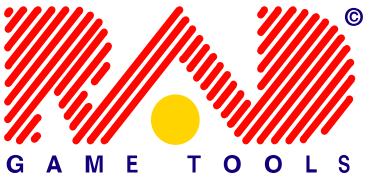
Miles Sound System SDK 6.1c



Acknowledgments

The Miles Sound System was written and designed by John Miles and Jeff Roberts. Special thanks also to Mitch Soule, Alicia Fukunaga, and Casey Muratori!

Additional programming by Dan Teven. Filter programming by Nick Skrepetos. Macintosh version originally ported by Software MacKiev <picture:kiev>(Volodya Prokurashko, Anatoly Klimashevsky, Anton Turov, Lena Alexandrova, Andrey Markin).

The Extended MIDI (XMIDI) standard was designed in collaboration with the musicians, programmers, and developers at Origin Systems in Austin, Texas. In particular, we gratefully acknowledge the expert assistance of Martin Galway, Dana Glover, Chris Roberts, Marc Schaefgen, Gary Scott Smith, Nenad Vugrinec, and Kirk Winterrowd in the evolution of the XMIDI standard. Thanks also to Ken Arnold and George ("The Fat Man") Sanger for their indispensable advice and support.

RAD's automatic documentation system was written by Casey Muratori. Typesetting by the TeX and LaTeX typesetting systems.

Introduction

Welcome to the Miles Sound System!

The Miles Sound System has been the standard in video game audio for almost ten years now. Ten years! It always surprises me to remember that Miles started out before Windows 3.0, before the Mac went to PowerPC, way before even the plain old Pentium chip!

Somehow, it just makes sense that on its tenth anniversary in 2001, Miles will be shipping on more platforms than ever! We've added MacOS support for the surprisingly resurrected Macintosh market and we're working on console platforms for next year. I guess I shouldn't be surprised to see Miles still going strong in other ten years!

After ten years of development, Miles has incorporated pretty much everything that related to game audio. Name a technology and it is in there somewhere - digital, MIDI, 3D, DSP, MP3, redbook and more!

This documentation will help you get a handle on these various APIs. We spend quite a bit of time working on our docs and hope they get you up and running fast. Please give us feedback on what worked for you and what didn't. Just as with the Miles API itself, we strive for continual improvement in our documentation as well. Your feedback is crucial in this regard!

Documentation is a pain to read when you just want to get up and running quickly. RAD realizes this, so we've retooled our docs to be more focused on specific tasks and problems. Really, the only documentation that you should force yourself to read is the **Overview** for your game's platform, and the "**FAQs and How Tos**" *Section*.

The "FAQs and How Tos" section is the best way to learn about using Miles - there are lots of "getting started" FAQs as well as tons of hard won trouble-shooting FAQs as well.

I love these new FAQs actually - I think it's a much more effective way to learn the ins and outs of a product than poring over a reference manual - I hope you like them too!

Also, note that our printed documentation doesn't include the Miles Reference Guide (function, type and preference descriptions - over 600 additional pages). Check out the online documentation for fully hyper-linked and cross-referenced discussions - it's awesome!

Again, please send feedback to me as you use Miles - what confused you, what you liked, what you hated, what you'd like to see in future versions, etc. Your feedback is what keeps Miles vital, fun to use, and powerful after all these years!

Thanks,

Jeff Roberts - RAD Game Tools - E-mail: miles@radgametools.com.

Contents

Chapter 1

Overview for Win32

1.1 Overview for Win32: Miles on Windows 95, 98, Me, NT and 2000

Discussion

Miles is most commonly used on the Win32 platform. Win32 is both wonderful and horrible (often simultaneously).

Miles is extremely easy to use and integrate under this platform. We'll discuss everything you need to know to get up and running!

1.2 Installation for Win32

Discussion

The Miles installer will be launched automatically when you insert the Miles SDK CD into an auto-play capable machine. If auto-play is not enabled on your machine, you can run the installation program manually by executing Miles.exe in the root directory of the Miles SDK CD.

Once the installer has launched, you will be presented with three installation options:

1) Install the Miles Sound System, which will install the files you need to build programs that incorporate Miles.

2) Install RAD Video Tools, which will install the tools used to compress video files into Bink or Smacker formats (as well as a bunch of cool sample videos).

3) Install Granny demos, which will install demonstrations of RAD's 3D animation product.

All installation options are largely automatic, and will not require much further user input unless you would like to change anything (like the installation directory).

1.3 Integrating Miles into a Win32 Build Environment

Discussion

To build an application that uses Miles, you must include the Miles SDK in the build process. There are four simple steps to do this:

1) Include **MSS.H** in your source files.

2) Link with the **MSS32.LIB** import library.

3) Place the **MSS32.DLL** file in the directory that contains your application.

4) Copy the all of the other files in the **\redist\win32** directory into a subdirectory off your application directory.

And that's it - you should now be able to call the Miles functions!

1.4 Distributing Miles with Win32 Applications

Discussion

To distribute an application that uses Miles, you must include the Miles DLL (**MSS32.DLL**) in the same directory as your application, and you must copy all of the other files in the **\redist\win32** directory to a directory accessible by your application at run-time.

Do **not** place the **MSS32.DLL** in the Windows or the Windows System directories! The **MSS32.DLL** *must* be placed in your application directory! The Miles DLL will refuse to be loaded from a system directory.

1.5 Sound under Win32

Discussion

For MIDI output, the Miles Sound System can use the system MIDI hardware through the midiOut API, or it can synthesize the MIDI stream itself and feed it through the digital output system. MIDI is pretty a simple environment.

For digital audio output, however, things are a little more complicated. Miles can be configured to use either DirectSound (the preferred method) or the old waveOut audio system.

DirectSound shipped about a year after Windows 95 first shipped. It has been built into Windows 98, Windows Me, and Windows 2000. NT 4 with service pack 3 also supports DirectSound, but only in *emulated* mode.

DirectSound can run in *native* mode or *emulated* mode. Native mode means that DirectSound is talking directly to your sound card hardware. Emulated mode means that DirectSound is talking to old waveOut audio drivers (through an extra layer of buffering).

DirectSound in emulation mode usually works terribly. To avoid this (and to support old Win95 and old NT machines that don't have DirectSound at all), Miles can use the old waveOut sound system directly. WaveOut was the

original way to play sound in Windows (it was added to Win16 way back as an add-on to Windows 3.0 - yes, audio was once considered a chargeable "add-on" by Microsoft).

WaveOut mode is nice, because it will work on pretty much any machine, but, unfortunately, it has serious latency problems. These latency issues were, in fact, the primary reason Microsoft added DirectSound.

Under waveOut, latency can rise to as high as 300 milliseconds on some older sound cards. DirectSound's latency, by comparison, rarely hits even 50 ms.

In practice, we've found that you usually don't have to worry about supporting old sound cards through waveOut, because a game's graphics usually require the latest Direct3D and/or DirectDraw anyway (so you know DirectSound will be installed).

So, by default, to deal with all of these issues, Miles tries to use DirectSound first, and then falls back to waveOut mode if DirectSound won't load or if it is running in emulation mode. So, you get DirectSound's good latency if running in native mode, and you still get waveOut's compatibility when necessary.

1.6 Memory management under Win32

Discussion

Under Win32, memory management is pretty easy - you can pass in any plain old memory pointer for Miles to process. If you want your memory allocations to be compatible with MacOS or DOS, though, you should use the AIL_mem_alloc_lock() function. If porting isn't an issue, though, just use whatever memory allocator is convenient (malloc, resource locks, whatever).

For internal memory use, Miles allocates its own memory directly from the operating system.

1.7 Callbacks under Win32

Discussion

Read this section carefully if you plan to use Miles timer or other callbacks (such as an end-of-sample callback). If you don't intend to use callbacks, then this section won't apply to you.

Under Win32, callbacks are usually made to your application on a background thread. The ideal callback function is one that does nothing more than writes to one or two flag variables and then immediately returns back to Miles. Under Win32, other operations are *allowed*, but not recommended!

So, unlike MacOS or DOS, you can do whatever you want inside a Miles callback under Win32 - there are no OS-mandated restrictions. However, if you want to maintain easily ported code, then you should read about the callback restrictions for the other platforms you intend to support. Other platforms are much more strict than Win32 when it comes to callbacks.

In any case, you do have to take general multithread precautions in your callbacks. For example, if you use the C runtime library inside a callback, then you need to link with the multithreaded libraries. If you access your own internal data structures, then you need to make sure that the accessor functions can be called by two threads at once (unless you protect with a critical section lock).

Also remember that all global or static variables that are modified by a callback function must be declared with the *volatile* modifier. Otherwise, the callback function and the foreground may not be able to communicate after the compiler has performed all of its optimizing tricks.

You should also minimize the total amount of work inside a Miles callback. During a callback, Miles is suspended, so other background tasks won't run - mixing will be interrupted, for example, if you take too long.

Again, though, the best advice for callbacks is to simply post a message or set a flag that can be operated on by your foreground task. This technique works on all platforms and you'll never need to worry about the complications that multithreading can cause.

1.8 Sharing the CPU under Win32

Discussion

Under Win32, Miles does most of its work in a background thread. This means that Miles usually does its work without you even having to worry about it. The mixing and buffer submission to the sound hardware all happens without your intervention.

The only time you may need to specifically give Miles CPU time is if the system thread scheduler starts performing badly and the background Miles thread starves. Under the Windows 95, 98, and Me, the thread scheduler usually has the most trouble when performing disk I/O.

Miles provides several techniques to handle thread-starvation. See the AIL_get_timer_highest_delay() function as well as the "My sound is skipping - what can I do?" *FAQ* for more details.

1.9 Call Logging under Win32

Discussion

Miles contains a powerful diagnostic feature that allows developers to create a timestamped list of all of their application's foreground Miles calls, along with their parameters and results. This list is written to a plain text file as an aid in tracking down hard-to-find application bugs.

To take advantage of MSS's built-in logging, create an "[**MSS**]" section in **WIN.INI**, followed by a string of the form, "**MSSDebug=filename.txt**". During execution of all subsequent Miles calls, a timestamped API call log will be written to the specified file.

For example, to dump the API call log to a file name **foo.txt**, add the following lines to **WIN.INI**:

[MSS]
MSSDebug=foo.txt

You can also view MSS's internal function calls by adding an extra line reading, "**MSSSysDebug=1**".

1.10 Miles Examples for Win32

Discussion

The Miles SDK supplies several example programs for Win32. To create these examples in an IDE-based environment, just create a new project and include the appropriate C source file (**Demo.C**, for example), and then add the Miles import library to the project (**MSS32.LIB**). The Win32 example files are found in the **\examples\win\src** and **\examples\console\src** directories.

For an example that uses digital sound, MIDI sound (with DLS), and CD (redbook) audio, see the **Demo.C** file.

For an example that uses the Quick API, see the **Quick.C** file.

For an example that uses the 3D digital audio API, see the **Exam3D.C** file.

For an advanced example of Internet voice chat, see the **MSSChat.C, MSSChtC.CPP, and MSSChtS.CPP** files.

For simple, console-based examples of digital audio playback, MIDI playback (hardware based and DLS), DSP filters, and more, see the examples in the **\examples\console\src** directory.

Chapter 2

Overview for MacOS

2.1 Overview for MacOS: Miles on Macintosh

Discussion

The Miles Sound System now supports MacOS! MacOS is strange to develop under - some pieces are advanced, some pieces seem comically backwards. Miles was developed on MacOS 9 - MacOS X should ship soon and is expected to be a much, much better development environment.

Miles is easy to use and integrate under this platform. This chapter will discuss everything you need to know!

2.2 Installation for MacOS

Discussion

The Miles SDK is provided as a self-extracting compressed archive in HQX format. To install it, simply double-click on the **MilesSDK.hqx** file on the Miles SDK CD.

Once installed, you can access this documentation online by clicking on the **Miles SDK Help** alias in the Miles SDK folder.

2.3 Sound under MacOS

Discussion

For digital output, the Miles Sound System uses the Sound Manager. Miles allocates a single sound buffer, and then mixes all of its sound channels into that single buffer.

For MIDI output, you will usually use MSS's internal DLS software synthesizer, but Miles can also use QuickTime as a MIDI output layer (QuickTime uses a *lot* more CPU time, though).

2.4 Integrating Miles into a MacOS Build Environment

Discussion

To build an application that uses MSS, you must include the MSS SDK in the build process. There are four simple steps to do this:

1) Include **MSS.H** in your source files.

2) Link with the **Miles Shared Library** file (in CodeWarrior, just add the shared library into your project).

3) Place the **Miles Shared Library** file in the folder containing your application.

4) Copy the other files in the **::redist::mac** folder into a subfolder off your application's folder.

And that's it - you should now be able to call the Miles functions!

2.5 Distributing Miles with MacOS Applications

Discussion

To distribute an application that uses Miles, you must include the **Miles Shared Library** in the same folder as your application, and you must copy the other files in the **::redist::mac** folder to a folder accessible by your application at run-time.

Do **not** place the **Miles Shared Library** in the System or Extension folders! Miles **must** always be placed in your application folder.

2.6 Memory management under MacOS

Discussion

By default, Miles handles memory management under MacOS by calling the MacOS memory functions directly. You can, however, override the Miles allocation and freeing functions with the AIL_mem_use_malloc() and AIL_mem_use_free() functions.

By default, Miles tries to allocate memory in this order:

1) Miles first tries to allocate memory from the application's memory partition.

2) Next, Miles tries to allocate memory from the MultiFinder temporary memory block.

3) Finally, Miles will try to allocate memory from the temporary system memory pool.

Using this set of memory fallbacks, Miles will usually find the memory it needs no matter where it may be hiding.

2.7 Callbacks under MacOS

Discussion

Read this section carefully if you plan to use Miles timer or other callbacks (such as an end-of-sample callback). If you don't intend to use callbacks, then this section won't apply to you.

Under MacOS, callbacks are usually made to your application on an interrupt-based system callback. The ideal callback function is one that does nothing more than writes to one or two flag variables and then immediately returns back to Miles. Under MacOS, the system is in a vulnerable state inside a callback, so you must be careful what code you execute!

The most important thing to avoid in a Miles callback is calling a function that can cause memory rearrangement. That means you must not call any Toolbox, Miles or other function that may cause memory to move. Check the individual Toolbox and Miles function descriptions to make sure that it can be called at interrupt time.

Also remember that all global or static variables that are modified by a callback function must be declared with the *volatile* modifier. Otherwise, the callback function and the foreground may not be able to communicate after the compiler has performed all of its optimizing tricks.

You should also minimize the total amount of work inside a Miles callback. During a callback, Miles is suspended, so other background tasks won't run - mixing will be interrupted, for example, if you take too long.

Again, though, the best advice for callbacks is to simply post a message or set a flag that can be operated on by your foreground task. This technique works on all platforms and you'll never need to worry about the complications that multithreading can cause.

2.8 Sharing the CPU under MacOS

Discussion

Under MacOS, Miles does most of its work in background interrupt functions. This means that Miles usually does its work without you even having to worry about it. The mixing and buffer submission to Sound Manager all happens without your intervention.

The only time you may need to specifically give Miles CPU time is if you are using the QuickTime MIDI synthesizer which requires you to service it with the AIL_serve() function. The Miles synthesizer usually sounds better and takes less CPU time, though, so this usually is unnecessary.

2.9 Call Logging under MacOS

Discussion

Miles contains a powerful diagnostic feature that allows developers to create a timestamped list of all of their application's foreground Miles calls, along with their parameters and results. This list is written to a plain text file as an aid in tracking down hard-to-find application bugs.

To take advantage of MSS's built-in logging, just create a text file called "**MssDebug.log**' in your application directory. If this file exists, then, during execution of all subsequent Miles calls, a timestamped API call log will be written to the file.

2.10 Miles Examples for MacOS

Discussion

The Miles SDK supplies several example programs for MacOS. Each example comes with a convenient CodeWarrior 6 project ready-to-build. The MacOS example files are found in the **::examples::mac::src** and **::examples::console::src** folders.

For an example that uses digital sound, MIDI sound (with DLS), and CD (redbook) audio, see the **Demo Project**.

For an example that uses the Quick API, see the **Quick API Project**.

For an example that uses the 3D digital audio API, see the **3D Example Project**.

For an advanced example of Internet voice chat, see the **Miles Chat Client Project, and Miles Chat Server Project**.

For simple, SIOUX console-based examples of digital audio playback, MIDI playback (hardware based and DLS), DSP filters, and more, see the **Console Examples Projects**.

Chapter 3

Overview for DOS

3.1 Overview for DOS: Miles on DOS with 32-bit DOS Extenders

Discussion

The Miles Sound System has always supported DOS (it was John Miles' first supported platform way back in 1991)! DOS is a little tough to develop on now-a-days though, because most of the tools necessary to do the job are incompatible with today's machines, or are even discontinued completely.

This platform is almost exclusively used by standup video arcade embedded machines now. It will be interesting to watch whether this market will gradually migrate to Linux in the future.

Not all of the Miles SDK is supported on DOS - there is no 3D sound, voice chat, microphone input or DSP filter support available on this platform.

As it was the initial Miles platform, MSS is extremely easy to use and integrate under DOS. This chapter will discuss everything you need to know!

3.2 Installation for DOS

Discussion

The Miles SDK is installed for DOS just like it is for Win32 - see the Installation for Win32 *Section* for details. You will need some version of Windows to perform the installation.

3.3 Sound under DOS

Discussion

Under DOS, the Miles Sound System provides of a set of loadable drivers, or device-dependent code modules which interact with the host machine's sound-generating hardware, and Miles itself.

Audio driver files for the DOS version of MSS can be recognized by their ". DIG" and ".MDI" file suffixes for the digital audio and XMIDI music drivers, respectively. MSS's driver file structure is the key to its expandability and ease of use. Each unique sound module standard supported by MSS has its own driver which contains enough self-descriptive information to drive its sound module under control of an application developed before that sound module was supported. This "encapsulated" approach helps guard against obsolescence when new sound hardware appears in the marketplace. It is possible to add

to an application's list of compatible sound modules without changing a single byte of existing application code, unless unique new performance features must be supported. Even after a product's release, new drivers and data files can be distributed to end-users via the Internet.

Each DOS driver is written in real-mode 80x86 assembly language or C as a single-segment program containing an INT 66H interrupt handler, various general-purpose support routines, and device-specific code for the supported audio hardware. The drivers' low-level functions are never directly called by the application. Instead, most driver functions serve to pass data back and forth between the sound module and the SDK, while others provide the driver with a means of describing its own capabilities in terms the SDK can understand. DOS drivers for MSS conform to the MSS DDK, which is an "open standard" available to all OEMs, developers, and other interested parties. This standard is defined by the MDD DDK documentation, provided with this documentation.

In fact, the majority of the loadable sound drivers were actually written by the sound card manufacturers. If you have a card that you'd like to be supported, give the manufacturer a call and have them get in touch with us to obtain a DDK.

3.4 Integrating Miles into a DOS Build Environment

Discussion

To build an application that uses MSS, you must include the MSS SDK in the build process. Miles for DOS supports either the Watcom C/C++ or the Borland C++ with Powerpack environments. There are three simple steps to do this:

1) Include **MSS.H** in your source files.

2) Link with the **MSSWAT.LIB or MSSBOR.LIB** library.

