VESA®

Video Electronics Standards Association

860 Hillview Court, Suite 150
Milpitas, CA 95035

Phone: (408) 957-9270
FAX: (408) 957-9277

VESA BIOS Extension/ Audio Interface (VBE/AI)

Version: 1.0
Adopted: February 11, 1994

IMPORTANT NOTICE: This is the approved specification from the VESA Software Standards Committee.

Purpose

The VESA Audio Interface's intent is to provide a single, low level API for certain sound technologies, with the goal that the interface is extensible enough for use from the low end, to the high end of today's PC audio devices.

Summary

The low level interface approach allows the interface to be supported within a ROM BIOS, a DOS device driver, TSR, or virtualized via Windows, Windows NT, or OS/2.

Intellectual Property

Trademarks

All trademarks used in this document are property of their respective owners. Refer to the table of contents for a complete list of trademarks.
VESA, VBE/AI Video Electronics Standards Association

Patents

VESA proposals and standards documents are adopted by the Video Electronics Standards Association without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the proposals or standards document.

Support for this Specification

If you have a product which incorporates this standard, you should ask the company that manufactured your product for assistance. If you are a software, display, controller or manufacturer, VESA can assist you with any clarification you may require. All questions must be in writing to VESA, in order of preference by:

FAX: 408-957-9277 Email: support@vesa.org
Mail to: VESA
860 Hillview Court, Suite 150
Milpitas, CA 95035

Acknowledgments

This document was made possible by the joint efforts of the members of VESA Software Standards Committee. In particular, the following individuals and their companies contributed with time and knowledge.

VBE/AI Work group Leader

Rick Allen, Media Vision

Committee Meeting Participants

Jeff Li, Acer
Jean-Yves Deschenes, Adlib MultiMedia, Inc.
Alan Alvarez, Advanced Graphics Comp. Corp.
Jason Blochowiak, Argo Games.
Ed Calloway, ATI Technologies, Inc.
Mike Collins, Broderbund, Inc.
Brad Haakenson, Cirrus Logic, Inc.
Keith Uhlin, Cirrus Logic, Inc.
Brian Oberholtzer, Crystal Semiconductor
Lee Mulcahy, Digispeech
Shui Choi, Engras, Inc.
Albert Mak, ESS Technology, Inc.
James A. Prasad, ESS Technology, Inc.
Rod Dewell, Excalibur Software
James Tao, Focus Information

Mike Liebow, Forte Technologies, Inc.
Mike Travers, Forte Technologies, Inc.
Craig Travers, Forte Technologies, Inc.
Lary Liang, Genoa Systems Corp.
Eileen Ong, Genoa Systems Corp.
Richard Chen, Genoa Systems Corp.
Uwe Maurer, Graphics Decisions
Greg Moore, IBM-Boca Raton
Brian Marquardt, Interactive Multimedia (IMA)
Craig Wiley, Integrated Circuit systems (ICS)
Bob Davies, Intel
Jeff Snowman, Interactive Multimedia
Guy Tiphane, Logitech
Spike McLarty, Logitech
Michael Z. Land, Lucas Arts.
Peter McConnell, Lucas Arts.
Rick Allen, Media Vision, Inc.
Doug Cody, Media Vision, Inc.
Perry Cook, Media Vision, Inc.
Jon Kim, Media Vision, Inc.
Ben Mejia, Microsoft, Inc.
Mike Rozak, Microsoft, Inc.
John Miles, Miles Design.
David Hawley, National Semiconductor
Scott McCoy, National Semiconductor
Paul Sidenblad, Olivetti ATC
Jim Martin, Phoenix Technologies
Al Rock, PictureTel
Ygal Arbel, S3, Inc.
Mike Fung, S3, Inc.
Steven Fischer, Sierra Semiconductor, Inc.
Stuart Goldstein, Sierra Semiconductor, Inc.
Kerchen Heller, Sierra Semiconductor, Inc.
Geoff Snowman, Snowman & Associates
John Ratcliff, The Audio Solution
Gustave Hamaguchi, Toshiba America
Rick Silverman, Trident Microsystems
Michael Mala, Trident Microsystems
Hseuh-Lin Kung, ULSI Systems
Tom Ryan, VESA
Andrew Lambrecht, VLSI Technology
Carmine Bonanno, Voyetra
Bob Starr, Yamaha, Inc.
Kats, Takahashi, Yamaha, Inc.
Nick Emoto, Yamaha, Inc.
Rod Wood, Yamaha, Inc.

Revision History

- Remaining processes to be defined:

- VESA vendor data type registration procedures for MIDI patch types.
- VESA vendor data type registration procedures for WAVE compression data types.

0.98

- Ran the spelling checker to correct typographical errors.

0.97

- Added the "wsWavePrepare" as a result of review process. This allows the WAVE device to preprocess the wave audio data to it's internal format. A corresponding feature bit was added to the WAVE info structure to indicate the need to make this function call.
- Added the "wsWaveRegister" as a result of review process. This allows the application to register WAVE blocks and receive a handle for each. The routines, wsPlayBlock and wsRecordBlock will now take handles instead of far pointers and the length is now a reserved field.
- wsPauseIO, wsResumeIO, and wsStopIO now have an integer parameter that is reserved for future use.
- Full duplex is now limited to the wsPlayCont and wsRecordCont functions to have a consistent approach for applications, since this is the optimal approach.
- In the WAVE feature bits, the bit indicating support for variable sample rates is been expanded to cover variable rates for playback and record, in both mono and stereo.
- msPreLoadPatch function will now return an error code if the driver fails the download. In addition, a provision was made so the application can fail the download.
- One additional bit has been added to the MIDI features bits. This indicates the device supports a remote patch loading feature, enabling it to load it's own patches.
- Added an integer to both the OPL2 and OPL3 patch formats to indicate the patch type. This way the driver can correctly interpret both patch types for internal use.
- In the WAVE and MIDI device checks, the GETDMAIRQ message's return values have been expanded to cover devices that use more than one DMA or IRQ channel.
- The MIDI device check, MIDITONES, was missing the **Parameter/Return Value** descriptions. This was added after being spotted in review.
- A subsection in the VESA General Data Structures, was added to document the sizes of data variables.
- Additional comments and discussions have been added, as part of the review process.
- Added the standard Microsoft RIFF wrapper around the `<"vail">` patch library chunk.

0.96

- Additional discussion was added to various areas.

- Application guidelines have been added at the end of the documentation.
- One addtional command line switch has been defined for the TSR drivers to process.
- The WAVE service, wsPCMInfo has a new parameter for compression block sizes.
- The WAVECOMPRESSION device check now uses the HIWORD for block size.
- MIDI voice stealing has been expanded to work on a channel basis instead of globally on or off.
- General API Subfunction #2 used for query can be used to return query information on 16 bit real mode or 32 bit flat model services structures.
- MIDI device check for controller support was removed in favor of supporting the General MIDI controllers as a base set.

0.95

- Added the "msUnloadPatch" prototype to the MIDI services structure. The function was already described in the text, but the prototype was missing.
- Additional cleanup in the MIDI patch library format.

0.94

- A new function was added to the MIDI services to let the application poll for MIDI input data.
- Two feature bits were added to the MIDI info structure to indicate MIDI input via interrupts, or MIDI input via polling. If neither bit is set, then there is no MIDI input at all.
- A new function was added to the VOLUME services to allow the application to reset the channel to default settings.
- Added a memreq field to the volume info structure since the application will need to know the size of the services structure.
- Additional clarification has been added for opening and closing Volume devices.
- The patch library chunk types have preliminary been defined. Final approval of the chunk type names will occur with completion of the VESA membership in Microsoft's MDK registration.

0.93

- The patch library format is starting to conform to the Microsoft standard. It will be finalized with VESA's membership in Microsoft's MultiMedia vendor registration.
- A WAVE device check was added to let the application query for the smallest unit of transferable data the hardware supports.
- The Volume feature bit for indicating support of 360 degree sound positioning has been extended to two bits indicating Azimuth and Phi support, respectively. This allows card that support Surround Sound, and Qsound to indicate such support.
- All call backs to applications now have an additional parameter, a 32 bit long, appended to the end of the parameter list. This allows for future additions to the interface. Currently the value in this new parameter will be zero (0).

- A VOLUME device check was added to allow the application to query the current state of a parametric equalizer.

0.92

- Vendor Board ID has been reduced to just the installed board number since the rest of the information is contained in the ASCII text fields.
- VBE/AI int 10, Function 0x4F allows subfunction numbers above 0x80 through 0xEF to be defined by the vendor. Also, Device Check message numbers 0x80 through 0xEF are now available for vendor definition.
- A software code version number has been added to each device class info structure.

0.91

- Full duplex PCM record/playback has been finalized. There is a new device check to enable/disable full duplex operation within the WAVE device. The features bits have been restored in the document.
- In the MIDI API, msPreloadPatch and msUnloadPatch have another parameter, the channel number. This change is designed to specifically support channel #10 (9 zero based), the percussive channel, since each key number is a different patch.

0.90

- The document has undergone a major cleanup to add more descriptions to all sections, correct inconsistencies in parameter passing and naming conventions, as well as, bring all the function descriptions up to date from the work group meetings.

0.81

- Vendor data field in the services structure has been removed. If the vendor needs additional memory when the driver is opened, this memory should be added to the memory request field.
- The limit of a maximum of 500 callbacks per second, requested by the drivers, has been softened to a suggested maximum.
- All services structures have a new field for future expansion in later revisions of the VBE/AI spec.
- The Vendor Board ID field in all info structures, has been adjusted to make better use of its bits.
- General subfunction #1 now uses the DL register, instead of the DH, to pass parameters. This is done for consistency of the interface.
- Full duplex feature bit has been removed until full duplex WAVE playback/record is reviewed.

0.71

- Added the OPL2 patch definition in the VBE/AI Data definitions.
- Added an OPL3 patch definition in the VBE/AI Data definitions.

- Added a MIDI device check to query the size of the FIFOs on MIDI transmitter/receivers.
- Added a MIDI device check to query the supported MIDI controllers.
- Added a MIDI feature bit to indicate if the device supports MIDI time stamping on MIDI input.
- Added General API subfunction #5 so a program can request a driver to unload.
- Added General API subfunction #6 for drivers to unlink from the driver chain.
- Removed the DMA/IRQ/BASE I/O ADDRESS fields from all info structures. This information is now obtained by performing device checks. Also, a device check has been added to request MEMORY ADDRESS information, if applicable.

0.70

- Removed PolyMIDIMsg function, and expanded MIDImsg to support multiple messages.
- The "ChipId" field in the info structures has been defined to report Vendor, Board Product, and Installed Board #, so it has been renamed to "BoardId".
- The volume API has been revised. No major functional additions or deletions have occured, but just tweaking of the parameters. Some DeviceCheck queries have been added; as well as, more documentation on the whole interface.

0.61

- Reformatted into VESA document style in preparation for distribution of proposal.
- Added more text strings to the INFO structures for Vendor ID, etc.
- Added more MIDI error codes.
- Added a block passing protocol for handling large MIDI patches.
- General API interface now conforms to VBE 2.0 (1.2?) specification for register definition and usage.
- Added the user preference field in the WAVE and MIDI info structures.

0.06

- General subfunction #0 of the general query interface was removed since it whole functionality can be accomplished by using function #1. This reduces some redundant action by the driver.
- General subfunction #1 has been broadened to query either individual device classes, or the entire installed base of device classes.
- General subfunction #2 can perform all "DeviceCheck" functions. This was only possible after opening the device. Now it can be performed during the query process.
- General subfunction #5 is reserved for future use as a driver loading/unloading function.
- General subfunction #6 is reserved for future use to allow the driver's 32 bit counterpart to hook into the driver's General API set. This allows the 32 bit interface to be added at a later time without creating code changes for the application.
- WAVE Play/Record continuous calls had a correction to the 2nd parameter. It was defined as a long, but now is a 32 bit far pointer.
- WAVE vendor specific data in the services structure has been expanded to 64 bytes.

- WAVE error codes have been refined.
- VOL function, "SetABSVolume" has been renamed to "SetVolume".
- VOL function, "SetVolume" has a new parameter. This word controls the volume in two ways. The first control sets the maximum volume level. The second control sets the volume, from zero, up to the maximum setting.
- MIDI vendor specific data in the services structure has been expanded to 64 bytes.
- The driver handle is now the first parameter in all application callbacks. This allows the application to have one callback and handle multiple devices.
- MIDI devices will recognize the International Midi Association, General Midi, channel assignments.

0.05

- Major changes have been done to every section. As a result of input from different groups, changes have been made to correct many deficiencies. Overall, the major changes have been:
- Major reworking has occurred in the General API functions.
- The concept of giving the drivers the ability to unload from the system has been dropped. Once a driver is loaded, it will stay in place.
- The query functions have been changed to allow specific device class queries instead of performing general queries.
- The data formats, and device services have modified to make the whole interface more supportable by host operating systems, such as, OS/2, Windows, and Windows NT. The supported PCM data formats have been reduced to 8 and 16 bits. The MIDI interface picks up one function to handle poly MIDI messages with embedded deltas.
- The naming conventions are moving closer to the Microsoft Windows orientation.
- Removed the PCM single sample I/O interface, which makes the interface wholly block oriented.
- Reduced a lot of variables and bit definitions from the various Info and Services structures.
- Dropped the Vendor ID (VID) and Technology ID (TID) from the info structures. For WAVE and VOLUME devices, these fields were added only for consistency in the structures. The MIDI drivers used these fields to identify the underlying hardware. The new approach places the ID on the patch format, so IDI devices now identify which patches they support, not the underlying hardware.
- The "DeviceCheck" function has been added to all device Services. This allows the application to query the device for specific information, rather than checking a fixed field in any of the structures. This allows for future growth without causing the structures to be redefined on each revision.

0.04

- Interim step to version 0.05.

0.03

- Initial proposal.

Table of Contents

1. Introduction and Scope

1.1. Goals

- Standard Software API for DOS developers.
- Labeling standards for the customer.
- A DOS API supportable by ROM BIOS, as a DOS TSR or driver, or as a virtualized extension of the installed Windows, Windows NT, or OS/2 drivers.
- Pascal calling convention for multi-program language support.
- Standard Volume/Mixing control for each mixer device and master volume channels.
- Low level design provides low memory usage and little redundant code.

1.2. VBE/AI Architecture

The VESA Audio Interface is strictly a low level, hardware independent, interface giving applications access to hardware functionality. Since it is designed as a low level interface, it's services are somewhat rudimentary, thus require more programming from the application. Additional software, in the form of mid to high level functions, can isolate the application from majority of the tedious programming details.

The VESA Audio Interface defines one unique Application Programming Interface (API) for each device type. Each device type is defined as a "device class".

Since multiple hardware devices can reside in one machine, the VESA Audio Interface allows for multiple instances of each device class API. Each API may reside in a single software driver, or several may be combined within one software driver. The organization is transparent to the application, which sees each API as a separate unit of functionality.

