomVer	Contains the ROM version of the device.
m_hTask	Contains the task of the application that is currently owning the device.
m_DevCaps	Contains the driver's capabilities in supporting the device. This member is BIT-ORed.
m_DevName[32]	Contains the official name of the device registered in the system.
m_SndEngine[16]	Contains the name of the sound engine supported by the device.
m_RegKey[256]	Contains the registry key which information are stored (Windows 95).

Return The return value is AWE_NO_ERR if the operation is successful, and an error code otherwise.

See Also AWE_GET_DEVICE_NODE, AWE_GET_NUM_DEVS

AWE_QUERY_EFXT_SUPPORT

Actions An application sends this message to the AWE Manager to retrieve a list of supported Effect Types.

Parameters *lParam1*

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Buffer size in characters.
m_Buffer	Far pointer of buffer area.

The manager upon returning will fill or change the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Flag	Number of entries stored in buffer.
m_Buffer	String entries of effect types.

The **m_SizeUsed** includes the '\0's used to delimit entries. Current version of MIDI drivers support only one Effect Type. The string representations (not including quotes) are as follows:

String	Type	Sub-Index
"Reverb & Chorus"	0	0 - Reverb
		1 - Chorus
		2 - Bass
		3 - Treble

The buffer area will be filled with Effect Types supported by the current MIDI driver. These entries are in string format, each terminated with a '\0'. The last entry of the list will have two consecutive '\0's. An example is:

"Reverb & Chorus\0\0"

With the example above, the **m_SizeUsed** is 17 and **m_Flag** is 1.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The followings are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application.

If the buffer is not sufficient to complete the operation, AWE Manager will not update string entries into the buffer. However, other return parameters will still be updated.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

AWE_ERR_SYSMEM_INSUFFICIENT

See Also AWE_QUERY_EFXV_SUPPORT, AWE_GET_EFX, AWE_SELECT_EFX

AWE_QUERY_EFXV_SUPPORT

Actions An application sends this message to the AWE Manager to retrieve a list of supported Type Variations.

Parameters *lParam1*

Specify a far pointer to CParamObject, *LPPARAMOBJECT*. When the application is calling the manager, the following fields of the structure are used,

Member	Remarks
m_SubIndex	The Sub-Index of the effect types.
m_TypeIndex	The effect types index.

The 'Reverb & Chorus' Effect Type supports four sets of variations: Reverb, Chorus, Bass, and Treble. For example, if the application wishes to query the variations supported by Chorus of 'Reverb and Chorus', then **m_SubIndex** will be 1 and **m_TypeIndex** will be 0. The manager will fill the following field when returning:

Member	Remarks
m_SubIndex	Total number of Sub-Index available for current selected Effect Types.

When returning, AWE Manager will update the **m_SubIndex** to contain the maximum number of Sub-Index available for the Effect Type. Each set of variations has the following entries and variation index.

String Equivalent	Vari-Index	Type	Sub-Index
"Room 1"	0	0	0
"Room 2"	1	0	0
"Room 3"	2	0	0
"Hall 1"	3	0	0
"Hall 2"	4	0	0
"Plate"	5	0	0
"Delay"	6	0	0

"Panning Delay"	7	0	0
"Chorus 1"	0	0	1
"Chorus 2"	1	0	1
"Chorus 3"	2	0	1
"Chorus 4"	3	0	1
"Feedback Delay"	4	0	1
"Flanger"	5	0	1
"Short Delay"	6	0	1
"Short Delay FB"	7	0	1
"-12dB"	0	0	2
"-8dB"	1	0	2
"-6dB"	2	0	2
"-4dB"	3	0	2
"-2dB"	4	0	2
"0dB"	5	0	2
"2dB"	6	0	2
"4dB”	7	0	2
"6dB”	8	0	2
"8dB”	9	0	2
"10dB”	10	0	2
"12dB"	11	0	2
"-12dB"	0	0	3
"-8dB"	1	0	3
"-6dB"	2	0	3
"-4dB"	3	0	3
"-2dB"	4	0	3
"0dB"	5	0	3
"2dB"	6	0	3
"4dB”	7	0	3
"6dB”	8	0	3
"8dB”	9	0	3
"10dB”	10	0	3
"12dB"	11	0	3

lParam2

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. Upon calling, the application will fill the following fields:

Member	Remarks
m_Size	Indicates the size of buffer available.
m_Buffer	Far pointer to a buffer area.

The Manager upon returning, will fill the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Flag	Number of entries written into the buffer.
m_Buffer	String entries of type variations.

The **m_SizeUsed** includes the '\0's used to delimit entries. The buffer area will be filled by the manager with the entries of Type Variation supported by the current MIDI driver. These entries are in string format, each terminated with a '\0'. The last entry of the list will have two consecutive '\0's. As an example, we have:

```
"Chorus 1\0Chorus 2\0Chorus 3\0Chorus 4\0Feedback
Delay\0Flanger\0Short Delay\0Short Delay FB\0\0"
```

The **m_Flag** will have 8 and **m_SizeUsed** will be filled with 87.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip a General Protection Fault in Windows.

If the buffer is not sufficient to complete the operation, AWE Manager will not update string entries into the buffer. However, other return parameters will still be updated.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_EFXT_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

AWE_ERR_SYSMEM_INSUFFICIENT

See Also AWE_QUERY_EFXT_SUPPORT, AWE_GET_EFX, AWE_SELECT_EFX

AWE_GET_EFX

Actions An application sends this message to the AWE Manager to retrieve the currently in use Effect Type and its Type Variations if any.

Parameters *lParam1*

Specify a far pointer to CParamObject, *LPPARAMOBJECT*. The manager will fill the following field with the following appropriate values:

Member	Remarks
m_TypeIndex	The effect types index.
m_SubIndex	Number of variations index returned.

m_VariIndex[..]	The type variations index.

The **m_VariIndex[..]** is an array type. 'Reverb & Chorus', only 0 and 1 are supported respectively. The 'Reverb and Chorus' has four sets of variations ('Reverb', 'Chorus', 'Treble' and 'Bass') and is the largest. This might change in future. To access them, simply index the **m_VariIndex[..]** with the appropriate numbers e.g. **m_VariIndex[0]** for Reverb and **m_VariIndex[1]** for Chorus. For unused indices, e.g. **m_VariIndex[5], m_VariIndex[6],** it must be zeroed out. If the Effect Types specified by **m_TypeIndex** does not have variations, the **m_VariIndex[..]** will not be used.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error is:

AWE_ERR_USERID_INVALID

See Also AWE_QUERY_EFXT_SUPPORT, AWE_QUERY_EFXV_SUPPORT,

AWE_SELECT_EFX

AWE_SELECT_EFX

Actions An application sends this message to inform the AWE Manager of the Effect Types and Variations that it wishes to use.

Parameters *lParam1*

Specify a far pointer to CParamObject, *LPPARAMOBJECT.* The application should fill the following fields with appropriate values:

Member	Remarks
m_TypeIndex	The effect types index.
m_VariIndex[..]	The type variations index.

The **m_VariIndex[..]** is an array type. At this moment, for 'Reverb & Chorus', only 0 and 1 are supported respectively. The 'Reverb and Chorus' has two sets of variations (one for each of 'Reverb' and 'Chorus') and is the largest. This might change in future. To access them, simply index the **m_VariIndex[..]** with the appropriate numbers e.g. **m_VariIndex[0]** for Reverb and **m_VariIndex[1]** for Chorus. If the Effect Types specified by **m_TypeIndex** does not have variations, the **m_VariIndex[..]** will not be used.

String	Type	Sub-Index
"Reverb & Chorus"	0	0 - Reverb

1 - Chorus

If the Type Index is pointing to 'Reverb & Chorus', then **m_VariIndex[0]** should contain the Vari-Index of 'Reverb' and **m_VariIndex[1]** should have index of 'Chorus'.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_EFXT_CHANGE_NOT_ALLOWED

AWE_ERR_EFXT_INVALID

AWE_ERR_EFXV_INVALID

See Also AWE_SELECT_EFX_EX, AWE_QUERY_EFXT_SUPPORT, AWE_QUERY_EFXV_SUPPORT, AWE_GET_EFX

AWE_SELECT_EFX_EX

Actions An application sends this message to inform the AWE Manager of the Effect Types and Variations that it wishes to use. This is an extended version and should try to aviod using AWE_SELECT_EFX as far as possible.

Parameters *lParam1*

Specify a far pointer to CParamObject, *LPPARAMOBJECT*. The application should fill the following fields with appropriate values:

Member	Remarks
m_TypeIndex	The effect types index.
m_VariIndex[..]	The type variations index.

The **m_VariIndex[..]** is an array type. 'Reverb & Chorus', only 0 and 1 are supported respectively. The 'Reverb and Chorus' has four sets of variations ('Reverb', 'Chorus', ‘Treble’ and ‘Bass’) and is the largest. This might change in future. To access them, simply index the **m_VariIndex[..]** with the appropriate numbers e.g. **m_VariIndex[0]** for Reverb and **m_VariIndex[1]** for Chorus, etc. For unused indices, e.g. **m_VariIndex[5], m_VariIndex[6],** it must be zeroed out. If the Effect Types specified by **m_TypeIndex** does not have variations, the **m_VariIndex[..]** must be set to zero.

String	Type	Sub-Index
"Reverb & Chorus"	0	0 - Reverb
		1 - Chorus
		2 - Bass
		3 - Treble

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_EFXT_CHANGE_NOT_ALLOWED

AWE_ERR_EFXT_INVALID

AWE_ERR_EFXV_INVALID

See Also AWE_QUERY_EFXT_SUPPORT, AWE_QUERY_EFXV_SUPPORT, AWE_GET_EFX

AWE_QUERY_SYN_SUPPORT

Actions An application sends this message to the AWE Manager to request a list of available or supported Emulations for Synthesizer Bank.

Parameters *lParam1*

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer available.
m_Buffer	Far pointer to a buffer area.

Upon returning, the manager will update the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Flag	Number of entries written into the buffer.
m_Buffer	String entries of synthesizer emulation.

The buffer area will be filled by the manager with the entries of Synthesizer Emulation supported by the current MIDI driver. These entries are in string format, each delimited by a '\0'. The last entry of the list will have two consecutive '\0's. Using the above example, we have:

```
"General MIDI\0GS\0MT 32\0User Custom Synth\0\0"
```

The size returned includes the '\0's used to delimit two entries. The current version of MIDI driver supports the following Synthesizer Emulation:

String	**Idx**	**Entry**	**File**
"General MIDI	0	GM	SYNTHGM.SBK
"GS"	1	GS	SYNTHGS.SBK
"MT 32"	2	MT	SYNTHMT.SBK
"User Custom Synth"	3	USER	SYNTHUSR.SBK

The bank files can be located in the default directory of AWE. This default directory is usually a sub-directory of the Sound Blaster path. The Sound Blaster path can be located from the environment variable "`SOUND`". The default name is used whenever a user specified file cannot be located. The path specified by the user can be found in the `SBWIN.INI` file under "AWE32" section. The entry name for each respective synthesizer emulation are listed in the table. Hence,

```
        [AWE32]
        GM = C:\SBANK\USER1.SBK
```

will have an assignment of `USER1.SBK` file as the General MIDI Bank.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks

The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip General Protection Fault in Windows

If the buffer is not sufficient to complete the operation, AWE Manager will not update string entries into the buffer. However, other return parameters will still be updated.