3) Copy the files in the **\redist\dos** directory into a subdirectory off your application's directory.

And that's it - you should now be able to call the Miles functions!

3.5 Distributing Miles with DOS Applications

Discussion

To distribute an application that uses Miles, you must include the files in the **\redist\dos** directory in a directory accessible by your application at runtime.

After you perform your installation, you should also then run the ***SetSound*** utility to allow the user to choose the type of sound card in his machine.

3.6 Memory management under DOS

Discussion

By default, Miles handles memory management under DOS by calling the C runtime *malloc* and *free* functions directly and then locking the memory with AIL_vmm_lock().

You can, however, override the Miles allocation and freeing functions with the AIL_mem_use_malloc() and AIL_mem_use_free() functions.

3.7 Callbacks under DOS

Discussion

Read this section carefully if you plan to use Miles timer or other callbacks (such as an end-of-sample callback). If you don't intend to use callbacks, then this section won't apply to you.

Under DOS, callbacks are usually made to your application on a background interrupt. The ideal callback function is one that does nothing more than writes to one or two flag variables and then immediately returns back to Miles. Under DOS, the system is in a vulnerable state inside a callback, so you must be careful what code you execute!

The most important thing to remember is you can't call any functions that call any DOS interrupt. I/O is **especially** not allowed!

Only 1.5 KB of stack space is available to DOS callback functions. This amount is sufficient for nearly all "callback-safe" actions. Any function that requires a larger stack may switch to its own with the AIL_switch_stack() function.

All code and data accessed by the callback function should be page-locked for use in a VMM environment. All callback functions must also be compiled with stack checking turned off. This usually requires a change to your compiler command line (add "-s" in Watcom C++, for example).

Also remember that all global or static variables that are modified by a callback function must be declared with the *volatile* modifier. Otherwise, the callback function and the foreground may not be able to communicate after the compiler has performed all of its optimizing tricks.

You should also minimize the total amount of work inside a Miles callback. During a callback, Miles is suspended, so other background tasks won't run - mixing will be interrupted, for example, if you take too long.

Again, though, the best advice for callbacks is to simply post a message or set a flag that can be operated on by your foreground task. This technique works on all platforms and you'll never need to worry about the complications that multithreading can cause.

3.8 Sharing the CPU under DOS

Discussion

Under DOS, Miles does most of its work in background interrupts. This means that Miles usually does its work without you even having to worry about it. The mixing and buffer submission to low-level sound drivers all happens without your intervention.

The only time you may need to specifically give Miles CPU time is when you are streaming files with the high-level streaming functions. DOS applications can't do I/O in the background, so if you are streaming, you have to periodically call AIL_service_stream().

Miles does take control of the system timer interrupt. If your game already uses the hardware timer, then you are usually best off converting your interrupt routines into Miles timers. They are just as accurate and allow everything to get along nicely. If you must control the clock, then contact RAD for details on patching the required functions in the Miles source code.

3.9 Call Logging under DOS

Discussion

Miles contains a powerful diagnostic feature that allows developers to create a timestamped list of all of their application's foreground Miles calls, along with their parameters and results. This list is written to a plain text file as an aid in tracking down hard-to-find application bugs.

To take advantage of MSS's built-in logging, set the DOS environment variable "MSS_DEBUG" to a filename to write the log to. During execution of all subsequent Miles calls, a timestamped API call log will be written to the specified file.

For example, to dump the API call log to a file name **foo.txt**, execute the following command at the DOS prompt:

set MSS_DEBUG=foo.txt

You can also view MSS's internal function calls by running the extra command, "**set MSS_SYS_DEBUG=1**".

3.10 Miles Examples for DOS

Discussion

The Miles SDK supplies several example programs for DOS. To create these examples, you can simply run the **DoMakeD.BAT** in the source directory.

The DOS example files are found in the **\examples\console\src** directory. There are examples of digital audio playback, MIDI playback (hardware based and DLS), DSP filters, and more.

Chapter 4

Overview for Win16

4.1 Overview for Win16: Miles on Windows 3.1

Discussion

The second platform that Miles was ported to was Win16. This is the only 16-bit platform that Miles supports (although internally MSS is mostly 32-bit assembly even on Win16).

Win16 is pretty tough to debug - the 16-bit debuggers don't run on the 32-bit Windows platforms, so you generally end up living in Soft-Ice. We may drop support for the platform in the future - it is a lot of work just to keep it running.

Not all of the Miles SDK is supported on Win16 - there is no 3D sound or DSP filter support available on this platform.

Miles is, however, easy to use and integrate under this platform. This chapter will discuss everything you need to know to get up and running!

4.2 Installation for Win16

Discussion

The Miles SDK is installed for Win16 just like it is for Win32 - see the Installation for Win32 *Section* for details.

4.3 Integrating Miles into a Win16 Build Environment

Discussion

To build an application that uses Miles, you must include the Miles SDK in the build process. There are five simple steps to do this:

1) First, you must build the Win16 libraries and DLL. To do this, edit "MSS16DEF.MAK" to point to your 16-bit Microsoft C Compiler. Then run the "DoMake16.BAT" batch file in the \src\win and the \src\asi directories to build everything.

2) Include **MSS.H** in your source files.

3) Link with the **MSS16.LIB** import library.

4) Place the **MSS16.DLL** file in the directory containing your application.

5) Copy all of the other files in the **\redist\win16** directory into a subdirectory off your application directory.

And that's it - you should now be able to call the Miles functions!

4.4 Distributing Miles with Win16 Applications

Discussion

To distribute an application that uses Miles, you must include the Miles DLL (**MSS16.DLL**) in the same directory as your application, and you must copy the all of the other files in the **\redist\win16** directory to a directory accessible by your application at run-time.

Do **not** place the **MSS16.DLL** in the Windows or the Windows System directories! The MSS16.DLL **must** be placed in your application directory! The Miles DLL will refuse to be loaded from a system directory.

4.5 Sound under Win16

Discussion

For MIDI output, the Miles Sound System can use the system MIDI hardware through the midiOut API, or it can synthesize the MIDI stream itself and feed it through the digital output system. For digital audio output on Win16, Miles uses the waveOut audio system.

WaveOut mode is nice, because it works on nearly any machine, but, unfortunately, it has terrible latency problems. This was, in fact, the primary reason Microsoft added DirectSound under Win32.

Under waveOut, latency can rise to as high as 300 ms on some older sound cards. There is, unfortunately, no solution to this problem - it's just the nature of the platform.

4.6 Memory management under Win16

Discussion

Under Win16, you **must** use the Miles memory allocation routines for any memory pointer that you pass into MSS. This allows Miles to ensure that the memory is locked and fixed into place to satisfy the Windows multimedia system.

To allocate memory for Miles use, call AIL_mem_alloc_lock(). To free memory, call AIL_mem_free_lock().

For internal memory use, Miles just allocates its own memory directly from these same two routines.

4.7 Callbacks under Win16

Discussion

Read this section carefully if you plan to use Miles timer or other callbacks (such as an end-of-sample callback). If you don't intend to use callbacks, then this section won't apply to you.

Under Win16, callbacks are usually made to your application on a Windows multimedia timer callback. Multimedia timer callbacks are the Windows equivalent to a DOS interrupt. The ideal callback function is one that does nothing more than writes to one or two flag variables and then immediately returns back to Miles.

The most important thing to remember is you can't call most Windows functions inside a multimedia timer. I/O is **especially** not allowed!

All code and data accessed by the callback function should reside in a FIXED segment (you set this in your project's DEF file). Make sure that all data passed to Miles is allocated from AIL_mem_alloc_lock() function.

Also remember that all global or static variables that are modified by a callback function must be declared with the *volatile* modifier. Otherwise, the callback function and the foreground may not be able to communicate after the compiler has performed all of its optimizing tricks.

You should also minimize the total amount of work inside a Miles callback. During a callback, Miles is suspended, so other background tasks won't run - mixing will be interrupted, for example, if you take too long.

Again, though, the best advice for callbacks is to simply post a message or set a flag that can be operated on by your foreground task. This technique works on all platforms and you'll never need to worry about the complications that multithreading can cause.

4.8 Sharing the CPU under Win16

Discussion

Under Win16, Miles does most of its work in a background multimedia callback

and a system timer. This means that Miles usually does its work without you even having to worry about it. The mixing and buffer submission to the sound hardware all happens without your intervention.

The only time you may need to specifically give Miles CPU time is if you aren't going to be servicing the Windows message queue for a while. To keep the audio playing in this situation, call AIL_serve() periodically during the delay.

4.9 Call Logging under Win16

Discussion

Miles for Win16 uses the same call logging feature as Win32. See the Call Logging under Win32 *Section* for details.

4.10 Miles Examples for Win16

Discussion

The Miles SDK supplies several example programs for Win16. To use these examples in an IDE-based environment, just create a new project and include the appropriate C source file (**Demo.C**, for example), and then add the Miles import library to the project (**MSS16.LIB**). The Win16 example files are found in the **\examples\win\src** directory.

For an example that uses digital sound, MIDI sound (with DLS), and CD (redbook) audio, see the **Demo.C** file.

For an example that uses the Quick API, see the **Quick.C** file.

Chapter 5

Overview for Win32s

5.1 Overview for Win32s: Miles on Windows 3.1 with 32-bit Extensions

Discussion

Win32s is a really strange platform. Think of it as a 32-bit DOS Extender for 16-bit Windows. Basically, Win32s lets *some* Win32 programs run unchanged on 16-bit Windows 3.1.

Win32s is almost impossible to debug - SoftIce even has trouble with Win32s applications. Usually, you'll find yourself doing *fprintf*s all over the place. This is probably the last version of Miles to support Win32s - it's just too painful.

When running on Win32s, you link to **MSS32.DLL**, but it internally immediately thunks down to **MSS16.DLL**. Almost all of the issues that apply to Win16 also apply to Win32s.

Not all of the Miles SDK is supported on Win32s - there is no 3D sound, input support, or DSP filter support available on this platform.

This chapter will discuss everything you need to know to get up and running on this somewhat brittle platform!

5.2 Installation for Win32s

Discussion

The Miles SDK is installed for Win32s just like it is for Win32 - see the Installation for Win32 *Section* for details.

5.3 Integrating Miles into a Win32s Build Environment

Discussion

To build an application that uses Miles, you must include the Miles SDK in the build process. There are four simple steps to do this:

1) Include **MSS.H** in your source files.

2) Link with the **MSS32.LIB** import library.

3) Place the **MSS32.DLL** and the **MSS16.DLL** files in the directory containing your application.

4) Copy the all of the other files in the **\redist\win16** directory into a subdirectory off your application directory.

And that's it - you should now be able to call the Miles functions!

5.4 Distributing Miles with Win32s Applications

Discussion

To distribute an application that uses Miles, you must include the Miles DLLs (**MSS32.DLL** and **MSS16.DLL**) in the same directory as your application, and you must copy the all of the other files in the **\redist\win16** directory to a directory accessible by your application at run-time.

Do **not** place the **MSS16.DLL** or **MSS32.DLL** files in the Windows or the Windows System directories! The Miles DLLs **must** be placed in your application directory! They will refuse to be loaded from a system directory.

5.5 Sound under Win32s

Discussion

Since Win32s uses the Win16 version of Miles, sound output on Win32s works exactly the same as under Win16. See the "Sound under Win16" *Section* for details.

5.6 Memory management under Win32s

Discussion

Under Win32s, you **must** use the Miles memory allocation routines for any memory pointer that you pass into MSS. This allows Miles to ensure that the memory is locked and fixed into place to satisfy the Windows multimedia system.

To allocate memory for Miles use, call AIL_mem_alloc_lock(). To free memory, call AIL_mem_free_lock().

For internal memory use, Miles just allocates its own memory directly from these same two routines.

5.7 Callbacks under Win32s

Discussion

Under Win32s, callbacks are **not** called at interrupt time, but are distributed by the Windows messaging system instead (Win32s doesn't allow callbacks at interrupt time). This means that your callbacks won't be executed unless you poll *PeekMessage* or *GetMessage* regularly.

5.8 Sharing the CPU under Win32s

Discussion

Under Win32s, Miles does most of its work in a background multimedia callback and a system timer. This means that Miles usually does its work without you even having to worry about it. The mixing and buffer submission to the sound hardware all happens without your intervention.

The only time you may need to specifically give Miles CPU time is if you aren't going to be servicing the Windows message queue for a while. To keep the audio playing in this situation, call AIL_serve() during the delay.

5.9 Call Logging under Win32s

Discussion

Miles for Win32s uses the same call logging feature as Win32. See the Call Logging under Win32 *Section* for details.

5.10 Miles Examples for Win32s

Discussion

The Miles SDK supplies several example programs for Win32s. To use these examples in an IDE-based environment, just create a new project and include the appropriate C source file (**Demo.C**, for example), and then add the Miles import library to the project (**MSS32.LIB**). The Win32s example files are found in the **\examples\win\src** directory.

For an example that uses digital sound, MIDI sound (with DLS), and CD (redbook) audio, see the **Demo.C** file.

For an example that uses the Quick API, see the **Quick.C** file.

Chapter 6

Miles FAQs and How Tos

6.1 FAQs and How Tos

Discussion

The Miles Sound System is a big API - pretty much anything related to audio is in there. MSS is logical, though, so once you "get it", most things will be second nature. Coming up to speed, however, can be a little daunting.

This is probably most important section in that regard - it will get you up and running quickly - pore over it carefully!

Q: **How do I initialize the Miles Sound System for use?**
A: The first step in using Miles is to initialize the library. You can initialize Miles with the full API or the Quick API.

To open Miles with the full Miles API, follow these steps:

1) Call AIL_set_redist_directory() to point to game local directory that should now contain the contents of the platform-specific "\redist" directory.

2) Call AIL_startup() to initialize Miles.

3) Call the appropriate open function for the subsystems that you plan to use: AIL_open_digital_driver() for 2D digital audio, AIL_open_XMIDI_driver() for MIDI, etc. Under Win32, make sure that you have created an HWND at this point (DirectSound requires this).

If you use the full Miles API to initialize Miles, be sure to call AIL_shutdown() to shut everything down when your game is closed.

To open Miles with the Quick API, follow these steps:

1) Call AIL_set_redist_directory() to point to game local directory that should now contain the contents of the platform-specific "\redist" directory.

2) Call AIL_quick_startup() with the appropriate settings for your application. For Win32, make sure that you have created an HWND at this point (DirectSound requires it).

If you use the Quick API to initialize Miles, be sure to call AIL_quick_shutdown() to shut everything down when your game is closed.

Q: **How do I play a digital sound?**
A: To play a digital sound with the **full Miles API**, follow these steps:

1) Allocate an HSAMPLE with AIL_allocate_sample_handle().

2) Call AIL_init_sample() to initialize the sample to default values.

3) Load the file from the disk with AIL_file_read() (or your own file IO routines).

4) Point the HSAMPLE at the loaded disk file image with AIL_set_sample_file().

5) Finally, call AIL_start_sample() to begin playing the HSAMPLE.

>To play a digital sound with the **Quick API**, follow these steps:

1) Call AIL_quick_load() to load the file.

2) Play the file with AIL_quick_play().

Q: **How do I play an XMIDI or MIDI file?**
A: To play an XMIDI file, follow these steps:

1) Allocate an HSEQUENCE with AIL_allocate_sequence_handle().

2) Load the file from the disk with AIL_file_read() (or your own file IO routines).

3) Call AIL_init_sequence() to point the HSEQUENCE at the loaded disk file image.

4) Play the file with AIL_start_sequence().

To play a standard MIDI file, call the AIL_MIDI_to_XMI() function to convert the MIDI file into an XMIDI file image and then follow the XMIDI steps.

Q: **How do I play an XMIDI file with a DLS instrument set?**
A: To play an XMIDI file using a DLS instrument set, follow these steps:

1) Open the DLS synth with AIL_DLS_open().

2) Load the instrument set from the disk with AIL_file_read().

3) Use AIL_extract_DLS() to uncompress the instrument set, if necessary. You call tell if you need to uncompress the file by checking the type of the file with the AIL_file_type() function.

4) Load the instrument set into the synthesizer with AIL_DLS_load_memory().

5) Now follow the steps in the "How do I play a MIDI file?" *FAQ*.

Q: **How do I stream a file off a hard disk or CD?**
A: To stream a digital sound off the disk, follow these steps:

1) Open the file to stream with AIL_open_stream().

2) Play the stream with AIL_start_stream().

Q: **How do I play a track off a CD?**
A: To play a CD track, follow these steps:

1) Open the redbook device with AIL_redbook_open().

2) Get the start and stop points with AIL_redbook_track_info().

3) Call AIL_redbook_play() to begin playing the track.

Q: **How do I loop a track off a CD?**
A: To loop a CD track, you have to do a little more work, because you can't instruct the hardware itself to loop. To handle this, follow these steps:

1) Open the redbook device with AIL_redbook_open().

2) Get the start and stop points with AIL_redbook_track_info().

3) Call AIL_redbook_play() to begin playing the track.

4) Subtract the starting and stopping points to calculate the length of the track, and then add a second or two of slop time.

5) Create a timer (system or Miles) for the length of time of the song.

6) Restart the CD track when the timer expires (and reset your timer for the next loop).

Q: **What kind of audio decompression does Miles support?**
A: Miles supports IMA ADPCM decompression, MPEG Layer 3 decompression, and Voxware voice decompression.

IMA ADPCM is a simple format that provides 4 to 1 compression on 16-bit data. There is essentially no CPU hit for decoding ADPCM - it's that simple. Like all ADPCM variants, it is somewhat hissy, though.

MPEG Layer 3 (or MP3) is a very complicated format that usually provides 11 to 1 compression with no audible loss. It does use a fair amount of CPU, though. See the "What is MPEG Audio and MP3?" *FAQ* for more details about MP3.

The Voxware voice codecs are complicated formats that provide up to 200 to 1 compression on voice data (fairly compressed-sounding, though). They are only for voice data, though - they can't do anything at all with music or sound effects. The Voxware codecs are used almost exclusively for Internet voice chat.

Q: **What is MPEG Audio and MP3?**
A: MPEG audio is a global standard that was invented to compress the audio streams that accompany MPEG video streams. Because it was invented almost 8 years ago, MPEG audio needed to be scalable in playback complexity. The MPEG group, therefore, came up with three levels of compression complexity: MPEG Layer 1, MPEG Layer 2, and MPEG Layer 3. Each level provided almost twice the audibly lossless compression ratio (MP1 provides 3 to 1, MP2 provides 6 to 1, and MP3 provides 11 to 1), but each level required more and more CPU horsepower.

However, now, 8 years later, even the sophisticated Layer 3 compression standard only uses a small percentage of the total power of a modern desktop CPU. This means we can now take advantage of the massive size savings that MPEG Layer 3 can provide! For example, at the usual compression ratio of 11 to 1, this means you could store up to 13.5 hours of audio on one CD! And remember, this is audibly lossless compression - you can't hear any difference!

Before MSS, one bad aspect to using MP3 compression was the fact that two overseas companies owned the compression and decompression patent rights to the MP3 algorithm. That meant that if you wanted to add MP3 to your commercial product, then you had to go through a convoluted and lengthy licensing process to obtain an expensive license. Worse, you were simply paying for the patent rights - you still had to write the MP3 decoder itself!