There are two functional levels to the VESA Audio Interface architecture. The first level is the General API for device query, acquire and release. Applications use software INT 10 to access this first functional level. The second level is FAR calls to the device's API set. Once a device is opened, the application is given a list of entry points within the VBE/AI device.

1.2.1.Audio Device Supported

Currently, the VBE/AI specification defines three device classes: WAVE, MIDI, and VOLUME.

Wave Audio Services
The WAVE Audio Services present a block oriented interface for playing/recording digital audio data. Industry standard PCM data size and formats are included to provide

cross O/S compatibility. The following features are supported in the WAVE Audio services:

8/16 bit PCM
Mono/Stereo
Compression/decompression
Standard Sample rates of 8000Hz,11025Hz.,22050Hz, 44100Hz.
Variable Sample rates up to 48kHz

MIDI Services
The MIDI services are designed as a General MIDI playback device that handles individual MIDI messages. For devices supporting a true external MIDI connection, services are presented to allow recording (input) the incoming data stream.
The following set of devices can be supported in VBE/AI MIDI architecture:

FM
Wave Table
Wave Synthesis
Wave Guide
MIDI Output
MIDI Input
Allows for custom patches.

Volume Control
One API structure has been defined for supporting volume and mixer devices. Volume services may be found as standalone devices, or as a subservices structure in the WAVE and MIDI device classes. Total Volume devices will always be found as standalone services. The Volume Control API supports the following feature set:

Individual mixer channel control.
Master volume channel control.
Panning, from simple stereo to 3D space.
Filtering.
Tone Control.

1.2.2.Devices Not Supported

CDROM control, which is covered by the Microsoft CDROM Extensions.
Effects Processors. This class of device will be expanded in future version of the VBE/AI specification.

2.VESA General API for All Devices

The VBE/AI drivers will be present in memory for use by applications, much the same as the ROM BIOS. This approach means the drivers will be preloaded into memory before the application starts execution. The application is not responsible for loading the drivers into memory.

For normal operation, the application will use the general API in the following manner:

1. Perform subfunction #1 to locate a handle to the device.
2. Perform subfunction #2 to query the device for information.
3. Perform subfunction #3 to open the device.
4. Perform subfunction #4 to close the device when done.

All drivers will chain into a linked list of VBE/AI drivers found in the VESA INT 10 services. The linked list approach allows for an unlimited number of drivers to be loaded. This is critical since a wide variety of audio boards may be present in one system, at the same time. When a driver is loaded, it will attempt to chain itself in via the VBE/AI chain using subfunction #6. Failing that, will chain directly into the INT 10 interrupt chain. This dual approach is required to facilitate proper unloading of TSR drivers from the linked list. The concept is further explained in the subfunction #6.

As each driver loads, it will establish an instance number for each API it presents to the application. This instance will become the "handle" for that API. The driver will enumerate handles by deriving the highest handle number from the current list of drivers, then incrementing the number by one (1) for each of it's API's.

The following is a description of the first functional level of the VESA Audio Interface.

Design Notes for the General VBE/AI interface:

- This is the only interface that does not use the Pascal calling convention. All parameters pass to these subfunctions are passed in registers, so there will be a finite set of parameters available to each subfunction.

- Calls to this interface are done with a software interrupt instruction, so all returns are done via an IRET.

2.1.Subfunction #0 VBE/AI Driver Check

Application Description:

This subfunction is the first call to the VESA interface and may be used to determine if any VBE/AI APIs are present.

Driver Internal Operation:

This subfunction call conforms to the VESA VBE specification, subfunction 0 architecture. It allows the application to query for current status of the VBE interface.

Input:

AX = 0x4F13
BX = 0 the VESA subfunction #
CX = 0 reserved for future use.
ES:DI = 0 reserved for future use.

Output:

AX != 0x004F if unsuccessful
AX = 0x004F if successful:
BL = the VESA driver version in nibble packed format, where the high nibble is the major version, the low nibble is the minor version.
BH,CX,DX,SI are undefined, therefore assumed to be corrupted.
ES:DI is unchanged.

2.2.Subfunction #1: Get Next Device Handle

Application Description:

This subfunction allows the application to query the device classes for each devices handle.

On the first call, the application will load a NULL, in CX, and a device class in DL, then perform the subfunction call.

If a device responds, CX will be returned with a valid handle. The application now holds the handle to the first device within the specific class.

On subsequent calls, the application will pass the same handle received in CX, unaltered, back into subfunction #1. If there is another device in chain, CX will return holding it's handle.

If there are no more devices available in the class, as defined in the DL register, CX will return NULL. CX may return NULL on the first subfunction #1 call if there are no devices within the specified class.

Driver Internal Operation:

First the application passes in an NULL handle (0) in CX, and a device class in DL. When a matching device class driver sees this invalid handle, it will load it's own handle into BX, then return to the application.

On subsequent calls, the application will pass in the handle of the last responding device. When the device see's own handle in CX, it will zero out the handle, then chain to the next device driver. This next driver will see the NULL, then load it's handle and return to the application.

Ultimately, the application will query one too many times resulting in an NULL handle being returned. This tells the application that no more devices are available for this device class.

Input:

AX = 0x4F13
BX = 0x01, the VESA subfunction.#
CX = zero (0) for the first call, or
the handle received from any prior calls.
DL = Device Class ID, or 0 for all device class queries.

Output:

AX != 0x004F, the call was unsuccessful.
AX = 0x004F:
if CX != 0, it holds the handle to a valid
device class. This handle can be used for other
VESA subfunction calls to the device.
if CX = 0, no more devices are available in
the specified device class.

Assume working registers BX,DX modified.

2.3.Subfunction #2: Query Device Class Info

Application Description:

This subfunction allows the application to retrieve, and in some functions, set, information about the installed device.

The most important query performed is number 2, which returns the main structure, defined as the "GeneralDeviceClass". Within this structure is the specific device class "Info" structure indicating the hardware's feature set. Refer to each Device Class for further structure definition.

Other queries return either a structure of variables, defined by the class of device, or the length of that given structure. All returned structures will be copied into the callers buffer pointed to by SI:DI.

Volume Info and Services structures are accessed through this query method.

The query process can be broken down into two sets of queries: general device queries, and specific device queries.The general query numbers are 1 through 6. To execute a general device query, AX holds the VBE function, 0x4F13, BX holds the subfunction number, 0x02, CX holds the device handle, and DX holds the query number:

query #1.	Get the length of the General Device Class Structure in SI:DI.
query #2.	Return a copy of the General Device Class Structure pointed to by SI:DI.
query #3.	Return the length of the Volume Info Structure for the device in SI:DI DX is NULL if the device class does not have a volume control.
query#4.	Get a copy of the Volume Info Structure for the device pointed to by SI:DI. DX is NULL if the device class does not have a volume control.
query #5.	Return the length of the Volume Services Structure for the device in SI:DI. DX is NULL if the device class does not have a volume control.
query #6.	Get a copy of the Volume Services Structure for the device pointed to by SI:DI. DX is NULL if the device class does not have a volume control.

The specific device queries are the same set as defined in each device's "DeviceCheck" function call. To execute a specific device query, AX holds the VBE function, 0x4F13, BX holds the subfunction number, 0x02, CX holds the device handle, and DX holds the device check function #, as defined within each device class. The registers SI:DI hold the 32 bit long parameter required by each device check. For a definition of each specific device query, refer to the "DeviceCheck" function found in each device services structure.

Driver Internal Operation:

This is a simple subfunction for the device driver. If the handle matches, it just has to process one of six queries or device check functions. NOTE Since the device check functions are being executed without requiring the device to be opened, the driver device checks must not have any dependencies upon an opened state!

Input:

AX = 0x4F13
BX = 0x02, the VESA subfunction #

CX = the device handle.
DL = the query #:

1. Get the length of the Device Info Structure.
2. Return a copy of the Device Info Structure.
3. Get the length of the Volume Info Structure for the device.
4. Get a copy of the Volume Info Structure for the device.
5. Get the length of the Volume Services Structure for the device.
6. Get a copy of the Volume Services Structure for the device.

DH = 0 for retrieving 16 bit real mode sizes and structures.
DH = 1 for retrieving 32 flat model sizes and structures. If 32 bit flat model is not supported, AX will return the appropriate indication.
SI:DI = A pointer to a memory block of X bytes in length for the functions that perform a structure copy. The length is determined by calling one of the functions to return the length of the structure in question.

Output:

AX != 0x004F, the call is unsuccessful, the other registers are undefined.
AX = 0x004F, the call is successful, the query returns these register values:

query #1. SI:DI = length of the Device Info Structure, in bytes.
query #2. Copies the Device Info Structure into the callers memory.
query #3. SI:DI = length of the Volume Info Structure, in bytes.
query #4. Copies the Volume Info Structure into the callers memory.
query #5. SI:DI = length of the Volume Services Structure, in bytes.
query #6. Copies the Volume Services Structure into the callers memory.

Specific "DeviceCheck" return results are returned in SI:DI. See each device for the "DeviceCheck" functions, parameters, and results.

Assume working registers BX,CX,DX modified.

2.4.Subfunction #3: Open Device

Application Description:

This subfunction allows the application to open WAVE and MIDI devices, and "own" the device as long as required. If the devices opens successfully, the device will remain "owned" by the application, until closed via Subfunction #4. NOTE: An application should always use the driver with the highest user preference. The preference is found in the Info structure, or read via a device check. If matching preference levels are found, then just use the first driver with that matching preference level.

NOTE: Volume devices cannot be opened with this service since all the info and services structures are acquired through subfunction #2 queries. An open call to a stand alone Volume device class will return a failure code. The normal approach to accessing volume devices is the query process in subfunction #2. The purpose here is to present one common approach to acquiring all Volume devices. Since opening a device implies exclusive ownership, this would be inappropriate for this class of device. Exclusive ownership is not available for volume devices.

Driver Internal Operation:

If the device is available for use, a copy of it's services structure is placed in the application's memory block (the segment/selector portion is passed in the SI register). After some internal housekeeping, the driver returns with SI:CX pointing to the application's memory block.

The memory block made available by the application will be used by the device for most of it's memory storage needs. This allows the driver to reside in ROM with little data requirements when idle. The size of the block is determined by the 'memreq' field in the device's Info structure.

If the device is unavailable for any reason, SI:CX will return NULL.

Input:

AX = 0x4F13
BX = 0x03, the VESA subfunction #
CX = The driver handle #.
DX = 0 for the drivers 16 bit interface,
DX = 1 for the drivers 32 bit interface.
SI = A segment/selector to a block of memory of the size required by the driver. NOTE: the memory starting offset is assumed to be zero. See the "memreq" field in the Device Info structure for the size required.

Output:

AX != 0x004F if unsuccessful.
AX = 0x004F if successful:
SI:CX = a far pointer to a structure that contains a table of service functions. The driver will return a zero if the requested API is not available (maybe already in use).

Assume working registers BX,DX modified.

Refer to the specific device class for a description of the Device Services Structure.

2.5.Subfunction #4: Close Device

Application Description:

This subfunction allows either the Wave or MIDI device driver to be closed by the application. It is very important that the application perform a close call before terminating, since the driver will continue to use the memory provided in the open call. The application MUST close all devices before terminating to keep the driver from touching the memory after the application has ended.

NOTE: The Volume device is a special case that can not be closed. Since the services are not acquired via subfunction #3 open call, then it doesn't need to be closed via subfunction #4.

When the application performs this call, the driver will perform the following steps:

For WAVE Audio Drivers:

Stops audio.
Frees the DMA, clears any IRQ requests.
All queued buffers are returned to the caller.

For MIDI Drivers:

Turns off all voices.
Frees the DMA, clears any IRQ requests.
Frees any of the applications patch data memory blocks.

For VOLUME drivers:

The close call is are ignored by Volume devices.

Once the driver is finished executing this call, it MUST NOT attempt to use the memory provided by the application at the open call. The memory provided by the application can now be considered freed up by the driver.

Input:

AX = 0x4F13
BX = 0x04, the VESA subfunction #
CX = The device handle

Output:

AX != 0x004F if unsuccessful
AX = 0x004F if successful

Assume working registers BX,CX,DX modified.

2.6.Subfunction #5 Driver Unload Request

Application Description:

This subfunction is presented to allow the application to ask the VBE/AI TSRs to unload from memory. The only driver that is expected to unload would be one written as a DOS TSR. True DOS drivers and ROM BIOS code are unable to unload from memory, so this call has limited usefulness only within DOS. Once the subfunction has been executed successfully, the application must then finish the TSR unloading. See the following note on unloading TSRs.

Driver Internal Operation:

Once the driver is requested to unload, it will execute subfunction #6 to determine if it can unload. If not, it will fail the call.

Input:

AX = 0x4F13
BX = 0x05 the VESA subfunction #.
CX = the device handle.

Output:

AX != 0x004F if unsuccessful
AX = 0x004F if successful:
BX = the PSP segment of the unloading TSR driver, so the application can terminate the process and release the driver's memory to the DOS pool.

Assume working registers CX,DX modified.

NOTE: The technical aspects of unloading TSRs go beyond the scope of this document. For more information on unloading TSR's, refer to the book, "Undocumented DOS", published by Addison Wesley.

2.7.Subfunction #6 Driver Chaining

Application Description:

This subfunction allows VBE/AI drivers to be hooked into the calling chain of the VBE list of drivers. The application should never call this subfunction. It is considered an internal subfunction for drivers use only.

Driver Internal Operation:

The purpose of subfunction #6, is to provide an alternate to chaining into INT 10 directly. It provides a safe way to load and, possibly, unload the drivers.

This subfunction has two distinct functions. The first is to allow drivers to link into the driver list. The second step allows drivers to unlink from the list.

Every driver must perform the first step, in order to link itself into the driver list. If the driver executes this subfunction, and finds the subfunction fails, then it must directly hook the INT 10 interrupt vector. Once it does this, it becomes a permanent link in the chain, so it must never unload. This architecture is beneficial since it means only one VBE/AI driver will ever hook INT 10 directly. All other drivers can be chained into the list, thus can be removed via the second step of this subfunction.

This second step, to unlink from the driver list, is designed for TSR implementations since no other form of DOS driver/BIOS has the ability to unload. The internal driver operation handles the incoming unload call by comparing the callers address (in DX:CX) with it's own next-in-chain vector. If the callers address matches, the caller's next-in-chain address (in SI:DI) will be stored as the driver's next-in-chain vector. At this point the caller is detached from the driver list.

NOTE: It should be noted that during normal operation, the callers next-in-chain address might be NULL.Since this is true, all drivers must always make a validity check of the next-in-chain vector before attempting to execute the next-in-chain address. If the next-in-chain vector is invalid, the driver will just perform an IRET instead of attempting to chain on.

NOTE: The second step of this subfunction allowing the driver to detach itself from the driver list does not mean the driver memory is free. The TSR code must still use the aid of an application to return it's memory to the DOS memory pool. For more information on unloading resident TSRs, refer to the book, "Undocumented DOS", published by Addison Wesley. Once the unchaining call is successfully completed, the driver must not fail to unload.