Return

The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

AWE_ERR_SYSMEM_INSUFFICIENT

See Also

AWE_SELECT_SYN_BANK, AWE_GET_SYN_BANK

AWE_GET_SYN_BANK

Actions An application sends this message to the AWE Manager to query for the current Synthesizer Bank and its descriptor.

Parameters *lParam1*

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer available.
m_Buffer	Far pointer to a buffer area.

Upon returning, the manager will update the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Flag	The synthesizer emulation index.
m_Buffer	String of current bank descriptor.

The buffer area will be filled by the manager with the descriptor of the current Synthesizer Bank. The **m_Flag** will contain the Synth Bank Index to the available emulation.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip General Protection Fault in Windows.

If the buffer is not sufficient to complete the operation, AWE Manager will not update string entries into the buffer. However, other return parameters will still be updated.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

See Also AWE_QUERY_SYN_SUPPORT, AWE_SELECT_SYN_BANK

AWE_SELECT_SYN_BANK

Actions An application sends this message to inform the AWE Manager of the emulation it wishes to use as current Synthesizer Bank.

Parameters

lParam1

Specify a word data type. This word contains the SBank Index, pointing to the desired Synthesizer Emulation. The following shows a list of available Emulations loadable into Synthesizer Bank:

String	Idx	Entry	File
"General MIDI"	0	GM	SYNTHGM.SBK
"GS"	1	GS	SYNTHGS.SBK
"MT 32"	2	MT	SYNTHMT.SBK
"User Custom Synth"	3	USER	SYNTHUSR.SBK

For Windows 3.1x, the bank files can be located in the default directory of AWE. This default directory is usually a sub-directory of the Sound Blaster path. The Sound Blaster path is located from the environment variable "`SOUND`". The default name is used whenever a user specified file cannot be located. The path specified by the user can be found in the `SBWIN.INI` file under the "AWE32" section. The entry name for each respective synthesizer emulation is listed in the table. Hence,

```
[AWE32]
GM = C:\SBANK\USER1.SBK
```

will have an assignment of `USER1.SBK` file as the General MIDI Bank.

For Windows 95 and Windows NT, the path names for bank files are located in the Registry. If the specified path is not found, the driver will look for the file in Windows' SYSTEM directory.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks

This function does not allow an application to specify a file name and overwrite the original Synthesizer Bank instrument files. The AWE Manager uses a predetermined name which is known to both the MIDI driver and itself. To use a user-define emulation as Synthesizer Bank, copy it to the default AWE directory and rename it to `SYNTHUSR.SBK`.

Return

The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_SBANK_INVALID

AWE_ERR_PATHNAME_INVALID

AWE_ERR_SYSMEM_INSUFFICIENT

AWE_ERR_DRAM_INSUFFICIENT

See Also

AWE_QUERY_SYN_SUPPORT, AWE_GET_SYN_BANK

AWE_LOAD_USER_BANK

Actions

An application sends this message to inform the AWE Manager that it wishes to load a bank of instruments into the User Bank area. The source can be either in a file or in memory blocks.

Parameters

lParam1

Specify a word data type. The application should fill the word with the desired User Bank number. The valid range of User Bank numbers is 1 to 127.

lParam2

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer area.
m_Flag	Indicates if buffer contains a path name.
m_Buffer	Far pointer to a buffer area.

The **m_Flag** field is used to indicate whether the User Bank will be loaded from a file or from a chunk of memory buffer. The value of m_Flag should be either OPER_MEMORY or OPER_FILE where OPER_MEMORY indicates loading from memory buffer pointer by the **m_Buffer** and OPER_FILE indicates loading from a file.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks

The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip a General Protection Fault in Windows.

It is recommend that the application use the macro defined by this API library. The OPER_FILE and OPER_MEMORY are constant macros used to distinguish between file loading or memory operations.

Return

The return value will be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_PATHNAME_INVALID

AWE_ERR_USER_OBJ_INVALID

AWE_ERR_UBANK_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

AWE_ERR_SYSMEM_INSUFFICIENT

AWE_ERR_DRAM_INSUFFICIENT

See Also AWE_LOAD_USER_INSTR, AWE_CLEAR_USER_BANK

AWE_GET_USER_BANK

Actions An application sends this message to the AWE Manager to request for the descriptor describing the User Bank.

Parameters *lParam1*

Specify a word data type. This word contains the UBank Index, pointing to one of the 127 User Banks. Valid range is 1 to 127.

lParam2

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer available.
m_Buffer	Far pointer to a buffer area.

Upon returning, the manager will update the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Buffer	String of current bank descriptor.

The buffer area will be filled by the manager with the descriptor of the selected User Bank. The string ends with two consecutive '\0's. Size returned includes the '\0's.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to a trip General Protection Fault in Windows.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_UBANK_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

See Also AWE_LOAD_USER_BANK, AWE_CLEAR_USER_BANK

AWE_CLEAR_USER_BANK

Actions An application sends this message to remove a loaded User Bank from memory.

Parameters *lParam1*

Specify a word data type. This word contains the UBank Index indicating which User Bank to be removed from memory. Valid range is 1 to 127.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value will be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_UBANK_INVALID

See Also AWE_LOAD_USER_BANK, AWE_CLEAR_USER_BANK

AWE_LOAD_USER_INSTR

Actions An application sends this message to download a chunk of instrument presets into the AWE DRAM. The loading can be either from file or direct from memory area.

Parameters *lParam1*

Specify a far pointer to CParamObject, *LPPARAMOBJECT*. The application should fill the following fields with appropriate values:

Member	Remarks
m_UBankIndex	The index to designate User Bank.
m_InstrIndex	The index to an Instrument within the UBank.

The **m_InstrIndex** will be the instrument offset of the User Bank specified by UBank Index. The **m_InstrIndex** must be a valid value between 0 to 127.

lParam2

Specify a far pointer to a CBufferObject data type, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer area.

m_SizeUsed	Indicates Instrument number of the bank located by m_Buffer.
m_Flag	Indicates if buffer contains a path name.
m_Buffer	Far pointer to a buffer area.

The **m_Flag** field is used to indicate whether the User Bank will be loaded from a file or from a chunk of memory buffer. The value of **m_Flag** should be either OPER_MEMORY or OPER_FILE where OPER_MEMORY indicates loading from memory buffer pointer by the **m_Buffer** and OPER_FILE indicates loading from a file. The **m_SizeUsed** is loaded with the Instrument index which will be loaded from the bank located by the **m_Buffer** variable.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks

The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip a General Protection Fault in Windows.

It is recommend that the application use the macro defined by this API library. The OPER_FILE and OPER_MEMORY are constant macros used to distinguish between file loading or memory operations.

This function will overwrite any instrument presets previously attached to the m_InstrIndex. No error or warning message is given.

Return

The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_PATHNAME_INVALID

AWE_ERR_USER_OBJ_INVALID

AWE_ERR_INSTR_INVALID

AWE_ERR_UBANK_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

AWE_ERR_SYSMEM_INSUFFICIENT

AWE_ERR_DRAM_INSUFFICIENT

See Also

AWE_CLEAR_USER_INSTR

AWE_GET_USER_INSTR

Actions

An application sends this message to the AWE Manager to request for an instrument descriptor of the Synthesizer Bank or User Bank.

Parameters

lParam1

Specify a far pointer to a CParamObject data type, *LPPARAMOBJECT*. The application should fill the following field with appropriate values:

Member	Remarks
m_UBankIndex	The index to destinate User Bank.
m_InstrIndex	The index to an Instrument within the UBank.

The **m_InstrIndex** will be the instrument offset of the User Bank specified by UBank Index. The **m_InstrIndex** must be a valid value between 0 to 127.

lParam2

Specify a far pointer to CBufferObject, *LPBUFFEROBJECT*. When the application calls the manager, the following fields are used:

Member	Remarks
m_Size	Indicates the size of buffer available.
m_Buffer	Far pointer to a buffer area.

Upon returning, the manager will update the following fields:

Member	Remarks
m_SizeUsed	Number of characters from buffer used.
m_Buffer	String of current instrument descriptor.

The buffer area will be filled by the manager with the descriptor of the instrument.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks

The following are some important points to note when using this function.

The location passed in as pointer must be valid memory locations allocated by the parent application.

If the buffer is not sufficient to complete the operation, AWE Manager will not update string entries into the buffer. However, other return parameters will still be updated.

Return

The return value will be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_UBANK_INVALID

AWE_ERR_INSTR_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

See Also

AWE_LOAD_USER_INSTR, AWE_CLEAR_USER_INSTR

AWE_CLEAR_USER_INSTR

Actions An application sends this message to clear an instrument's presets located in the User Bank area.

Parameters *lParam1*

Specify a word data type. The word contains the UBank Index where the instrument is to be removed from.

lParam2

Specify a word data type. This word contains the instrument number in the User Bank specified by the UBank Index in *lParam1*. The instrument's presets will be deleted from the memory area.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_DEVICE_BUSY

AWE_ERR_INSTR_INVALID

AWE_ERR_UBANK_INVALID

See Also AWE_LOAD_USER_INSTR

AWE_GET_UBANK_PARENT

Actions An application sends this message to manager requesting the it identify the original parent of specified user bank. This parent is considered as the bank which loads the user bank.

Parameters *lParam1*

A far pointer to a *WORD* containing the User Bank index. The valid range for this is 1 to 127.

The driver upon returning will fill or change the contents of this word. Three types of values may be returned.

lParam1	Remarks
Unchanged	The User Bank in query actually is self loaded (i.e. parent of itself).
0 to 127	The User Bank is loaded in by other bank. The parent bank has the index from 0 to 127. 0 signify Synthesiser Bank.
-1	The User Bank in query is not loaded yet.

lParam2

Unused

Remark This message is introduced because there is no way to differentiate between a User Bank that is self-loaded or loaded by other User Bank.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible errors are:

AWE_ERR_USERID_INVALID

AWE_ERR_RESOURCE_INSUFFICIENT

See Also None.

AWE_QUERY_DRAM_SIZE

Actions An application sends this message to the AWE Manager to request for current available and maximum DRAM size in the AWE hardware.

Parameters *lParam1*

Specify a far pointer to a double word, *DWORD FAR* *. This location will be used to store the maximum amount of DRAM found on the hardware.

lParam2

Specify a far pointer to a double word, *DWORD FAR* *. This location will be used to store the current available amount of DRAM found on the hardware.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The unit for both *lParam1* and *lParam2* is words. That is, the return size is in terms of number of words.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error is:

AWE_ERR_USERID_INVALID

See Also None.

AWE_GET_VERSION

Actions This message retrieves the AWE Manager version number.

Parameters *lParam1*

Specify a far pointer to word, *WORD FAR* *. The location of this pointer will be filled with the current major version of the AWE Manager.

lParam2

Specify a far pointer to word, *WORD FAR* *. The location of this pointer will be filled with the current minor version of the AWE Manager.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks	None.
Return	The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error is: AWE_ERR_USERID_INVALID
See Also	None.

AWE_SEND_MIDI

Actions An application sends this message to the manager requiring it to service MIDI commands.