However, with the Miles Sound System, we have arranged through Fraunhofer and Thomson multimedia to provide a legally redistributable MP3 decoder for no extra cost! You must sign an extra MPEG license addendum (which, among other minor things, requires you to add a credit to Fraunhofer and Thomson in your game), but, other than that, you can use MP3 decompression just like our already built-in IMA ADPCM decompression. More about that in a minute, though.

Now, MPEG Layer 3 **encoding** is a completely different situation. RAD does sell an MPEG Layer 3 encoder, but does so completely separately from the Miles Sound System. This MP3 encoder is available from us on a per-machine basis for $95.00. Note that you cannot redistribute this encoder - it is for your use only. To obtain a redistributable encoder, give us a call and we can get you in touch with the right people at Fraunhofer.

The Miles MP3 encoder is based on the ISO reference source code. This means that while the sound quality of the compression is very good, the compression speed is very, very slow. It can take up to 5 minutes to compress a single 3 minute WAV file! The ISO reference source also has one other limitation - it can only compress 32 Khz, 44 Khz, and 48 Khz data rates. That means that if you try to compress a 22 Khz input file, the Miles encoder will upsample it to 32 Khz and then compress it (which obviously hurts the overall compression ratio).

Because of these limitations, you may want to look at alternative MP3 compressors - this is a highly competitive field and there are many very good, very fast MP3 encoders available (including ones that support other sample rates - 16, 22 and 24 Khz). See the MP3.COM (http://www.mp3.com) web site for links to many different encoders. The Fraunhofer encoders are generally found to be the very best (and are relatively inexpensive).

One feature that the other encoders don't have, however, is the ability to compress the instruments in a DLS file. Only the Miles MP3 encoder can currently perform this specific type of compression (which can compress DLS files up to 11 to 1). With this feature, the Miles MP3 encoder can give you

incredibly small digital music files. For example, on our web site are several DLS files that are under 1 MB is size, yet contain 7 to 10 MBs of instrument data!

OK, now let's talk about the Miles MP3 **decoder**. First off, the Miles MP3 decoder is just that: an MP**3** decoder. Miles cannot decode or playback MPEG Layer 1 or MPEG Layer 2 compressed audio. You must uncompress data compressed with the simpler formats with a non-Miles utility, and then recompress it using MP3. However, since MP3 is 4 times as small as MP1 and 2 times as small as MP2, this should be to your advantage once the data is converted.

Once the Miles MP3 decoder is installed, MP3 data is treated like a completely integrated sound type - the data doesn't even have to be decompressed before passing it to Miles! However, the MP3 decompression process does take more CPU than the comparatively simple IMA ADPCM algorithm. You won't be able to decompress 20 MP3 streams simultaneously on a 486, for example!

To solve this problem, Miles also allows you to pre-decompress MP3 data. This way, if CPU-performance is critical, then you can decompress the MP3 files ahead of time (say when you load the game level data). If, however, memory is critical and CPU-performance isn't important, then you could decompress all of the MP3 files on the fly. The choice is up to you!

Since an MPEG Layer 3 file contains just the compressed data (MP3 files don't really have an identifying header), you must use a new function called AIL_set_named_sample_file() to tell Miles that it is dealing with MP3-compressed data. There is one exception to this - Microsoft has defined a special type of WAV file that can contain MP3 data (basically the MP3 data is wrapped up into a RIFF chunk).

If you use this weird WAV file format, then Miles can use the same old AIL_set_sample_file() function that you've always used to load sound data. However, not all encoders (including the Miles MP3 encoder) support this WAV file wrapping system, so most of the time you should use the new AIL_set_named_sample_file() function (which works with either type of MP3 file).

Q: **What MP3 encoder should I use?**
A: The encoders are being upgraded so quickly that it's hard to give a recommendation that will be valid for any length of time.

We've generally found the Fraunhofer encoders to be the best (and are relatively inexpensive) - after all, they did first publish the format!

If you want to compress the DLS instruments in your files, then you need our encoder - it is the only one with the hooks necessary to encode DLS files. Our encoder isn't the best for general MP3 encoding, though.

Q: **I can't get Miles to play an MP3 file - what's going on?**
A: If you can't play an MP3 file, try these troubleshooting questions:

1) Did you copy all the redistribution files into a directory available to your application?

2) Did you call AIL_set_redist_directory() to tell Miles where this directory is? This directory is relative to the location of the Miles library file, so if you are using a relative path - make sure it is relative to the right directory!

3) If you are using the 2D API, are you using AIL_set_named_sample_file() to tell Miles what type of data it is?

4) Did you sign the license agreement for MP3 decoding? This is a separate license addendum that your company might have chosen not to sign. Look for the ASI codec library files (*.ASI in Win32) in the redistribution directory - are they present?

5) Is the file in question an MP3 file? Miles doesn't play MPEG Layer 1 or MPEG Layer 2 files which are often mistakenly labeled MP3 files. Try recompressing the file yourself just to be sure.

Q: **How can I lower the CPU hit of playing an MP3 file?**
A: The Miles MP3 decoder is amazingly fast - it contains both Intel and AMD 3DNow optimized versions. On a P3/300, full 44 Khz stereo decoding takes as little as 3% of the CPU.

You can cut the CPU hit almost linearly by dropping the sample rate, and/or the number of channels. For example, 44 Khz mono takes as little as 1.5%.

In general, we've seen very few places where the CPU hit of MP3 is a problem. Remember that if you are streaming audio in off a drive, you usually take a larger CPU hit just in IO overhead (which MP3 will actually mitigate because you are reading a tenth as much data).

Q: **I'm getting a glitch when I loop an MP3 file - what can I do?**
A: There are a few things you can do to minimize the glitch, but looped MP3s always glitch a little - **always**. The issue is that the MP3 file format uses a 50% frame overlap in each compressed frame. That means you can't just plug one MP3 frame up next to another - you get a big blip of odd synthesized audio when the decoder merges two completely unrelated chunks of data.

To avoid this, Miles clears the MP3 decoder whenever the audio loops (either at the end of the file or at the end of a sub-loop block). This always kicks out a short blip of silence in the output. **This is as good as MP3 can be looped** - there is no way to improve it - it's just a downside to the format itself.

So, you can't get any better than a little blip of silence, however, you can accidentally make it **even worse** when using bad sub-block loop points. To

avoid this, make sure the loop points that you choose occur at the beginning of a new MP3 frame. Most sound editors can display the frame points of an MP3 file.

Q: **How do I set a DSP filter on a sample?**
A: To set a DSP filter, you have to first find the filter that you want to apply. To find a filter, you enumerate through the possible filters looking for the one you want with the AIL_enumerate_filters() function. Code like this will find the low-pass DSP filter:

```
HPROVIDER low_pass;
char*     flt_name;
HPROENUM enumFLTs=HPROENUM_FIRST;
HPROVIDER cur_prov;
while (AIL_enumerate_3D_filters( &enumFLTs,
                                 &cur_prov,
                                 &flt_name ) )
{
  if (strcmp(flt_name,"LowPass Filter")==0)
  {
    low_pass=cur_prov;
    break;
  }
}
```

OK, we have a handle to the low-pass filter, so now we have to apply it to our HSAMPLE. To do this, we call the AIL_set_sample_processor() function, like this:

```
AIL_init_sample( S );
AIL_set_sample_processor( S, DP_FILTER, low_pass );
```

Make sure you call AIL_set_sample_processor() *after* the call to AIL_init_sample(). Now all that remains is setting the filter preferences that you'd like to control.

To do that, call the AIL_set_filter_sample_preference() function with one of the preferences described in the Digital Filter Services *Section<?Digital Filter Services>*. To finish our example, let's set the low-pass filter to kick in at 4000 Hz, like this:

```
F32 value=4000.0F;
AIL_set_filter_sample_preference( S, "Lowpass Cutoff", &value );
```

Q: **What is the difference between "simple" reverb and the DSP reverb filters?**
A: Simple reverb is built right into the Miles mixer. It applies a simple echo more than a "true" reverb. The simple reverb is nice because it is much, much faster than the reverb filters. The reverb filters, on the other hand, usually sound much better (depending on the type of effect that you are looking for, of course).

In general, most folks use the Reverb filters rather than the built-in simple reverb.

Q: **How can I play two HSAMPLEs from the same memory address?**
A: You don't have to do anything special to play from the same memory location in Miles.

Just allocate an HSAMPLE with AIL_allocate_sample_handle() - an HSAMPLE doesn't have any sample memory associated with it. Rather, **you** allocate your own memory and then just point it at this address with the AIL_set_sample_file() function.

Miles reads out of **your** memory! So, if you want to play two HSAMPLEs (or three, or ten, etc) from the same memory location, just pass the allocate two HSAMPLES and point each one to the same address with AIL_set_sample_file().

Q: **When can I free the memory I gave to MSS?**
A: Since Miles plays out of your memory buffers, you can't free the memory until you know that Miles is done with it.

For example, if you've pointed Miles at some audio data with AIL_set_sample_file(), you can't free the memory until AIL_sample_status() returns SMP_DONE. You can call AIL_end_sample() to instantly force the status to SMP_DONE.

Occasionally, a crash will occur when a Miles user is freeing sample memory that will cause them to think that Miles is still using the data even after the status has changed to SMP_DONE.

Over the last ten years, there have a fair number of folks convinced that this is what was happening (with varying degrees of certainty). In **all** cases, something else was causing the crash - memory corruption (which using causes a crash in *free*), another sample was also playing from the address, etc. Just remember - if SMP_DONE has been set, you can free the memory as long as another sample isn't using it. Guaranteed!

Q: **How do I start an audio file somewhere in the middle?**
A: In all of the Miles APIs (2D digital, MIDI, streaming, and 3D digital), when you call the "start" function (like AIL_start_sample(), for example), the sound will restart at the beginning of the audio file.

So, to play an audio file from the middle, open the sound file, skip to an offset, and *then* start the playback with one of a "resume" function (like AIL_resume_sample()).

Q: **Panning isn't working - what's going on?**
A: If panning doesn't work, then you probably just didn't open the sound driver in stereo mode. Make sure you used 2 for the number of output channels in the call to AIL_open_digital_driver().

Q: **Which 3D sound provider should I use?**
A: There are pros and cons to using any of the providers. Read about the details on each provider in the 3D Digital Audio Services *Section<?3D Digital Audio Services>*.

In any case, we recommend that you do **not** make the choice for the user. Instead, you're usually better off displaying the current list of providers and allowing the user to make their own choice.

Q: **How do I open a 3D digital provider?**
A: Opening a "3D provider" is the first step in using 3D sound. A 3D provider is a loadable module that provides 3D services to MSS. There are a bunch of different 3D technologies that you can choose from.

In general, you won't be able to auto-detect the "best" provider to use. Some of the providers even depend on the end-user's speaker configuration (like Dolby Surround). Because of this, you should choose a default 3D provider, and then require the user to select one of the more specific providers. The best provider to use as a default is the Miles Fast 2D provider - it works on all sound cards and doesn't cost much CPU.

So, the first step in opening a 3D provider is enumerating all of the available 3D providers with AIL_enumerate_3D_providers(). This is performed with code something like this:

```
#define MAX_PROVIDERS 64
char*     provider_names[MAX_PROVIDERS];
HPROVIDER provider_handles[MAX_PROVIDERS];
HPROVIDER provider_default=0;
U32       provider_count=0;
HPROENUM enum3D=HPROENUM_FIRST;
while (AIL_enumerate_3D_providers( &enum3D,
                                   &provider_handles[provider_count],
                                   &provider_names[provider_count] ) )
{
  if (strcmp(provider_names[provider_count],"Miles Fast 2D Positional Audio")==0)
    provider_default=provider_handles[provider_count];
  provider_count++;
}
```

OK, so now we have an array of descriptive strings as well as a handle for each provider (including our default provider). Note that **all of the providers** will be in this list (even providers that won't necessarily work on the user's hardware). The only way to tell if a particular provider will actually work on an end-user's machine is to try to open it with the AIL_open_3D_provider() function. RAD doesn't usually recommend automatically culling the list (it can take a long time just to try to open every provider) - just display them all and let the user choose.

So, at this point, we can either display a menu to let the user choose the provider, or just use the default provider. To open the provider, we simply call the AIL_open_3D_provider() function. Remember to check the return result from this call - it will fail if a user has chosen a provider that his machine doesn't support.

Q: **How do I play a 3D digital sound?**
A: Once you have a 3D provider *opened*, playing a 3D digital sound is comparatively easy. Just follow these steps:

1) Allocate an H3DSAMPLE with AIL_allocate_3D_sample_handle().

2) Load the file from the disk with AIL_file_read() (or your own file IO routines).

3) Call AIL_set_3D_sample_file() to point the sample at the file image that you want to play (use AIL_set_3D_sample_info() to play RAW audio data.)

Note that 3D audio data has to be mono (not stereo) and has to be uncompressed (not IMA ADPCM or MP3). See the "How can I play stereo or compressed digital files with the 3D API?" *FAQ* for more details.

4) Call AIL_set_3D_sample_distances() to set the 3D volume falloff values. See the "My 3D sound isn't falling off like I expected..." *FAQ* for more details on 3D distances.

5) Finally, call AIL_start_3D_sample() to begin playing the H3DSAMPLE.

Q: **My 3D sound isn't falling off like I expected - what's going on?**
A: This is a common source of confusion. Basically, the most important thing to remember is that the maximum distance has **nothing** to do with the volume falloff curve! **Only the minimum distance matters for the falloff curve.**

The volume of a sample falls off by half every time the minimum distance is doubled. So, if you set your minimum distance to 3, then this falloff curve is in effect:

Distance: less than or equal to 3 meters. Volume: 100%.
Distance: 6 meters. Volume 50%.
Distance: 12 meters. Volume 25%.
Distance: 24 meters. Volume 12.5%.
Distance: 48 meters. Volume 6.25%.
etc...

So, you can see that the maximum distance doesn't affect the volume falloff curve at all! The maximum distance just sets a hard clip point - the volume is forced to be zero.

Also note that some DirectSound drivers don't implement the maximum distance correctly with the DSBCAPS_MUTE3DATMAXDISTANCE flag in effect (which Miles uses). At the maximum distance on these drivers, the volume doesn't drop to zero, instead, it just stops falling off.

Q: **How can I play stereo or compressed digital files with the 3D API?**
A: The 3D API takes mono, uncompressed data only. The reason for this

requirement is that some providers to download the sample data directly onto the sound card.

So, if you have compressed or stereo data, you need to convert it before passing it to the AIL_set_3D_sample_file() or AIL_set_3D_sample_info() functions.

For MP3 files, call AIL_decompress_ASI() to decompress the data.

For IMA ADPCM files, call AIL_decompress_ADPCM() to decompress the data.

For stereo files, call AIL_process_digital_audio() to convert the data. In general, you should just not use stereo sound data, though. It makes sense to convert MP3 or ADPCM files on-the-fly to save disk space, but it doesn't really make any sense to ship stereo files and then just down-convert them to mono at runtime.

Q: **I get a crash when I call AIL_digital_handle_release while the 3D digital API is playing - what's going on?**
A: Some of the 3D providers use the DirectSound handle that the 2D subsystem opens. If you release the 2D handle, you need to also release the 3D system. However, the 3D system may be more complicated to release, though, because the audio samples could have been downloaded to the hardware itself.

Note that opening, closing and re-opening the 3D sound driver is stressing the sound hardware drivers in an unusual way. As with all hardware drivers, when you step off the common path, you run the risk of exposing driver bugs.

So, for maximum compatibility, RAD recommends just not releasing the drivers *at all* if you are using 3D sound. This isn't that big of a downside anymore - most sound drivers allow multiple applications to access the hardware at once now, anyway.

If you still want to **release** your digital handle regardless of the driver situation, you need to follow these steps:

1) Call AIL_close_3D_provider() to close the 3D resources.

2) Call AIL_digital_handle_release() to release the 2D handle.

Then, when you want to **reacquire** the sound resources:

1) Call AIL_digital_handle_reacquire() to re-open the 2D handle.

2) Call AIL_open_3D_provider() to re-open the 3D system.

3) Reallocate all of your H3DSAMPLES and resubmit the audio buffers with AIL_set_3D_sample_file().

Q: **My 3D audio is a lot louder than my 2D audio - what can I do?**
A: Some 3D hardware doesn't have very good balance between their 3D samples

and 2D samples. There isn't a good way to automatically deal with this, because it's a sound card-specific problem. The best solution is to give the user a slider in your normal volume-setting options screen and let them balance the two types of audio themselves. Then, during the game, use AIL_set_3D_sample_volume() and AIL_set_sample_volume() to adjust the volume for each sample.

Q: **How do I set an EAX reverb room style?**
A: Use the AIL_set_3D_room_type() to set the EAX room first, and then call AIL_set_3D_sample_effects_level() to turn the effects volume up for the 3D samples that should have reverb.

Q: **How does Miles do sound prioritization?**
A: It doesn't. Philosophically, RAD has never been a big fan of handling sound prioritization at the library level. The way Miles works, the application can just allocate sample handles until it hits the system-imposed limit, and the application is responsible for reusing those handles in whatever way it sees fit. The application can use whatever criteria it wants to bump playing sounds (age, importance, distance, volume, etc.), or decide not to bump any at all if the new sound isn't that important.

A lot of that flexibility goes away, or becomes awkward to access, if you make Miles responsible for killing existing sounds in favor of new ones.

Q: **I can't seem to change the volume of my XMIDI sequences, what's going on?**
A: This is probably because your XMIDI sequence doesn't have any default volume events. Miles can't change the volume of a sequence until the baseline level is specified with a MIDI volume event. Have your musician insert volume events at the beginning of the sequence and you should be able to then control the volume with AIL_set_sequence_volume().

Q: **What's the deal with callbacks - are they threads, interrupts, system timers, or what?**
A: It all depends on the platform:

On *Win32*, callbacks are usually made from a background thread.

On *MacOS*, callbacks are usually made from an interrupt-based system callback.

On *DOS*, callbacks are usually made from a hardware interrupt.

On *Win16*, callbacks are usually made from a Windows multimedia timer (which is essentially an interrupt).

On *Win32s*, callbacks are usually made from the Windows message pump.

Q: **How much latency does Miles have?**
A: The AIL_digital_latency() function can roughly report the current latency. This section will discuss where this latency comes from on each of the Miles platforms.

On Win32 with DirectSound, the default latency is 64 milliseconds (not counting the OS, sound driver and sound hardware latency which is out of our control). You can reduce or increase the latency by changing the DIG_DS_MIX_FRAGMENT_CNT preference. If you shorten the latency, you increase the chance of skipping. If you lengthen the latency, you can ride out longer thread starvations.

On Win32 with waveOut or Win16, the latency depends on the sound card and sound card drivers. Miles calculates the minimum latency that won't cause skipping at runtime. On most cards, this will be 150 ms to 200 ms, but on some sound cards (especially older ISA cards), this latency can rise to as high as 300 ms! These excessive latency times with waveOut environment are due to fundamental design shortcomings in the Microsoft architecture, and cannot be reliably overcome by any external techniques.

However, under waveOut, the MSS architecture does permit a sound effect to be started with essentially zero latency, as long as no other sound effects are currently being played! So, if you have a situation where latency is critical, then you can improve waveOut's performance dramatically by letting all other sounds complete before starting the latency-critical sound effect.