Step #1 - Chaining into the driver linked list:

Input:

AX = 0x4F13
BL = 0x06, the VESA subfunction #
BH = 0 to request the driver to chained into the linked list.
DX:CX holds this driver's entry point.

Output:

AX = 0x004F if successful
DX:CX = the address of the next-in-chain driver. This address MUST be retained and used to chain to the next driver during the entire time the driver is loaded.
AX != 0x004F if unsuccessful, The driver is now allowed to directly chain to the INT 10 interrupt vector. If this occurs, the driver becomes a permanent member of the chain; therefore, must never unload, via the second step of this subfunction call.

Assume working registers BX, modified.

Step #2 - Unchaining from the driver linked list:

Input:

AX = 0x4F13
BL = 0x06, the VESA subfunction #
BH = 1 to request the driver to be unchained from the linked list.
DX:CX = this driver's entry point, identical to the first half of this subfunction call.
SI:DI = the next-in-chain driver address that was received in the first half of this call.

Output:

AX = 0x004F if successful
AX != 0x004F if unsuccessful. If the function fails, the driver must NOT attempt any further unloading, nor should the application assume the driver is able to be removed from memory. The driver has become a fixed member of the chain that cannot be freed. Becoming a fixed member of the chain will occur with the first driver chained. Since this entire linkage scheme depends upon having at least one driver loaded, the first driver must always be available to process the VBE/AI chain-in requests.

2.8.Function #7: 32-bit interface loading

reserved for appending 32 bit services to the driver query capabilities - to be defined later

3.VESA General Data Structures

3.1.General Device Class structure for all devices

When a device is queried via subfunction #1, it will return the following structure. The entire structure contents are read-only for the application. There are no fields that the application may modify.

```
typedef struct {

     // housekeeping...

     char gdname[4];     // name of the structure
     long gdlength;      // structure length

     // generalities...

     int  gdclassid;     // type of device
     int  gdvesaver;     // version of VESA driver

     union {
          WaveInfo   gdwi;
          MIDIInfo   gdmi;
          VolumeInfo gdvi;
     } u;

} GeneralDeviceClass, far *fpDC;
```

3.2.General Device class Structure field descriptions

```
char gdname[4];
long gdlength;
```

These two fields are for structure ID only. Vendor name, and other specific information is stored in other fields.

```
int gdclassid;
```

This 16 bit quantity identifies the class of device:

0 - unknown.
1 - Wave Audio Device.
2 - MIDI Device.

3 - Volume Control Device.

```
int gdvesaver;
```

This is the current version of VESA driver definition.

```
union {
  WaveInfo   gdwi;
  MIDIInfo   gdmi;
  VolumeInfo gdvi;
} u;
```

This union contains one of the Info structures for a device class. Check the classid field to determine the structure type. Each structure is explained in the following sections.

NOTE: Having the Volume Info structure appearing in this structure is a special case, where, for instance, a line input connection has a Volume Services API, but is not associated with any WAVE or MIDI services.

3.3.Variable Data Sizes.

Throughout the specification, reference is made to various sizes of data variables. This section documents the variables, in number of bits.

char	8 bit byte
int	16 bit word.
long	32 bit word.
char far *	32 bit segment:offset pointer.
void far *	32 bit segment:offset pointer.
void huge *	32 bit segment:offset pointer.
pascal far *	a pointer to a function that uses the pascal calling convention.

TRUE	This is a non-zero value. The data size is determined by the context used at each point within the specification.
FALSE	This is a binary zero value. The data size is determined by the context used at each point within the specification.
NULL	This is a binary zero value. The data size is determined by the context used at each point within the specification.

LOWORD	the low 16 bits of a 32 bit long.
LOWORDLOWBYTE	Bits D0-D7 of a 32 bit long
LOWORDHIGHBYTE	Bits D8-D15 of a 32 bit long

HIWORD	the high 16 bits of a 32 bit long.
HIWORDLOWBYTE	Bits D15-D23 of a 32 bit long

HIWORDHIGHBYTE	Bits D24-D31 of a 32 bit long

4.WAVE Audio Services

4.1.Introduction

The VESA WAVE Audio services provide a low level interface for playback and recording of PCM data. The interface is oriented around block I/O to/from the device. Both polled and DMA devices can perform block I/O using this API, transparent to the application.

4.2.Theory of Operation

Once a WAVE audio application has determined the presence of the devices, it may open a device for it's use. Assuming the device opens without error, the application must now make some adjustments. It will have to query the feature bits to determine the best way to play/record audio.

The two main areas of adjustment will be for PCM data size, and sample rate limitations.

For PCM data size, an application can standardize on the lowest common denominator, as the safest approach. For the more savvy, using the larger data size, and making runtime adjustments on the data to scale it to fit, maybe an acceptable approach.

Supported sample rates vary between most manufactures. The VESA Audio Interface has adopted a base set, with the possibility of variable rates in between. For some applications, selecting the base set may be acceptable. For others, performing real-time interpolation to other rates may be an acceptable approach.

The basic interface is geared toward block oriented devices. This type of device may use a DMA channel, or may have on-board FIFO memory for caching blocks of data. The interface, for the most part, is the same for the application.

Non-DMA Block oriented devices can be supported as Block I/O devices that use timer tick callbacks to keep data moving. The Disney Sound Source with a 16 byte FIFO, or Media Vision AudioPort with a 1024 byte FIFO, are two examples.

If the device driver requires timer tick callbacks, then the application will be responsible for calling the driver at the specified rate.

Before starting to play or record, the application must fill in the two callbacks in the WAVE Services Structure. This allows the application to receive callbacks when the block I/O is completed (emptied/filled).

To begin playing, the application will first set the sample rate, data size, etc., through the "wsPCMInfo" routine, then will register one or more blocks with "wsWaveRegister", and

being playback by calling "wsPlayBlock". When the block is done playing, a callback to the "wsApplPSyncCB" vector will occur to indicate the end has occurred. The application can now make another call to "wsPlayBlock" to start playing another registered block. The process for recording is similar when using "wsRecordBlock", except the PCM blocks will be full of new recorded data and sent back via the "wsApplRSyncCB" call back routine.

For using the continuous block approach, the application allocates and manages the fixed buffer. The only limitation is that the buffer must conform to the 64k limitations of the XT DMA architecture. This is the lowest common denominator for all hardware, thus covers non-DMA and DMA devices alike. See "wsPlayCont" for more details on buffer limitations.

The continuous block approach will take a fixed circular buffer, divide it into integral amounts, fill it, call the driver to play, then receive a callback as each integral amount is processed by the driver. It is up to the application to keep track of which portions of the circular buffer are empty. This is possible by use of the "wsApplPSyncCB" callback. As each portion of the buffer is played, the device driver calls the application, thus creating a sync signal the application can use.

The model of continuous recording is identical to the continuous playback functions, except when callbacks are made to the application, the buffers will be filled, not emptied of data. Also, callbacks are performed via the "wsApplRSyncCB" vector.

To pause and resume audio transfers, the application may call "wsPauseIO" and "wsResumeIO". To terminate an audio transfer, the application should use "wsStopIO", not "wsPauseIO". Conversely, it would be incorrect to use "wsStopIO" to pause a data transfer since current buffer position information will be lost.

4.3.WAVE Info Structure

```
typedef struct {

     // housekeeping

     char winame[4];     // name of the structure
     long wilength;      // structure length

     // hardware vendor name

     long wiversion;     // driver code version
     char wivname[32];   // vendor name
     char wiprod[32];    // vendor product name
     char wichip[32];    // vendor chip hardware description
```

```
    char wiboardid;     // installed board #
    char wiunused[3];   // unused 3 bytes

    // device specific information

    long wifeatures;    // feature bits
    int  widevpref;     // user preference field.

    int  wimemreq;      // memory required for driver use.
    int  witimerticks;  // # of timer callbacks per second

    int  wiChannels;    // 1 = mono, 2 = stereo. Stereo is
                        // assumed to be interleaved data.
                        // stereo always supports mono too.

    int  wiSampleSize;  // Bit field of max sample sizes

} WAVEInfo, far *fpWAVInfo;
```

4.4.WAVE Info structure field descriptions

```
char winame[4];     // name of the structure
long wilength;      // structure length
```

These two fields are for structure ID only. Vendor name, and other specific information is stored in other fields.

```
long wiversion;
```

This field is the software version number of the driver. It's value is defined by the vendor. The data will be the BCD format.

```
char wivname[32];
```

This field is an ASCIIZ text string containing the vendor name. The NULL terminator is part of the declared size.

```
char wiprod[32].
```

This field is an ASCIIZ text string for the board level product name. The NULL terminator is part of the declared size.

```
char wichip[32].
```

This field is an ASCIIZ text string for the device specific name, such as "Yamaha OPL3". The NULL terminator is part of the declared size.

```
char wiboardid
```

This field defines the board # installed in the machine. Some hardware allows for multiple card of the same type to be installed in the same machine. This field identifies which board number this driver is addressing.

```
long wifeatures;
```

This is a bit field listing the features of supported by this device driver. Here are the current definitions.

```
0x00000001 8000Hz Mono Playback.
0x00000002 8000Hz Mono Record.
0x00000004 8000Hz Stereo Record.
0x00000008 8000Hz Stereo Playback.
0x00000010 8000Hz Full Duplex Play/Record.

0x00000020 11025Hz Mono Playback.
0x00000040 11025Hz Mono Record.
0x00000080 11025Hz Stereo Record.
0x00000100 11025Hz Stereo Playback.
0x00000200 11025Hz Full Duplex Play/Record.

0x00000400 22050Hz Mono Playback.
0x00000800 22050Hz Mono Record.
0x00001000 22050Hz Stereo Record.
0x00002000 22050Hz Stereo Playback.
0x00004000 22050Hz Full Duplex Play/Record.

0x00008000 44100Hz Mono Playback.
0x00010000 44100Hz Mono Record.
0x00020000 44100Hz Stereo Record.
0x00040000 44100Hz Stereo Playback.
```

All the above bits indicate the minimal set of sample rates supported by the device.

```
0x00080000 44100Hz Full Duplex Play/Record.
```

This bit indicates the device can be used for both playing and recording WAVE audio, simultaneously.

```
0x08000000 Wave Prepare Data call is required.
```

This bit indicates the driver must first prepare the data before it is played, or after it is recorded.

```
0x10000000 Variable Mono   Playback Sample Rates.
0x20000000 Variable Stereo Playback Sample Rates.
0x40000000 Variable Mono   Record Sample Rates.
0x80000000 Variable Stereo Record Sample Rates.
```

These bits indicates the device supports more than the base set of sample rates. Use the `DeviceCheck` function to find the actual rates.

The device can support more sample rates than defined in the standard set.

```
int  widevpref;
```

The users preference of selected devices will be reflected in this field. The DeviceCheck function is used to set the users preference. The numbering will be 0 for highest preference to X, in single increments, for the lowest preference.

```
int  wimemreq;
```

The WAVE Audio driver (as with all VBE/AI drivers...) uses very little internal memory in the unopened state. For the device to be used, the application must make available memory for the device. Each device driver may have different memory requirements.

```
int  witimerticks;
```

The device may require X number of call backs per second from the application. A value of zero (0) indicates no timer tick call- backs are needed. The theoretical maximum may be several thousand a second, but a practical maximum should be 500. This is only a suggested maximum limit for consideration of slower machines.

```
int  wiChannels;
```

This field indicates the number of DAC/ADC "channels" available. One (1) means the device is monoaural, two (2) means it is a stereo device.

```
int  wiSampleSize;
```

This is a bit field indicating the different PCM sample sizes that the hardware can handle. The bit meanings are currently defined as:

```
0x0001  8 bit playback
0x0002 16 bit playback
```

```
0x0010  8 bit record
0x0020 16 bit record
```

Design Notes:

▪The application must allocate the requested memory, size specified by the "wimemreq" field, and pass it to the driver in the open call, subfunction #3. When the application closes the driver, the memory is free to be reused by the application. The pointer must be a segment:offset, where the offset has a value of zero.

4.5.WAVE Audio Services Structure

The following is a description of the contents of the WAVE services. For the most part, this structure is read-only to the application. The only two fields the application can write are the callback fields.

```
typedef struct {

     // housekeeping

     char wsname[4];     // name of the structure
     long wslength;      // structure length

     char wsfuture[16];  // reserved for future use

     //device driver supplied function

     long (pascal far *wsDeviceCheck)(int,long);
     long (pascal far *wsPCMInfo    )(int,long,int,int,int);
     int  (pascal far *wsPlayBlock  )(int,long);
     int  (pascal far *wsPlayCont   )(void far *,long,long);
     int  (pascal far *wsRecordBlock)(int,long);
     int  (pascal far *wsRecordCont )(void far *,long,long);
     int  (pascal far *wsPauseIO    )( int );
     int  (pascal far *wsResumeIO   )( int );
     int  (pascal far *wsStopIO     )( int );

     int  (pascal far *wsWavePrepare)(int, int, int, void
                                  far *,long);
     int  (pascal far*wsWaveRegister)(void huge*, long);

     int  (pascal far *wsGetLastError)( );
     void (pascal far *wsTimerTick  )( );

     // device driver run-information time data

    void(pascal far*wsApplPSyncCB)(int,void far*,long,long);
    void(pascal far*wsApplRSyncCB)(int,void far*,long,long);

} WAVEService, far *fpWAVServ;
```

4.6.WAVE Services Structure field descriptions

```
char wsname[4];     // name of the structure
long wslength;      // structure length
```

These two fields are for structure ID only. Vendor name, and other specfic information is stored in other fields.

```
long (pascal far * wsDeviceCheck  )( int, long);
long (pascal far * wsPCMInfo      )( int, long,int,int,int);
int (pascal far * wsPlayBlock    )( int, long);
int (pascal far * wsPlayCont     )( void far *, long,long)
int (pascal far * wsRecordBlock  )( int, long);
int (pascal far * wsRecordCont   )( void far *, long,long);
int (pascal far * wsPauseIO      )( int );
int (pascal far * wsResumeIO     )( int );
int (pascal far * wsStopIO       )( int );
int (pascal far*wsWavePrepare)(int,int,int,void huge*,long);
int (pascal far*wsWaveRegister)   (void huge *, long);
int (pascal far * wsGetLastError )( );
void (pascal far * wsTimerTick    )( );
```

The above set of indirect function calls encompass the complete WAVE audio services. each function will be discussed below.

```
void (pascal far *wsApplPSyncCB )(int,void far *,long,long);
void (pascal far *wsApplRSyncCB )(int,void far *,long,long);
```

These fields are loaded by the application with a far entry point within the application. The callback functions allow the driver to return empty (playback mode) or full (record mode) blocks back to the application. Either of these calls may be performed at interrupt time, such as, at the end of a block interrupt. The parameters to the application are:

Parameters:

```
int         The driver's handle.
void far *  The applications pointer to the block,
            or NULL if returning a block handle.
            The handle is returned in parameter
            #3, instead of the block length.
long        The length of the block, in bytes
long        Reserved, currently zero (0).
```

The application is responsible for saving all registers. The CPU Flags register may be modified.