Parameters *lParam1*

A *DWORD* containing the MIDI message. This message follows the format as specified by the Standard MIDI Message Format. The message is decoded as:

```
00000000 0ccccccc 0bbbbbbb 1aaaaaaa
High-Order Byte          Low-Order Byte
```

where,

aaaaaaa:	Specify the status byte.
bbbbbbb:	Specify the first data byte. (0-127)
ccccccc:	Specify the second data byte. (0-127)

lParam2

Unused

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks This message is provided so that MIDI events can inserted synchronously without go through the MMSYSTEM. The MIDI command accepted here are generic commands drafted. If a command is not supported by the driver, it will not be executed

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error is:

AWE_ERR_USERID_INVALID

AWE_ERR_DRIVER_BUSY

See Also None.

AWE_ISHANDLE

Actions An application sends this message to the AWE Manager to identify if a handle is a successful candidate of the manger. Applications may use this message to validate a handle before committed to any operations.

Parameters *lParam1*

Specify a declared handle type, *AWEHANDLE*. This is the handle used in query.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks The following are some important points to note when using this function :

This message is only meaningful to be called after successful acquisition of the Manager.

The location passed in as pointer must be valid memory locations allocated by the parent application. If the locations are invalid, it is possible to trip General Protection Fault in Windows.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise. Possible error is:

AWE_ERR_USERID_INVALID

See Also None.

AWE_IS_DEVICE_FREE

Actions An application sends this message to the AWE Manager to identify if a device is free and available. Application may wish to do so before attempting to open the device.

Parameters *lParam1*

Specify a DWORD data type. Indicates the device number in query.

lParam2

Unused.

hUserID

Current ID assigned to the application by the AWE Manager during initialization.

Remarks None.

Return The return value would be AWE_NO_ERR if the operation is successful, and an error code otherwise

See Also None.

Error Messages

The error messages generated by the AWE Manager are described below. They can be divided into three categories:

- General Error Messages
- Invalid Parameter Messages
- Resource Contention Messages

General Error Messages

AWE_NO_ERR	No error message. This message will be display most of the time to indicate a successful operation.
AWE_ERR_UNDEFINE_ERROR	A reserved error message indicating an error not defined during development. A new error message may be added to it at a later stage when the cause of the error is identified.

Invalid Parameter Messages

AWE_ERR_DEVICE_DRV_INVALID	This error message is return when the SBAWE32.DRV MIDI driver is not located.
AWE_ERR_USERID_INVALID	An invalid user ID is used to gain access to the AWE Manager. This ID should be obtained from the AWE_OPEN API call.
AWE_ERR_EFXT_INVALID	An invalid Effect Type Index is specified. This error happens when an application specifies an Effect Type Index that is not currently supported by the MIDI driver.
AWE_ERR_EFXV_INVALID	An invalid Type Variation index is specified. This error happens when an application specifies a Type Variation index that is not supported by the MIDI driver.
AWE_ERR_SBANK_INVALID	An invalid Synthesizer Bank index is specified. This error occurs when the MIDI driver cannot support the specified Synthesizer Bank index.
AWE_ERR_PATHNAME_INVALID	An invalid path name to a file is specified. This happens when an operation, which requires a user file or a predetermined path, cannot locate the desired file.
AWE_ERR_USER_OBJ_INVALID	An user object module has an incorrect format. This error occurs when an application attempts to download an unrecognized bank of

	instruments (or single instrument) into a specific User Bank area. This module uses the SBK format.
AWE_ERR_INSTR_INVALID	An invalid instrument index is specified. This index is only supported within the range of 0 to 127, i.e. 128 instruments.
AWE_ERR_UBANK_INVALID	An invalid User Bank index is specified. This index is only supported within the range of 1 to 127. An error occurs when the application points to a bank index higher or lesser than the legal range.
AWE_ERR_MSG_INVALID	Indicates that an undefined API message is received by the manager.
AWE_ERR_ACCESS_NOT_PERMITTED	When an application calls the manager without first acquiring the MMSystem, the AWE Manager will not allow the application to gain access to it.
AWE_ERR_VERSION_INVALID	Indicates that a incompatible MIDI driver is used. This driver does not conform to the required version number of the Manager.

Resource Contention Messages

AWE_ERR_DLL_BUSY	Notifies the application that another application has already accessed the manager and is still accessing it. This error message is to prevent multiple applications calling the manager simultaneously.
AWE_ERR_DEVICE_BUSY	Indicates that the MIDI driver is performing hardware specific operations. The calling application may have to wait till it is ready.
AWE_ERR_RESOURCE_INSUFFICIENT	Indicates that the calling application has provided insufficient memory buffer space required to complete the current query operation.
AWE_ERR_SYSMEM_INSUFFICIENT	Indicates that the AWE Manager is unable to allocate any more memory from Windows for internal use.
AWE_ERR_EFXT_CHANGE_NOT_ALLOWED	Indicates that effect types cannot be changed while MIDI driver is in play back mode.
AWE_ERR_DRAM_INSUFFICIENT	Indicates that the MIDI driver has detected insufficient memory available on the AWE hardware to complete an operation.

Windows Programming Guide

This chapter provides you with examples on how to use the API provided by AWEMAN. The examples provided, in fragments of code, are sufficient to let you start accessing and manipulating the Sound Blaster AWE32. This guide will show, in step by step examples, how the AWE32 features can be programmed using the DLL provided.

This guide assumes that you are proficient in 'C' and Windows programming basics. The code present in the examples assume that the DLL is statically linked to your program.

This chapter consists of the following sections :

- Opening and closing the DLL
- Query for support
- Retrieving selections
- Issuing selections

Opening and closing

The first step in your application is to obtain the AWE device by calling the API provided in the AWEMAN.DLL. This can be done during your program's initialization in WinMain.

```
#include "windows.h"
#include "AWE_DLL.H"

/* Global variables */
extern AWEHANDLE hDeviceID;
.
.
int PASCAL WinMain (
   HANDLE   hInst,
   HANDLE   hPrevInst,
   LPSTR    lpCmdLine,
   int      nCmdShow )
{
   MSG msg;
   LRESULT lResult;
   HWND hWnd;
   WORD num;
        .
        .
   /* Perform your program's initialization and create main window */
   InitApplication(hInst);
   hWnd = CreateWindow("DEMO", "AWE Application", WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      NULL, NULL, hInst, NULL);
        .
        .

   /* Check for number of devices available in the system */
   /* System which does not support multiple card (like Win3.1) will */
   /* simply return 1 if there is an AWE hardware. */
   lResult = AWEManager(0,AWE_GET_NUM_DEVS,(LPARAM)(WORD FAR *)&num,0);
   if (lResult != AWE_NO_ERR)
   {
   /* Do error processing here! */
        .
        .
      return NULL;
   }

   /* Open the AWE Manager and retrieve a registered ID. */
   /* The 0 in last parameter refers to the first device available in */
   /* the system. The value in this parameter should never be greater */
   /* or equals to num */
   lResult = AWEManager((AWEHANDLE)0, AWE_OPEN, (LPARAM)&hDeviceID,
               0);
   if(lResult != AWE_NO_ERR)
   {
   /* Do error processing here! */
      .
      .
      return NULL;
   }
      .
      .
   /* Acquire & dispatch messages until a WM_QUIT msg is received */
   while (GetMessage(&msg,NULL,NULL,NULL))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   /* Close AWE Manager */
   AWEManager(hDeviceID, AWE_CLOSE, 0L, 0L);

   return (msg.wParam);
}
```

If the calling application is running on a system that supports multiple AWE hardware (like Windows 95), it should use the AWE_GET_NUM_DEVS API to determine the number of devices available in the system. If there is a need to selectively pick from one of the devices, the application can use the AWE_GET_DEVICE_CAPS API to retrieve capabilities of individual AWE devices.

Remember to close the AWE Manager by calling AWE_CLOSE as shown in the example above. This will allow other applications to use the manager after you have finished using it.

Querying for supports

The query class API provided by the Manager allows you to query the AWE's current available resources. It is important to query for available support before issuing them. This is because, the features provided by the Manager may increase or changes as time goes past. Generally, the options available of querying are Synthesizer Bank, Effect Types and Type Variations. In addition, the memory available on the AWE can also be queried.

The following function retrieves the available support and updates the global variables. This example can be added in your program's initialization routine.

```
#include "string.h"
#include "windows.h"
#include "AWE_DLL.H"

#define   BUF_SIZE   255

AWEHANDLE hID;
char      cSynthName[4][100]; // Buffer to store support strings
char      cEffectType[3][50];
char      cReverbList[8][50];
char      cChorusList[8][50];
char      cTrebleList[12][50];
char      cBassList[12][50];
DWORD     dwMemAvail;
DWORD     dwMemMax;

char *GetElement( char *source, int index, int size )
//
```

```
// Return the indexed element in a list of string.
// Each element terminated by '0'.
// Last element terminates with two '0'.
//
{
   char  *dest = source;
   char  *end  = source + size;
   int   i;

   for ( i=0; i < index; i++ )
   {
      // Traverse the pointer till a \0 is met.
      while ( *dest != 0 && dest < end ) dest++;
      ++dest;                        // skip the \0.
   }
   return dest;                      // return head of next string.
}

void RetrieveSupport( void )
//
// Retrieve Synth and Effect Type supported by the MIDI driver.
//
{
   char           scratch[BUF_SIZE];
   WORD           i;
   CBufferObject  buffer;
   CParamObject   param;

   //
   // SYNTHESIZER BANK
   // Get Synthesizer Emulation List
   buffer.m_Size        =  BUF_SIZE;
   buffer.m_Buffer      =  (LPSTR)&scratch[0];

   AWEManager(hID, AWE_QUERY_SYN_SUPPORT,
            (LPARAM)(LPBUFFEROBJECT)&buffer, 0L);

   // Add into the Synth buffer
   for ( i=0; i<buffer.m_Flag; i++ )
      strcpy(&cSynthName[i][0],(const char*) GetElement(&scratch[0],i,
         (int)buffer.m_SizeUsed));

   //
   // EFFECTS TYPE
   // Get Effects Type List
   // Same buffer settings as before.
   AWEManager(hID, AWE_QUERY_EFXT_SUPPORT,
           (LPARAM)(LPBUFFEROBJECT)&buffer, 0L);

   // Add into the EfxType buffer
   for ( i=0; i<buffer.m_Flag; i++ )
      strcpy(&cEffectType[i][0],(const char*) GetElement(&scratch[0],i,
         (int)buffer.m_SizeUsed));

   //
   // REVERB TYPE VARIATIONS
   // Get REVERB Type Variations List
   // Same buffer settings as before.
   param.m_SubIndex     =  REVERB;
   param.m_TypeIndex    =  REVERB_CHORUS;
   AWEManager(hID,AWE_QUERY_EFXV_SUPPORT,(LPARAM)(LPPARAMOBJECT)&param,
                   (LPARAM)(LPBUFFEROBJECT)&buffer);

   // Add into the Reverb Buffer
   for ( i=0; i<buffer.m_Flag; i++ )
      strcpy(&cReverbList[i][0], (const char*)
               GetElement(&scratch[0], i, (int) buffer.m_SizeUsed));

   //
   // CHORUS TYPE VARIATIONS
   // Get CHORUS Type Variations List
   // Same buffer settings as before.
   param.m_SubIndex     =  CHORUS;
   AWEManager(hID,AWE_QUERY_EFXV_SUPPORT,(LPARAM)(LPPARAMOBJECT)&param,
                   (LPARAM)(LPBUFFEROBJECT)&buffer);

   // Add into the Chorus Buffer
```

```
        for ( i=0; i<buffer.m_Flag; i++ )
           strcpy(&cChorusList[i][0],(const char*) GetElement(&scratch[0],i,
              (int)buffer.m_SizeUsed));

        //
        // TREBLE TYPE VARIATIONS
        // Get Treble Type Variations List
        // Same buffer settings as before.
        param.m_SubIndex     =  TREBLE;
        AWEManager(hID,AWE_QUERY_EFXV_SUPPORT,(LPARAM)(LPPARAMOBJECT)&param,
                            (LPARAM)(LPBUFFEROBJECT)&buffer);

        // Add into the Treble Buffer
        for ( i=0; i<buffer.m_Flag; i++ )
           strcpy(&cTrebleList[i][0],(const char*) GetElement(&scratch[0],i,
              (int)buffer.m_SizeUsed));

        //
        // BASS TYPE VARIATIONS
        // Get Bass Type Variations List
        // Same buffer settings as before.
        param.m_SubIndex     =  BASS;
        AWEManager(hID,AWE_QUERY_EFXV_SUPPORT,(LPARAM)(LPPARAMOBJECT)&param,
                            (LPARAM)(LPBUFFEROBJECT)&buffer);

        // Add into the Bass Buffer
        for ( i=0; i<buffer.m_Flag; i++ )
           strcpy(&cBassList[i][0],(const char*) GetElement(&scratch[0],i,
              (int)buffer.m_SizeUsed));


        // QUERY MEMORY STATUS
        // Get Memory status for both available and maximum.
        AWEManager(hID,AWE_QUERY_DRAM_SIZE,(LPARAM)(DWORD FAR*)&dwMemMax,
                            (LPARAM)(DWORD FAR*)&dwMemAvail);
    }
```