On MacOS, Miles has been written so that digital latency will rarely (if ever) exceed 100 milliseconds and will never exceed 140 milliseconds regardless of the speed of your machine. Just like Windows, though, Sound Manager imposes some amount of its own latency as well (exactly how much depends on the machine and version of Sound Manager).

On DOS, the default latency is 100 milliseconds. You can reduce or increase the latency by changing the DIG_LATENCY preference. If you shorten the latency, you increase the chance of skipping. Under DOS, there is little reason to change the default latency.

Q: **My sound is skipping on Win32 - what can I do?**
A: Under Win32, mixing happens on a background thread (on the other Miles platforms, mixing occurs on an interrupt). Unlike interrupts, threads are subject to starvation - that is, depending on CPU load, one thread can be neglected for a period of time.

When that happens to the Miles mixing thread, then the digital audio can "skip" momentarily. Fortunately, on NT based Win32 platforms, this rarely happens - you really have to try to cause a skip on NT. Unfortunately, on the Windows 9x based platforms, it is fairly easy to cause the Miles thread to starve. The easiest way to trigger starvation is to do a lot of sustained disk IO on an IDE-based device.

So, regardless of the cause of the problem, we just want the user to have a good experience. A simple way to fix the problem is to just mix ahead farther - if we mix ahead 200 milliseconds, then we can tolerate a thread dropout of up to 200 ms with no trouble! The only problem is that now our latency is 200 ms - fine for some games, but terrible for others!

Fortunately, there is a good compromise. By using the AIL_get_timer_highest_delay() function, you can track the length of the thread starvations as they occur.

So, let's say you have a game that skips when you read in the level data from the disk. The best way to fix the problem is something like this:

1) Call AIL_get_timer_highest_delay() before you start the IO (this clears the previous high value).

2) Do all your IO.

3) Call AIL_get_timer_highest_delay() and record it.

Run your game on a couple machines and keep track of the high value. It's usually going to be something around 150 milliseconds, but get a good reading with your game's pattern of IO.

Now we just bump up the latency during the IO and then bump it back down after - you get the best of both worlds - good latency in the general case, and enough mix ahead to handle the IO dropouts. Once you know roughly what latency you have to cover, simply do something like this:

1) Call AIL_set_preference() on DIG_DS_MIX_FRAGMENT_CNT to your highest timer delay divided by 8 (each fragment in Miles is 8 ms).

2) Call AIL_serve() to make sure that the mix-ahead buffer is filled.

3) Do all your IO.

4) Call AIL_set_preference() on DIG_DS_MIX_FRAGMENT_CNT to set it back to its default value of 8.

Q: **My application is crashing - what should I try?**
A: Here's my quick list of things to ask yourself when you get a crash. Most of them seem really obvious, but one of these questions solves pretty much every technical support query we get!

Go through the questions one by one and **don't assume** the answer to any of them - actually explore each question. Without fail, the most complicated crashes are usually reduced to one simple assumption that was never completely confirmed.

1) What call is crashing? Does it crash every time, or just once in a while?

2) If you cut that call into one of the Miles sample applications, does it still crash?

3) Is the problem data-dependent? Do other sound files work?

4) Don't overlook the obvious - what are the contents of the parameters being passed to the crashing function? Are they valid? Double-check them at run-time in the debugger.

5) Are you using the latest version of Miles? See the "What is the latest version of MSS?" *FAQ* for details.

6) Make sure that the header file, the DLLs and shared libraries, and all of the add-ons files (the ASIs, FLTs, M3Ds, etc) are all the same version! Scan your drive for the Miles library files - make sure there are none on your drive except the very latest version.

7) Miles is used in a **lot** of games (about 700 per year). If you are getting a crash, try to imagine what is different about your application - are you doing anything you'd guess is unusual?

8) Try never freeing your sound memory buffers. You'll leak memory, but if you don't crash, then you are probably freeing a piece of memory before Miles is done with it. See the "When can I free the memory I gave to MSS?" *FAQ* for details.

9) Does it happen on a specific machine? On a specific OS? Does it only happen with a particular sound card? Do you have the latest sound card drivers for your sound card?

10) Does it only happen with a particular 3D provider?

11) Examine the Miles call log (see the call logging sections in the Overview sections). Does anything appear unusual?

12) Build the Miles DLL in debug mode - can you tell what's happening from the crash location in the source code? See the "How can I rebuild the Miles source code?" *FAQ* for details.

13) Try simplifying the problem. Turn off everything unnecessary (DSP filters, spacialization, reverb, etc.) - does it affect the crash?

14) Are you using callbacks? Try turning them off completely - does it still crash?

15) Finally, if you've been able to reproduce a crash in Miles, send me the example (keep it short and simple - the console applications in the \Examples directory are a good place to start) and we'll get it fixed for you. RAD can usually fix reproducable problems within days!

Q: **Where is the Miles source code? How can I rebuild it (in debug mode)?**
A: Almost all of the source code is included with Miles. How you rebuild it depends on the platform:

To build Miles on Win32:

1) Edit **MSS32DEF.MAK** in the "\src\win" directory and modify it to point to your installation of Microsoft Visual C and MASM (use short pathnames).

2) If you want to build in debug mode, edit the line "Debug=0" to "Debug=1" in the **MSS32DEF.MAK** file.

3) Run the **DoMake.BAT** file in "\src\win", "\src\m3d", "\src\asi", and the "\src\flt" directories. The debug files will be generated in the "\redist\win32" directory.

To build Miles on MacOS:

1) Load the **Miles Release Project** or the **Miles Debug Project** in the "\src\mac" directory into CodeWarrior and do a "rebuild all".

To build Miles on DOS:

1) Edit the WATCOM.MAK or BORLAND.MAK to modify the paths to your installation of the compiler and MASM.

2) If you want to build in debug mode, edit the line "Debug=0" to "Debug=1" in the **WATCOM.MAK** or **BORLAND.MAK** files.

3) Run the **DoMake.BAT** for Watcom or the **DoMakeB.BAT** for Borland in the "\src\dos" directory.

To build Miles on Win16:

1) Edit **MSS16DEF.MAK** in the "\src\win" directory and modify it to point to your installation of Microsoft Visual C 1.52 and MASM (use short pathnames).

2) If you want to build in debug mode, edit the line "Debug=0" to "Debug=1" in the **MSS16DEF.MAK** file.

3) Run the **DoMake16.BAT** file in "\src\win" and "\src\asi" directories. The debug files will be generated in the "\redist\win16" directory.

Q: **How can I tell what version of Miles that I have?**
A: It depends on when and what version you'd like to check.

At runtime, you can call the AIL_MSS_version() function to get the library version.

You can get the header file version by looking at the top of the **MSS.H** file.

You can check the version of any of the Windows DLLs (including M3Ds, ASIs, and FLTs) by right-clicking on the file in Explorer, choosing "Properties" and then clicking on the Version tab.

You can check the version of the Mac shared libraries (including M3Ds, ASIs, and FLTs) by using the Finder menu option "Get Info...".

Q: **What is the latest version of MSS?**
A: You can always see what the latest version of Miles is (and see what has changed) by checking the Miles web-site (http://www.radgametools.com/MilesSndSys/changes.html).

Q: **How do I access files from a big packed file?**
A: Just use the AIL_set_file_callbacks() function to point to your own routines for doing IO. The streaming API can also take a system file handle if you don't want to bother with implementing callbacks.

Q: **I'm setting a preference and it isn't having any effect - what's going on?**
A: Usually, when this happens, you are setting the preference after it has a chance to take effect. For example, if you set DIG_MIXER_CHANNELS to TRUE, but do it after you've called AIL_open_digital_driver(), then it won't have a chance to apply.

Also remember that preferences are reset by a call to AIL_startup(), so don't set your preferences too early or forget to reset them if you ever re-startup Miles.

Q: **I want a small installation - do I need all of the files in the \redist directories?**
A: No, you only have to use the files that you intend to use. You have to have the main Miles library file (MSS32.DLL under Win32, for example), but the other files are optional.

If you aren't using any of the filters, then you don't need any of the *.FLT files, for example.

Each of the M3D files are discussed in the 3D Digital Audio Services *Section<?3D Digital Audio Services>* - if you aren't planning to support a particular 3D technology, just don't ship the corresponding M3D file.

The ASI files provide compression and decompression support in Miles. The MP3DEC.ASI file is the MP3 decoder. The MSSV12.ASI, MSSV24.ASI, and MSSV29.ASI files are the Voxware voice chat codecs. If you don't use one or more of these audio codecs, then you don't need to ship them!

Q: **I'm using Windows 9x or NT - do I have to call the AIL_mem_alloc_lock function?**
A: Only if you want to make porting easier to MacOS or DOS. Under Win32, you can just pass in any memory buffer to Miles - malloc-ed, GlobalAlloc-ed,

resource locked, etc.

Q: **How careful do I have to be when shutting down?**
A: Not very - if you simply call AIL_shutdown(), Miles will shut everything that you have opened (the digital system, the MIDI system, any open streams, the 3D providers, etc).

Miles doesn't, however, automatically free any memory that you have allocated with AIL_mem_alloc_lock() or AIL_file_read(), so make sure to free all your Miles allocated memory before the call to AIL_shutdown().

Q: **This function returns a bunch of return data, but I only want one value - is there a simpler way to use it?**
A: Yes, throughout Miles, it is legal to pass in zero for almost every return pointer. If the pointer you pass Miles is zero, then it just doesn't set that return value.

For example, in AIL_sample_ms_position() both the total length of the sample and the current position are returned. If you just need the total length, you can pass in zero for the current position parameter and Miles just won't return it.

Q: **The Internet voice chat example has a lot of latency or it doesn't work on my LAN - what's going on?**
A: There are two equally important subsystems when doing Internet voice chat. The first is the sound system - this subsystem captures the data, compresses, decompresses it, and then outputs it. This is the side of the problem that Miles solves: it has the input, voice chat codecs, and killer output side.

The other half of the problem is the network layer. This is usually a UDP-based network transport layer that your game also uses for communication. The key thing to remember is that Miles doesn't have **any network layer**! A good network layer is at least as big of a problem as sound itself, and it's way out of MSS's domain. There are, in fact, complete licensable products that *just* supply the network communication for a game.

So, the voice chat examples that we supply are just that - examples. You can use the examples to see how to call the input, codec, and output systems, but they won't help you on the network side. We just implemented the least complicated protocol possible to keep the rest of the example easy (it's already fairly complex).

So, the examples use the high-level TCP/IP protocol for communication - this works fine on a LAN with access to a DNS server, but it definitely will have trouble over the Internet unless you replace the network pieces.

We have extensive documentation on integrating the voice chat codecs with a generic network layer - see the "Implementing Voice Chat" *Section<?Implementing*

Voice Chat> for more details.

Q: **What are these ASI, M3D, and FLT files? What is RIB?**
A: RIB stands for RAD Interface Broker - it is RAD's mechanism for expanding and enhancing the system through the use of dynamically-loadable modules. These modules, called RIBs, are designed in conformance with the new RAD Interface Broker (RIB) standard. ASI, M3D and FLT files are all RIB-based loadable modules.

ASI files are sound codecs - MP3 is provided in the **MP3DEC.ASI** file, for example.

M3D files are 3D sound providers - EAX support is provided in the **MSSEAX.M3D** file. There are quite a few M3D files now.

FLT files are digital sound filters - low-pass filtering is performed in the **LowPass.FLT** file. There are a bunch of FLT files!

This is all you really need to know about RIB to use Miles, but if you are interesting in the technical underpinnings of RIB, read on...

The RIB standard allows either MSS or the application itself to scan a directory for installable modules, load them, query their attributes, and take advantage of the features they offer at runtime, all in a manner that's completely transparent to your application. RIBs deliver many of the same benefits of "component-oriented" design that are commonly associated with more complex and proprietary industry standards such as Microsoft's COM model, but on an open, cross-platform basis, and with several key features designed to avoid many of the headaches that eventually come to plague users of other runtime object models.

As with most component-based models in use today, RIB components represent individual functions and data elements as collective members of named interfaces. Unlike more familiar object models such as those associated with MS COM and even C++ itself, however, an application's view of a particular RIB interface and its elements is determined completely by the application itself, and not the RIB component provider or any of its header files. The MSS RIB manager implements true dynamic binding at no additional cost in runtime performance. By way of illustration, three different ways to instantiate the same object might look like:

C++ With Static Binding

```
MyStream = new MP3Stream;
```

MS COM With Runtime Binding

```
MP3StreamFactory->QueryInterface(IID_Stream,
                                 (void **) &MyStream);
```

RAD Interface Broker With Runtime Binding

```
STR_OPEN STR_open;
...
HATTRIB OUTPUT_SAMPLE_RATE
...
RIB_INTERFACE_ENTRY MyStreamInterface[] =
{
   FN(STR_attribute),
   FN(STR_open),
   FN(STR_close),
   FN(STR_process),
   FN(STR_set_preference),
   AT("Output sample rate",    OUTPUT_SAMPLE_RATE),
   AT("Output sample width",   OUTPUT_BITS),
   AT("Output channels",       OUTPUT_CHANNELS)
}
HSTREAM STR = RIB_find_file_provider("Audio stream",
                                     "Input file types",
                                     "*.MP3");
RIB_request(STR,
            "Audio stream",
             MyStreamInterface);
MyStreamHandle = STR_open();
```

It's apparent at a glance that the RIB-based example is, if nothing else, more complex. What benefits are being gained with all that extra code? The answer lies in a fundamental aspect of the RIB system's design: RIB interfaces specify content, not structure. RIB components never reveal the exact layout of their interfaces to the application. A RIB interface name is only used to refer to a collection of available, related functions and data attributes. Unlike a C++ class, a RIB interface does not expose implementation details which have to be artificially segregated from the outside world with "private" or "public" labels. And unlike an MS COM interface, a RIB interface does not introduce dangerous dependencies on the underlying language's virtual-function table model, including the number of functions an interface provider offers and the order in which its function prototypes appear in a class declaration. As suggested by the example above, each RIB application is able to specify its own view of the functions and data exposed by its component providers. The binding of the provider's resources with the application's declarations takes place dynamically, via the *RIB_request* function. There are several advantages to this approach:

- No more obsolete header files. A RIB component's header file contains only the typedef'ed prototypes for the functions exported by the component at its time of release. The application declares variables of these types to receive function pointers when *RIB_request* is called to resolve its requested functions by name. Data attributes (named read-only values) and preferences (named readable/writable values) are resolved in a similar manner, through the use of tokens which are passed down from the provider to the caller via *RIB_request*.

- No more broken interfaces. A RIB interface does not contain or imply any implementation details such as the order in which its functions appear in an internal virtual-method table. Consequently, new functions and data members may be exposed in subsequent releases, without "breaking" any existing application compiled with an older version of the component's header file. Similarly, different providers of the same interface (e.g., an MPEG audio decoder and an IMA ADPCM audio decoder) can export provider-specific attributes and functions, which the application can either enumerate or ask for explicitly at runtime. Applications are also free to request functions and attributes which

may not be supported by all providers of a given interface, taking advantage of optional features when available and degrading gracefully otherwise.

- Full runtime browsability. All communication between RIB components and their calling applications takes place via functions, attributes, and preferences whose names are presented as plain-text ASCII strings. This applies as well to the names of the components' interfaces themselves. The RIB manager API allows the names, members, and types of all interface elements to be enumerated in a standardized fashion, making it possible for general-purpose component browsers to be developed for all providers of a given interface.

What does this mean to you, as the creator of your own MSS applications? The good news is that we've made it easy for you to take advantage of the new RIB providers without getting involved in the details of the RIB component model. MSS's API has been expanded to make access to both the 3D audio and MPEG compression feature sets as painless as possible. Most application authors will never come into direct contact with the RIB manager API that's built into MSS. For the curious, however, the complete RIB API documentation is available in RIB.H, with the new audio stream interface documented in MSSASI.H. Other RIB interfaces, including the mixer services and 3D audio services, are so intimately associated with the MSS system that their declarations are part of MSS.H itself.

In short, RAD believes that the component-oriented RAD Interface Broker model is the best way to make sure that MSS grows with your needs. We feel it's one of the more exciting features of the new system, even though its implications may be lost amidst the excitement of the new features and tools!

Chapter 7

Implementing Voice Chat

7.1 Implementing Voice Chat

Discussion

When implementing voice chat, you use a combination of the Voxware voice chat codecs, the Miles digital input API, the Miles digital output API, and, finally, a networking layer. Since Miles itself doesn't supply a built-in networking layer (which is tough to do in the general case, because multi-player games need such strict control of the network pipe), it was difficult to decide how to expose and document the Miles voice chat features.

So, what we decided to do in the end, was to create a complete client-server voice chat example using the simplest possible networking layer (TCP with WinSock). With this solution, you can see a fairly full-featured application that you can adapt to your game's networking layer with little trouble. This means that this chapter will be fairly different than the rest of the Miles documentation - instead of a long list of functions, we'll walk through the client (MSSCHT. CPP) and the server (MSSCHTS.CPP) source code, step-by-step, describing each portion of the code in detail.

7.2 Voxware Voice Chat Codecs

Discussion

The Miles Sound System includes Audio Stream Interface (ASI)-based support for the popular Voxware MetaVoice compression and decompression libraries. Available for use by MSS developers on a royalty-free basis, this package includes the RT29 and RT24 fixed-rate codecs which are capable of compression and decompression of speech at bit rates of 2900 and 2400 bits per second, respectively; and the VR12 variable-rate codec which operates at varying bit rates between 600 and 1800 bits per second. Compared to standard 8-kHz 16-bit monaural voice input at 128,000 bits per second, these bit rates correspond to compression ratios between 50:1 and 100:1!

It's important, however, to understand that the Voxware codecs are not general-purpose audio compression resources. They achieve their spectacular performance by emulating the acoustical characteristics of the human vocal tract with specific, highly-refined DSP techniques and data models. Consequently, the MetaVoice technology should not be viewed as a competitor to IMA ADPCM, MP3, and other popular compression standards, but rather as a complementary technology that addresses the bandwidth/performance shortcomings of conventional codecs with respect to speech compression. Put another way, if you try to compress music or sound-effects samples with Voxware's low-bandwidth codecs, you'll be unlikely to recognize, much less appreciate, the decompressed result.

Using these high-performance codecs effectively in your application requires that you understand their capabilities and limitations, and plan your communications architecture carefully. To use the Miles Sound System's codecs for voice communications in a networked environment, you'll need to be comfortable with four basic aspects of the job. Most of these points are illustrated for demonstration purposes in the MSSCHTC.CPP (client) and MSSCHTS.CPP (server) example programs included with MSS.

7.3 Working with Voice Input

Discussion

Before you can compress and transmit voice data, you'll need to acquire it. We've provided an easy-to-use set of API functions for voice input (see the Digital Input Services chapter), which allow the application to control and monitor the recording of audio data from the user's microphone or other input source. MSSCHTC.CPP demonstrates how to open a device for digital input and obtain incoming data from it by means of a callback function. The speech-compression codecs are not hard-wired into MSS's audio input/output architecture – other voice-input mechanisms may be used by the application, including calling DirectSoundInput or the waveIn API directly for the absolute lowest possible latency. However, three aspects of voice input should be kept in mind regardless of how the data is obtained.