Design Notes:

For devices such as the Sound Blaster, ThunderBoard, etc, which require an internal speaker connection to be made at runtime, this will be performed by the driver, transparently to the application.

4.7.WAVE Driver Functions

The WAVE audio interface is designed as a low level approach to playing/recording digital audio. This approach means much of the work of handling the WAVE audio stream must happen in the application.

The facilities within this driver allow for:

- 8/16 bit PCM play and record of mono or stereo PCM data.
- Single buffer transfers, via DMA or other means.
- Circular buffer transfers via the auto-initialization mode of the DMA controller, or other means.
- Application independence from direct hardware programming of the audio card, IRQ and DMA channels.

The API does not require the use of a DMA channel. Parallel port devices, etc., can use the timer tick function to process the data, then perform end of block callbacks to the application.

```
long (pascal far *wsDeviceCheck) (int, long);
```

Description:

This function is a conglomerate service to provide the application one call to derive all sorts of miscellaneous information about the device. The first parameter is a message number; the second, a 32 bit parameter to the message. Device checks of value above 0x80 are vendor specific, defined by each vendor.

```
message WAVECOMPRESSION (0x11)
```

This message allows the application to ask the device if it supports a specific hardware compression/decompression. Contact the VESA office for the current list of compression types.

Parameter:

LOWORD The parameter is an enumerated compression type. Contact The VESA office for the most current list of enumerated compression types.

HIWORD Blocking size. If this value is zero, the driver must just validate the enumerated compression type without considering the block size. If this is non-zero, then the driver can respond if it can handle

the block size, for this type of compression.

NOTE: Hardware implementation of compression schemes is best supported by applications if there is a software only solution. The VBE/AI philosophy sees hardware compression as an accelerator to the software.

Return Value:

DWORD FALSE if no match (no h/w support) is available.
TRUE if a matching compression scheme found.

`message WAVEDRIVERSTATE (0x12)`

This message returns the state of the driver, that is, if idle, or currently processing any data.

Parameter:

None.

Return Value:

LOWORD indicates the current driver status:
-1 if the driver is not opened.
0 - idle
1 - busy
0x80 - MSB OR'ed in to indicate a paused state.

`message WAVEGETCURRENTPOS (0x13)`

For playback, returns the byte count of played samples within the current block. For record, returns the byte count of recorded samples in the currently filling block. Both values are a count in bytes. For continuous block modes, the returned byte count will be the current position of the pointer from the beginning offset of the buffer. This value should be as accurate as possible for reliable usefulness to applications that make use of real-time additive synthesis.

Parameter:

None.

Return Value:

DWORD - The number of samples played or recorded in the current block. NOTE: Since many audio cards do not have hardware support for

current sample count information, this value will only be an estimate of the true position. The driver must return the most accurate position possible down to the sample.

`message WAVESAMPLERATE (0x14)`

This message allows the application to query the device for support of specific sample rates. Since sample rates on the various cards are not exact matches, the rule of thumb should allow a match if the hardware can come within 5 percent of the applications sample rate.

Parameter:

The 2nd parameter holds the monaural sample rate.

Return Value:

DWORD - 0 if the sample rate is not supportable.
Any other result is the closest match the hardware can perform, within a maximum delta of 5 percent.

`message WAVESETPREFERENCE (0x15)`

This message allows the user to select his/her own order of driver use, if there are multiple VBE/AI devices installed in the machine. An application can query the preference field in the info structure to determine the users preference. A zero (0) indicates highest preference for this device. The preference value is a simple incrementing count, from 0, to X, in increments of one (1).

Parameter:

The 2nd parameter holds the new preference value.
A negative one (-1) will not be set, so this function can also be used to query the current setting without having to perform the Query Device Class Info API call.

Return Value:

LOWORD - The old preference value.

`message WAVEGETDMAIRQ (0x16)`

This message allows the application to find out if the device uses an IRQ or DMA channel. This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

HIWORDLOWBYTE - The first DMA channel used by the device. A 0xFF means no DMA channel is used. If the most significant bit is set, the device support auto initialize DMA.

HIWORDHIGHBYTE - The second DMA channel used by the device. A 0xFF means no DMA channel is used. If the most significant bit is set, the device support auto initialize DMA.

LOWORDLOWBYTE - The first IRQ channel used by the device. A 0xFF means no IRQ channel is used.

LOWORDHIGHBYTE - The second IRQ channel used by the device. A 0xFF means no IRQ channel is used.

`message WAVEGETIOADDRESS (0x17)`

This message allows the application to find out the device's base I/O address. NOTE: This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

LOWORD - The base I/O address. For MPU-401 devices, this might be 0x330, or 0x300. For the early Adlib Music Synthesizer card, this might be 0x388.

`message WAVEGETMEMADDRESS (0x18)`

This message allows the application to find out the device's base memory starting address, and size. NOTE: This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

LOWORD - The base memory address, in the form of a real mode segment address, with an assumed offset of zero (0). -1 indicates memory mapping is not used.
HIWORD - The size of the addressable memory space , in bytes. A -1 indicates memory mapping is not used.

`message WAVEGETMEMFREE (0x19)`

This message allows the application to find out how much memory is available on the audio card. Some cards use on-board memory to cache data during record and playback. The value returned here will be the current state of the card, therefore; it may change during runtime.

Parameter:

none.

Return Value:

DWORD indicating the amount of free on-board memory, 0 if none.

`message WAVEFULLDUPLEX (0x1A)`

It is a requirement that the application use the PlayCont/RecordCont functions for full duplex operations. This will provide a stable audio stream for listening, less erratic recorded data, and better synchronization of audio to external events.

This message disables/enables/queries the state of full duplex hardware operation. Full duplex means having the ability to play and record simultaneously at the same sample rate and same block length. The default state of the driver is to have full duplex disabled. The application will query the feature bits to determine if the feature is supported, then makes this call to enable the operation. Once this is done, the wave services will now allow PlayCont/RecordCont operations to occur. In normal operation, one request would pre-empt the other.

Once full duplex is enabled, the application will perform a pair of matching calls to PlayCont and RecordCont. The two processes will not begin until the second call is made. This way the driver will be able to synchronize both I/O operations.

The other calls, such as pause/resume/stop, etc. will affect both streams. As each

process hits the end of a buffer division, the application will receive the appropriate callback through either "wsApplPSyncCB" or "wsApplRSyncCB".

Parameter:

0 Disable full duplex operation.
1. Enable full duplex operation.
2. Query the state of the is flag. Returns a 0 for disabled, 1 for enabled.

Return Value:

LOWORD Always returns the current state of the full duplex flag. A zero (0) is returned if disabled, a one (1) is returned if enabled.

`message WAVEGETBLOCKSIZE (0x1B)`

This message returns the count, in samples, of the smallest unit of transferable data that the hardware supports. Here are some examples of how to respond to this query. For a simple DMA device, such as the original Sound Blaster, this call would return a one (1), since the Sound Blaster uses single sample DMA transfers on low order DMA channels. For the Pro Audio Spectrum, which can use a high order DMA channel, 8 bit samples are transferred two at a time; therefore, would respond with a 2 to this query. The Yamaha Gold Sound Standard chipset, as used on the Adlib Gold 1000 card, requires the data to be transferred in 128 byte blocks, so on a low order DMA, in 8 bit mode, the returned value would be 128.

Parameter:

LOWORD(PCM data size of 8 or 16)
HIWORD (compression type)

Return Value:

DWORD - the smallest transferable unit of samples supported by the hardware.

`message WAVEGETPCMFORMAT (0x1C)`

This device check returns two values: the supported formats of 8 and 16 bit PCM, and the current enabled state.

The return value is a bit field where each bit indicates a supported format. The following is a listing showing, in hexadecimal format, each bit definition.

0x00000001 8 bit signed data.
0x00000002 8 bit unsigned data.
0x00000010 16 bit signed data.
0x00000020 16 bit unsigned data.

Parameter:

none.

Return Value:

LOWORD the bit field of the supported formats.
HIWORD the current enabled state.

`message WAVEENAPCMFORMAT (0x1D)`

Parameter:

LOWORD - A bit field ,as declared in the WAVEENAPCMFORMAT device check. This parameters must have only one bit set for each PCM data size. This will enable that format until changed by another WAVEENAPCMFORMAT device check, or the device is closed.

Below are some examples of legal and illegal settings with the assumption that the hardware supports a mask of 0x00000033.

The following are legal values:

0x00000012 enables 8 bit unsigned and 16 bit signed format.
0x00000022 enables 8 bit unsigned and 16 bit unsigned format.

The following are illegal values:

0x00000012 8 bit unsigned data, 16 bit is not enabled.
0x00000031 8 bit signed, both 16 bit signed and unsigned enabled.

Return Value:

none.

```
long (pascal far *wsPCMInfo)(int,long,int,int,int);
```

Description:

This call sets up the play/record data stream parameters. The driver will calculate the best match sample rate and return it back to the application. The new parameters will become effective immediately, until changed by another "wsPCMInfo" call. This call may be performed at any time when the device is opened. It will take effect immediately. For devices that have an onboard FIFO, the FIFO data is not guaranteed to be played at the old rate. The sample rate change may affect the data in the FIFO. During a recording session, any data in the FIFO during the sample rate change may have been recorded at the old sample rate.

All blocks registered via the "wsWaveRegister" call will assume the current state, as defined by the last "wsPCMInfo" call.

Parameters:

int - channels: 1 for mono, 2 for stereo.
long - sample rate: up to 48kHz.
int - the compression type may be specified for hardware assist.
int - blocking size for the compression data.
int - PCM data size of 8 or 16.

Return Value:

long - 0 if in error (see wsGetLastError()).
Returns the best match sample rate if there are no errors.

```
int (pascal far *wsPlayBlock) (int, long);
```

Description:

This function performs a single transfer of a registered wave block to the audio card. When the application calls this function, if the driver is currently busy with another buffer, it will stop that process, perform the wsApplPSync callback, then proceed with this new request. Once the transfer begins, control will be returned to the application. The full transfer will be done as a background process, transparent to the application.

When playing wave audio blocks, keep in mind, the driver only sees one block at a time, thus plays only one block at a time. There is no assumption that one block will follow another. Once the transfer is completed, the hardware returns to an idle state. This low level approach is intended only to facilitate transfer of the applications' data to/from the audio card.

For non-DMA devices, the timer tick callbacks can be used to output data to the device. Using the timer ticks this way dictates that the hardware have a modest FIFO for handling blocks of data since the timer tick callback cannot happen at rates faster than about 2ms.

Drivers that make use of the DMA will have to validate the applications buffer under any running Virtual DMA Service before attempting to program the DMA controller. One way around this complication is to break down the users' buffer into chunks aligned on 4k boundaries, and play each one. Once the whole buffer has played, then give the sync callback to the application.

The driver will handle all 64k alignment problems. The application will be notified of completion via the callback (assuming the wsApplSyncCB field is loaded) only when the entire block transfer is completed.

NOTE: The callback to the application is expected to occur when the data has played out the DAC, i.e., has been heard by the user. In the case of devices using a FIFO, the driver must hold the callback until the data has been passed through the DAC. Once played out, the driver then makes the callback to the application. If a FIFO device returns the handle upon loading the FIFO, then the application will notified too early that the data has been heard. This is a potential problem for applications using the callback for synchronization of multiple events.

NOTE: Once the block is registered with the "wsWaveRegister" function, the data must remain in place for the driver to have access to it. The application, on the other hand, is not allowed to touch the data in the buffer since it may have been moved to the audio card's on-board memory. Once the buffer has been unregistered, the application can then modify the contents of the block.

Parameters:

int - the registered wave block handle.
long - NULL - reserved for future use.

Return Value:

FALSE if the block cannot be played due to any of the possible error conditions. Call "wsGetLastError" to determine the error.

```
int (pascal far *wsRecordBlock)(int, long);
```

Description:

This function performs single transfer of data from the audio card into the registered Wave block. When the application calls this function, if the driver is currently busy with

another buffer, it will stop that process, perform the "wsApplRSync" callback, then proceed with this new request. Once the transfer begins, control will be returned to the application. The full transfer will be done as a background process, transparent to the application.

When recording WAVE audio blocks, keep in mind, the driver only sees one block at a time, thus records only one block at a time. There is no assumption that one block will follow another. Once the transfer is completed, the hardware returns to an idle state. This low level approach is intended only to facilitate transfer of the applications' data to/from the audio card.

For non-DMA devices, the timer tick callbacks can be used to record data from the device. Using the timer ticks this way dictates that the hardware have a modest FIFO for handling blocks of data, since the timer tick callback cannot happen at rates faster than about 2ms.

Drivers that make use of the DMA will have to validate the memory under any running Virtual DMA Service before attempting to program the DMA controller. One way around this complication is to break down the users' buffer into chunks aligned on 4k boundaries, and record each one. Once the whole buffer is filled with recorded data, the sync callback is given to the application.

The driver will handle all 64k alignment problems. The application will be called back (assuming the wsApplRSyncCB field is loaded) only when the entire block transfer is completed.

NOTE: Once the block is registered with the "wsWaveRegister" function, the block must remain in place for the driver to have access to it. The application, on the other hand, is not allowed to touch the data in the buffer since it may not be moved to the buffer until it is unregistered. Once the buffer has been unregistered, the application can then process the contents of the block.

Parameters:

int - the registered Wave block handle.
long - NULL - reserved for future use.

Return Value:

FALSE if the block cannot be played due to any of the possible error conditions. Call "wsGetLastError" to determine the error.

```
int (pascal far*wsPlayCont)(void far*,long,long);
int (pascal far*wsRecordCont)(void far*,long,long);
```

Description:

These play and record modes function as if the DMA controller were programmed in auto-initialize mode. For devices that support the DMA, this is in fact, what happens. For non-DMA devices, such as, parallel port devices, the driver is given enough information to emulate the auto-init DMA mode.

When the application calls this function, if the driver is currently busy with another buffer, it will stop that process, return the buffer to the caller, then proceed with this new request. Once the transfer begins, control will be returned to the application. The full transfer will be done as a background process, transparent to the application.

This approach assumes the application will manage the DMA buffer, but allows the driver to manage the hardware. The DMA process (or pseudo DMA process for non-DMA devices) will loop in the caller's buffer. For the playback process, the application is expected to preload the buffer with data before making this call. For best playback performance, at least two buffer divisions worth of data should be preloaded before making this call.

After each buffer division is processed, the driver will call the application with a pointer to the completed buffer division. As a rule, the driver will call the application before touching the next buffer division. For DMA devices, a few samples into the next division may be processed due to interrupt latency, etc. Therefore, the application should make allowances by preloading enough divisions to keep the driver from playing invalid data.

During the record process, as each buffer division is loaded with new data, the driver will call the application with a pointer to the filled portion. As with the playback process, the hardware will continue filling the next buffer division. The application will be notified as each division is filled.

The sample rate will be the same for all processed data, until the stream is complete, terminated, or changed via a PCMInfo call.