Note that there is no error handling implemented in the above functions. It is assume that the buffer size used to retrieve the strings is sufficient. If you do not wish to allocate the scratch buffer from the stack, you will have to query for the required buffer size. Using the size obtained you can then use it for dynamic allocation. To query for buffer size required, specify a buffer of 1 character. This will cause the AWE Manager to return the error AWE_ERR_RESOURCE_INSUFFICIENT. From the CBufferObject, the member m_SizeUsed will contain the number of bytes required to contain the entire list of strings.

Retrieving selections

The 'Get' class API provided by the Manager allows you to retrieve the AWE's current selection. You can retrieve the current in use selection from Effect Types and Variations. For Synthesizer Bank, User Bank and Instruments in the bank, these API will be used to retrieve their descriptors.

The following set of functions retrieve the current selection and updates the global variables. This example can be added to your program.

```
#include "string.h"
#include "windows.h"
#include "AWE_DLL.H"

#define   BUF_SIZE 255

AWEHANDLE       hID;
enum SBANK      m_CurSynthBank;       // Current synthesizer bank
enum TYPEINDEX  m_CurEfxType;         // Current effects type
enum VARIINDEX  m_CurTypeVari[6];     // Current type variations
char            cUBankDescriptor[30]; // User Bank Descriptor

void GetCurrentSelection( void )
//
```

```
// Retrieve the current Hardware Settings
//
{
   CParamObject         param;
   CBufferObject        buffer;
   char                 scratch;

   // Get current Synthesizer Bank from AWEMAN
   buffer.m_Size = sizeof(scratch);
   buffer.m_Buffer = (LPSTR)&scratch;

   AWEManager(hID,AWE_GET_SYN_BANK,(LPARAM)(LPBUFFEROBJECT)&buffer,0L);
   // should return an error, since buffer not enough
   // Ignore error, since we only want m_Flag => SBANK INDEX

   m_CurSynthBank = (enum SBANK)buffer.m_Flag;

   // Get Current Effects Type and Variations
   param.m_VariIndex[REVERB] = param.m_VariIndex[CHORUS] = 0;

   AWEManager(hID, AWE_GET_EFX_EX, (LPARAM)(LPPARAMOBJECT)&param,0L);

   m_CurEfxType = (enum TYPEINDEX) param.m_TypeIndex;
   m_CurTypeVari[0] = (enum VARIINDEX) param.m_VariIndex[0];
   m_CurTypeVari[1] = (enum VARIINDEX) param.m_VariIndex[1];
   m_CurTypeVari[2] = (enum VARIINDEX) param.m_VariIndex[2];
   m_CurTypeVari[3] = (enum VARIINDEX) param.m_VariIndex[3];
   m_CurTypeVari[4] = (enum VARIINDEX) param.m_VariIndex[4];
   m_CurTypeVari[5] = (enum VARIINDEX) param.m_VariIndex[5];
}

void GetUBankDescriptor( int nUBNum )
//
// Retrieve the User Bank's Descriptor.
//
{
   char           scratch[BUF_SIZE];
   CBufferObject  buffer;

   buffer.m_Size = BUF_SIZE;
   buffer.m_Buffer = (LPSTR)&scratch;

   AWEManager(hID, AWE_GET_USER_BANK, (LPARAM)(WORD) nUBNum,
                    (LPARAM)(LPBUFFEROBJECT) &buffer);

   if (buffer.m_SizeUsed == 1)
      strcpy((char *) &cUBankDescriptor[0],
             (const char*)"NO DESCRIPTOR\0");
   else
      strcpy((char *) &cUBankDescriptor[0], (const char*) &scratch[0]);
}

char *GetInstrDescriptor( int nBNum, int nINum, char *desc )
//
// Retrieve the Instrument's Descriptor.
//
{
   char           scratch[BUF_SIZE];
   CBufferObject  buffer;
   CParamObject   param;

   param.m_UBankIndex = nBNum;
   param.m_InstrIndex = nINum;

   buffer.m_Size = BUF_SIZE;
   buffer.m_Buffer = (LPSTR)&scratch;

   AWEManager(hID,AWE_GET_USER_INSTR,(LPARAM)(LPPARAMOBJECT)&param,
              (LPARAM)(LPBUFFEROBJECT)&buffer);

   if (buffer.m_SizeUsed == 1)
      strcpy(desc,(const char*)"NO DESCRIPTOR\0");
   else
      strcpy(desc,(const char*)&scratch[0]);

   return desc;
}
```

Note that there is no error handling implemented in the above functions. It is assume that the buffer size used to retrieve the strings is sufficient. Unlike the query class API, the size required to contain the descriptor will not be returned if the input buffer size is not large enough. This is because the descriptor store in the memory has a maximum size of 20 bytes.

Issuing selections

The select class API provided by the Manager allows you to instate selection to the AWE's features. These API allow you to configure the AWE's Effect Types, Variations and Synthesizer Bank. For User Bank, a user object module must be specified when using the API. The object module for Banks has the SBK extension. Individual instruments and banks of instruments uses the same format.

The following functions demostrate how to set the features and load a user define SBK file into the bank. This example can be added in your program.

```
#include "windows.h"
#include "AWE_DLL.H"

#define   BUF_SIZE      255
#define   MAX_USER_BANK 127
#define   MAX_INSTR     127

AWEHANDLE      hID;
enum SBANK     m_CurSynthBank;       // Current synthesizer bank
enum TYPEINDEX m_CurEfxType;         // Current effects type
enum VARIINDEX m_CurTypeVari[6];     // Current type variations
int            m_CurUserBank;        // Current user bank number
char           cUBankDescriptor[30]; // User Bank Descriptor

void SetSynth( int nSNum )
//
// Configure the hardware with current Synth.
//
{
   LRESULT lResult;

   lResult = AWEManager(hID, AWE_SELECT_SYN_BANK,
                        (LPARAM)(WORD)nSNum, 0L);

   if (lResult != AWE_NO_ERR)
   {
      // TO DO:
      // DisplayErrorMsg( lResult );
   }
}

void SetEffects( void )
//
// Configure the hardware with current effects.
//
{
   CParamObject   param;
   LRESULT        lResult;

   param.m_TypeIndex = m_CurEfxType;
   param.m_VariIndex[0] = m_CurTypeVari[0];
   param.m_VariIndex[1] = m_CurTypeVari[1];
   param.m_VariIndex[2] = m_CurTypeVari[2];
   param.m_VariIndex[3] = m_CurTypeVari[3];
   param.m_VariIndex[4] = m_CurTypeVari[4];
   param.m_VariIndex[5] = m_CurTypeVari[5];

   // TO DO: Display HOUR GLASS

   lResult = AWEManager(hID, AWE_SELECT_EFX_EX,
                        (LPARAM)(LPPARAMOBJECT) &param, 0L);

   if (lResult != AWE_NO_ERR)
   {
      // TO DO:
```

```
      // DisplayErrorMsg( lResult );
   }

   // TO DO: Restore normal arrow cursor
}

void SetUserBank( int nUBNum, char *filename, int nStrlen )
//
// Configure the User Bank.
//
{
   CBufferObject  buffer;
   LRESULT        lResult;

   // validate the range first!
   m_CurUserBank = min( MAX_USER_BANK, max( nUBNum, 1 ) );

   // TO DO: Display HOUR GLASS

   buffer.m_Size = nStrlen;
   buffer.m_Flag = OPER_FILE;
   buffer.m_Buffer = (LPSTR)filename;
   lResult = AWEManager(hID,AWE_LOAD_USER_BANK,(LPARAM)(WORD)nUBNum,
                       (LPARAM)(LPBUFFEROBJECT)&buffer);

   if (lResult != AWE_NO_ERR)
   {
      // TO DO:
      // DisplayErrorMsg( lResult );
   }
   else
   {
      // Update the global user bank descriptor buffer!
      GetUBankDescriptor( nUBNum );
   }

   // TO DO: Restore normal arrow cursor
}

void SetInstr( int nUBNum, int nINum, int nFINum, char *filename,
         int nStrlen )
//
// Download an instrument from a bank file.
// nFINum - Instrument number from the SBK file to download from.
// nINum  - Instrument number in the nUBNum bank to download to.
//
{
   char           scratch[BUF_SIZE];
   CBufferObject  buffer;
   CParamObject   param;
   LRESULT        lResult;

   // validate the range first!
   nINum= min( MAX_USER_BANK, max( nUBNum, 0 ) );
   nINum= min( MAX_INSTR, max( nINum, 0 ) );

   // TO DO: Display HOUR GLASS

   param.m_UBankIndex = nUBNum;
   param.m_InstrIndex = nINum;

   buffer.m_SizeUsed = nFINum;
   buffer.m_Size = nStrlen;
   buffer.m_Flag = OPER_FILE;
   buffer.m_Buffer = (LPSTR)filename;

   lResult = AWEManager(hID,AWE_LOAD_USER_INSTR,(LPARAM)(LPPARAMOBJECT)
                       &param,(LPARAM)(LPBUFFEROBJECT)&buffer);

   if (lResult != AWE_NO_ERR)
   {
      // TO DO:
      // DisplayErrorMsg( lResult );
   }
   else
   {
      // TO DO: Update the instrument descriptor if necessary!
   }
```

```
    // TO DO: Restore normal arrow cursor
}
```

PART IV MIDI NRPN Implementation

What Is MIDI Non-Registered-Parameter-Number?