Use 16-bit mono audio at 8 kHz. The Voxware codecs are optimized for most efficient compression and reproduction at a sampling rate of 8 kHz. This is the standard sample rate used in almost all commercial and military speech-communication applications, including the US telephone system. The Voxware ASI codec implementations are designed solely for use with 16-bit PCM audio at 8 kHz, and will not support any other format.

Use full-duplex input. "Full duplex" simply means that the sound hardware is configured for simultaneous, independent recording and playback of audio data. With a full-duplex sound card, you don't have to stop playback in order to record speech, and vice versa. This doesn't mean that you have to leave recording on continuously, or that you have to continually send recorded audio data over the network - just that you don't have to shut down the playback subsystem in order to acquire input data. The Miles Sound System's integrated voice-input API supports only full-duplex audio hardware. Almost any sound card sold in the last five years will support full-duplex operation.

Choose voice-operated, continuous, or "push to talk" operation. Depending on the nature of your application (i.e., how "noisy" it is, how much network bandwidth is available, and under what circumstances your users will want to communicate) you'll need to decide how to control the flow of speech input.

Voice-operated (VOX) control may be appropriate in cases where the application doesn't generate a continuous stream of loud, raucous sound effects that would tend to cause false triggering. This can be implemented by leaving the input subsystem active all the time, and watching the incoming PCM input data buffers for an average signal level that exceeds a predetermined threshold, indicating that the user is speaking. Since you'll be dealing with 16-bit signed PCM data, you can simply examine the magnitude (absolute value) of individual samples or groups of samples in the buffer to determine their relative loudness. Typically it makes sense to average amplitude values over a period of time - several dozen milliseconds or more - in order to make sure the user has really begun speaking. VOX control may be an especially good match for applications that are bundled with telephone-operator-style headsets, or that otherwise require their use.

Continuous speech streaming may be useful in cases where network bandwidth is not particularly constrained, and/or your application supports only a few simultaneous users. This is the model implemented by the MSSCHTC and MSSCHTS example programs. It simply implies that the input API is left active on a constant basis, with all incoming data compressed and transmitted to the server for mixing and broadcasting.

Push-to-talk (PTT) offers a worthwhile compromise between the VOX and continuous operation models. The input subsystem may or may not be active on a continuous basis, but voice data is compressed and transmitted to the server only when the user holds down a particular key or input-device button. In noisy game environments with many simultaneous users, PTT is likely to be the mode of choice. In a bandwidth-saving variation on the PTT theme, each client could ask the server for "permission" to speak upon receiving a PTT input, allowing the server to limit the amount of incoming data from clients who are trying to speak at the same time.

7.4 Accessing the Codecs Directly with the RIB Interface

Discussion

Most MSS applications don't need to access the library's ASI (Audio Stream Interface) layer directly - the details of communicating with individual ASI providers are encapsulated by high-level library functions such as the streaming and quick-integration APIs. However, real-time voice chat is an exception. To achieve good latency performance and full control over the compression/decompression process, it's necessary for the application to gain low-level access to the Voxware codecs by means of the ASI interface.

The ASI standard represents a very simple, generalized interface for MSS-compatible audio compression and decompression codecs. ASI uses the RAD

Interface Broker (RIB) architecture to allow various codecs to be enumerated and selected on the basis of any of their attributes and capabilities, including the standard file suffix of the data format they support. (See the RIB Architecture chapter for more information on the RIB interface model.) MSSCHTC.CPP and MSSCHTS.CPP demonstrate the process of selecting an ASI codec to handle a given input or output data format. The ASI codecs responsible for delivering the Voxware MetaVoice functionality include:

MSSV29.ASI: 2900-bps high-quality fixed-rate compressor/decompressor. Suffix: ".V29"

MSSV24.ASI: 2400-bps fixed-rate compressor/decompressor. Suffix: ". V24"

MSSV12.ASI: 1200-bps low-bandwidth variable-rate compressor/decompressor. Suffix: ".V12"

By using each ASI codec's file suffix as a descriptor of the data format in use, MSSCHTC and MSSCHTC illustrate a way to make your chat system independent of any particular data format. When you run the Miles chat front-end (MSSCHAT.EXE), it calls MSSCHTC.EXE with the codec to use specified on the Windows command line. The client will try to find an ASI codec capable of both compression and decompression of data with the specified suffix type. Likewise, it will inform the server during the logon process that all data coming from the client will be compressed in the specified format, and that all data sent back to the client should be delivered in the same format. If both the client and the server are able to locate an ASI provider for the specified format, the link can be successfully established.

As an example of direct ASI codec access, MSSCHTC uses RIB services to locate an ASI encoder (compressor) for the data format specified on its command line.

```
//
// Find ASI provider to encode voice data for transmission
//
transmit_encoder = RIB_find_file_provider("ASI codec",
                                          "Output file types",
                                           suffix);
if (transmit_encoder == NULL)
{
   show_error("Error: No ASI provider available to encode data type %s\n", suffix);
   exit(1);
}
show_success("Using ASI provider %X to transmit data type %s\n",
             transmit_encoder,
             suffix);
```

Once the transmit encoder provider is obtained, MSSCHTC calls RIB to obtain an interface to various functions supported by the ASI codec. (As with all RIB components, it is the requestor's responsibility to specify the interface it wants to use when communicating with a RIB service provider).

```
//
// Get ASI stream interface for transmit encoder
//
RIB_INTERFACE_ENTRY ASISTR[] =
{
   { RIB_FUNCTION, "ASI_stream_open",          (U32) &XMIT_stream_open,          RIB_NONE },
```

```
   { RIB_FUNCTION, "ASI_stream_close",          (U32) &XMIT_stream_close,          RIB_NONE },
   { RIB_FUNCTION, "ASI_stream_process",        (U32) &XMIT_stream_process,        RIB_NONE }
}
RIB_request(transmit_encoder,"ASI stream",ASISTR);
//
// Open stream with codec, registering callback function
//
transmit_stream = XMIT_stream_open(0,
                                   transmit_stream_callback,
                                   0);
if (transmit_stream == NULL)
{
   show_error("Could not open stream encoder\n");
   exit(1);
}
```

Subsequently, a similar block of code is used to locate an ASI provider capable of decompressing the same data format as it is received from the server. The only difference in this case is the use of the "Input file types" attribute as the search key passed to *RIB_find_file_provider*.

```
//
// Search for ASI codec capable of processing specified input file type
//
// This ASI provider will be used to decode data sent from the server
// to the client
//
receive_decoder = RIB_find_file_provider("ASI codec",
                                         "Input file types",
                                          suffix);
if (receive_decoder == NULL)
{
   show_error("Error: No ASI provider available to decode data type %s\n",
              suffix);
   exit(1);
}
```

To illustrate a "well-behaved" application's interaction with an ASI provider, the client receive provider's interface contains requests for a few key attributes and preferences, in addition to the same basic functions as those previously obtained from the transmit provider:

```
//
// Get ASI stream interface for receive decoder
//
RIB_INTERFACE_ENTRY RECVSTR[] =
{
   { RIB_FUNCTION,   "ASI_stream_attribute", (U32) &RECV.ASI_stream_attribute,    RIB_NONE },
   { RIB_FUNCTION,   "ASI_stream_open",      (U32) &RECV.ASI_stream_open,         RIB_NONE },
   { RIB_FUNCTION,   "ASI_stream_seek",      (U32) &RECV.ASI_stream_seek,         RIB_NONE },
   { RIB_FUNCTION,   "ASI_stream_close",     (U32) &RECV.ASI_stream_close,        RIB_NONE },
   { RIB_FUNCTION,   "ASI_stream_process",   (U32) &RECV.ASI_stream_process,      RIB_NONE },
   { RIB_FUNCTION,   "ASI_stream_set_preference", (U32) &RECV.ASI_stream_set_preference,
                                                                                  RIB_NONE },
   { RIB_ATTRIBUTE,  "Output sample rate",   (U32) &RECV.OUTPUT_SAMPLE_RATE,      RIB_NONE },
   { RIB_ATTRIBUTE,  "Output sample width",  (U32) &RECV.OUTPUT_BITS,             RIB_NONE },
   { RIB_ATTRIBUTE,  "Output channels",      (U32) &RECV.OUTPUT_CHANNELS,         RIB_NONE },
   { RIB_PREFERENCE, "Requested sample rate",(U32) &RECV.REQUESTED_RATE,          RIB_NONE },
};
RIB_request(receive_decoder,"ASI stream",RECVSTR);
//
// Open the receiver stream
//
// If the codec needs to inspect the stream data to
// configure itself, this call will block in the RECV_stream_CB()
// handler until the requested amount of data (typically only
// the first few bytes of the source stream) is received from the
// client.
//
RECV_read_cursor = 0;
RECV_write_cursor = 0;
RECV.stream = RECV.ASI_stream_open(0,
                                   RECV_stream_CB,
                                   0);
if (RECV.stream == NULL)
{
   show_error("Could not open stream decoder\n");
   exit(1);
}
```

These attributes and preferences are used to request operation at the standard 8 kHz sample rate (HW_RATE) and to obtain the actual data format used by the codec to configure the output stream's HSAMPLE:

```
//
// Request codec output rate which matches hardware rate
//
U32 req_rate = HW_RATE;
RECV.ASI_stream_set_preference(RECV.stream,
                               RECV.REQUESTED_RATE,
                              &req_rate);
//
// Stream is now open -- get its attributes and set output
// sample attributes accordingly
//
U32 nch  = RECV.ASI_stream_attribute(RECV.stream, RECV.OUTPUT_CHANNELS);
U32 rate = RECV.ASI_stream_attribute(RECV.stream, RECV.OUTPUT_SAMPLE_RATE);
U32 bits = RECV.ASI_stream_attribute(RECV.stream, RECV.OUTPUT_BITS);
S32 type;
if (nch == 2)
{
   type = ( (bits == 16) ? DIG_F_STEREO_16 : DIG_F_STEREO_8 ) |
          ( (bits == 16) ? DIG_PCM_SIGN    : 0 );
}
else
{
   type = ( (bits == 16) ? DIG_F_MONO_16   : DIG_F_MONO_8 | )
          ( (bits == 16) ? DIG_PCM_SIGN    : 0 );
}
AIL_init_sample             (stream);
AIL_set_sample_playback_rate(stream, rate);
AIL_set_sample_type         (stream, type, 0);
receive_buffer_size = AIL_minimum_sample_buffer_size(dig, rate, type);
show_info("Receive stream format: %d channels, %d Hz, %d bits\n",nch,rate,bits);
show_info("Receive buffer size=%d\n",receive_buffer_size);
```

In reality, as noted previously, all MSS ASI providers used for voice communication are internally hardwired for operation with 16-bit monaural PCM audio at a sample rate of 8 kHz, so the above code to query the stream attributes is not, strictly speaking, necessary. However, besides being good programming (or at least typing) practice, it serves as an example of how to obtain attributes and set preferences for an ASI stream. This general technique will be useful if the application wishes to take advantage of the Voxware warping and comfort-noise masking features described below.

Once the transmit and receive providers have been selected and their corresponding streams opened successfully, all that remains to be done on the client side with respect to ASI is the actual encoding and decoding of data. The following except from *receive_ASI_thread_procedure* in MSSCHTC.CPP illustrates the use of the *ASI_stream_process* function to turn compressed audio data back into playable PCM format:

```
//
// If fetch buffer empty, call ASI decoder to acquire decompressed data
// from server
//
// This will block in RECV_stream_CB() until enough data is received
// from the server to fill the fetch buffer
//
// Using the temporary fetch buffer allows us to read and decompress
// data while all of the receive buffers are full -- otherwise, we'd
// have to spin until an empty receive buffer becomes available
//
if (!fetch_buffer_full)
{
   S32 amount = RECV.ASI_stream_process(RECV.stream,
                                        fetch_buffer,
                                        receive_buffer_size);
   if (amount != receive_buffer_size)
   {
      //
      // Bad read
      //
```

```
      active = 0;
      return 0;
   }
   fetch_buffer_full = 1;
}
```

7.5 Performing the Compression and Decompression

Discussion

Under the ASI standard, all compression and decompression takes place within the provider's *ASI_stream_process* function. This function is extremely open-ended in nature, and completely symmetrical with respect to compression and decompression. The application simply calls it with a request for any desired amount of compressed or decompressed data, from bytes to megabytes. *ASI_stream_process* fulfills as much of the application's request as possible from the provider's own internal buffers, if any data is left over from a previous call. Subsequently, it invokes the application callback function that was originally passed to *ASI_stream_open* to request that the application send it enough source data to enable it to fulfill the compression or decompression request. In this case, MSSCHTC.CPP's *RECV_stream_CB* function is called by the *ASI_stream_open* handler to obtain compressed data from the server, one "frame" at a time, until enough frames of data have arrived to fulfill the application's request:

```
// ----------------------------------------------------------------------------
// RECV_stream_CB()
// Runs from client receive thread
// ----------------------------------------------------------------------------
S32 CALLBACK RECV_stream_CB(U32       user,
                            void FAR *dest,
                            S32       bytes_requested,
                            S32       offset)
{
   //
   // A seek offset of 0 occurs only when the stream is opened (initial seek to
   // beginning to determine stream characteristics) and when the first frame
   // of the stream is read after opening (when actually beginning to stream
   // data).  Since we are dealing with a non-seekable input stream, we
   // must handle this second case by maintaining a separate read and write
   // cursor in the frame buffer.  The second 0-offset seek is guaranteed
   // to occur while the initial frame data is still in the buffer.
   //
   // Most voice codecs won't perform any seek operations at all -- this
   // functionality is here primarily to support MP3 streaming and similar
   // operations.  A chat server designed strictly for use with the Voxware
   // codecs can safely ignore the offset parameter altogether.
   //
   //
   if (offset != -1)
   {
      RECV_read_cursor = offset;
   }
   //
   // Data will typically be requested one frame at a time by the ASI codec
   // (although frame header components may be requested separately, a few words
   // at a time).  A return value of less than bytes_requested indicates the
   // end of the stream has been reached.  We don't support fragmented frames,
   // so we'll return 0 to the codec if the client disconnects while we're
   // polling it for data.  Any ASI codec designed for compatibility with
   // IP streaming MUST accept 0-byte return values at any stage.
   //
   S32 needed = bytes_requested;
   while (1)
   {
      //
      // Get as much data as possible from frame buffer
      //
```

```
      if (RECV_read_cursor < RECV_write_cursor)
      {
         S32 n = min(needed,
                     RECV_write_cursor - RECV_read_cursor);
         memcpy(dest,
                &RECV_frame[RECV_read_cursor],
                 n);
         dest              = ((C8 *) dest) + n;
         RECV_read_cursor += n;
         needed           -= n;
      }
      //
      // If all requested data has been read, return
      //
      if (needed <= 0)
      {
         //
         // Keep read/write cursors in first half of frame buffer
         //
         if (RECV_read_cursor >= RECEIVE_FRAME_SIZE)
         {
            RECV_read_cursor  -= RECEIVE_FRAME_SIZE;
            RECV_write_cursor -= RECEIVE_FRAME_SIZE;
         }
         return bytes_requested;
      }
      //
      // We need more data -- read it into the frame buffer
      //
      S32 result = NET_poll_for_data(&RECV_frame[RECV_write_cursor],
                                     needed);
      if (result == -1)
      {
         //
         // Client disconnected, return 0 to abort current frame
         //
         return 0;
      }
      if (result == 0)
      {
         //
         // No data available yet
         //
         Sleep(10);
      }
      else
      {
         //
         // Advance the write cursor
         //
         RECV_write_cursor += result;
      }
      //
      // Block until more data comes in or server disconnects
   }
}
```

As the comments suggest, this code is somewhat more robust than it has to be for simple voice communication with the Voxware codecs. Many ASI decompressors, notably the MP3 decoder, can output different PCM data formats depending on the exact format of the compressed source data. In such a case, the application needs to be able to determine the sample format required to play the decoded data immediately after calling the *ASI_stream_open* function. So a typical ASI codec must examine the first few bytes of the data when opening a stream, in order to report the stream's format to the application. This is the only case in which a "seek" operation must be supported by the codec when dealing with streamed data, so it's a good practice to support valid offset parameters in an ASI decompressor's callback function if you expect to support more general codecs in the future. (Typically the ability to seek 1 or 2 KB backwards in a stream after the first frame is fetched is sufficient for most popular data formats).

Turning uncompressed PCM data from the input API or data file into compressed data suitable for transmission to the server is an almost identical process,

thanks to the inherent symmetry of the ASI specification. MSSCHTC.CPP's *transmit_ASI_thread_procedure* calls the transmit ASI provider as follows:

```
//
// See if more transmit data is needed
//
if (current_send_offset == TRANSMIT_PACKET_SIZE)
{
   //
   // Request data from ASI encoder to send to server
   //
   // This will block in XMIT_stream_CB( ) until enough input
   // data is available to satisfy the request
   //
   S32 amount = XMIT_stream_process(transmit_stream,
                                    send_buffer,
                                    TRANSMIT_PACKET_SIZE);
   if (amount != TRANSMIT_PACKET_SIZE)
   {
      //
      // Bad read
      //
      active = 0;
      return 0;
   }
   current_send_offset = 0;
}
```

As in the receiver code above, the *XMIT_stream_process* function calls the application-supplied callback function *XMIT_stream_CB* to fetch enough data to fulfill the application's request. In this case, *XMIT_stream_CB* blocks until sufficient data is available from the input API:

```
// ----------------------------------------------------------------------------
// XMIT_stream_CB( )
// ----------------------------------------------------------------------------
S32 CALLBACK XMIT_stream_CB(U32       user,
                            void FAR *dest,
                            S32       bytes_requested,
                            S32       offset)
{
   //
   // Block transmission thread in this routine until specified
   // number of bytes available from input, or disconnection occurs
   //
   S32 bytes_sent = 0;
   while (bytes_requested > 0)
   {
      //
      // Allow other threads to run
      //
      Sleep(3);
      //
      // Exit if transmit thread killed by main thread
      //
      // Returning 0 here will cause transmit_thread_procedure() to exit
      // when its ASI_stream_process( ) call fails
      //
      if (!threads_active)
      {
         return 0;
      }
      //
      // Get input buffer to send
      //
      if ((input_buffer_tail == -1) ||
          (input_buffer_tail == input_buffer_head))
      {
         //
         // No input buffers available
         //
         continue;
      }
      //
      // Transmit data from input buffer at queue tail
      //
      C8 FAR *src       = &input_buffer [input_buffer_tail] [input_read_offset];
      S32     src_avail = INPUT_BUFFER_SIZE - input_read_offset;
      S32 send_amount = min(bytes_requested,
                            src_avail);
      memcpy(dest,
             src,
             send_amount);
      dest              = ((C8 *) dest) + send_amount;
```

```
            input_read_offset += send_amount;
            bytes_requested   -= send_amount;
            bytes_sent        += send_amount;
            if (input_read_offset == INPUT_BUFFER_SIZE)
            {
               //
               // Entire buffer has been sent; advance tail pointer
               //
               if (input_buffer_tail == N_INPUT_BUFFERS-1)
               {
                  input_buffer_tail = 0;
               }
               else
               {
                  ++input_buffer_tail;
               }
               input_read_offset = 0;
            }
         }
         return bytes_sent;
      }
```

Use of the ASI codec interface in MSSCHTS.CPP is very similar to the above examples in MSSCHTC.CPP.