Device drivers that make use of the DMA, will have to validate the memory under any running Virtual DMA Service before attempting to program the DMA controller. The application should make a practice of using buffers less than 4k in size, aligned on 4k boundaries, to avoid problems with Virtual DMA host environments. If the VDS validation fails, or the buffer crosses a 64k boundary, the driver may elect to fail the call. Even though the PC's second DMA controller supports 128k memory size transfers, the VBE/AI interface accepts a maximum size of 64k for the circular buffer.

NOTE: During PlayCont operations, the callback to the application is expected to occur when the data has played out the DAC, i.e., has been heard by the user. In the case of devices using a FIFO, the driver must hold the buffer until the data has been passed through the DAC. Once played out, the driver then makes the callback to the application. If a FIFO device returns the buffer upon loading the FIFO, then the application will notified too early that the data has been heard. This is a potential problem for applications using the callback for synchronization of multiple events.

Parameters:

void far *A pointer to the buffer (adjusted to reside within a 64k page).
long This is the buffer length, in bytes.
long This parameter is an integral division of the dma buffer length (the second parameter), in bytes. This allows the driver to callback the application after every integral division of data has been processed.

Suggested value for the buffer length is 4096 maximum. The suggested integral division is 2048. This divides the buffer in half. An integral amount of 1024 divides the buffer in quarters.

Return Value:

FALSE if in error. Possibly buffer crosses a 64k boundary, or VDS failed.
See GetLastError() for more possibilities.
TRUE if running.

```
int (pascal far *wsPauseIO) ( int );
```

Description:

This function pauses an active audio data stream. No action is taken if the device is idle.

Parameters:

int - Reserved for future definition. MUST BE NULL.

Return Value:

int TRUE if paused successfully.
FALSE if channel was inactive or already paused.

```
int (pascal far *wsResumeIO) ( int );
```

Description:

This function resumes the paused audio stream. No action is taken if the device is in a running state.

Parameters:

int - Reserved for future definition. MUST BE NULL.

Return Value:

int TRUE if resumed successfully.
FALSE if channel was inactive or already running.

```
int (pascal far *wsStopIO) ( int );
```

Description:

This function stops the audio stream, if running. It resets the hardware, and returns any buffers to the application. No action is taken if the device is already idle.

Parameters:

int - Reserved for future definition. MUST BE NULL.

Return Value:

None.

```
long(pascal far*wsWavePrepare)(int,int,int,void
huge *,long);
```

Description:

This routine incorporates necessary functionality for devices that do not support the defined data formats of unsigned 8 bit and signed 16 bit audio. This routine allows the device to modify the wave audio data to its native format for playback, or to the standard formats, after recording.

This function allows the driver to modify the buffered data, but not to change the size of the buffer. Later revisions may allow for more extensive data conversions, but now it is limited to modifying 8 bit to 8 bit, and 16 bit to 16 bit samples.

The application will check the feature bits in the info structure to see if this call is necessary. If so, the call must be made any time before playing or any time after recording. This allows an application to load a file, preprocess it through this function, then store for later playback. This approach reduces the amount of real-time data manipulation.

Drivers that modify the buffer will return the buffer length as an indication of changes. Drivers that do not modify the buffer will return a zero, indicating no changes. The application is not expected to call this function if the feature bit is not set.

If the block is going to be played via "wsPlayBlock", it must be passed through this function before being registered via the "wsWaveRegister" function call. Conversely, if the block was recorded via "wsRecordBlock", it must be unregistered before the application calls this function to modify the contents of the block.

Parameters:

int	Control word for the conversion. 0 to convert for playback, or 1 to convert from a recording.
int	PCM data size of 8 or 16.
int	channels: 1 for mono, 2 for stereo.
void huge *	The huge pointer to the data buffer.
long	size of the buffer, in bytes.

Return Value:

long	Returns 0 if no changes were made, else the size of the buffer, in bytes.

```
int (pascal far * wsWaveRegister)(void huge * long);
```

Description:

This function is used by the application to registers a wave block for use by "wsPlayBlock" and "wsRecordBlock" functions. The drivers will allow up to 32 blocks to be registered, even though only one can be played at a time. This call must be performed before "wsPlayBlock" or "wsRecordBlock" since the returned handle will be passed to these functions.

The block will remain registered with the driver until the application calls this function to unregister the block. This is performed by passing a NULL pointer in parameter 1, and

the handle in parameter 2. Once the block is unregistered, it's handle will become invalid. The block cannot be referenced by that handle since the driver has removed it from it's internal tables.

The application must be careful how it uses the registered block handles. Since the handle value has meaning only to the driver, using the handle in compare instructions, etc., will result in unpredictable results, thus is not recommended.

The block being registered will assume all the parameters of the last "wsPCMInfo" call.

Once the block is registered with this function, the data must remain in place for the driver to have access to it. The application, on the other hand, is not allowed to touch the data in the buffer since it may have been moved to the audio card's on-board memory. Once the buffer has been unregistered, the application can then modify the contents of the block.

This function is only for use with the "wsPlayBlock" and "wsRecordBlock" functions. The continuous mode operations, "wsPlayCont" and "wsRecordCont", still require a pointer to a buffer with the application performing a back fill operation.

Parameters:

void huge *	The huge pointer to the data buffer. If NULL, this is an unregister request from the application.
long	size of the buffer, in bytes, or the handle of a block being unregistered.

Return Value:

int	If registering a Wave block, the value returned will be a unique handle, or NULL if the device has an error (Call "wsGetLastError" to determine the fault). Unregistering a Wave block never generates an error, so the application will ignore the return value.

```
void (pascal far *wsTimerTick) ( );
```

Description:

This function is called by the application at the timer resolution requested by the driver. (See the WAVE info structure for the resolution). If the driver does not request timer ticks, then the application may ignore this function.

This service allows, for example, a driver that feeds a parallel port audio device, to send the next portion of data when playing data, or to read the next portion of data into the recording buffer.

The driver must use some judgment in how much time is spent in this routine. If the driver has a lot of work to do, then it must break up the load across several timer ticks. At a high resolution of 500 ticks per second, the driver may end up spending more than 2ms moving data to the device, such as a parallel port. On some slower machines, an interrupt rate of 500 times per second may consume the entire bandwidth of the computer, when running any one of today's virtual memory managers. These software packages have to virtualize the hardware interrupts, and much more, thereby creating a lot of overhead on every hardware interrupt. For this reason, The driver should not assume the caller can make more than 500 calls per second (1 every 2ms).

This is not a reentrant function call. The application is expected to handle reentrant timer interrupts in case the driver takes more than one timer tick, thus avoiding stack overflow. When the application calls this function, interrupts are assumed to be enabled.

Parameters:

None.

Return Value:

None.

```
int (pascal far *wsGetLastError) ( );
```

Description:

When an error condition occurs, the driver will post an error number internally. This error code can be read by the application through this function call.

The driver will zero out the posted error after the application has read it.

Parameters:

None.

Return Value:

The last error number:

```
0x01 WAV_NOSUPPORT       The driver was asked to
                        perform an unsupported function
0x02 WAV_BADSAMPLERATE   Cannot handle the sample rate
0x03 WAV_BADBLOCKLENGTH  Block length is invalid
0x04 WAV_BADBLOCKADDR    Bad block Crossed 64k boundary
                        or Virtual DMA Services barfed.
0x05 WAV_BADLOSTIRQ      Application missed an IRQ
0x06 WAV_BADPCMDATA      unknown PCM size, format,
                        or compression type.
0x80 WAV_HWFAILURE       generic hardware failure.
```

Error messages 0x80 - 0xEF are vendor specific. 0x00-0x7F and 0xF0-0xFF are reserved by VESA.

Design Notes:

All calls to the driver assumes up to 256 bytes of stack available for the drivers use. If making a nested callback to the application, the application must use little stack memory during this callback.

5.MIDI Audio Services

5.1.Introduction

The MIDI API provides low level MIDI I/O services. The instrument and channel assignments conform to the General MIDI Specification from the MIDI Manufactures Association.

In an attempt to standardize the playback of musical data across the wide spectrum of MIDI devices, an architecture for handling patches has been adopted to normalizes the way each patch (musical instrument or sound effect) is used by the application. This normalization requires the architecture to be divided into two parts; a patch library and device API. The device API is the standard set of services, as described here in the MIDI Audio Services. The patch library is a disk resident file containing the driver's General MIDI instrument patches. If the MIDI device has the GM patches preloaded, the library, is not required.

The MIDI driver provides the application with low level, indirect, H/W access, without any hardware dependencies. A MIDI transmitter/receiver, such as the MPU-401, may have a driver that will send the MIDI stream directly to an external synthesizer. For other devices, such as the OPL3 FM synthesizer, the driver will be interpreting the MIDI stream to play the individual notes.

A driver that does not contain a preloaded set of General MIDI patches, may specify an external, disk resident, library of patches. It is up to the application to present the patch data to the driver before the instrument is used. This approach has several benefits. First, the memory size of the MIDI driver is greatly reduced since much of the patch data is static data, and not often used. Storing the data in a disk library makes maximum use of limited RAM resources. Secondly, if the application knows the format of a given patch, it can provide it's own custom patches to the driver as overrides to the default set. There's no rule saying the driver must ONLY use it's patches from the library. This gives the application a chance to make custom sounds for common hardware, such as the Yamaha OPL2, and use the VESA MIDI drivers to perform the work

In specifying an external patch library, the vendor hardware specific patch data now becomes a standard object to the application. The file format structure will be known to applications, but the contents, size, etc. of each patch will be specific to the vendor. An example of the file format is provided below.

The typical use of the patch library occurs in this fashion: The application will open a Synthesizer device and open the associated patch library. A MIDI data stream of notes will be playing, when a Program Change is encountered. The application will check to see

if the device has a preloaded patch, or requires one from the library. If the patch is in the library, the application will load the object into memory, strip off the 'vaid' header, then pass a pointer to the patch, to the driver. When the driver sees the program change, the patch will be available for loading into the device.

Proposed patch library format, conforming to Microsoft's LIST chunk format definitions:

```
<"vail">,<LEN> {
     <"ZSTR">,<LEN>,<DATA>               // copyright, etc
        "       "     "                      "
     <"vaip">,<LEN> {                    // patch pointers
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 0.
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 1.
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 2.
                    "       "      "       "         "
                    "       "      "       "         "
          <"vaid">,<LEN>,<DATA>           // patch 0 data
          <"vaid">,<LEN>,<DATA>           // patch 1 data
          <"vaid">,<LEN>,<DATA>           // patch 2 data
          <"vaid">,<LEN>,<DATA>           // patch 3 data
             "       "     "                    "
             "       "     "                    "
     }
}
```

The actual file format will have the <"vail"> chunk embedded in a <"RIFF"> chunk, so the actual file contents will be:

```
     <"RIFF">,<LEN> {
          <"vail">,<LEN> {
               ....
               ....
          }
     }
```

The patch file format has four VBE/AI list chunk types defined. The "vail" chunk begins a nested structure of chunks. Within the "vail" chunk, the first chunk type encountered may be the Microsoft "ZSTR" chunk. Any number of "ZSTR" chunks may appear before the first "vaip" chunk. The "ZSTR": chunk type is the standard Microsoft chunk for holding ASCIIZ strings. These messages contain any relevant information, such as copyright, etc.

The next chunk type, "vaip" begins the actual patch library. Within this chunk type, are two other chunks, "vaii", and "vaid". The "vaii" is the VAI Index containing an <OFFSET>, to the corresponding "vaid", or VAI Data chunk. Also in the "vaii" chunk,

the <DLEN> field holds the length of the corresponding "vaid" <DATA> field length. The <OFFSET> is a byte count starting from the "vaip", file position, inclusive. To calculate the file position of the first "vaid" chunk, take the file position of the "v" in the "vaip" chunk, plus the <OFFSET> in the first "vaip" chunk.

If the driver does specify a patch library (remember, the library is optional) it should be present somewhere on the user's hard driver or network. In order to find the file, the application should perform the following search:

1. Search the current directory.
2. Search each path in the environment PATH statement.
3. Search the root directory of drive C:

If the library is not found, the application can choose to skip the driver, or use it with just the default patch. Each VBE/AI MIDI device will have a default patch so something audible will play.

The patch library will hold 256 patches. The first 128 will be the General MIDI melodic patches. The second 128 will be the General MIDI percussive patches. The first General MIDI percussive patch starts at 35th entry in the second half of the library.

NOTE: The patch library format is a different file format, not associated with the Microsoft General MIDI file format. The MS General Midi file format introduces a form of hardware dependencies in the file format by predefining the channel assignments for certain levels of hardware feature sets. Only IMA MIDI channel assignments are recognized by the VBE/AI MIDI devices.

5.2.Theory of Operation

Once a MIDI application has found a VBE/AI MIDI device, it may open the device for it's use. Assuming the device opens without error, the application will now query the driver Info structure to query the feature bits for certain data; specifically, to find the patch library name, number of channels supported, timer tick callbacks needed, etc.

For an application to use the MIDI device, it will be required to perform all the data handling, tempo, and synchronization tasks. The low level VESA MIDI driver only provides the I/O to the device, and translation of MIDI messages to hardware specific meaning, to generate appropriate sounds.

The typical MIDI data stream is a sequence of Note On, Note Off, and Program Change event messages. The precise time each MIDI event is to occur, is known as delta time zero. The Application will be maintaining the tempo speed by calling the MIDI driver with MIDI messages that are presently at delta time zero. When the driver receives the MIDI events, it assumes the messages are to be processed immediately.

Before MIDI Program Changes occur, the application must perform some housekeeping to make sure the driver can play the new patch. Within each Info structure, the "features" bit field informs the application if all GM patches are preloaded. If so, then the application does not have to provide any additional information. If the patches are not preloaded, the application can query the "patches" bit field to determine if the particular patch is loaded. If not, then the app must then extract the patch from the library, and send it to the driver, via the "msPreloadPatch" function call. This can occur before delta time zero, and MUST occur before the program change event is handed to the driver. NOTE: for MIDI transmitter/receiver devices, the "msPreloadPatch" function will transmit the whole patch to the external device. This means the patch data within the library must be in the SYSEX format, with appropriate vendor information already embedded. For more information on patches, see the section on VESA Audio Data Formats. NOTE: All VBE/AI MIDI devices will have at least one preloaded default patch so the device will make minimal sound.

For applications that wish to perform custom sounds, sending patch information via the "msPreloadPatch" can override a loaded or default patch. The formats of each patch are defined by the hardware vendor. For the application to override an existing patch, it just has to present a new patch in the vendor specific format. For external boxes attached to a MIDI transmitter/receiver, the MIDI Sys-Ex command is used to perform such custom patch configurations. This approach is fully supported since a MIDI transmitter/receiver will pass that data on to the external device. For the most part, applications can maintain hardware independence by using the "msPreloadPatch" call allowing the dependencies to reside in both the driver and patch libraries.