Non-Registered Parameter Numbers are used to represent sound or performance parameters, and in the case of the EMU8000, SoundFont Parameters. NRPN can be transmitted via MIDI, as it is itself a pair of MIDI controller messages. NRPN consists of

NRPN MSB	MIDI Controller 99
NRPN LSB	MIDI Controller 98

NRPN MSB and LSB forms a value that indicates the desired sound parameter. After sending NRPN MSB and LSB messages, MIDI controllers 6 (Data Entry MSB) and 38 (Data Entry LSB) are sent to pass in the value for the sound parameter.

In general, to send a NRPN message, the following steps are required :

1. send NRPN MSB with MSB of sound parameter
2. send NRPN LSB with LSB of sound parameter
3. send Data Entry MSB with MSB of sound parameter value
4. send Data Entry LSB with LSB of sound parameter value

As NRPN and Data Entry messages are MIDI controller messages, any MIDI sequencer software that supports editing of controller message are capable of sending them.

Take note that NRPN is MIDI channel oriented, in other words, the NRPN values only affect the current instrument assigned on the MIDI channel where your NRPN values was sent.

How do I use SBAWE32 NRPN?

For SB AWE32 NRPN to be functional, NRPN MSB has to be 127, and NRPN LSB set to the desired parameter to be controlled (see the following for a list of available NRPN LSB for each parameter).

Data entry MSB with Data entry LSB together forms a 14bit number. The middle value 8192 (0x2000, Data MSB = 64 and Data LSB = 0) is taken as value 0. To convert from MSB and LSB to actual value, here is the equation:

Actual value = (MSB * 128 + LSB) - 8192

To convert a actual value into MSB and LSB, here are the steps:

MSB = (actual value + 8192) / 128

LSB = (actual value + 8192) % 128

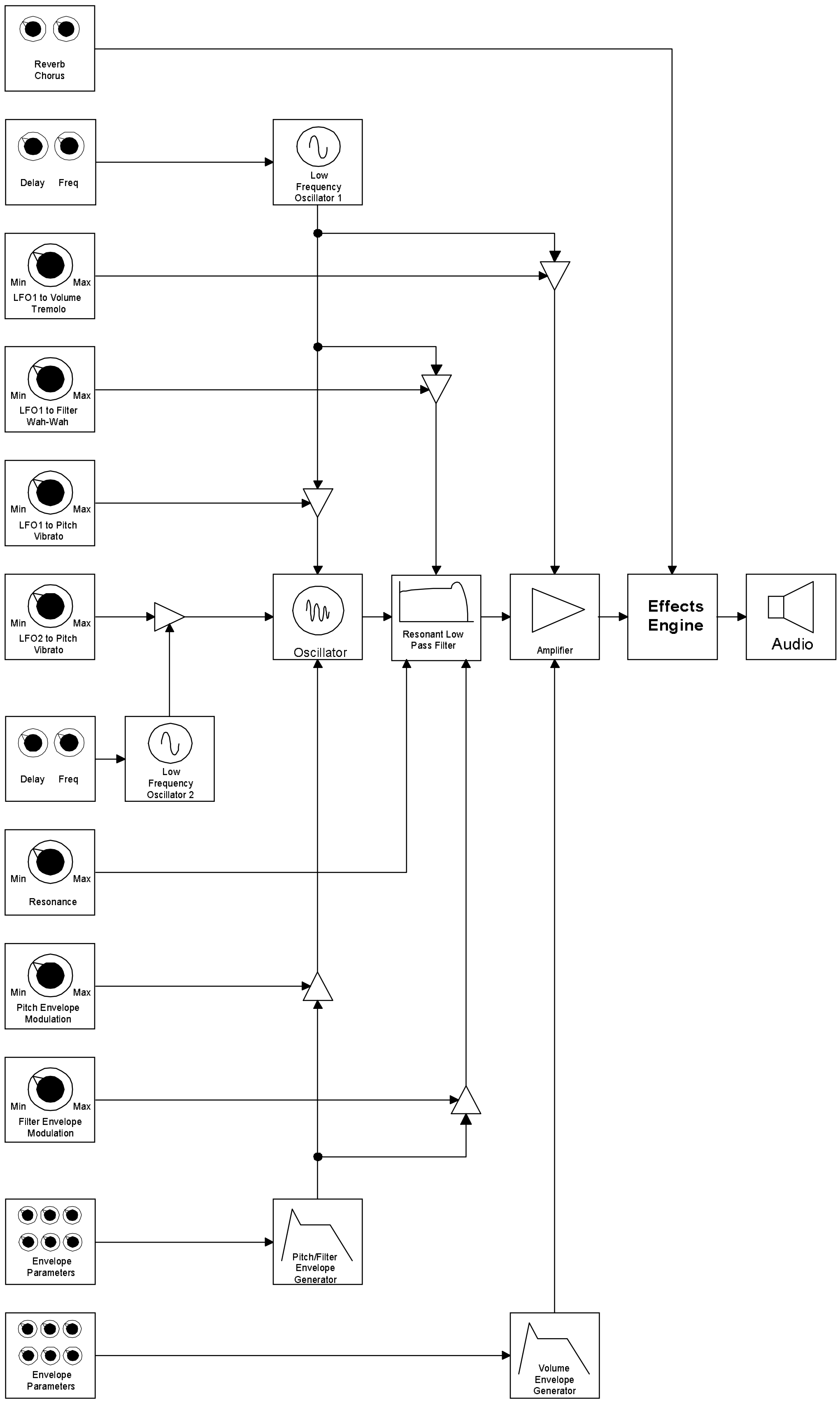
A "Reset All Controllers" message (MIDI controller 121) will restore the instrument's original SoundFont parameters.

The EMU8000 Sound Architecture

The EMU8000 has an extensive modulation implementation using two sine-wave LFO's (Low Frequency Oscillator) and two multi-stage envelope generators.

Modulation means to dynamically change a parameter of an audio signal, whether it be the volume (amplitude modulation, or tremolo), pitch (frequency modulation, or vibrato) or filter cutoff frequency (filter modulation, or wah-wah). To modulate something we would require a modulation source, and a modulation destination. In the EMU8000, the modulation sources are the LFOs and the envelope generators, and the modulation destinations can be the pitch, the volume or and filter cutoff frequency.

The EMU8000's LFO's and envelope generators provides a complex modulation environment. Each sound producing element of the EMU8000 consists of a resonant low-pass filter, two LFOs, in which one modulates the pitch (LFO2), and the other modulates pitch, filter cutoff and volume (LFO1) simultaneously. There are two envelope generators; envelope 1 contours both pitch and filter cutoff simultaneously, and envelope 2 contours volume. The output stage consists of an effects engine which mixes the dry signals with the Reverb/chorus level signals to produce the final mix. The diagram on the next page shows the typical blocks of an EMU8000 sound element.


Reverb
Chorus
Delay
Freq
Low Frequency Oscillator 1
Min
Max
LFO1 to Volume Tremolo
LFO1 to Filter Wah-Wah
LFO1 to Pitch Vibrato
LFO2 to Pitch Vibrato
Oscillator
Resonant Low Pass Filter
Amplifier
Effects Engine
Audio
Low Frequency Oscillator 2
Resonance
Pitch Envelope Modulation
Filter Envelope Modulation
Envelope Parameters
Pitch/Filter Envelope Generator
Envelope Parameters
Volume Envelope Generator

EMU8000 Sound Elements

The blocks within an EMU8000 sound element can be programmed to produce a variety of sound effects.

Oscillator

An oscillator is the source of an audio signal.

Low Pass Filter

The low pass filter is responsible for modifying the timbres of an instrument. The low pass filter's filter cutoff values can be varied from 100 Hz to 8000 Hz. By changing the values of the filter cutoff, a myriad of analogue sounding filter sweeps can be achieved. An example of a GM instrument that makes use of filter sweep is instrument number 87, Lead 7 (fifths).

Amplifier

The amplifier determines the loudness of an audio signal.

LFO1

An LFO, or Low Frequency Oscillator, is normally used to periodically modulate, i.e., dynamically change a sound parameter, whether it be volume (amplitude modulation), pitch (frequency modulation) or filter cutoff (filter modulation). It operates at sub-audio frequency from 0.042 Hz to 10.71 Hz. The LFO1 in the EMU8000 modulates the pitch, volume and filter cutoff simultaneously.

LFO2

The LFO2 is similar to the LFO1, except that it modulates only the pitch of the audio signal only.

Filter Resonance

A filter alone would be like an equaliser, making a bright audio signal duller, but the addition of resonance greatly increases the creative potential of a filter. Increasing the resonance of a filter makes it emphasis signal at the cutoff frequency, giving the audio signal a subtle "wah-wah", i.e., imagine a siren sound going from bright to dull and bright again periodically.

LFO1 to Volume (Tremolo)

As indicated in figure 1, LFO1's output is routed to the amplifier, with the depth of oscillation determined by LFO1 to Volume. LFO1 to Volume produces tremolo, which is a periodic fluctuation of volume. Lets say you are listening to a piece of music on your home stereo system. When you rapidly increase and decrease the playback volume, you are creating tremolo effect, and the speed in which you increases and decreases the volume is the tremolo rate (which corresponds to the speed at which the LFO is oscillating at) . An example of a GM instrument that makes use of LFO1 to Volume is instrument number 45, Tremolo Strings.

LFO1 to Filter Cutoff (Wah-Wah)

As indicated in figure 1, LFO1's output is routed to the filter, with the depth of oscillation determined by LFO1 to Filter. LFO1 to Filter produces a periodic fluctuation in the filter cutoff frequency, producing an effect very similar to that of a wah-wah guitar (see resonance for a description of "wah-wah"). An example of a GM instrument that makes use of LFO1 to Filter Cutoff is instrument number 19, Rock Organ.

LFO1 to Pitch (Vibrato)

As indicated in figure 1, LFO1's output is routed to the oscillator, with the depth of oscillation determined by LFO1 to Pitch. LFO1 to Pitch produces a periodic fluctuation in the pitch of the oscillator, producing a vibrato effect. An example of a GM instrument that makes use of LFO1 to Pitch is instrument number 57, Trumpet.

LFO2 to Pitch (Vibrato)

The LFO1 in the EMU8000 can simultaneously modulates pitch, volume and filter. LFO2, on the other hand, modulates only the pitch, with the depth of modulation determined by LFO2 to Pitch. LFO2 to Pitch produces a periodic fluctuation in the pitch of the oscillator, producing a vibrato effect. When this is couple with LFO1 to Pitch, a complex vibrato effect can be achieved.

Volume Envelope

The character of a musical instrument is largely determined by it's volume envelope, the way in which the level of the sound changes with time. For example, percussive sounds usually starts suddenly and then die away, whereas a bowed sound might take some time to start and then sustain at a more or less fixed level.

A six-stage envelope made up the volume envelope of the EMU8000. The six stages are delay, attack, hold, decay, sustain and release. The stages can be described as follows :

Delay	The time between when a key is played and when the attack phase begins
Attack	The time it takes to go from zero to the peak (full) level.
Hold	The time the envelope will stay at the peak level before starting the decay phase.
Decay	The time it takes the envelope to go from the peak level to the sustain level.
Sustain	The level at which the envelope remains as long as a key is held down.
Release	The time it takes the envelope to fall to the zero level after the key is released.

Using these six parameters can yield very realistic reproduction of the volume envelope characteristics of many musical instruments.