7.6 Integrating the Codecs with your Networking Architecture

Discussion

When it comes to picking a topology for your voice-networking subsystem, several options are available. Which one is right for your application will be determined by such factors as the nature of your application's existing network support (if any), availability of server-side and client-side network bandwidth, and availability of server-side and client-side processing power.

In a pure client-server model, as illustrated by the MSSCHTC and MSSCHTS example applications, only one compression and one decompression thread run on each client. Data is acquired at the client, compressed, and transmitted to the server. At the server side, incoming data from each client is decompressed and mixed with data from other clients, then transmitted back to each client. Each stream transmitted from the server must be mixed and compressed discretely to avoid echoing a given client's contribution to the mix back to that client. Server-side I/O bandwidth and processing overhead can be quite high unless special measures are taken to optimize bandwidth. Such measures might include permission-to-talk schemes such as outlined in the Push-to-talk section above, and/or simply limiting the number of clients allowed to contribute to the output mix at any given time.

A peer-to-peer model implies that each client runs one compression thread and as many decompression threads as are necessary to receive speech from all other clients who may be speaking. All decompression and mixing takes place redundantly at each client, and all compressed data must be sent to every other client. Lacking an adequate IP-multicast-compatible transport layer, the pure peer-to-peer model does not scale adequately beyond a few clients in the

general case. However, the peer-to-peer model may represent the simplest and most efficient way to implement person-to-person IP telephony applications and two-player games.

A distributed-server model may be an excellent compromise in massively-multiplayer games and online communities, especially in cases where centralized server resources are limited or nonexistent. In a distributed-server environment, each client station acts as a mixing server for a limited number of other clients, typically partitioned by "channels" or other logical divisions between groups of users that don't need to talk to each other on a continuous basis. Communication between any two users may still be established on demand by means of appropriate interfaces in the game or online environment, subject to a given number of simultaneous "connections" allowed at each client.

7.7 Mixing the Client Sound Data on the Server

Discussion

Any model other than peer-to-peer will require that data be buffered and mixed on one or more servers, in preparation for transmission to clients. Fortunately, given the tight constraints on PCM data formats used for voice communication, it's difficult to imagine an easier audio programming job. Mixing of 16-bit signed PCM data is accomplished by means of simple algebraic addition of the corresponding samples from each client buffer, followed by a hard clamping operation to avoid distortion due to sample data overflow. The lack of any need for sample rate conversion, volume scaling, or reformatting enables a simple C/C++ loop to do the job efficiently.

It's important to mix the data from each client at higher-than-16-bit precision, performing the clipping operation only after all samples have been added. Doing so actually reduces the amount of sample points that ultimately have to be clipped, since each incoming waveform sample point has a roughly equal chance of causing an overflow condition or canceling an existing one. MSSCHTS.CPP illustrates how 16-bit sample data may be mixed at 32-bit precision:

```
// -------------------------------------------------------------
// Merge data from all active clients into each client's
// mixer buffer
// -------------------------------------------------------------
for (i=0; i < N_CLIENTS; i++)
{
   CLIENT *C = &clients[i];
   //
   // If not actively streaming, skip further foreground processing
   //
   if (!C->in_use)
   {
      continue;
   }
   //
   // Clear client's mixer buffer to signed-PCM 0 value (0x0000)
   //
   memset(C->mixer_buffer,
          0,
          sizeof(C->mixer_buffer));
   //
```

```
   // Add samples from each contributing client's decode buffer to
   // target's mixing buffer
   //
   for (S32 j=0; j < N_CLIENTS; j++)
   {
      CLIENT *SRC = &clients[j];
      //
      // Avoid echoing each client's own traffic back to it, unless
      // we've enabled local echo as a diagnostic option
      //
      if (SendMessage(hear,BM_GETCHECK,0,0)==0)
      {
        if (i == j)
        {
           continue;
        }
      }
      //
      // Skip inactive or newly-logged-on clients that don't have
      // enough data buffered yet
      //
      if (SRC->samples_to_mix == 0)
      {
         continue;
      }
      //
      // Add 16-bit data to 32-bit output buffer
      //
      S16 *src  = SRC->fetch_buffer;
      S32 *dest = C->mixer_buffer;
      for (S32 m=0; m < samples; m++)
      {
         dest[m] += src[m];
      }
   }
   //
   // Clip and copy contents of client's mixer buffer to output buffer
   //
   C8 output_buffer[TRANSMIT_QUEUE_SIZE];
   S32 *src  =         C->mixer_buffer;
   S16 *dest = (S16 *) output_buffer;
   for (S32 m=0; m < samples; m++)
   {
      S32 s = src[m];
      if (s < -32768)
      {
         *dest++ = -32768;
      }
      else if (s > 32767)
      {
         *dest++ = 32767;
      }
      else
      {
         *dest++ = S16(s);
      }
   }
   //
   // Add buffer to transmit queue
   //
   C->transmit->put_data(output_buffer,
                         samples * sizeof(S16));
}
```

7.8 Implementation Details

Discussion

MSSCHTC.CPP and MSSCHTS.CPP both rely heavily on multithreading to keep the various compression and decompression processes out of each others' way. Most of the execution time in each thread is spent waiting for data to become available from either WinSock or the mixer. The Voxware ASI codecs are guaranteed to be 100% thread-safe, but especially on the server side with two threads per client, it makes sense to be extremely careful with any shared variables and data structures to avoid deadlocks. MSSCHTS.CPP's mixer

loop, which runs in the same thread as *WinMain*, uses CQueue, a rudimentary circular-buffer template, to isolate data being received and transmitted to each client.

Note that the data structures and synchronization methods in MSSCHTS and MSSCHTC have been chosen primarily for simplicity, and are intended more for illustrative purposes than as an attempt to wring every last cycle of performance out of the Win32 thread scheduler and WinSock API.

7.9 Bandwidth Optimization and Channel Reliability Tips

Discussion

Any real-world implementation of networked voice communication is likely to be a complex affair. Below are some hints on how to get the most out of the MSS voice chat codecs.

Use UDP networking. The MSSCHTS and MSSCHTC example programs delivered with MSS use TCP/IP with the Winsock 1.X API, for simplicity. It's important to note, though, that TCP/IP is a stream-oriented rather than a packet-oriented protocol. TCP/IP provides certain guarantees of data integrity, but those guarantees come at a price: packet size and delivery latency. Because the MSS chat codecs operate on a frame-by-frame basis, it's more efficient to transport the data for each frame in an independent UDP datagram, rather than forcing the TCP layer to treat the compressed data as fragments of a continuous stream. The UDP protocol imposes significantly less overhead on data communications than TCP, but it does require significantly more programming effort to handle missing or duplicated packets in a robust fashion. If you are working on a high-performance real-time multiplayer game, chances are you're already using UDP. In such a case, your task is simply to add ASI compression and decompression support to your existing network code.

Optimize the size of each transmitted frame. There is a subtle inefficiency in the general-purpose ASI interface that becomes evident when working with the Voxware codecs. The compressed frame size of each ASI codec is a fixed number of bytes, obtainable by querying the "Maximum frame size" attribute on the decompressor provider (or, alternatively, by querying the "Minimum input block size" attribute on an open decompressor stream). The inefficiency arises from the minuscule size of a frame of Voxware-compressed data. The V29 codec uses a 67-bit frame; the V24 codec a 54-bit frame; and the V12 codec deals with frame sizes which vary between 2 and 41 bits. The ASI frame size attribute always reports frame size at byte granularity, resulting in the effective waste of several bits per frame - which is a significant quantity of data at such small frame sizes! If your networking layer is efficient enough to take advantage of otherwise-unused bits in outgoing and incoming voice packets, or if you

wish to pack multiple frames' worth of data into a single transmission unit, you may want to query the S32 "Actual bits encoded last frame" attribute in the encoder's "ASI stream" interface, to determine the exact number of bits in each compressed frame.

Use warping and comfort-noise masking to adapt to varying network conditions. Two of the most powerful, network-friendly features of the Voxware MetaVoice codecs delivered with the Miles Sound System are "warping" and "comfort noise." The ASI codecs expose these features through the F32 "Warp factor" and S32 "Comfort noise frames" preferences in the decoders' "ASI stream" interfaces.

Warping enables the application to slow down or speed up the rate at which the Voxware decoder consumes network data. A warp factor of 1.0 causes decoded data to be produced at the standard Voxware rate of 180 bytes per frame, every frame. A warp factor of greater than 1.0 causes the Voxware codec to "slow down" the generation of output voice data, occasionally generating an extra 180-byte output frame without demanding any new input data. Interestingly, the voice does not undergo any pitch change whatsoever! Similarly, a warp factor of less than 1.0 causes the codec to "speed up" the output voice data, occasionally swallowing an input frame without generating any output data for that frame. By varying the warp factor around 1.0, your code can dynamically adjust for changes in packet arrival times without resorting to large buffers that may substantially increase perceived latency.

Comfort noise is a similar bandwidth-optimization feature. When a network connection becomes so unreliable that the latencies can't be easily handled by warping alone, you can set the "Comfort noise frames" preference to a nonzero value. Subsequent calls to the decoder's *ASI_stream_process* function will not fetch any data from the application callback, but instead will generate a "masking tone" based on the previous characteristics of the stream's voice, such that a human listener is unlikely to notice one or two missing packets. This is substantially preferable to the usual alternative of allowing the stream to drop out completely. Just set the "Comfort noise frames" preference to the number of subsequent frames you would like to mask, and the codec will do the rest.

Neither MSSCHTC.CPP nor MSSCHTS.CPP implements comfort noise or warping, or any other form of packet-arrival timing compensation. To use warping and comfort noise within the context of your own networking code, simply follow these examples:

```
//
// Get preference handles for warping and comfort noise
//
HATTRIB WARP_FACTOR,
COMFORT_NOISE_FRAMES;
RIB_INTERFACE_ENTRY ASISTR[] =
{
   PR("Warp factor",            WARP_FACTOR),
   PR("Comfort noise frames",   COMFORT_NOISE_FRAMES),
};
RIB_request(ASI,"ASI stream",ASISTR);
//
// Adjust Voxware warping/masking preferences
//
```

```
if (/* need to increment warp factor */)
{
   S32 w = ASI_stream_attribute(stream, WARP_FACTOR);
   F32 warp = *(F32 *) &w;
   if (warp < 3.0F) warp += 0.1F;
   ASI_stream_set_preference(stream, WARP_FACTOR, &warp);
}
if (/* need to decrement warp factor */)
{
   S32 w = ASI_stream_attribute(stream, WARP_FACTOR);
   F32 warp = *(F32 *) &w;
   if (warp > 0.1F) warp -= 0.1F;
   ASI_stream_set_preference(stream, WARP_FACTOR, &warp);
}
if (/* need to request 1 frame of comfort noise */)
{
   S32 c = ASI_stream_attribute(stream, COMFORT_NOISE_FRAMES);
   c = c + 1;
   ASI_stream_set_preference(stream, COMFORT_NOISE_FRAMES, &c);
}
```

Chapter 8

Miles Sound System Tools Reference

8.1 Miles Sound Studio

Discussion

The Miles Sound Studio is a utility capable of all of the common tasks that a Miles Sound System user will require. It contains tools for digital sound files, MIDI files, and DLS files. The Miles Sound Studio is available for Windows or MacOS.

All of the neat things that the Miles Sound Studio can do are available from the buttons along the bottom of the window. In Windows, you simply highlight a file, and then click one of the buttons. In MacOS, you drag a file from a Finder folder window onto the button to apply. We'll describe each button next.

The Compress button: this option will compress either a digital wave file, a DLS file, or DLS data embedded in an XMIDI file with IMA ADPCM or MPEG Layer 3 compression. ADPCM compression provides a 4-to-1 space-savings with very little quality loss, MP3 provides 11-to-1 space-saving with no quality loss. ADPCM and MP3 compressed wave files can be played back directly by the Miles Sound System.

Note that compressed DLS files generally have the extension ".MLS".

The Decompress button: this option will decompress either a digital wave file, a MLS file, or MLS data embedded in an XMIDI file that was previously compressed with the Compress option. Note that this isn't a lossless operation - compressing and then decompressing a file will result in different sound data.

The Merge XMI with DLS button: this option will merge together a DLS or MLS file into a MIDI or XMIDI file. This allows you distribute your MIDI files with built-in instruments - kind of like a high-rent MOD file! For further space-savings, you can also choose to remove the instruments that aren't used in the MIDI sequence. You can even compress the instrument data as it is merged into the new file. You must highlight both a DLS and a MIDI file for this option. To highlight more than one file at a time, hold the 'Control' key down as you click on the files.

Note that XMIDI files with embedded DLS data generally have the extension ".MSS".

The Unmerge XMI and DLS button: this option will unmerge the XMIDI and DLS pieces of a merged-XMIDI file into two separate files. This is useful when you need to edit the DLS instruments that were previously embedded into a merged-XMIDI file.

The Filter DLS with MIDI button: this option will use one or more MIDI or XMIDI files to filter out unused instruments in a DLS or MLS file. This option allows you to shrink huge DLS files (General MIDI sets, for example) into smaller DLS files that include only the instruments that are used by the specified MIDI files. You can highlight as many MIDI files as you want to filter with,

so you can even create perfect 'application-wide' DLS files (DLS files that can play any MIDI file in your application). If the data is uncompressed, you can also choose to compress it as it is filtered. You must highlight both a DLS and at least one MIDI file for this option. To highlight more than one file at a time, hold the 'Control' key down as you click on the files.

The Convert to XMIDI button: this option will convert Standard MIDI Format 0 and Format 1 sequence files to the Extended MIDI (XMIDI) format used by the Miles Sound System's Version 2.0 and Version 3.0 drivers. This conversion preserves all of the information needed to faithfully reproduce the sequence's original sound, while discarding elements of the MIDI storage format that are redundant, inefficient, or otherwise unnecessary for real-time playback.

An XMIDI file containing a single translated MIDI sequence often requires from 10% to 30% less storage space, and can be performed with substantially less background processing time. The MIDI data track is compressed mostly by the elimination of all Note Off messages, which is made possible by appending each note's duration in quantization intervals to its original Note On event. Track-specific MIDI meta-events (such as Sequence/Track Name, Instrument Name, and End of Track events) are stripped from the data stream, while a single End of Track meta-event (FF 2F 00) is placed at the end of the XMIDI sequence.

In addition to the prequantized MIDI event stream, the XMIDI sequence format also includes information such as master lists of Global Timbre Library entries needed by the sequence and target offsets for any Sequence Branch Index controllers present in the sequence. To aid developers with specialized needs, XMIDI files are stored in accordance with Electronic Arts' Interchange File Format (EA IFF 85) standard . For example, using conventional IFF access routines, MSS applications may load and interpret XMIDI timbre list (TIMB) chunks manually, using the data to prepare the synthesizer for immediate playback of sequences at a later time.

Refer to the Extended MIDI (XMIDI) Specification *Section* for a detailed description of the XMIDI IFF structure.

Note that XMIDI files generally have the extension ".XMI".

The Convert to RAW button: this option will convert a highlighted digital wave file into a RAW file. A raw file is a digital sound file that has no header or trailer information (sometimes useful in double-buffered sound playback).

Note that RAW sound files generally have the extension ".RAW".

The File Type Button: this option will display the file type of the highlighted file. It's handy when you can't remember if you already embedded DLS data into a file, or whether you compressed it, etc, etc. It can detect PCM wave files, IMA ADPCM wave files, other wave files, standard MIDI files, XMIDI files,

XMIDI files with embedded uncompressed DLS data, XMIDI files with embedded compressed DLS data, uncompressed DLS files, and compressed DLS files.

The MIDI List button: this option will display the contents of the selected MIDI file. This list file will display every MIDI event in the file. It is useful for trouble-shooting problem MIDI files. You can also choose to view the Roland MT-32 / LAPC-1 Programmer's Reference Guide with the MIDI listing. In addition to all standard MIDI file events, this option also recognizes and displays each Miles Sound System-specific meta-event and Extended MIDI Control Change event.

Below is a sample MIDI list:

```
MIDI Listing version 4.0a         Copyright (C) 1991-98 RAD Game Tools, Inc.
------------------------------------------------------------------------------
Offset $0000: Chunk MThd, length $0006
    MIDI file format: 1
    # of tracks: 2
    MIDI division: 120 delta-time ticks per quarter-note
Offset $000E: Chunk MTrk, length $0019
    DTime 0000: meta-event: key signature 0 0
    DTime 0000: meta-event: tempo in uS/MIDI quarter-note = 666666
    DTime 0000: meta-event: time signature 4/4, c = 024, b = 008
    DTime 0000: meta-event: end of track
Offset $002F: Chunk MTrk, length $0048
    DTime 0000: meta-event: sequence/track name "PIANO"
    DTime 0000: meta-event: instrument name "Acou Piano 2"
    DTime 0000: Ch 02 program change to 001
    DTime 0000: Ch 02 main volume MSB = 127
    DTime 0000: Ch 02 note 048 on, attack velocity 091
    DTime 0092: Ch 02 note 048 off
    DTime 0028: Ch 02 note 048 on, attack velocity 088
    DTime 0106: Ch 02 note 048 off
    DTime 0014: Ch 02 note 048 on, attack velocity 098
    DTime 0079: Ch 02 note 048 off
    DTime 0001: Ch 02 note 050 on, attack velocity 088
    DTime 0035: Ch 02 note 050 off
    DTime 0007: Ch 02 note 052 on, attack velocity 090
    DTime 0091: Ch 02 note 052 off
    DTime 0027: Ch 02 MSS callback trigger controller 000
    DTime 0000: meta-event: end of track
```

The DLS List button: this option will display the contents of the selected DLS file. This list file will display all of the instrument information inside the DLS file. You can also choose whether to view the articulation data for each instrument. Refer to the Downloadable Sounds Level 1 Version 1.0 specification, available at the M.M.A. web-site (http://www.midi.org), for guidelines on the interpretation of DLS data files.

8.2 Miles Sound Player

Discussion

The Miles Sound Player is an easy-to-use application for Win32 and Win16 that can play PCM wave files, IMA ADPCM wave files, MPEG Layer 3 files, MIDI files, and XMIDI files. It can play any of the MIDI types with the system's MIDI hardware or with the MSS software synthesizer. The Miles Sound Player is available for Windows or MacOS.

This application is launched from the Miles Sound Studio when you choose to play a sound file.

Controls:

The **slider** allows you to both view and change the current playback position.

The **CPU** indicator tells you what percentage of the CPU is being used by the digital processor and/or software synthesizer.

The **Filter** switch turns on and off sound filtering - filtering takes a little extra CPU time.

The **Reverb** switch turns on and off global reverb - reverb also takes extra CPU time.

The **Rate** pull down allows you to choose the rate for digital synthesis - the faster the rate, the more CPU time is required.

The **16-bit** switch turns on and off 16-bit output mode - 16-bit quality takes very little extra CPU time.

The **Stereo** switch turns on and off stereo output mode - stereo takes extra CPU time.

The **MIDI** pull down allows you to choose five different MIDI playback options:

"Auto-choose MIDI device": tells the player to use the system MIDI hardware for normal MIDI and XMIDI sequences, and to use the software-synthesizer for XMIDI files with embedded DLS data.

"Use MIDI hardware": tells the player to use the system MIDI hardware in all cases.