5.3.MIDI Info Structure

```
typedef struct {

     // housekeeping

     char miname[4];     // name of the structure
     long milength;      // structure length

     // hardware vendor name

     long miversion;     // driver code version
     char mivname[32];   // vendor name
     char miprod[32];    // vendor product name
     char michip[32];    // vendor chip/hardware description
     char miboardid;     // installed board number
     char miunused[3];   // unused data

     char milibrary[14]; // patch library file name

     long mifeatures;    // feature bits
     int  midevpref;     // user set preferences field

     int  mimemreq;      // memory required for driver use
     int  mitimerticks;  // # of timer callbacks per second

     int  miactivetones; // # of tones available (polyphony)

} MIDIInfo, far *fpMIDInfo;
```

5.4.MIDI Info structure field descriptions:

```
char miname[4];     // name of the structure
long milength;      // structure length
```

These two fields are for structure ID only. Vendor name, and other specific information is stored in other fields.

```
long miversion;
```

This field is the software version number of the driver. It's value is defined by the vendor. The data will be the BCD format.

```
char mivname[32];
```

This field is an ASCIIZ text string containing the vendor name, copyright, etc. The NULL terminator is part of the declared size.

```
char miprod[32].
```

This field is an ASCIIZ text string for the board level product name. The NULL terminator is part of the declared size.

```
char michip[32].
```

This field is an ASCIIZ text string for the device specific name, such as "Yamaha OPL3". The NULL terminator is part of the declared size.

```
char miboardid
```

This field defines the board # installed in the machine. Some hardware allows for multiple card of the same type to be installed in the same machine. This field identifies which board number this driver is addressing.

```
char milibrary[14];
```

This field is an ASCIIZ text string naming the vendors patch library for this device. It is a standard DOS file name. The NULL terminator is part of the declared size.

```
long mifeatures;
```

This is a bit field listing all the features of supported by this device driver. Here are the current definitions.

`0x00000001`
: Reserved for later GM extensions.

`0x00000002`
: Reserved for later GM extensions.

`0x00000004`

: Reserved for later GM extensions.

`0x00000008`
: Reserved for later GM extensions.

`0x00000010 Midi Transmitter/Receiver`

This device is a MIDI transmitter/ receiver. It requires other external devices to play sound. This device may be used to receive MIDI data from external devices.

`0x00000040 Patches preloaded`

This device contains all the patches necessary to play a GM data stream.

`0x00000080 Internal Time Stamping supported`

On MIDI transmitter/receivers, the MIDI input callback has a parameter that contains the MIDI byte time stamp. This feature bits indicates if that parameter is valid, I.E. generated by the hardware. If this bit is not set, the application will have to generate it's own time stamping.

`0x00000100 MIDI interrupt driven input supported.`

On MIDI receivers, the MIDI input is driven via hardware interrupts. This is then normal approach, which allows the MIDI driver to callback the application whenever a byte is received. If this bit is set, then the MIDI polled bit should not be set.

`0x00000200 MIDI polled input.`

On this type of MIDI receiver, there is no hardware support for interrupt driven input, so the application will have use the "`msPollMIDI`" routine to generate "`msApplMIDIIn`" callbacks. If this bit is set, then the MIDI interrupt bit should not be set.

`0x0000400 MIDI remote patch loading supported.`

This bit indicates the driver can perform remote loading of patches. When set, the application may call the "msPreloadPatch" call with a NULL far pointer to indicate a patch load request. The other

```
int  midevpref;
```

The user will set his/her preference of selecting devices using this field. The Device Check function will receive the users preference. The numbering will be 0 for highest preference to X, in single increments, for the lowest preference.

```
int mimemreq;
```

The MIDI driver (as with all drivers...) uses very little internal memory in the unopened state. For the device to be used, the application must make available memory for the device. Each device driver may have different memory requirements.

```
int mitimerticks;
```

The device may require X number of call backs per second from the application. A value of zero (0) indicates no timer tick call backs are needed. The theoretical maximum may be several thousand a second, but a practical maximum should be 500. This is only a suggested maximum limit for consideration of slower machines.

```
int miactivevoices;
```

This field is a count of the maximum number of voices that can play at once. For an OPL2, the maximum, for rhythmic mode, is 9. For a Sound Canvas, the number would be 24.

5.5.MIDI Services Structure

Simple structure of the standard MIDI data stream

The actual MIDI Services Structure:

```
typedef struct {

     // housekeeping

   char msname[4];            // name of the structure
   long mslength;             // structure length

   int  mspatches[16];        // patch-is-loaded bit field
   char msfuture[16];         // reserved for future use

     // device driver supplied function

   long (pascal far*msDeviceCheck) (int,long);
   int (pascal far*msGlobalReset) ();
   int (pascal far*msMIDImsg    ) (char far *,int);
   void (pascal far*msPollMIDI   ) (int);
   int (pascal far*msPreLoadPatch)(int,int,void huge*,long);
   int (pascal far*msUnLoadPatch) (int,int);
   void (pascal far*msTimerTick  ) ();
   int  (pascal far* msGetLastError)();

     // application supplied functions
```

```
   void (pascal far*msApplFreeCB)(int,int,char far*,long);
   void (pascal far*msApplMIDIIn)(int,int,char,long);

} MIDIServ, far *fpMIDServ;
```

5.6.MIDI Services Structure field descriptions:

```
char msname[4];     // name of the structure
long mslength;      // structure length
```

These two fields are for structure ID only. Vendor name, and other specific information is stored in other fields.

```
long (pascal far *msDeviceCheck  )(int, long);
int  (pascal far *msGlobalReset  )();
int  (pascal far *msMIDImsg      )(char far *, int);
void (pascal far *msPollMIDI     )(int );
int  (pascal far *msPreLoadPatch )(int,int,void huge*,long);
int  (pascal far *msUnLoadPatch  )(int,int);
void (pascal far *msTimerTick    )();
int  (pascal far *msGetLastError )();
```

The above set of indirect function calls encompass the complete MIDI audio services. each function will be discussed in the following MIDI Driver Functions section.

```
void (pascal far *msApplFreeCB )(int,int,char far *,long);
```

This field is filled by the application with a far entry point within the application. This callback function allows the driver to return patch blocks back to the application. The application is responsible for saving all registers. The CPU Flags register may be modified.

Parameters:

int	The driver's handle.
int	The patch #.
void far *	The applications pointer to the patch.
long	Reserved, currently zero (0).

```
void (pascal far *msApplMIDIIn )( int, int, char, long );
```

This field is filled by the application with a far entry point within the application. This callback function allows a MIDI receiver to pass MIDI data bytes back to the application.

The application is responsible for saving all registers. The CPU Flags register may be modified

Parameters:

int	The driver's handle
int	The time stamp.
char	the MIDI byte received.
long	Reserved, currently zero (0).

```
int  patches[16];
```

This table defines which patches are currently loaded. The table is large enough to hold the state of all 256 patches; 128 for melodic, 128 for percussive. When a bit is set to 1, it indicates the patch is preloaded, and available for a program change. The application can check the appropriate bit to determine if a patch must be preloaded before a program change is issued.

5.7.MIDI Driver Functions

```
long (pascal far *msDeviceCheck) (int, long);
```

Description:

This function is a conglomerate service to provide the application call to derive all sorts of information about the device. The first parameter is a message number; the second, a 32 bit parameter to the message. Device checks of value above 0x80 are vendor specific, defined by each vendor.

`message MIDITONES (0x11)`

This message asks the device for the current count of unused tones. This way, the application can determine the number of sounds still available for use. A tone is defined as the minimal addressable unit of sound. On the Roland Sound canvas, this is a "partial". On the Yamaha OPL2, this is a 2-op voice.

Parameter:

None.

Return Value:

LOWORD - The count of available tones.

NOTE: In the case of MIDI devices that do not know the tone count, such as MIDI transmitter/receivers, this function can just return 0xFFFF to all calls.

`message MIDIPATCHTYPE (0x12)`

This message asks the device if it understands a particular patch format type. Contact the VESA office for a full listing of Registered Patch types.

Parameter:

The long is a number listing the predefined patch type.

Return Value:

DWORD FALSE if the patch type is not supported by the device driver.
TRUE if the device driver understands the patch type.

`message MIDISETPREFERENCE (0x13)`

This message allows the user to select his/her own order of driver use, if there are multiple VBE/AI devices installed in the machine. An application can query the preference field in the info structure to determine the users preference. A zero (0) indicates highest preference for this device. The preference value is a simple incrementing count, from 0, to X, in increments of one (1).

Parameter:

The 2nd parameter holds the new preference value.
A negative one (-1) will not be set, so this function can also
be used to query the current setting without having to perform
the Query Device Class Info API call.

Return Value:

LOWORD The old preference value.

`message MIDIVOICESTEAL (0x14)`

This message allows the application to enable or disable the driver's voice stealing on a per channel basis. If voice stealing is to be enabled for all channels, (which is the default state), the HIWORD will be 0xFFFF, each bit corresponding to the appropriate channel. To disable a channels voice stealing, set the corresponding HIWORD bit to one (1).

NOTE: This calls persistence only lasts through the current open state. Once the device is closed, then the driver will revert back to all channels allowing voice stealing.

Parameter:

LOWORD = 0 disable voice stealing. Send the mask in the high word
= 1 enable voice stealing. Send the mask in the high word.
= 2 return the current state of the voice stealing flag. High word is used as an AND mask to the current setting. All 1 bits matching this mask will be returned to the app.
HIWORD Bit mask of 16 channels where the lsb is channel 0, and the msb is channel 15.

Return Value:

if parameter = 2, then a return value of TRUE indicates
voice stealing is enabled, else FALSE for voice stealing is disabled.

NOTE: In the case of MIDI devices that cannot control voice stealing, such as

MIDI transmitter/receivers, this function can just return 0xFFFF to all calls.

`message MIDIGETFIFOSIZES (0x15)`

This message allows the application to find out the physical FIFO size for MIDI transmitter/receivers. This call is only valid for that type of device.

Parameter:

none.

Return Value:

LOWORD - The size, in bytes, of the MIDI receiver FIFO or FALSE if not supported.
HIWORD - The size, in bytes, of the MIDI transmitter FIFO or FALSE if not supported.

`message MIDIGETDMAIRQ (0x16)`

This message allows the application to find out if the device uses an IRQ or DMA channel. This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

HIWORDLOWBYTE - The first DMA channel used by the device. A 0xFF means no DMA channel is used. If the most significant bit is set, the device support autoinitialize DMA.
HIWORDHIGHBYTE - The second DMA channel used by the device. A 0xFF means no DMA channel is used. If the most significant bit is set, the device support autoinitialize DMA.
LOWORDLOWBYTE - The first IRQ channel used by the device. A 0xFF means no IRQ channel is used.
LOWORDHIGHBYTE - The second IRQ channel used by the device. A 0xFF means no IRQ channel is used.

`message MIDIGETIOADDRESS (0x17)`

This message allows the application to find out the device's base I/O address. NOTE: This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

LOWORD - The base I/O address.
For MPU-401 devices, this might be 0x330, or 0x300. For the early Adlib Music Synthesizer card, this might be 0x388.

`message MIDIGETMEMADDRESS (0x18)`

This message allows the application to find out the device's base memory starting address, and size. NOTE: This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

LOWORD - The base memory address, in the form of a real mode segment address, with an assumed offset of zero (0). -1 indicates memory mapping is not used.
HIWORD - The size of the addressable memory space , in bytes. A -1 indicates memory mapping is not used.

`message MIDIGETMEMFREE (0x19)`

This message allows the application to find out how much patch memory is available on the midi card. Some cards use on-board memory to cache data. The value returned here will be the current state of the card, so it may change during runtime.

Parameter:

none.

Return Value:

DWORD - amount of free memory, 0 if none.

`message MIDIDRIVERSTATE (0x1A)`

This message returns the state of the driver, that is, if idle, or currently processing any data.

Parameter:

None.

Return Value:

LOWORD indicates the current driver status:
-1 if the driver is not opened.
0 - if opened.

```
int (pascal far *msGlobalReset) ( );
```

Description:

Resets the driver and all voices to an inactive state. All pointers to patches that have been retained will now be returned to the application, via the ApplFreeCB callback.

Parameters:

None.

Return Value:

None.

```
int (pascal far *msMIDImsg) ( char far *, int);
```

Description:

This function receives a far pointer to a block of MIDI messages. These messages will be the standard MIDI messages, such as, NOTE ON, NOTE OFF, Program Change, etc. This

function is called at delta time 0, meaning ALL data within the block is to be acted upon immediately. The driver will process all the data in the block before returning to the application. Since this occurs, the "msMIDImsg" and ""msPreloadPatch"" function calls should not be considered re-entrant since this will cause one data stream to preempt another, causing a breakdown in the MIDI protocol.

Even though the application may pass down complete MIDI messages, the drivers are required to handle the data as if it came in one byte at a time. This means the driver will have to handle running status, and process the MIDI protocol.

As far as the hardware will allow, all drivers will support the following MIDI messages:

Note Off	(8n)	
Note On	(9n)	
Poly Key	(An)	
Controllers	(Bn)	
	Modulation	(1)
	Main Volume	(7)
	Pan	(10)
	Expression	(11)
	Sustain	(64)
	Reset all Controllers	(121)
	All Notes Off	(123)
	Reverb	(91)
	Chorus	(93)
Patch Change	(Cn)	
Channel Pressure	(Dn)	
Pitch Bend	(En)	
System Messages	(Fn)	

Parameters:

char far * - Pointer to the MIDI messages, such as a NOTE ON message, etc.
int - this is the length of the midi message.

Return Value:

None.

```
void (pascal far *msPollMIDI ) ( int );
```

Description:

This function allows the application to poll a MIDI receiver for data. The normal MIDI receiver will have hardware interrupt capabilities, so this function is not normally needed.

In the case of a polled-input-only type of device, the application will have to use this function to generate "`msApplMIDIIn`" callbacks. The feature bits in the info structure indicate if the device uses interrupt driven input, or polled input.

All data received on the MIDI receiver will be passed back to the application via the "`msApplMIDIIn`" callback.

Parameters:

int — A count, in microseconds, for the routine to loop before returning to the caller. This is a wait-for-data-available count, so it does not include the time used to send data to the application via the "`msApplMIDIIn`" callback.

Return Value:

None.

```
int (pascal far*msPreLoadPatch)(int,int,void huge*,long);
```

Description:

This routine receives two integers indicating the channel and patch number of an upcoming program change. The huge pointer points to a driver specific patch, assumed to have come from the driver's patch library. The length of the patch is passed in as the forth parameter. There is a special case for the third parameter which allows it to be NULL. This special case is discussed below.

The channel information, passed in parameter #1, is only used to distinguish the percussive channel (#9 - zero based) from the rest of the channels. The percussive channel is unique in that each note is really a separate patch, versus the melodic channels having only one patch that can be played on 84 notes. In order to play instruments on the percussive channel, it will be required to download each instrument patch to the driver, if it is not preloaded. This is no different that downloading patches to the melodic channels, except occurs at a higher rate and can have multiple patches coexisting on the same channel. Melodic channels only allow one patch to be active on a channel at a time.