Pitch and Filter Envelope

The pitch and filter envelope is similar to the volume envelope in that it has the same envelope stages. The difference between them is that whereas the volume envelope contours the volume of the instrument over time, the pitch and filter envelope contours the pitch and filter values of the instrument over time. The pitch envelope is particularly useful in putting the finishing touches in simulating a natural instrument. For example, some wind instruments tends to go slight sharp when they are first blown, and this characteristics can be simulated by setting up a pitch envelope with a fairly fast attack and decay. The filter envelope, on the other hand, is useful in creating synthetic sci-fi sound textures. An example of a GM instrument that makes use of the filter envelope is instrument number 86, Pad 8 (Sweep).

Pitch/Filter Envelope Modulation

These two parameters determine that modulation depth of the pitch and filter envelope. In the wind instrument example above, a small amount of pitch envelope modulation is desirable to simulate it's natural pitch characteristics.

SB AWE32 MIDI NRPN List

Note: "Realtime" means that the parameter can also affect a sustaining note. For example, filter sweep on a sustaining sound can be achieved by sending continuous NRPN LSB 21 (initial filter cutoff).

NRPN LSB 0 (Delay before LFO1 starts)

Realtime	: No
Range	: [0, 5900]
Unit	: 4 milliseconds

LFO1 Delay from 0 to 22 seconds.

NRPN LSB 1 (LFO1 Frequency)

Realtime	: Yes
Range	: [0, 127]
Unit	: 0.084Hz

LFO1 frequency from 0Hz to 10.72 Hz.

NRPN LSB 2 (Delay before LFO2 starts)

Realtime	: No
Range	: [0, 5900]
Unit	: 4 milliseconds

LFO2 Delay from 0 to 22 seconds.

NRPN LSB 3 (LFO2 Frequency)

Realtime	: Yes
Range	: [0, 127]
Unit	: 0.084Hz

LFO2 frequency from 0Hz to 10.72 Hz.

NRPN LSB 4 (Envelope 1 delay time)

Realtime	: No
Range	: [0, 5900]
Unit	: 4 milliseconds

Envelope 1 Delay from 0 to 22 seconds.

NRPN LSB 5 (Envelope 1 attack time)

Realtime	: No
Range	: [0, 5940]
Unit	: Milliseconds

Envelope 1 attack time from 0 to 5.9 seconds.

NRPN LSB 6 (Envelope 1 hold time)

Realtime	: No
Range	: [0, 8191]
Unit	: Milliseconds

Envelope 1 hold time from 0 to 8 seconds.

NRPN LSB 7 (Envelope 1 decay time)

Realtime	: No
Range	: [0, 5940]
Unit	: 4 Milliseconds

Envelope 1 decay time from 0.023 to 23.7 seconds.

NRPN LSB 8 (Envelope 1 sustain level)

Realtime	: No
Range	: [0, 127]
Unit	: 0.75dB

Envelope 1 sustain level from full level down to off (0.75 dB step).

NRPN LSB 9 (Envelope 1 release time)

Realtime	: No
Range	: [0, 5940]
Unit	: 4 milliseconds

Envelope 1 release time from 0.023 to 23.7 seconds.

NRPN LSB 10 (Envelope 2 delay time)

Realtime	: No
Range	: [0, 5900]
Unit	: 4 milliseconds

Envelope 2 Delay from 0 to 22 seconds.

NRPN LSB 11 (Envelope 2 attack time)

Realtime	: No
Range	: [0, 5940]
Unit	: Milliseconds

Envelope 2 attack time from 0 to 5.9 seconds.

NRPN LSB 12 (Envelope 2 hold time)

Realtime	: No
Range	: [0, 8191]
Unit	: Millisecond

Envelope 2 hold time from 0 to 8 seconds.

NRPN LSB 13 (Envelope 2 decay time)

Realtime	: No
Range	: [0, 5940]
Unit	: 4 milliseconds

Envelope 2 decay time from 0.023 to 23.7 seconds.

NRPN LSB 14 (Envelope 2 sustain level)

Realtime	: No
Range	: [0, 127]
Unit	: 0.75dB

Envelope 2 sustain level from full level down to off.

NRPN LSB 15 (Envelope 2 release time)

Realtime	: No
Range	: [0, 5940]
Unit	: 4 milliseconds

Envelope 2 release time from 0.023 to 23.7 seconds.

NRPN LSB 16 (Initial Pitch)

Realtime	: Yes
Range	: [-8192, 8191]
Unit	: cents

Pitch tuning between -8192 and 8191 cents.

NRPN LSB 17 (LFO1 to Pitch)

Realtime	: Yes
Range	: [-127, 127]
Unit	: 9.375 cents

If data value is greater than 0, this will cause a positive (from 0 to maximum) of 1 octave shift at LFO peak. On the other hand, if data value is smaller than 0, this will cause a negative (from o to minimum) of 1 octave shift at LFO peak.

NRPN LSB 18 (LFO2 to Pitch)

Realtime	: Yes
Range	: [-127, 127]
Unit	: 9.375 cents

If data value is greater than 0, this will cause a positive (from 0 to maximum) of 1 octave shift at LFO peak. On the other hand, if data value is smaller than 0, this will cause a negative (from o to minimum) of 1 octave shift at LFO peak.

NRPN LSB 19 (Envelope 1 to Pitch)

Realtime	: No
Range	: [-127, 127]
Unit	: 9.375 cents

If data value is greater than 0, this will cause a positive (from 0 to maximum) of 1 octave shift at envelope peak. On the other hand, if data value is smaller than 0, this will cause a negative (from 0 to minimum) of 1 octave shift at envelope peak.

NRPN LSB 20 (LFO1 to Volume)

Realtime	: Yes
Range	: [0, 127]
Unit	: 0.1875 dB

Data value smaller than 64 causes a positive phase (from 0 to maximum) volume modulation via LFO1 with magnitude of 12 dB at LFO peak. On the other hand, data value greater than or equals to 64 causes a negative phase (from 0 to minimum) volume modulation via LFO1 with magnitude of 12 dB at LFO peak.

NRPN LSB 21 (Initial Filter Cutoff)

Realtime	: Yes
Range	: [0, 127]
Unit	: 62Hz

Filter cutoff from 100Hz to 8000Hz.

NRPN LSB 22 (Initial Filter Resonance Coefficient)

Realtime	: No
Range	: [0, 127]

The EMU8000 had a built in resonance coefficient table comprising of 16 entries. Values 0-7 will select the first (0) entry, values 8-15 selects the second (1) entry and so on.

Coeff	Low Fc(Hz)	Low Q(dB)	High Fc(kHz)	High Q(dB)	DC Attn(dB)
0	92	5	Flat	Flat	-0.0
1	93	6	8.5	0.5	-0.5
2	94	8	8.3	1	-1.2
3	95	10	8.2	2	-1.8
4	96	11	8.1	3	-2.5
5	97	13	8.0	4	-3.3
6	98	14	7.9	5	-4.1
7	99	16	7.8	6	-5.5
8	100	17	7.7	7	-6.0
9	100	19	7.5	9	-6.6
10	100	20	7.4	10	-7.2
11	100	22	7.3	11	-7.9
12	100	23	7.2	13	-8.5
13	100	25	7.1	15	-9.3
14	100	26	7.1	16	-10.1
15	100	28	7.0	18	-11.0

NRPN LSB 23 (LFO1 to Filter Cutoff)

Realtime	: Yes
Range	: [-64, 63]
Unit	: 56.25 cents

Positive data value causes a positive phase (from 0 to maximum) filter modulation via LFO1 with magnitude of 3 octave at LFO peak. On the other hand, negative data value causes a negative phase (from 0 to minimum) filter modulation via LFO1 with magnitude of 3 octave at LFO peak.

NRPN LSB 24 (Envelope 1 to Filter Cutoff)

Realtime	: No
Range	: [-127, 127]
Unit	: 56.25 cents

Data value greater than 0 causes a positive phase (from 0 to maximum) filter modulation via Envelope1 with magnitude of 6 octave at envelope peak. On the other hand, value smaller than 0 causes a negative phase (from 0 to minimum) filter modulation via Envelope 1 with magnitude of 6 octave at envelope peak.

NRPN LSB 25 (Chorus Effects Send)

Realtime	: No
Range	: [0, 255]

Chorus send, with 0 being the driest (no chorus effects processing), and 255 being the wettest (full chorus effect processing).

NRPN LSB 26 (Reverb Effects Send)

Realtime	: No
Range	: [0, 255]

Reverb send, with 0 being the driest (no reverb effects processing), and 255 being the wettest (full reverb effect processing).

PART V Audio Spatialization API

Overview

The Audio Spatialization Library (ASL) provides programmers a low-level access to audio spatialization algorithms implemented on the SB AWE32. The library provides programmers the ability to create and move in 3D space basic audio spatialization objects such as sound emitters and receivers. The library attempts to control the apparent location of sound emitters relative to a receiver by modeling a small set of physical and psycho-acoustic phenomena.

Spatial Audio Overview

The technology of spatial audio breaks down into two basic categories, depending on the techniques employed to produce the spatial experience.

Passive Stereo Enhancement

Passive Stereo Enhancement systems are, as the name implies, methods for improving the three dimensional nature of an already captured sound image, without any user control of apparent sound position. A Stereo Enhancement system is any system which takes as its sole realtime input a two channel stereo audio signal, and outputs a (hopefully improved) two channel stereo audio signal. The systems may, of course, have controls which determine the degree of enhancement. These systems have no possibility for real time positioning and controlled movement of sound sources, since they can only accept a signal that has already had all of its components mixed together.

Parameteric 3D Audio

In Parametric Spatialization, one or more monophonic channels are reproduced according to realtime dimensional parameters specifying the position of objects in the soundfield. Unlike other techniques, Parametric Spatialization determines the stereo audio image synthetically, based on realtime input. Parametric Spatialization is thus interactive and user controllable.

Which 3D Audio Cues are most robust?

People can localize sounds in 3D space well, but in fact, a variety of consistent cues (or clues) are required by our perception mechanism to determine where a sounds is coming from in three dimensional space. Suprisingly, one of the most important perceptual mechanisms that we use to determine a sound's location is head movement! Why this is true tells us a lot about what we can honestly expect from synthetic spatial audio systems.

By far, the most robust cue for perceiving spatial location is "lateralization", or the ability to determine whether a sound is on our left or right side. Even people who are deaf in one ear are fairly good at lateralization.

The next best cue is making the determination whether a sound is on the median plane or not. Substantial research shows that we are also good at this distinction.

Beyond these two factors, people are actually quite weak at determining spatial location, without head movement. This may seem quite suprising, but research subjects who have their head immobilized in research test conditions consistently identify sounds which are actually coming from in front of them as coming from behind them. Some speculate that this so called front-back confusion may have an evolutionary advantage, since unseen sounds might be predators! Elevation cues are also quite fragile.

Finally, another of the most important cues for a sound's location is vision! If we see an object that we believe to be making a sound, then we think the sound is coming from that object. This phenomenon has been exploited for years by ventriloquists. This phenomenon can be exploited just as well by games developers in that on-screen visual cues can also help establish the apparent location of a sound source.