"Use software synthesizer": tells the player to use the Miles software synthesizer in all cases.

"Use Vortex DLS hardware": tells the player to use the Aureal Vortex DLS hardware for playback.

"Use SonicVibes DLS hardware": tells the player to use the S3 SonicVibes DLS hardware for playback. To use the software synthesizer with normal MIDI and XMIDI sequences, you must also choose a DLS file as the instrument set - click the **"GM DLS file..."** button to choose this DLS file.

8.3 MIDIECHO for Windows - MIDI Data Receiver and Interpreter

Discussion

The **MIDIECHO** for Windows program receives incoming MIDI Channel Voice and System Exclusive messages via a standard Windows midiIn driver, and sends them to a specified Miles Sound System Extended MIDI (XMIDI) driver.

MIDIECHO is a valuable resource for musicians who wish to develop music and sound-effects arrangements for MSS applications. Composers can receive immediate feedback when developing IBM-specific XMIDI compositions on a Macintosh or Atari ST-based MIDI sequencer platform, simply by connecting the host's MIDI interface to a reasonable-quality MIDI interface on a "slave" PC system running **MIDIECHO**.

When starting **MIDIECHO**, you must choose the capture and playback options.

You can choose the drivers for MIDI input, MIDI output, and digital output. Digital output is used for the software synthesizer and digital trigger sound effects with wave library files (now obsolete - you should use DLS for all future digital sample triggering). You can also choose the DLS Provider DLL to use - use MSS32.DLL for the software synthesizer, S3BASE.DLL for Sonic Vibes DLS hardware, or VORT_DLS.DLL for Aureal Vortex DLS hardware.

Once you've made all of your choices, click the Start MIDI Echo button, and the main MIDI Echo text window will appear. When playing complicated MIDI sequences, the display may fall behind the MIDI music - to avoid this, simply press Alt-Enter to switch the screen to full-screen text mode.

When **MIDIECHO** is running, the **+ and -** keys may be used to adjust the "voice alert limit," or the number of notes which MSS assumes the synthesizer in use can play simultaneously. During a MIDI performance, **MIDIECHO** presents a continuously-updated graphical display of polyphonic note activity. When the voice alert limit is exceeded, the portion of the bar graph to the right of the limit indicator will be shown in bright red, and the "total overflows" indicator above the graph will be incremented. This feature may be helpful when diagnosing note "dropout" problems that occur due to partial overflow.

When running on slower PC platforms with dense MIDI traffic, **MIDIECHO** may exhibit undesirable tempo variations due to program overhead. If this problem occurs, the **T** key can be used to disable the continuously scrolling MIDI event trace which appears in **MIDIECHO**'s uppermost window.

The **A** key (All Notes Off) can be pressed at any time to "kill" any stuck notes which may occur while running **MIDIECHO**.

During playback, if a combination of XMIDI Patch Bank Select controller and MIDI Program Change number values is used to set up a timbre which cannot be found in the specified Global Timbre Library file, **MIDIECHO** will complain with an error message. The **E** key can be used to enable or disable a beep tone from the PC speaker when note overflow occurs, or when instrument-loading errors are reported.

The **F** key (Filter) can be pressed to toggle the software synthesizer's filtering mode.

The **R** key (Reverb) can be pressed to toggle the software synthesizer's reverb mode.

MIDIECHO re-transmits incoming data on all 16 MIDI channels to the MSS driver via the AIL_send_channel_voice_message() call. The program re-transmits all MIDI Channel Voice messages. (MIDI System Exclusive messages are retransmitted as well, but this feature should be used with caution since "timing slips" inherent in System Exclusive transmission can cause strange, misleading effects in the music).

MIDIECHO is intended to receive normal MIDI Channel Voice data, not synthesizer-ready XMIDI data. Timing problems may occur if a dense XMIDI data stream (as from a Miles Sound System application or the MSS Version 2. 0 **CAKEPORT** driver) is fed to the **MIDIECHO** host's input port.

8.4 MIDIECHO for MacOS - MIDI Data Receiver and Interpreter

Discussion

The *MIDIECHO* for Mac program receives incoming MIDI Channel Voice and System Exclusive messages via a QuickTime MIDI input, and sends them to a specified Miles Sound System Extended MIDI (XMIDI) driver.

Note: To use a MIDI keyboard, you have to install the Opcode OMS library by Opcode Systems, Inc.

MIDIECHO can be useful for musicians who wish to develop music and sound-effects arrangements for MSS applications. Composers can receive immediate feedback when playing compositions on a MIDI sequencer platform, simply by connecting the hosts MIDI interface to a reasonable-quality MIDI interface on a "slave" Mac system running **MIDIECHO**.

When starting **MIDIECHO**, you must choose the capture and playback options.

With the drop-down menus, "**MIDI Input Device**" and "**MIDI Output Device**", you can choose the drivers for MIDI input and MIDI output.

The "**DLS Bank File**" button allows selecting a DLS file for playing music. If you selected QuickTime as MIDI Output Device, you can skip selection of the DLS Bank File.

The "**WVL File**" button allows selecting a Wave Library file for playing music. If you selected QuickTime as MIDI Output Device, you can skip selection of the Wave Library File.

Once you've made all of your choices, click the OK button, and the main **MIDIECHO** window will appear.

When **MIDIECHO** is running, the voice limit slider may be used to adjust the "voice alert limit", or the number of notes which MSS assumes the synthesizer in use can play simultaneously. During a MIDI performance, **MIDIECHO** presents a continuously updated graphical display of polyphonic note activity.

When the voice alert limit is exceeded, the progress bar begins iterating, and the Total Overflow indicator below the graph will be incremented. This feature may be helpful when diagnosing note "dropout" problems that occur due to partial overflow.

When running on a slower Mac with dense MIDI traffic, **MIDIECHO** may exhibit undesirable tempo variations due to program overhead. If this problem occurs, the MIDI Trace checkbox can be used to disable the continuously scrolling MIDI event trace, which appears in **MIDIECHO**s uppermost window.

The "**All Notes Off**" button can be pressed at any time to "kill" any stuck notes, which may occur while running **MIDIECHO**.

During playback, if a combination of XMIDI Patch Bank Select controller and MIDI Program Change number values is used to set up a timbre which cannot be found in the specified Global Timbre Library file, **MIDIECHO** will complain with an error message.

The "**Error Beep**" checkbox can be used to enable or disable the Alert Sound when note overflow occurs, or when instrument-loading errors are reported.

The "**Filtering**" checkbox can be marked to toggle the software synthesizer's filtering mode.

The "**Reverberation**" checkbox can be marked to toggle the software synthesizer's reverb mode.

MIDIECHO re-transmits incoming data on all 16 MIDI channels to the MSS driver via the AIL_send_channel_voice_message() call. The program retransmits all MIDI Channel Voice messages (MIDI System Exclusive messages are retransmitted as well, but this feature should be used with caution since "timing slips" inherent in System Exclusive transmission can cause strange, misleading effects in the music.)

MIDIECHO is intended to receive normal MIDI Channel Voice data, not synthesizer-ready XMIDI data. Timing problems may occur if a dense XMIDI data stream is fed to the **MIDIECHO** hosts input port.

8.5 MIDIECHO for DOS - MIDI Data Receiver and Interpreter

Discussion

The **MIDIECHO** program receives incoming MIDI Channel Voice and System Exclusive messages via a Roland MPU-401-compatible MIDI interface, and sends them to a specified Miles Sound System Extended MIDI (XMIDI) driver.

MIDIECHO is a valuable resource for musicians who wish to develop music and sound-effects arrangements for MSS applications. Composers can receive immediate feedback when developing IBM-specific XMIDI compositions on a Macintosh or Atari ST-based MIDI sequencer platform, simply by connecting the host's MIDI interface to the MPU-401 interface on a "slave" PC system running **MIDIECHO**.

MIDIECHO has the following command-line syntax:

```
MIDIECHO driver.MDI  [driver.DIG]  [wavelib.WVL]  [GTL_prefix]
        [/MA:xx]     [/MI:xx]
        [/A:xx]      [/I:xx]       [/D:xx]
        [/DA:xx]     [DI:xx]       [/DD:xx]
        [/PR:xx]
```

where driver.MDI specifies the Miles Sound System .MDI driver to be used for playback performance, and **GTL_prefix** optionally indicates the pathname and filename prefix under which Yamaha or Roland MT-32 Global Timbre Library files are stored. Other optional parameters include **/A:xx, /I:xx, and /D:xx,** which specify the hexadecimal I/O port address, IRQ number, and DMA channel to be used by the MSS playback driver, and the **/MA:xx and /MI:xx** parameters which specify the address and IRQ setting of the Roland MPU401-compatible MIDI interface used to receive the incoming data. The **/MA and /MI** parameters are required only if the Roland interface is not addressable at its default jumper settings I/O address $330, IRQ 2. Similarly, **/A and /I, and /D** are necessary only if the sound adapter used for playback is configured at a non-standard I/O address or interrupt setting.

By specifying the filename of an MSS-compatible digital audio driver [**driver.DIG**] and a wave library [**wavelib.WVL**] created by the **WAVELIB** program, it is possible to use **MIDIECHO** as a composition aid for XMIDI-based sound-effect sequences or sequences which use "virtual wave synthesis" to play rhythm or melodic instrument sounds. The **/DA:xx, /DI:xx, and /DD:xx** parameters may be used to specify an explicit port address, interrupt number, and DMA channel for the digital driver, if necessary.

By default, the **MIDIECHO** program included with MSS is configured for a default pitch-bend range of +/- 2 semitones. The **/PR:xx** option permits other pitch-bend range values to be specified without sending MIDI RPN data, which may be useful in cases where **MIDIECHO** is used to compose sequences for playback on the MT-32 or other non-General MIDI platforms (including older MSS 2.0 XMIDI drivers).

While **MIDIECHO** is running, the **+ and -** keys may be used to adjust the "voice alert limit," or the number of notes which MSS assumes the synthesizer in use can play simultaneously. During a MIDI performance, **MIDIECHO** presents a continuously-updated graphical display of polyphonic note activity. When the voice alert limit is exceeded, the portion of the bar graph to the right of the limit indicator will be shown in bright red, and the "total overflows" indicator above the graph will be incremented. This feature may be helpful when diagnosing note "dropout" problems that occur due to partial overflow.

When running on slower PC platforms with dense MIDI traffic, **MIDIECHO** may exhibit undesirable tempo variations due to program overhead. If this problem occurs, the **T** key can be used to disable the continuously scrolling MIDI event trace which appears in **MIDIECHO**'s uppermost window.

The **A** key (All Notes Off) can be pressed at any time to "kill" any stuck notes which may occur while running **MIDIECHO**.

During playback, if a combination of XMIDI Patch Bank Select controller and MIDI Program Change number values is used to set up a timbre which cannot be found in the specified Global Timbre Library file, **MIDIECHO** will complain with an error message. The **E** key can be used to enable or disable a beep tone from the PC speaker when note overflow occurs, or when instrument-loading errors are reported.

MIDIECHO re-transmits incoming data on all 16 MIDI channels to the MSS driver via the AIL_send_channel_voice_message() call. The program re-transmits all MIDI Channel Voice messages. (MIDI System Exclusive messages are retransmitted as well, but this feature should be used with caution since "timing slips" inherent in System Exclusive transmission can cause strange, misleading effects in the music).

MIDIECHO is intended to receive normal MIDI Channel Voice data, not synthesizer-ready XMIDI data. Timing problems may occur if a dense XMIDI data stream (as from a Miles Sound System application or the MSS Version 2. 0 **CAKEPORT** driver) is fed to the **MIDIECHO** host's input port.

8.6 SETSOUND - MSS Sound Configuration Utility

Discussion

The **SETSOUND** program may be distributed with MSS applications as an interactive sound driver installation and configuration utility.

SETSOUND requires no command-line parameters. It is intended to be launched after the application's end-user installation program finishes executing, as well as anytime the end user wishes to reconfigure his/her sound options after

installation is complete. To avoid memory problems inherent with spawned DOS-extended executables, it is advisable to launch **SETSOUND** as the last step in the application's INSTALL.BAT batch file, rather than through the use of the C *spawn* or *exec* functions. Alternately, **SETSOUND.C** can be compiled to **SETSOUND.OBJ** and bound at link time to the application's end-user installation program, simply by renaming **SETSOUND.C's** *main* function.

When executed, **SETSOUND** presents a series of self-explanatory configuration menus. These menus allow the user to select and configure a MIDI music or digital audio driver for use by the MSS application, or to disable sound and music entirely. When either the Select and configure MIDI music driver or Select and configure digital audio driver options are chosen, **SETSOUND** will scan the current working directory for all MSS .DIG or .MDI drivers and present a list of these drivers to the user. All MSS drivers which are shipped with the application appear on the selection list, along with any drivers which are added to the directory by the user himself. This powerful feature ensures that an application which uses **SETSOUND** as its sound configuration program will not become "obsolete" as new MSS drivers are released. In accordance with RAD Game Tools's guidelines for driver installation, **SETSOUND** makes no attempt to install any sound drivers to determine if the supported hardware is present. Instead, DOS environment information (including the **BLASTER, ULTRASND, and SNDSCAPE** environment variables, as well as the contents of **AUTOEXEC.BAT** and **CONFIG.SYS**) is used to help determine which driver should be the "default" choice when the selection menu is first presented.

If the selected driver accepts I/O configuration information, **SETSOUND** next displays a menu offering the user the ability to Attempt to configure sound driver automatically or Skip auto-detection and configure sound driver manually. No attempt is made to auto-detect devices without the user's permission, except in cases where the driver is able to determine the correct I/O configuration without manipulating the hardware directly. (For example, the Pro Audio Spectrum drivers are able to obtain I/O information from the Media Vision device driver **MVSOUND.SYS**, so the configuration menu does not appear when one of these drivers is selected.) If the manual configuration option is selected, a series of menus is presented to obtain the correct I/O address, IRQ setting, and/or DMA channel assignments for the supported device. Extensive error-checking and feedback is employed throughout the installation and configuration process to help reduce sound-related technical support issues.

When a driver is successfully selected and configured, **SETSOUND** creates a plain-text ASCII file (called **MDI.INI** for MIDI music drivers or **DIG.INI** for digital audio drivers) which specifies the driver name, device name, and I/O settings which correspond to the user's sound hardware choice. (An example **.INI** file is presented in the AIL_read_INI() function description. If the No MIDI music or No digital audio option is selected, any existing **.INI** file is deleted entirely.) The **.INI** files created by **SETSOUND** are used by the

AIL_install_DIG_INI() and AIL_install_MDI_INI() functions to load, configure, and initialize the digital audio and/or MIDI music driver selected by the user, with no additional effort on the part of the application program.

To distribute an application with **SETSOUND**, you must distribute **SETSOUND.EXE** and its "driver list file," **MSSDRVR.LST**, and its Window 9x configuration utility, **MSSW95.EXE**. These three files should all exist in the same directory. **MSSW95.EXE** is spawned by **SETSOUND** when running under a DOS box in Windows 9x. It can also be run from the command line to display the machine's Windows 9x configuration.

MSSDRVR.LST is a text file which describes all "known" MSS drivers that existed when the current version of **SETSOUND** was created, and is used by **SETSOUND** to provide additional helpful information about each driver as the user scrolls through the selection menu. Drivers which are not listed in MSSDRVR.LST appear on the **SETSOUND** selection menu, but no supplemental information will be presented to the user for these drivers, and they cannot appear as the default choice in the selection menu. **SETSOUND** uses the current working directory for all of its file access, so all MSS drivers delivered with an application must be copied into a working subdirectory on the user's hard disk during installation, along with MSSDRVR.LST and (preferably) **SETSOUND.EXE** itself. The DIG.INI and MDI.INI files will be created and stored in this subdirectory as well. Whether **SETSOUND** is actually copied to the user's sound directory or left on the application's distribution CD, *please be sure that **SETSOUND** can be executed by the end user at any time after installation!* Otherwise, the user will be "stuck" with his/her original sound configuration choice, with no alternative but to re-install the application when upgrading or changing sound hardware.

The **SETSOUND.EXE** distributed from RAD Game Tools was bound with the Causeway DOS extender. This means that unless you rebuild **SETSOUND.EXE**, you don't need to distribute DOS4GW.EXE. Note that if you rebuild **SETSOUND.EXE**, you must either link it with an .EXE-bound DOS extender such as Causeway, DOS/4GW Professional or FlashTek X/32, or make sure that a copy of the DOS extender (DOS4GW.EXE) is available in the current path at the time **SETSOUND** is executed.

Applications which employ only one type of audio support (either digital audio or MIDI music) should be shipped with drivers of only the required type (.DIG or .MDI). **SETSOUND** will automatically reconfigure its menus to omit all references to the "unsupported" audio format.

Internationalization of the **SETSOUND** program is possible; however, the interactive nature of its text menu system requires modifications to text strings within **SETSOUND.C** itself. The file MSSDRVR.LST may be easily modified to support other languages, but it does not contain all of the text strings used by **SETSOUND**. Contact RAD Game Tools for information on how to modify the specially-formatted text strings used by **SETSOUND** to present its menus

and dialog boxes.

8.7 GLIB - Extended MIDI (XMIDI) Global Timbre Librarian

Discussion

The **GLIB** program creates Global Timbre Library files for use by Miles Sound System Extended MIDI (XMIDI) driver applications.

The Global Timbre Library, or GTL, is the set of timbres, or sound description data for musical instruments or sound effects, which are intended for use by a given synthesizer during the course of an entire Miles Sound System application. Each supported synthesizer may have its own timbre data format, so an application may need as many GTL files as it has XMIDI drivers. Many synthesizers (such as the Ad Lib) contain no built-in timbres, and thus require at least a small GTL file in order to make any sounds at all. Others, such as the Roland MT-32, contain a number of built-in timbres, and do not require GTL support unless the musician wishes to supplement the built-in timbres with "custom" timbres created with a third-party editor program.

GLIB is responsible for creating each Global Timbre Library file according to instructions in a standard ASCII catalog file, or "catfile." The catfile is created by a musician or programmer who is responsible for deciding how some or all of the 128 possible MIDI Program Change ("Patch") numbers are mapped to as many as 16,000 possible timbres of both the built-in and custom types, at various points during the application's execution. A single catfile may serve as the "script" for every Global Timbre Library file used by an application, but the usual practice is to maintain a separate catfile for each individual synthesizer's GTL file.

Each timbre specification in the **GLIB** catalog file consists of an *assignment statement* of the form.

```
timbre(bank,patch) = ...
```

In an assignment statement, the '...' expression on the right side of the assignment statement (or rvalue) tells **GLIB** where to find the source data for the timbre (in an Ad Lib Instrument Maker bankfile, for example). The **timbre( )** expression on the left (or lvalue) specifies the destination "address" of the timbre in the Global Timbre Library file being created. This "address" consists of a timbre **bank number** and a **patch** (MIDI Program Change number) mapping for the timbre. A musician who is composing a MIDI sequence for use with an MSS XMIDI driver may "request" that a custom timbre be associated with a particular patch during playback, simply by prefixing each MIDI Program Change message in the sequence with an XMIDI Patch Bank Select controller

(114) which specifies the bank number under which the desired timbre for the patch was stored in the Global Timbre Library file. For each synthesizer, only one timbre may be mapped to a particular patch at any given time.