If the patch is too large for the application to handle, it may be sent, in multiple chunks, to the driver on successive calls. To do this, the application must perform a two step process: query the driver to find the size needed for each block, then pass that portion of the patch to the driver. To receive the required patch length, the application calls the driver with a zero block length (forth parameter). In this special case, all other parameters may be NULL as well. The return value will be the required patch length size. The application now passes this much of the patch to the driver. The size required may change

between successive calls, so the application must determine the length of each block before sending in the patch data. The last block may be shorter than the size specified by the driver, which is the norm for variable length patches. The application must continue this two step process until the entire patch is downloaded. If the application cannot perform the full download, it can terminate the process by calling "msUnloadPatch" before sending the last block to the driver. If the driver cannot handle any portion of the patch download, it will return the MID_PATCHINFAIL error code and return all retained patch blocks for this patch, to the application via the "msApplFreeCB" callback.

If the driver returns a length of zero (0) for the minimum patch size, this allows the application to determine the length of each block passed in.

A MIDI transmitter/receiver, such as the MPU-401, doesn't require this call to receive patches. If the application does perform this call on MIDI transmitter/receivers, the driver will consider this another form of the "msMIDImsg" call. Any block passed in will have been stripped of the VBE/AI patch header by the application, so the rest of the data will be sent out the MIDI transmitter. The patch data may be sent out this way, as long as it conforms to the MIDI message SYSEX standard. The drivers will not provide the SYSEX headers, so it must be present within the patch data itself.

This routine may be called any time before delta time 0 to cache the patch before it is played. As a matter of fact, it is suggested that all patches be downloaded before the music sequence is played. This way, the application can be given the most consistent tempo during playback. There is a danger of the audio stalling if the application has to load patches from the library in real-time.

The "vaid" patch header data will be removed by the application before passing the patch to the driver.

NOTE: The application must always send in a non-zero value (I.E. 1 to X) for the length of data, when performing a patch download. Otherwise, if the value is zero, then the driver will just return the next block size.

NOTE: There is a special case where a patch download can pass a NULL pointer. This works with devices that support the MIDI remote load feature bit. If the device supports this bit, it will perform the patch location and downloading via it's own means. The length of the block can be set to any arbitrary non-zero value (most likely -1). The driver will perform it's work, and return success or failure, in the same way a normal download will return success or failure. This feature is added to the existing definition of this function, so the driver will still handle the normal process of patch downloading.

Parameters:

int	the MIDI channel number (zero based).
int	the patch number (zero based).

void huge *	a huge pointer to the start of the object or NULL for the special case of devices supporting remote patch loading.
long	the length of the patch block, or zero to query the next block length needed by the driver.

Return Value:

If the forth parameter is zero (0), then the driver will return the length needed for the next patch download. A return value of 0 means the application can determine the length.

If the forth parameters is non-zero, a patch, or portion thereof, is being downloaded by the application. If driver responds with zero, or with an error code of MID_PATCHINFAIL.

appropriate
error code, TRUE to this call, then
the patch memory was retained by the driver. It can be freed up by performing a "msUnload Patch" call.

```
int (pascal far* msUnloadPatch) ( int, int )
```

Description:

This routine allows the application to tell the driver to release a preloaded patch. The patch will be returned via the "`msApplFreeCB`" the standard callback mechanism.

The two parameters are channel and patch number. The channel information is used to distinguish between percussive and melodic channels. An unload request on the percussive channel will return the patch for the percussive instrument. An unload request on the melodic channels will return the matching patch number, regardless of which melodic channel the patch is associated with.

If the driver is holding multiple blocks for the patch, each block will be returned, one at a time, to the application via the callback.

This routine may be called at any time.

Parameters:

int	the MIDI channel number.
int	the patch number.

Return Value:

None. The patch pointer will be returned via the "`msApplFreeCB`" callback.

```
void (pascal far *msTimerTick) ( );
```

Description:

This function is called by the application at the timer resolution requested by the driver. (See the MIDI info structure for the resolution). If the driver does not request timer ticks, then the application may ignore this function.

The driver must use some judgment in how much time is spent in this routine. If the driver has a lot of work to do, then it must break up the load across several timer ticks. At a high resolution of 500 ticks per second, the driver may end up spending more than 2ms moving data to the device. On some slower machines, an interrupt rate of 500 times per second may consume the entire bandwidth of the computer, when running any one of today's virtual memory managers. These software packages have to virtualize the hardware interrupts, and much more, thereby creating a lot of overhead on every hardware interrupt. For this reason, The driver should not assume the caller can make more than 500 calls per second (1 every 2ms).

This is not a reentrant function call. The application is expected to handle reentrant timer interrupts in case the driver takes more than one timer tick, thus avoiding stack overflow. When the application calls this function, interrupts are assumed to be enabled.

Parameters:

None.

Return Value:

None.

```
int (pascal far *msGetLastError)( );
```

Description:

When an error condition occurs, the driver will post an error number internally. This error code can be read by the application through this function call.

The driver will zero out the posted error after the application has read it.

Parameters:

None.

Return Value:

The last error number:

```
0x01 MID_NOSUPPORT   The driver was asked to perform an
                     unsupported function
0x02 MID_UNKNOWPATCH  The driver doesn't understand
                     the patch.
0x03 MID_ALLTONESUSED all tones are active, no more
                     can be played
0x04 MID_BADMIDIMSG  messages lost a channel msg.
0x05 MID_PATCHINCOMP  The application passed in an
                     incomplete patch.
0x06 MID_PATCHINFULL  preload patch list is full.

0x07 MID_BADLOSTIRQ   No MIDI-in callback - data lost
0x08 MID_PATCHINFAIL  The driver is failing a patch
                     download.

0x80 MID_HWFAILURE    Generic hardware failure msg
```

Error messages 0x80 - 0xEF are vendor specific. 0x00-0x7F and 0xF0-0xFF are reserved by VESA.

6.Volume Control Services

6.1.Introduction

The Volume control API structure has been defined for supporting both volume and mixer devices. In the following section any reference to volume devices will include mixer. All access to the volume information and services structures is performed via subfunction #2, Query Device Class Info queries.

6.2.Theory of Operations

The API for volume and mixer controls are considered a non-exclusive device, that is, may be accessed by more than one application at a time. This allows for mixer TSR applications to control the volumes of devices at any moment in time - a requirement when the foreground application does not provide such control.

The volume settings for the function calls are always expressed in linear whole numbers. Within the volume driver, no runtime data should ever be stored in the applications memory, since a close call is not be required by the application.

6.3.Volume Info Structure

```
typedef struct {

     // housekeeping

     char viname[4];     // name of the structure
     long vilength;      // structure length

     // hardware vendor name

     long viversion;     // driver code version
     char vivname[32];   // vendor name
     char viprod[32];    // vendor product name
     char vichip[32];    // vendor chip/hardware description
     char viboardid;     // installed board #
     char viunused[3];   // unused 3 bytes

     char vicname[24];   // text name of the mixer channel

     long vifeatures;    // feature bit field

     unsigned int vimin; // minimum volume setting
     unsigned int vimax; // maximum volume setting
```

```
    unsigned int vicross;  // attenuation/gain crossover

} VolumeInfo, far *fpVolInfo;
```

6.4.Volume Info Structure field descriptions

```
char viname[4];     // name of the structure
long vilength;      // structure length
```

These two fields are for structure ID only. Vendor name, and other specific information is stored in other fields.

```
long viversion;
```

This field is the software version number of the driver. It's value is defined by the vendor. The data will be the BCD format.

```
char vivname[32];
```

This field is an ASCIIZ text string containing the vendor name, copyright, etc. The NULL terminator is part of the declared size.

```
char viprod[32];
```

This field is an ASCIIZ text string for the board level product name. The NULL terminator is part of the declared size.

```
char vichip[32];
```

This field is an ASCIIZ text string for the device specific name, such as "MVA508 Mixer". The NULL terminator is part of the declared size.

```
char viboardid;
```

This field defines the board # installed in the machine. Some hardware allows for multiple card of the same type to be installed in the same machine. This field identifies which board number this driver is addressing.

```
char vicname[24];
```

This field is an ASCIIZ string containing the channel name. The NULL terminator is part of the declared size.

```
long vifeatures;
```

This field holds the feature bits, currently defined as:

`0x00000001 Stereo Volume control available`

Stereo left/right volume control is available.

`0x00000004 Low Pass Filter control is available`

A low pass filter settings is programmable on this device through the "vsFilterControl" API call.

`0x00000008 High Pass Filter control is available`

A high pass filter settings is programmable on this device through the "vsFilterControl" API call.

`0x00000010 Parametric Tone control is available`

Tone sliders are available on this device. Query via device check to determine the range of settings.

`0x00000020 Selectable output paths`

The output of this volume control is selectable for recording purposes. One path is for playback only; the other for record and/or playback. On the record path, playback is not assumed, but is allowable, so the user can hear what the audio sounds like during recording.

`0x00000100 Field position of Azimuth supported.`

Horizontal sound positioning is available beyond simple panning found in left/right volume controls, i.e., greater than 180 degrees. This feature bit would be set for devices supporting Surround Sound, Qsound, etc.

`0x00000200 Field position of Phi supported.`

Vertical elevation sound positioning is available.

`0x80000000 Master control device`

This bit just indicates that this volume control is the master volume device for the board.

6.5.Volume Services Structure

```
typedef struct {

     // housekeeping

     char vsname[4];     // name of the structure
     long vslength;      // structure length

     char vsfuture[16];  // reserved for future use

     // Volume/Mixing Services

     long (pascal far *vsDeviceCheck)  ( int,long );
     long (pascal far *vsSetVolume)    ( int,int,int );
     long (pascal far *vsSetFieldVol)  ( int,int,int );
     int  (pascal far *vsToneControl)  ( int,int,int,int );
     long (pascal far *vsFilterControl)( int,int,int );
     void (pascal far *vsOutputPath  ) ( int );
     void (pascal far *vsResetChannel) ( );
     int  (pascal far *vsGetLastError) ( );

} VolumeService, far *fpVolServ;
```

6.6.Volume Services Structure field descriptions

```
long (pascal far *vsDeviceCheck  )( int,long );
long (pascal far *vsSetVolume)    ( int,int,int );
void (pascal far *vsSetFieldVol  )( int,int,int )
void (pascal far *vsToneControl  )( int,int,int,int );
void (pascal far *vsFilterControl)( int,int,int );
void (pascal far *vsOutputPath   )( int );
void (pascal far *vsResetChannel )( );
int  (pascal far *vsGetLastError )( );
```

These functions provide all the volume/mixer controls available to the applications.

6.7.Volume Driver Functions

```
long (pascal far *vsDeviceCheck)( int, long );
```

Description:

This function is a conglomerate service to provide the application a call to derive all sorts of information about the device. The first parameter is a message number; the second, a 32 bit parameter to the message. Device checks of value above 0x80 are vendor specific, defined by each vendor.

`message VOLFILTERRANGE (0x11)`

This message allows the application to query the device for the number of cut-off points in the low or high pass filter.

Parameter:

1 for the low pass filter
2 for the high pass filter.

Return Value:

LOWORD - a count of cut-off points, from 0 to 65535.

`message VOLFILTERSETTING (0x12)`

This message allows the application to query the driver for information about each one of the filter settings, starting from the first to the last, one per call. In order to start from the beginning, perform the VOLFILTERRANGE query to select which filter to search. All subsequent calls to this query will return the cut-off point numbered from 0 to 65535, as well as, the cut-off point in Hertz.

Parameter:

none

Return Value:

LOWORD - the current cutoff point number.
HIWORD - the current cutoff point, in Hertz.

`message VOLFILTERCURRENT (0x13)`

This message allows the application to query the current setting of the low or high pass filter.

Parameter:

1 for the low pass filter
2 for the high pass filter.

Return Value:

LOWORD - the current cutoff point number, from 0 to 65535.
HIWORD - the current cutoff point, in Hertz.

`message VOLTONERANGE (0x14)`

This message allows the application to query for the number of Parametric Equalizers controls supported by the hardware. Each equalizer is numbered so the application can query the current state of each one individually using the "VOLGETTONESETTING" device check.

Parameter:

none.

Return Value:

The count of equalizers, from 0 to 65535

`message VOLTONESETTING (0x15)`

This message allows the application to query the driver for information about each one of the parametric equalizer settings, starting from the first to the last, one per call. In order to start from the beginning, perform the preceding VOLTONERANGE query to reset the starting point of the search. All subsequent calls to this query will return the center frequency, boost cutoff, and width (Q) of the equalizer.

Parameters:

none

Return Value:

LOWORD - the center frequency of the equalizer, in Hertz.
HIWORDLOWBYTE - the maximum width of the equalizer, in increments of 50hz, so a returned value of 5 would be 250Hz in band width.
HIWORDHIBYTE - The maximum boost cut setting, from 0 to 256, where a center of 127 is zero attenuation. All settings over 127 imply gain.

`message VOLTONECURRENT (0x16)`

This message allows the application to query the state of a given parametric equalizer setting.

Parameter:

int equ The equalizer number, as determined by DeviceCheck VOLTONESETTING enumeration. Refer to the "VOLTONERANGE" device check for more info.

Return Value:

LOWORD - the center frequency of the equalizer, in Hertz
HIWORDLOWBYTE - the current width of the equalizer, in increments of 50hz, so a returned value of 5 would be 250Hz in band width.
HIWORDHIBYTE - The current boost cut setting, from 0 to 256, where a center of 127 is zero attenuation. All settings over 127 imply gain.

`message VOLPATH (0x17)`

This function returns the current output path connection. This allows applications to query if the volume device is in a record or playback state.

Parameter:

none

Return Value:

FALSE if connected to the playback path.
TRUE if connected to the record path.

NOTE: this setting may change automatically if the record path connection is an exclusive connection between volume devices. This means that one volume device, when switched to the record path, may automatically disconnect another volume device from record path, by forcing it back to the playback path.

`message VOLGETIOADDRESS (0x18)`

This message allows the application to find out the device's base I/O address. NOTE: This is provided as information only; the application is not expected to touch the hardware. Doing so will cause unpredictable results.

Parameter:

none.

Return Value:

LOWORD - The base I/O address of the card.
For MPU-401 devices, this might be 0x330, or 0x300.
For the early Adlib Music Synthesizer card, this might be 0x388.

```
long (pascal far *vsSetVolume)( int, int, int );
```

Description:

Sets the overall maximum volume setting for this mixer channel or volume device. There are two basic controls embedded in this call. One control is determined by the user, the second by the application. This allows the user to control the volume range with which the application can move. The user setting is an integer value, from 0 to 100, which the driver interprets as a multiplier of 0.00 to 1.00. The applications setting can range anywhere between VMIN and VMAX. All settings up to VCROSS start at full attenuation (-db) up to zero attenuation. All settings over VCROSS implies gain is now applied to the signal.

The default state of the volume can be determined by the hardware. The user volume setting always defaults to 100 as its initial value.