In contrast, an off-screen sound source can help to direct the viewer's attention to the (unseen) apparent source of that sound. The most recent generation of games adopts a first person perspective which is ideally suited for spatial audio, in that the user can direct the view of the game in different directions. If robust cues are employed to attract the viewer's attention, the viewer can be cued to direct the view towards the unseen sound. The strategy of directing the view from audio cues will only be as strong as the audio cues, however.

Head Tracking

If a 3D audio spatialization system is producing audio for headphones and the listener is using a head tracking system, so that the position and orientation of the listener's head is known in read time, the 3D audio system can provide the correct left-right cues to simulate the actual behavior that physical sounds have in physical space. Head tracking is widely known to be able to significantly enhance spatial audio cues dramatically.

People are not likely to start wearing head trackers and headphones in large numbers any time soon. For one thing, head tracking hardware is quite expensive. Secondly, it is cumbersome and isolates a person from the room environment.

Instead, people mostly listen to multimedia audio over speakers. Speaker playback of spatial audio is still quite practical, but there will always be limitations. The developer should not expect robust front-back and elevation cues without head tracking headphone audio displays.

However, distance cues and lateral cues should be very robust. Interestingly, distance cues are quite practical even with only one speaker!

AWE-32 Implementation of 3D Audio Cues

The AWE-32 implementation of Parametric 3D Audio provides strong lateralization and distance cues. The lateralization cues have been found to be quite robust either on speakers or on headphones. They do not seem to suffer a lot from a small "sweet spot" either. The sweet spot is the region midway between two stereo loudspeakers where the stereo effect is strongest. While the optimum AWE-32 spatial effect is still in the center between loudspeakers, listeners can be almost anywhere between a pair of loudspeakers and the lateralization effect will still be quite clear. Many existing 3D audio techniques have a very sensitive sweet spot, and are severly degraded by listening off-center.

Distance cues will work even on a single speaker.

Dynamic Versus Static Cues

Dynamic spatial cues are far more robust in both the real world and in synthetic spatialization. If a sound emitter (or a receiver) is moving, then the change in spatial location will create a much stronger impression than a sound that is fixed, relative to a receiver.

This phenomenon has been well-documented in the research literature on all spatial perception. We evolved to sense change, and static perceptual cues are usually unimportant, and hence tend to be ignored.

What sounds are best suited for spatialization?

Sound sources that have a broad spectrum and that have significant time domain amplitude envelope features are best suited for parametric 3D audio. Smooth steady state tones will localize very poorly. For example, with the new electronic telephones, it is often hard to figure out whose office their rings are coming from. The old fashioned bell ringers have sharp transients and broad spectral features, making then much easier to localize.

Typical sounds that will localize well include explosions, airplane and helicopter sounds, cracks, snaps and other game style sound effects.

Typical sounds that will localize poorly include pure steady tones and simple waveforms.

SB AWE32 Audio Spatialization Library

The SB AWE32 implementation of the Audio Spatialization Library imposes some limits because of its tight integration with the MIDI engine.

- The sound samples (wave files) have to be preloaded in a form of SoundFont

  It is recommended that users create SoundFonts using our Vienna SoundFont editor so that loop points of sound samples are marked properly. The loop length must be a least 256 samples. Note that the envelope and LFO parameters set in the SoundFonts are ignored. Once a SoundFont is loaded, multiple emitters can be created out of a sound sample in the SoundFont using **c3daSetEmitterMIDISource**.

- Multi-layering in the SoundFont is not supported

  Note that if multi-layering is present in a SoundFont, only the sound sample in the first layer is used.

- Only a maximum of MAX_EMITTER number of emitters can be created

  Emitters share the polyphonies with the MIDI engine. As more emitters are created, more polyphonies are taken away from the MIDI engine. When the MIDI engine has not enough polyphonies, note stealing will occur. This means that old notes will be turned off to make room for new notes. If you plan to play a dense piece of music, try to release some of the emitters first by calling **c3daDestroyEmitter**.

- Set emitter position update frequency to about 20Hz

  Update the emitter positions 20 times per second. At this rate, the library could provide the optimum spatialization effects.

Types and Structures

The types and structures used by the Audio Spatialization Library are:

- **c3daEmitter** - an emitter object.
- **c3daReceiver** - a receiver object.
- **c3daError** - the return type from all ASL functions. Possible values are c3daSUCCESS and c3daFAILURE.
- **c3daSoundState** - specifies a sound output or "play" state. Possible values are c3daSTART, c3daSTOP, and c3daPAUSE.
- The (x, y, z) coordinates are of type **int**. They must be in the range of between -32767 to 32767.

System Functions

This group of API consists of the following :

- c3daInit
- c3daEnd
- c3daSetDopplerEffect
- c3daSetMaxDistance

c3daInit

```
c3daError
PASCAL
c3daInit(VOID)
```

Actions Initialize the Audio Spatialization library and must be called before any other calls to the library.

Parameters None.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks A single receiver (with an orientation where the head is facing along the positive x-axis and the left ear is on the positive y-axis) is also automatically created and used as the active receiver. Use **c3daGetActiveReceiver** to retrieve it.

c3daEnd

```
c3daError
PASCAL
c3daEnd(VOID)
```

Actions Shutsdown the Audio Spatialization Library and releases all of its resources. **c3daEnd** must be called before an application exits.

Parameters None.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks Any calls to the library after a call to **c3daEnd** will result in strange and unusual behavoir (except for **c3daInit** to reinitialize the library).

c3daSetDopplerEffect

```
c3daError
PASCAL
c3daSetDopplerEffect(UINT uDoppEffect)
```

Actions Set the the doppler effect for all emitters. By using this function, the user can control the "magnitude" of the doppler effects that may be heard as a result of emitter position changes.

Parameters *uDoppEffect*

The valid range for *uDoppEffect* is 0-255 where a value of 0 turns off doppler effects completely and a value of 255 turns on the full effects of doppler. The default value is 127.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetMaxDistance

```
c3daError
PASCAL
c3daSetMaxDistance(int MaxDistance)
```

Actions Set the maximum distance that emitter volumes will attenuate across. At maximum distance, attenuation is at maximum (-96dB). By using this maximum distance, the user can adjust the distance attenuation that correspond to modeled world.

Parameters *MaxDistance*

The valid range for *MaxDistance* is 1 - 32767.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Emitter Functions

This group of API consists of the following :

- c3daCreateEmitter
- c3daDestroyEmitter
- c3daSetEmitterPosition

- c3daSetEmitterOrientation
- c3daSetEmitterSoundState
- c3daSetEmitterGain
- c3daSetEmitterPitchInc
- c3daSetEmitterMIDISource

c3daCreateEmitter

```
c3daError
PASCAL
c3daCreateEmitter(
      c3daEmitter FAR*  lpEm,
      int               x,
      int               y,
      int               z
)
```

Actions Initializes an emitter object and places an omni-directional sound emitter object in 3D space.

Parameters *lpEm*

Specify a pointer to an emitter object.

x, y, z

Specify the initial coordinates of the emitter. These coordinates are relative to the active receiver.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks By default an emitter is created in a "stopped" state. Use the **c3daSetEmitterSoundState** to change an emitter's sound output state.

c3daDestroyEmitter

```
c3daError
PASCAL
c3daDestroyEmitter(c3daEmitter FAR* lpEm)
```

Actions Destroys an emitter object. All resources associated with the emitter are released.

Parameters *lpEm*

Specify a far pointer to an emitter object.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetEmitterPosition

```
c3daError
PASCAL
c3daSetEmitterPosition(
      c3daEmitter FAR*  lpEm,
      int               x,
      int               y,
      int               z
)
```

Actions Place an emitter object at a position in 3D space relative to the active receiver.

Parameters *lpEm*

Specify a far pointer to an emitter object.

x, y, z

Specify the new coordinates of the emitter relative to the active receiver (0,1,0).

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks Calling this function could potentially alter the sound output due to the effects of the audio spatialization algorithms. Note that all receivers have a fixed orientation where the head is facing along the positive x-axis and the left ear is along the positive y-axis. To implement a receiver facing the positive y-axis, following transformation is needed before calling **c3daSetEmitterPosition**. Please refer to section **Implementing Receiver Orientation** for more information.

```
int new_x = y;
int new_y = -x;
int new_z = z;
c3daSetEmitterPosition(&EmitterObj, new_x, new_y, new_z);
```

c3daSetEmitterOrientation

```
c3daError
PASCAL
c3daSetEmitterOrientation(
      c3daEmitter FAR*  lpEm,
      int               x,
      int               y,
      int               z
)
```

Actions Set the orientation of an emitter.

Parameters *lpEm*

Specify a far pointer to an emitter object.

x, y, z

Specify the orientation vector.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks At this time this function has no effect since only omni-directional emitters can be created.

c3daSetEmitterSoundState

```
c3daError
PASCAL
c3daSetEmitterSoundState(
      c3daEmitter FAR*  lpEm,
      c3daSoundState    state
)
```

Actions Change the sound output state of an emitte.

Parameters *lpEm*

Specify a far pointer to an emitter object.

state

Possible values are: `c3daSTART`, `c3daSTOP`, and `c3daPAUSE`.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetEmitterGain

```
c3daError
PASCAL
c3daSetEmitterGain(c3daEmitter FAR* lpEm, UINT uGain)
```

Actions Change the output gain of an emitter.

Parameters *lpEm*

Specify a far pointer to an emitter object.

uGain

The valid range for *uGain* is 0-255 where 0 causes the emitter to be muted (but still running) and 255 represents unity gain (the default).

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetEmitterPitchInc

```
c3daError
PASCAL
c3daSetEmitterPitchInc(c3daEmitter FAR* lpEm, int inc)
```

Actions Increment or decrement the pitch of an emitter.

Parameters *lpEm*

Specify a far pointer to an emitter object.

inc

The increment or decrement in number of cents. A positive value increases the current pitch and a negative value decreases the current pitch. The range of permissible values depends on the sample sampling rate and current doppler shift. Typically it has a max of about 2000 cents for a 44.1kHz sample.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetEmitterMIDISource

```
c3daError
PASCAL
c3daSetEmitterMIDISource(
      c3daEmitter FAR*  lpEm,
      UINT              uBank,
      UINT              uProgram,
      UINT              uKeynum
)
```

Actions Associate a MIDI source with an emitter.

Parameters *lpEm*

Specify a far pointer to an emitter object.

uBank, uProgram, uKeynum

Specify the MIDI source.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Receiver Functions

This group of API consists of the following :

- c3daCreateReceiver
- c3daDestroyReceiver
- c3daSetActiveReceiver
- c3daGetActiveReceiver
- c3daSetReceiverPosition

c3daCreateReceiver

```
c3daError
PASCAL
c3daCreateReceiver(
      c3daReceiver FAR* lpRx,
      int               x,
      int               y,
      int               z
)
```

Actions Initialize a receiver object and places it in 3D space.

Parameters *lpRx*

Specify a far pointer to a receiver object.

x, y, z

Specify the initial coordinates of the receiver.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks The position of a receiver is used by the audio spatialization algorithms if/when this receiver becomes the active receiver. The orientation of receivers is fixed to be facing along the positive x-axis with the left ear along the positive y-axis.

c3daDestroyReceiver

```
c3daError
PASCAL
c3daDestroyReceiver(c3daReceiver FAR* lpRx)
```

Actions Destroy a receiver object. All resources associated with the receiver are released.