Parameters:

int ctrl This parameter determines one of two settings for the volume. One is for user control, the other is for the application:

```
#define USERSETTING  0x01
#define APPSETTING   0x02
```

The user settings will pass in a value of 00 through 100, which is used as a multiplier as a multiplier to the applications setting. The application will pass a setting between VMIN and VMAX.. The formula to calculate the final volume is:

setting = vol(a * u)

where: a is the applications setting
u is the user's multiplier.

int left This is the new left, or mono channel setting. A value of negative one (-1) is not set, but allowable to let the application read the current setting for either the user or applications setting.

int right This is the new right channel setting. A value of negative one (-1) is not set, but allowable to let the application read the current setting for either the user or applications setting.

Return Value:

HIWORD - the last right channel setting.
LOWORD - the last left channel setting.

```
void (pascal far *vsSetFieldVol)( int, int, int );
```

Description:

This function uses a spherical, i.e., polar, coordinate system to position the channel's sound in 3D space. The application of this volume control is added to the base setting performed by the "vsSetVolume" function. This allows the "vsSetVolume" control to be used as the master control, and the 3D sound used as an expression control.

The three coordinate values are:

Radius - The distance of the sound from the listener.
Theta, or Azimuth - the horizontal angle, in degrees, from the listener to the sound source.
Phi, or Elevation - The vertical position for an elevation que.

Radius has a range of zero to 512. This control establishes the volume from the sound, whereas; the other two controls establish the position.

Theta is the horizontal angle around the listener. An angle of 0 is directly in front of the listener. An angle of +90 is directly to the right, whereas; an angle of -90 is directly to the left. An angle of +180 or -180 is directly in back of the listener.

Phi determines the vertical elevation of the sound source. Negative 90 degrees is directly below the listener, all the way up to positive 90 degrees, being directly overhead.

Parameters:

int Radius	for distance of the sound from the listener (0 through 512)
int Theta	horizontal angle, in tenth of degrees. (0 through 1800)
int Phi	Phi is the vertical angle, in tenth of degrees.. (-900 through +900)

Return Value:

none

```
void (pascal far *vsToneControl)(int,int,int,int);
```

Description:

Sets the tone control setting for the given band.

Parameters:

int equ	The equalizer number, as determined by DeviceCheck VOLTONESETTING enumeration. Refer to the "vsDeviceCheck" function for more information.
int center	For parametric equalizers with programmable center frequencies, value will become the new center. The value is in Hertz. For non programmable device, this value is ignored.
int boost	This is the attenuation/gain control of the tone control device. A value of 0 is full attenuation, up to 127 of zero attenuation 128 to 256 imply gain settings.
int width	Also known a 'Q'. this is the width in 50Hz increments. A value of 0 through 255 gives a width of 0 to 12750Hz.

Return Value:

none

```
void (pascal far *vsFilterControl)(int, int, int);
```

Description:

Sets the new low, or high pass filter setting

Parameters:

int select	1 selects the low pass filter, 2 selects the high pass filter.
int cutoff	This is the cutoff frequency, of the filter, in Hertz. This is equivalent to the parametric equalizers center frequency.
int boost	This is the attenuation/gain control of the tone control device. A value of 0 is full attenuation, up to 127 of zero attenuation. 128 to

256 imply gain settings.

Return Value:

none

```
int (pascal far *vsOutputPath)( int );
```

Description:

For some mixers/volume devices that support selectable outputs, this function will switch the output path to one of two outputs. One path is assumed to be for playback only; the other for record and/or playback. On the record path, playback is not assumed, but is allowable, so the user can hear what the audio sounds like during recording. Since some mixers do not allow multiple paths to be connected to the record path at one time, it is up to the device to query the connection state of related volumes to determine the relationship.

Parameters:

int output path selection. A parameter value of FALSE selects playback. A parameter value of TRUE selects record.

Return Value:

int - the old setting.

```
void (pascal far *vsResetChannel)( );
```

Description:

This function resets all the volume, filter, and equalizers, back to a default state. Only the user setting will not be reset. If this needs to be reset as well, then the application can make a subsequent call to "`vsSetVolume`" with a new user setting of 100.

Parameters:

None.

Return Value:

None.

```
int (pascal far *vsGetLastError) ( );
```

Description:

Returns the last error number.

Parameters:

None.

Return Value:

Returns the last error code:

```
0x01 VOL_NOSUPPORT  Unsupported feature/function
                    was requested by the application
0x02 VOL_BADVALUE   A parameter value was out of range
0x80 MID_HWFAILURE  Generic hardware failure msg
```

Error messages 0x80 - 0xEF are vendor specific. 0x00-0x7F and 0xF0-0xFF are reserved by VESA.

7.VESA Audio Data Formats

7.1.MIDI Patch Library Format

Proposed patch library format, conforming to Microsoft's LIST chunk format definitions:

```
<"vail">,<LEN> {
     <"ZSTR">,<LEN>,<DATA>                // copyright, etc
        "       "     "                      "
     <"vaip">,<LEN> {                     // patch pointers
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 0.
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 1.
          <"vaii">,<LEN>,<OFFSET>,<DLEN>  // to patch 2.
                    "       "      "       "         "
                    "       "      "       "         "
          <"vaid">,<LEN>,<DATA>           // patch 0 data
          <"vaid">,<LEN>,<DATA>           // patch 1 data
          <"vaid">,<LEN>,<DATA>           // patch 2 data
             "       "     "                      "
             "       "     "                      "
     }
}
```

The actual file format will have the <"vail"> chunk embedded in a <"RIFF"> chunk, so the actual file contents will be:

```
     <"RIFF">,<LEN> {
          <"vail">,<LEN> {
               ....
               ....
          }
     }
```

The patch file format has four VBE/AI list chunk types defined. The "vail" chunk begins a nested structure of chunks. Within the "vail" chunk, the first chunk type encountered may be the Microsoft "ZSTR" chunk. Any number of "ZSTR" chunks may appear before the first "vaip" chunk. The "ZSTR" chunk type is the standard Microsoft chunk for holding ASCIIZ strings. These messages contain any relevant information, such as copyright, etc.

The next chunk type, "vaip" begins the actual patch library. Within this chunk type, are two other chunks, "vaii", and "vaid". The "vaii" is the VAI Index containing an <OFFSET>, to the corresponding "vaid", or VAI Data chunk. Also in the "vaii" chunk, the <DLEN> field holds the length of the corresponding "vaid" <DATA> field length. The <OFFSET> is a byte count starting from the "vaip", file position, inclusive. To calculate

the file position of the first "vaid" chunk, take the file position of the "v" in the "vaip" chunk, plus the <OFFSET> in the first "vaip" chunk.

NOTE: The patch library format is a different file format, not associated with the Microsoft General MIDI file format. The MS General Midi file format introduces a form of hardware dependencies in the file format by predefining the channel assignments for certain levels of hardware feature sets. Only IMA General MIDI channel assignments are recognized by the VBE/AI MIDI devices.

7.2.MIDI Patch Data Formats

The VBE/AI Patch Data Format will have a simple structure of patch type, then followed by patch data.

```
<PATCH DATA> := <PATCH TYPE>,<PATCH DATA>
```

The patch type will be a 16 bit word containing the registered patch type ID.
The patch data is a variable length block of data, defined by the hardware vendor.

7.2.1.OPL2 Patch Definition

```
typedef struct {
     char opl2ksl;
     char opl2freqMult;
     char opl2feedBack;  // used by operator 0 only
     char opl2attack;
     char opl2sustLevel;
     char opl2sustain;
     char opl2decay;
     char opl2release;
     char opl2output;
     char opl2am;
     char opl2vib;
     char opl2ksr;
     char opl2fm;        // used by operator 0 only
} opl2opr;

typedef struct {
     int  patchtype;     // indicates the patch type
     char opl2mode;      // 0 = melodic, 1 = percussive
     char opl2percVoice; // if mode=1, voice # to be used
     opl2opr opl2op0;    // Operator 0 specifics
     opl2opr opl2op1;    // Operator 1 specifics
     char opl2wave0;     // waveform for operator 0
     char opl2wave1;     // waveform for operator 1
```

```
} OPL2native;

typedef struct {
     VAIDhdr   vbehdr;   // the "vaid" chunk header
     OPL2native native;  // native OPL2 patch data
} OPL2patch;
```

7.2.2.OPL3 Patch Definition

```
typedef struct {
     int  patchtype;     // indicates the patch type
     char reg20h[4];     // a set of four registers
     char reg40h[4];     // for each of the 4 operators.
     char reg60h[4];
     char reg80h[4];
     char regE0h[2];
     char regC0h[2];
} OPL3native;


typedef struct {
     VAIDhdr   vbehdr;   // the chunk header
     OPL3native native;  // native OPL3 patch data
} OPL3patch;
```

7.3.WAVE Audio Data Formats.

The VBE/AI specification only defines uncompressed, 8 and 16 bit linear PCM data formats. Other formats may be included as needed via the VBE/AI compression registry. Vendors may add compressed data formats via this registry. Contact the VESA offices for more details.

The table below shows the layout of the mono and stereo data streams, from left to right. The notation for the samples is given here:

S# is the mono sample number.
L# is the left channel sample number
R# is the right channel sample number.
lsb is least significant byte of the 16 bit sample.
msb is most significant byte of the 16 bit sample.

```
Memory location     0       1       2       3.......>>
```

Mono 8 bit	S#0	S#1	S#2	S#3.....>>
Stereo 8 bit	L#0	R#0	L#1	R#1.....>>
Mono 16 bit	S#0lsb	S#0msb	S#1lsb	S#1msb..>>
Stereo 16 bit	L#0lsb	L#0msb	R#1lsb	R#1msb..>>

8. TSR Implementation Issues.

8.1.Common Command Line Parameters

For the purposes of a common user interface, each VBE/AI driver will process the following set of command line parameters. This doesn't mean the command line is applicable, just that the driver responds to each one in an appropriate manner.

```
/H        Helps
/?        helps
/B:xx     Board #
/A:xx     board Address
/D:xx     Dma channel
/I:xx     Irq channel
/MB:xx    Memory Base (segment)
/MR:xx    Memory Range
/P:xx     user Preference setting
/U        Unload this driver from memory
/V        Verify if Hardware is present
```

Here are a few rules for defining how to process the command line switches:

1. The switches will be order independent, meaning each one will be processed in the order found on the command line. If a switch is duplicated, the second switch will override the first switches settings.

2. The leading '/' or '-' will be optional.

3. The numeric value following the switch, designated as 'xx', will be in hexadecimal.

4. The Parameters will be case insensitive, so '/P:xx' and '/p:xx' will be processes as the same switch.

5. Since multiple APIs may be present per TSR, a way must be provided to direct switch settings to each API. Each API will be enumerated internally, from 0 to X, in single increments. In such a case, the TSR should print to the screen a list of it's enumerated services, following it's logo and copyright statement. When the 'xx' is listed as a parameter, it may have a second parameter following, separated by a comma. This second parameter will be the enumerated service number. The parameter would then look like:

   "/P:xx,xx "

6. The /V switch is a special purpose switch to verify if the hardware is present; the driver will not stay resident. All other command line switches are still valid except the /U switch. The driver will use any present switch, in its context, when determining if the hardware is present. If the hardware is not present, the driver will terminate with a return code of zero (0). If the hardware is present, the driver will terminate with a return code value of one (1). If the hardware is present, and has a VBE/AI driver, the driver will terminate with a return code of two (2). This return code is detectable via the DOS ERRORLEVEL.

9. Application Guidelines.

9.1.Minimum requirements for VBE/AI compliance.

9.1.1.General rules for Device Support.

When selecting a device, the application should perform the selection upon the following order:

1. Does the device have the right features?
2. Does this device have the highest user preference?
3. Is this device currently available? (i.e. not already opened.)

In the VBE/AI architecture, the application owns the system clock timer. No VBE/AI driver is allowed to directly commander the timer for it's own use. Therefore, the application is responsible for giving the driver timer tick callbacks at the requested intervals. Providing the necessary timing for VBE/AI drivers is a requirement of the application, to be VBE/AI compliant.

9.1.2.Wave Device Support.

Callback from the VBE/AI device to the application may occur at interrupt time, so the application must perform, fast, limited work. The application may perform such work as calling the driver to start the next audio block, but using callback to perform disk I/O is a bad practice for time sake and DOS re-entrancy issues.

The "wsPlayBlock" routine allows the application to play short, random sounds. For audio "streams" and for synchronization purposes, the "wsPlayCont" function should be used. The application receives callbacks at more accurate fixed intervals.

9.1.3.MIDI Device Support.

It is the responsibility of the application to handle the downloading of patches from the disk resident patch library, to the driver. The application should handle large patches using the block passing protocol defined in the "msPreLoadPatch" function call.

In order to have all the instrument patches ready for playing, it is suggested that application first download all needed patches before starting a sequence. This would avoid the situation of needing a patch, and not being able to download it fast enough to keep the tempo in time.

9.2.WAVE Audio Full Duplex Operation.

Full duplex is defined as having the ability to play and record simultaneously. This capability is an upcoming requirement in video conferencing where two way communication is a must. For the application using VBE/AI, this can be accomplished by two approaches. The application can open two WAVE audio devices, using one for record, the other for playback. The second approach is similar, but only one device, which supports full duplex, is opened, and used for both record and playback.

Technically, the VBE/AI design for full duplex allows the application to deal with both approaches in a simple manner. Since using the second approach of using one API for both record and playback functions presents the most problems, these guidelines are developed around the limitations of the full duplex hardware. In order handle both approaches using one coding scheme, settling on the limitations will allow for wider hardware support.

The VBE/AI single device full duplex operation requires the block sizes of the WAVE data to be the same. Also, the sample rate for both play and record must be the same rate. The following discussion addresses these limitations.

For the easiest full duplex operation, VESA requires the use of "wsPlayCont" and "wsRecordCont". This allows the application to handle arbitrary sized WAVE blocks for both incoming and outgoing WAVE data. Since the application owns the buffer, it is responsible for managing the data. The VBE/AI drivers are only responsible for moving the data from the buffer, to the hardware, and visa versa. Using the "wsPlayBlock" and "wsRecordBlock" functions is not possible in the scope defined in the VBE/AI specification.

Addressing the single sample rate issue, this normally will not be a problem. In the case of having to run disparate sample rates, the application will have to use some simple real-time techniques to down sample the data to the chosen hardware rate. Since the application owns the buffer, down sampling can be performed as the data is entered into the buffer.

10. Trademarks.

Adlib Music Synthesizer Card is a trademark of Adlib Multimedia, Inc.
Adlib Gold 1000 is a trademark of Adlib Multimedia, Inc.
Audio Port is a trademark of Media Vision, Inc.
General MIDI is a trademark of the International MIDI Association.
MPU-401 is a trademark of Roland, Inc.
OPL2 is a trademark of Yamaha Systems Technology.
OPL3 is a trademark of Yamaha Systems Technology.
Pro Audio Spectrum is a trademark of Media Vision, Inc.
Qsound is a trademark of Archer Communications, Inc.
Sound Blaster is a trademark of Creative Labs, Ltd.
Sound Blaster Pro is a trademark of Creative Labs, Ltd.
Sound Canvas is a trademark of Roland, Inc.
Sound Source is a trademark of Disney, Inc.
Surround Sound is a trademark of Dolby Labs, Inc.
Thunderboard is a trademark of Media Vision, Inc.