Parameters *lpRx*

Specify a far pointer to a receiver object.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks If `lpRx` is the active receiver, then destruction of this receiver will cause all sound output to cease until a new active receiver is specified.

c3daSetActiveReceiver

```
c3daError
PASCAL
c3daSetActiveReceiver(c3daReceiver FAR* lpRx)
```

Actions Set the current active receiver. The active receiver is the receiver for which audio spatialization algorithms are applied.

Parameters *lpRx*

Specify a far pointer to a receiver object.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks On the AWE, multiple receivers can be created; however, there can only be one active receiver at a time.

c3daGetActiveReceiver

```
c3daError
PASCAL
c3daGetActiveReceiver(c3daReceiver FAR* FAR* lpRx)
```

Actions Return the current active receiver.

Parameters *lpRx*

Specify a far pointer to a far pointer to a receiver object.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

c3daSetReceiverPosition

```
c3daError
PASCAL
c3daSetReceiverPosition(
      c3daReceiver FAR* lpRx,
      int               x,
      int               y,
      int               z
)
```

Actions Place a receiver in 3D space. Correspondingly, the positions of all existing emitters are updated internally.

Parameters *lpRx*

Specify a far pointer to a receiver object.

x, y, z

Specify the new coordinates of the receiver.

Return Return `c3daSUCCESS` upon success and `c3daFAILURE` otherwise.

Remarks If `lpRx` is the active receiver, then movement of this receiver may cause a perceivable change in sound due to the effects of the audio spatialization algorithms.

Programming Example

This chapter gives a simple example of using the Audio Spatialization API and position the emitter using a mouse pointer. Please refer to the chapter **SoundFont Bank And Downloadable DRAM**

Services and **Real and Protected Mode API Programming Guide** for explanation on using the MIDI and SoundFont libraries.

```
#include <dos.h>
#include <stdio.h>
#include "ctaweapi.h"

volatile int count = 0;
void (interrupt far* prev_intr)();    /* previous interrupt handler */
```

newintr is a interrupt handler that hooks on to interrupt 1CH to get a 18.2Hz timer interrupt. The variable **count** is incremented and serve as an indicator to the position update loop to update the emitter position.

```
void interrupt far newintr()
{
    ++count;
    _chain_intr(prev_intr);   /* chain to previous interrupt */
}
```

Start of program.

```
main()
{
    int current;
    int x, y;
    FILE *fp;
    long bsize[2];
    SOUND_PACKET sp;
    WAVE_PACKET wp;
    c3daEmitter em;
    union REGS regs;
    char preset[256];
    char packet[PACKETSIZE];
```

For simplicity, assume that EMU8000 is at base address 0x620. Other method would be to look at the BLASTER environment 'E' setting.

```
    if (awe32Detect(0x620)) {
        printf("Cannot detect SB AWE32\n");
        return -1;
    }
    awe32InitHardware();
    if (awe32DramSize == 0) {
        printf("Not enough DRAM\n");
        return -1;
    }

    awe32SoundPad.SPad1 = awe32SPad1Obj;      /* install GM presets */
    awe32SoundPad.SPad2 = awe32SPad2Obj;
    awe32SoundPad.SPad3 = awe32SPad3Obj;
    awe32SoundPad.SPad4 = awe32SPad4Obj;
    awe32SoundPad.SPad5 = awe32SPad5Obj;
    awe32SoundPad.SPad6 = awe32SPad6Obj;
    awe32SoundPad.SPad7 = awe32SPad7Obj;
    awe32InitMIDI();
```

Detect and initialize the mouse driver. This program won't run without a mouse.

```
    regs.x.ax = 0;
    int86(0x33, &regs, &regs);
    if (regs.x.ax == 0) {
        printf("Cannot initialize mouse\n");
        return -1;
    }
    regs.x.ax = 0x1;
    int86(0x33, &regs, &regs);
```

Loading a sound sample (sound.raw) as SoundFont into MIDI bank 1, instrument 0.

```
    sp.total_banks = 2;
    bsize[0] = 0;
    bsize[1] = awe32DramSize * 2;     /* use all available DRAM */
    sp.banksizes = bsize;
    awe32DefineBankSizes(&sp);
    fp = fopen("sound.raw", "rb");
    if (!fp) {
        printf("Cannot open \"sound.raw\"\n");
        return -1;
    }

    wp.tag = 0x101;
    wp.bank_no = 1;
    wp.sample_size = 49429;          /* hard-coded info about sound.raw */
    wp.samples_per_sec = 44100;
    wp.bits_per_sample = 16;
    wp.no_channels = 1;
    wp.looping = 1;
    wp.startloop = 0;
    wp.endloop = 49425;
    wp.release = 0;
    if (awe32WPLoadRequest(&wp)) {
        printf("awe32WPLoadRequest failed\n");
        return -1;
    }

    wp.data = packet;
    while (wp.no_wave_packets--) {
        fread(packet, 1, PACKETSIZE, fp);
        awe32WPStreamWave(&wp);
    }
    fclose(fp);

    wp.presets = preset;             /* SoundFont preset space */
    awe32WPBuildSFont(&wp);
```

Initialize the Audio Spatialization Library and creative an emitter base on the loaded sound sample.

```
    c3daInit();
    c3daCreateEmitter(&em, 1, 0, 60);
    c3daSetEmitterMIDISource(&em, 1, 0, 60);
    c3daSetEmitterSoundState(&em, c3daSTART);
```

Save current interrupt vector and hook on to interrupt 1CH.

```
    prev_intr = _dos_getvect(0x1c);
   _dos_setvect(0x1c, newintr);
```

The position update loop. Loop forever until a key is hit. Once count changes its value, update the emitter position.

```
    current = count;
    while (!kbhit()) {
        while (current == count) ;   /* wait til count changes */
        current = count;
        regs.x.ax = 3;
        int86(0x33, &regs, &regs);   /* get mouse position */
        x = regs.x.cx - 320;
        y = regs.x.dx - 96;
        /* rotate -90 degree along z-axis */
        c3daSetEmitterPosition(&em, -y/2, -x/2, 0);
    }
    getch();
```

Terminates.

```
    _dos_setvect(0x1c, prev_intr);
    regs.x.ax = 0x21;
    int86(0x33, &regs, &regs);
    awe32Terminate();

    return 0;
}
```

Implementing Receiver Orientation

For efficiency reasons, the Audio Spatialization Library does not provide the ability to change a receiver's orienation - all receivers have a fixed orientation where the head is facing along the positive x-axis and the left ear is along the positive-y-axis. This chapter gives a simple example of how to implement receiver orientation functionality.

The basic idea is that changing a receiver's orientation is equivalent to appropriately moving all the emitters around a receiver with a fixed orientation. This example uses yaw, pitch, and roll to transform a recevier's orientation from the default fixed orientation to a new one. To equivalently move the emitters, the "inverse" transform is applied to all the emitters.

```
/*
 * This example is sample code to handle receiver orientation since
 * the basic c3da core does not allow a change in receiver orienation.
 * There are two routines defined in this example:
 *
 *      setEmitterPosition - a replacement for c3daSetEmitterPosition
 *      setReceiverOrientation - a new routine to handle receiver orientation
 */

#include <math.h>

#include "ctaweapi.h"

#define NUMBER_OF_EMITTERS      4
#define DEG2RAD                 0.017453f;

/*
 * example utility structs
 */

typedef struct _Emitter {
    int           x;          /* world x position */
    int           y;          /* world y position */
    int           z;          /* world z position */
    c3daEmitter   em;         /* handle to c3da core emitter */
                              /* need to be "created" using  */
                              /* c3daCreateEmitter           */
} Emitter;

typedef struct _Receiver {
    float         a[9];       /* rotation matrix for yaw, pitch, roll */
                              /* A = | a[0]  a[1]  a[2] |     */
                              /*     | a[3]  a[4]  a[5] |     */
                              /*     | a[6]  a[7]  a[8] |     */
    c3daReceiver  rx;         /* handle to c3da core receiver */
                              /* need to be "created" using   */
                              /* c3daCreateReceiver           */
} Receiver;

/*
 * as an example, allocate some global emitters and one receiver
 */

Receiver       thisRx;
Emitter        Em[NUMBER_OF_EMITTERS];

/*
 * sample "world" setEmitterPosition routine
 */

void
setEmitterPosition( Emitter* pEm, int x, int y, int z )
{

    int        new_x, new_y, new_z;

    /*
     * save the world coordinates
     */
    pEm->x = x;
    pEm->y = y;
    pEm->z = z;

    /*
     * rotate this emitter so that it is relative to the fixed
     * receiver in the c3da core
     */

    new_x = thisRx.a[0]*x + thisRx.a[1]*y + thisRx.a[2]*z;
    new_y = thisRx.a[3]*x + thisRx.a[4]*y + thisRx.a[5]*z;
    new_z = thisRx.a[6]*x + thisRx.a[7]*y + thisRx.a[8]*z;

    /*
     * now tell the c3da core about the new position
     */

    c3daSetEmitterPosition ( &(pEm->em), new_x, new_y, new_z );

}

/*
 * sample "world" setReceiverOrientation routine
 *
 * Arguments:
 *   yaw - rotation around the z-axis (-180 to 180 degrees)
 *   pitch - rotation around the y-axis (-180 to 180 degrees)
 *   roll - rotation around the x-axis (-180 to 180 degrees)
 *
 * Notes:
 *   o order of rotations - roll, pitch, yaw
 *   o Instead of actually changing the orientation of the receiver,
 *     we equivalently move all the emitters around the fixed receiver
 *     in the c3da core.  The rotation matrix used to move the emitters
 *     is simply the transpose of the "composite" rotation matrix defined
 *     by yaw, pitch, roll and the order of rotations.
 */

void
setReceiverOrientation( int yaw, int pitch, int roll )
{

    float      yaw_f;
    float      pitch_f;
    float      roll_f;

    /*
     * convert yaw, pitch, and roll to radians
     */

    yaw_f = DEG2RAD * (float) yaw;
    pitch_f = DEG2RAD * (float) pitch;
    roll_f = DEG2RAD * (float) roll;

    /*
     * setup the inverse rotation matrix to handle yaw, pitch, and roll
     */

    thisRx.a[0] = cos(yaw_f)*cos(pitch_f);
    thisRx.a[1] = sin(yaw_f)*cos(pitch_f);
    thisRx.a[2] = -sin(pitch_f);
    thisRx.a[3] = -sin(yaw_f)*cos(roll_f) +
                   cos(yaw_f)*sin(pitch_f)*sin(roll_f);
    thisRx.a[4] = cos(yaw_f)*cos(roll_f) +
                   sin(yaw_f)*sin(pitch_f)*sin(roll_f);
    thisRx.a[5] = cos(pitch_f)*sin(roll_f);
    thisRx.a[6] = sin(yaw_f)*sin(roll_f) +
                   cos(yaw_f)*sin(pitch_f)*cos(roll_f);
    thisRx.a[7] = -cos(yaw_f)*sin(roll_f) +
                   sin(yaw_f)*sin(pitch_f)*cos(roll_f);
    thisRx.a[8] = cos(pitch_f)*cos(roll_f);

    /*
     * loop over the emitters letting them use the new orientation
     */

    for ( i = 0; i < NUMBER_OF_EMITTERS; i++ )
       setEmitterPosition( &Em[i], Em[i].x, Em[i].y, Em[i].z );
```

```
}
```