The nature of the '...' rvalue expression depends on the GTL file's target synthesizer. The following rvalue expressions are currently supported by **GLIB**:

BNK_inst(string,num): Refers to the name of an Ad Lib instrument to be found in an Ad Lib Instrument Maker bankfile which was previously specified with a BNK_file( ) expression (see below). num normally specifies the number of semitones by which to transpose notes played with the instrument. (This value is added to any transposition value specified with the Ad Lib Gold Instrument Maker.) If the timbre is to be played in MIDI channel 10 (to emulate a Roland MT-32 rhythm key), num specifies the MIDI note number corresponding to the pitch at which the instrument should sound. In such a case, the bank number specified in the lvalue must be 127, while the patch number specifies the MIDI key number which causes the instrument to be played.

MTB_file("filename"): Refers to the name of a Big Noise MT-32 Editor/ Librarian file containing a complete bank of 64 Roland MT-32 timbres. The 64 timbres will be mapped to 64 sequential patches beginning at the patch number given in the timbre( ) lvalue expression. (If this sequential mapping causes any timbre's patch number to exceed 127, the patch bank number is incremented at that point and the patch number itself is incremented modulo 128.) Note that only the timbres, or "tones," in the file are used by **GLIB**; the "bank timbre patch change" and "rhythm key table" information in the file is ignored.

BIN_file("filename"): Refers to the name of a binary file containing a single timbre image in the target Miles Sound System driver's native timbre format. This "generic" expression is useful in cases where the application designer wishes to use a proprietary editor with its own timbre file format, but is not otherwise needed.

In addition to timbre assignment statements, **GLIB** recognizes two "declarative" expressions:

outfile("filename"): Specifies the name of a Global Timbre Library to create. An **outfile( )** specification must appear in the catfile before any timbres are assigned to the outfile. When this statement is encountered, any GTL file being created by a previous **outfile( )** statement is closed and saved to disk.

BNK_file("filename"): Specifies the Ad Lib Instrument Maker bankfile from which subsequent **BNK_inst( )** statements should retrieve timbres for inclusion in the Global Timbre Library file. The Ad Lib Instrument Maker program (version 1.5 or greater) is used to create the .BNK "source file" format for all Yamaha YM3812-based adapters, including the Ad Lib, Sound Blaster, Sound Blaster Pro, Pro Audio Spectrum, Thunderboard, and others.

Beginning with **GLIB** version 1.02, bankfiles from the Ad Lib Gold Instrument Maker program may be used interchangeably with those created by the

original Ad Lib Instrument Maker. The Ad Lib Gold Instrument Maker may be used to create 4-operator timbres for the Ad Lib Gold card and other sound adapters based on the Yamaha YMF262 (OPL3) integrated circuit, as well as 2-operator timbres for standard YM3812-compatible adapters. To guard against inadvertent use of 4-operator timbres in GTL files intended for use with YM3812 drivers, **GLIB** will issue a warning message if any **BNK_inst( )** statement assigns a 4-operator timbre to an output file whose suffix is ".AD".

All characters appearing on a line after a semicolon are considered to be comments, and ignored. Blank lines are also acceptable.

The catalog file used to create a Global Timbre Library is of extreme importance in determining how well a given XMIDI sequence will sound when "tweaked" for optimum Ad Lib playback quality. Supplied with the Miles Sound System is **SAMPLE.CAT**, a sample **GLIB** catalog file which creates an Ad Lib-compatible Global Timbre Library file, and **SAMPLE.BNK**, an Ad Lib Instrument Maker bankfile containing approximately 200 "canned" instruments supplied with the Ad Lib and Sound Blaster developer kits. The sample Ad Lib GTL file **SAMPLE.AD** was created by the following command:

```
GLIB sample.cat
```

The created **SAMPLE.AD** should be used only for development, testing, and experimental purposes, as its instruments are not "adjusted" with the Instrument Maker program to emulate the Roland synthesizer faithfully in important areas such as octave register, envelope timing, relative volume level, and general tonal quality. For shipping purposes, you should usually use The Fat Man's FM patches that are supplied in the **/REDIST/DOS** directory.

As an example of a **GLIB** catalog file, **SAMPLE.CAT** is reproduced here in its entirety.

```
;
;SAMPLE.CAT
;Global Timbre Library assignments for sample MSS V2.0 applications
;
;Usage: GLIB sample.cat
;
outfile("sample.ad")            ;create Ad Lib Global Timbre Library
BNK_file("sample.bnk")          ;name of Ad Lib Instrument Maker bankfile
;
;Bank 0: Melodic instruments to emulate Roland MT-32 built-in timbres
;
timbre(0,0) = BNK_inst("piano1",0)            ;Acou Piano 1
timbre(0,1) = BNK_inst("piano3",0)            ;Acou Piano 2
timbre(0,2) = BNK_inst("piano4",0)            ;Acou Piano 3
timbre(0,3) = BNK_inst("elpiano1",0)          ;Elec Piano 1
timbre(0,4) = BNK_inst("elpiano2",0)          ;Elec Piano 2
timbre(0,5) = BNK_inst("elpiano1",0)          ;Elec Piano 3
timbre(0,6) = BNK_inst("pianof",0)            ;Elec Piano 4
timbre(0,7) = BNK_inst("piano1",0)            ;Honkytonk
timbre(0,8) = BNK_inst("organ2",0)            ;Elec Org 1
timbre(0,9) = BNK_inst("organ1",0)            ;Elec Org 2
timbre(0,10) = BNK_inst("organ1",0)           ;Elec Org 3
timbre(0,11) = BNK_inst("organ4",0)           ;Elec Org 4
timbre(0,12) = BNK_inst("pipes",0)            ;Pipe Org 1
timbre(0,13) = BNK_inst("pipes",0)            ;Pipe Org 2
timbre(0,14) = BNK_inst("pipes",0)            ;Pipe Org 3
timbre(0,15) = BNK_inst("accordn",0)          ;Accordion
timbre(0,16) = BNK_inst("harpsi4",0)          ;Harpsi 1
timbre(0,17) = BNK_inst("harpsi4",0)          ;Harpsi 2
timbre(0,18) = BNK_inst("harpsi4",0)          ;Harpsi 3
timbre(0,19) = BNK_inst("elclav2",0)          ;Clavi 1
timbre(0,20) = BNK_inst("elclav2",0)          ;Clavi 2
timbre(0,21) = BNK_inst("elclav2",0)          ;Clavi 3
```

```
timbre(0,22) = BNK_inst("celesta",0)          ;Celesta 1
timbre(0,23) = BNK_inst("celesta",0)          ;Celesta 2
timbre(0,24) = BNK_inst("sftbrss1",0)         ;Syn Brass 1
timbre(0,25) = BNK_inst("sftbrss1",0)         ;Syn Brass 2
timbre(0,26) = BNK_inst("sftbrss1",0)         ;Syn Brass 3
timbre(0,27) = BNK_inst("sftbrss1",0)         ;Syn Brass 4
timbre(0,28) = BNK_inst("bass2",0)            ;Syn Bass 1
timbre(0,29) = BNK_inst("bass2",0)            ;Syn Bass 2
timbre(0,30) = BNK_inst("bass2",0)            ;Syn Bass 3
timbre(0,31) = BNK_inst("bass2",0)            ;Syn Bass 4
timbre(0,32) = BNK_inst("fantapan",0)         ;Fantasy
timbre(0,33) = BNK_inst("mars",0)             ;Harmo Pan
timbre(0,34) = BNK_inst("syn1",0)             ;Chorale
timbre(0,35) = BNK_inst("tincan1",0)          ;Glasses
timbre(0,36) = BNK_inst("mars",0)             ;Soundtrack
timbre(0,37) = BNK_inst("moon",0)             ;Atmosphere
timbre(0,38) = BNK_inst("trainbel",0)         ;Warm Bell
timbre(0,39) = BNK_inst("synbass1",0)         ;Funny Vox
timbre(0,40) = BNK_inst("bells",0)            ;Echo Bell
timbre(0,41) = BNK_inst("bells",0)            ;Ice Rain
timbre(0,42) = BNK_inst("oboe1",0)            ;Oboe 2001
timbre(0,43) = BNK_inst("bass1",0)            ;Echo Pan
timbre(0,44) = BNK_inst("bass2",0)            ;Doctor Solo
timbre(0,45) = BNK_inst("bass2",0)            ;Schooldaze
timbre(0,46) = BNK_inst("javaican",0)         ;Bellsinger
timbre(0,47) = BNK_inst("csynth",0)           ;Square Wave
timbre(0,48) = BNK_inst("strings1",0)         ;Str Sect 1
timbre(0,49) = BNK_inst("strnlong",0)         ;Str Sect 2
timbre(0,50) = BNK_inst("strings1",0)         ;Str Sect 3
timbre(0,51) = BNK_inst("koto1",0)            ;Pizzicato
timbre(0,52) = BNK_inst("violin",0)           ;Violin 1
timbre(0,53) = BNK_inst("violin1",0)          ;Violin 2
timbre(0,54) = BNK_inst("violin",0)           ;Cello 1
timbre(0,55) = BNK_inst("cello",0)            ;Cello 2
timbre(0,56) = BNK_inst("contrab",0)          ;Contrabass
timbre(0,57) = BNK_inst("harp1",0)            ;Harp 1
timbre(0,58) = BNK_inst("harp",0)             ;Harp 2
timbre(0,59) = BNK_inst("guitar1",0)          ;Guitar 1
timbre(0,60) = BNK_inst("guitar1",0)          ;Guitar 2
timbre(0,61) = BNK_inst("elguit2",0)          ;Elec Gtr 1
timbre(0,62) = BNK_inst("elguit1",0)          ;Elec Gtr 2
timbre(0,63) = BNK_inst("sitar1",0)           ;Sitar
timbre(0,64) = BNK_inst("bbass",0)            ;Acou Bass 1
timbre(0,65) = BNK_inst("bass2",0)            ;Acou Bass 2
timbre(0,66) = BNK_inst("bass2",0)            ;Elec Bass 1
timbre(0,67) = BNK_inst("bass2",0)            ;Elec Bass 2
timbre(0,68) = BNK_inst("bbass",0)            ;Slap Bass 1
timbre(0,69) = BNK_inst("bbass",0)            ;Slap Bass 2
timbre(0,70) = BNK_inst("bass1",0)            ;Fretless 1
timbre(0,71) = BNK_inst("bass1",0)            ;Fretless 2
timbre(0,72) = BNK_inst("flute1",0)           ;Flute 1
timbre(0,73) = BNK_inst("flute1",0)           ;Flute 2
timbre(0,74) = BNK_inst("flute1",0)           ;Piccolo 1
timbre(0,75) = BNK_inst("flute1",0)           ;Piccolo 2
timbre(0,76) = BNK_inst("flute2",0)           ;Recorder
timbre(0,77) = BNK_inst("flute2",0)           ;Pan Pipes
timbre(0,78) = BNK_inst("sax1",0)             ;Sax 1
timbre(0,79) = BNK_inst("sax1",0)             ;Sax 2
timbre(0,80) = BNK_inst("sax1",0)             ;Sax 3
timbre(0,81) = BNK_inst("softsax",0)          ;Sax 4
timbre(0,82) = BNK_inst("clar1",0)            ;Clarinet 1
timbre(0,83) = BNK_inst("clarinet",0)         ;Clarinet 2
timbre(0,84) = BNK_inst("oboe",0)             ;Oboe
timbre(0,85) = BNK_inst("oboe",0)             ;Engl Horn
timbre(0,86) = BNK_inst("bassoon",0)          ;Bassoon
timbre(0,87) = BNK_inst("harmonca",0)         ;Harmonica
timbre(0,88) = BNK_inst("trumpet4",0)         ;Trumpet 1
timbre(0,89) = BNK_inst("trumpet4",0)         ;Trumpet 2
timbre(0,90) = BNK_inst("tromb2",0)           ;Trombone 1
timbre(0,91) = BNK_inst("tromb1",0)           ;Trombone 2
timbre(0,92) = BNK_inst("frhorn1",0)          ;Fr Horn 1
timbre(0,93) = BNK_inst("frhorn2",0)          ;Fr Horn 2
timbre(0,94) = BNK_inst("tuba1",0)            ;Tuba
timbre(0,95) = BNK_inst("brass1",0)           ;Brs Sect 1
timbre(0,96) = BNK_inst("brass2",0)           ;Brs Sect 2
timbre(0,97) = BNK_inst("vibra2",0)           ;Vibe 1
timbre(0,98) = BNK_inst("vibra3",0)           ;Vibe 2
timbre(0,99) = BNK_inst("marimba",0)          ;Syn Mallet
timbre(0,100) = BNK_inst("belshort",0)        ;Windbell
timbre(0,101) = BNK_inst("belshort",0)        ;Glock
timbre(0,102) = BNK_inst("belshort",0)        ;Tube Bell
timbre(0,103) = BNK_inst("xylofone",0)        ;Xylophone
timbre(0,104) = BNK_inst("marimba",0)         ;Marimba
timbre(0,105) = BNK_inst("koto1",0)           ;Koto
timbre(0,106) = BNK_inst("fstrp2",0)          ;Sho
timbre(0,107) = BNK_inst("flute",0)           ;Shakuhachi
timbre(0,108) = BNK_inst("flute",0)           ;Whistle 1
timbre(0,109) = BNK_inst("flute",0)           ;Whistle 2
timbre(0,110) = BNK_inst("flute2",0)          ;Bottleblow
timbre(0,111) = BNK_inst("flute2",0)          ;Breathpipe
timbre(0,112) = BNK_inst("bdrum3",0)          ;Timpani
```

```
timbre(0,113) = BNK_inst("tom",0)             ;Melodic Tom
timbre(0,114) = BNK_inst("sdrum2",0)          ;Deep Snare
timbre(0,115) = BNK_inst("synsnr1",0)         ;Elec Perc 1
timbre(0,116) = BNK_inst("synsnr1",0)         ;Elec Perc 2
timbre(0,117) = BNK_inst("synsnr2",0)         ;Taiko
timbre(0,118) = BNK_inst("synsnr2",0)         ;Taiko Rim
timbre(0,119) = BNK_inst("cymbal",0)          ;Cymbal
timbre(0,120) = BNK_inst("shppizz",0)         ;Castanets
timbre(0,121) = BNK_inst("triangle",0)        ;Triangle
timbre(0,122) = BNK_inst("synbass4",0)        ;Orche Hit
timbre(0,123) = BNK_inst("phone1",0)          ;Telephone
timbre(0,124) = BNK_inst("chirp",0)           ;Bird Tweet
timbre(0,125) = BNK_inst("mars",0)            ;One Note Jam
timbre(0,126) = BNK_inst("bells",0)           ;Water Bells
timbre(0,127) = BNK_inst("meri",0)            ;Jungle Tune
;
;Rhythm key assignments (to Ad Lib melodic instruments)
;
;Played in channel 10 to emulate built-in Roland MT-32 rhythm sounds
;
;(Bank 127 reserved for use with "rhythm" emulation instruments)
;
timbre(127,35) = BNK_inst("bdrum",41)         ;Acou BD
timbre(127,36) = BNK_inst("bdrum",41)         ;Acou BD
timbre(127,37) = BNK_inst("tom",55)           ;Rim Shot
timbre(127,38) = BNK_inst("snare2",60)        ;Acou SD
timbre(127,39) = BNK_inst("snare2",60)        ;Hand Clap
timbre(127,40) = BNK_inst("snare2",60)        ;Elec SD
timbre(127,41) = BNK_inst("tom",41)           ;Acou Low Tom
timbre(127,42) = BNK_inst("clsdhi",84)        ;Clsd Hi Hat
timbre(127,43) = BNK_inst("tom",41)           ;Acou Low Tom
timbre(127,44) = BNK_inst("bcymbal",84)       ;Open Hi Hat 2
timbre(127,45) = BNK_inst("tom",48)           ;Acou Mid Tom
timbre(127,46) = BNK_inst("bcymbal",84)       ;Open Hi Hat 1
timbre(127,47) = BNK_inst("tom",48)           ;Acou Mid Tom
timbre(127,48) = BNK_inst("tom",55)           ;Acou Hi Tom
timbre(127,49) = BNK_inst("cymbal3",72)       ;Crash Cym
timbre(127,50) = BNK_inst("tom",55)           ;Acou Hi Tom
timbre(127,51) = BNK_inst("bcymbal",84)       ;Ride Cym
timbre(127,54) = BNK_inst("bcymbal",84)       ;Tambourine
timbre(127,56) = BNK_inst("tom",48)           ;Cowbell
timbre(127,60) = BNK_inst("tom",55)           ;High Bongo
timbre(127,61) = BNK_inst("tom",41)           ;Low Bongo
timbre(127,62) = BNK_inst("tom",55)           ;Mt High Conga
timbre(127,63) = BNK_inst("tom",55)           ;High Conga
timbre(127,64) = BNK_inst("tom",48)           ;Low Conga
timbre(127,65) = BNK_inst("tom",55)           ;High Timbale
timbre(127,66) = BNK_inst("tom",41)           ;Low Timbale
timbre(127,67) = BNK_inst("snare2",60)        ;High Agogo
timbre(127,68) = BNK_inst("snare2",48)        ;Low Agogo
timbre(127,69) = BNK_inst("bcymbal",84)       ;Cabasa
timbre(127,70) = BNK_inst("bcymbal",84)       ;Maracas
timbre(127,71) = BNK_inst("bcymbal",84)       ;Smba Whis S
timbre(127,72) = BNK_inst("bcymbal",84)       ;Smba Whis L
timbre(127,73) = BNK_inst("bcymbal",84)       ;Quijada
timbre(127,75) = BNK_inst("bcymbal",84)       ;Claves
```

8.8 CLAD - Creative Labs and Ad Lib (TM) Converter

Discussion

The **CLAD** program translates instrument data between the Ad Lib and Creative Labs (Sound Blaster) native file formats. .INS and .SBI files are used by the Ad Lib and Creative Labs voice editing tools, respectively.

CLAD has the following command-line syntax:

```
CLAD < infile.ins | infile.sbi > < outfile.ins | outfile.sbi >
```

where **infile** represents the instrument file (in either Ad Lib (.INS) or Sound Blaster (.SBI) format) to translate, and **outfile** represents the file to be created in the new format.

The use of the .INS and .SBI file suffixes is mandatory.

INS2SBI, a similar program which accompanies the Sound Blaster Developer Kit, can also perform a one-way instrument file translation from the Ad Lib format to the Sound Blaster format. However, **INS2SBI** contains several bugs; **CLAD** should be used for this purpose instead.

8.9 MIDILOG - MIDI File Event Filter

Discussion

The **MIDILOG** program strips any non-standard headers and trailers from a Standard MIDI file, and optionally translates certain MIDI events in the file.

The ability to translate MIDI events makes **MIDILOG** a powerful tool when dealing with sequencer programs that do not allow certain events to be embedded in a MIDI file, but which are capable of recording other types of events which may be replaced later with the events originally desired by the musician or programmer.

MIDILOG has the following command-line syntax:

```
MIDILOG infile outfile [logfile]
```

where **infile** refers to the original MIDI file to be translated, **outfile** specifies the MIDI output file, and **logfile** specifies an optional ASCII text file containing a list of event translation rules to apply to the **infile's** MIDI events while copying them to the **outfile**.

An event translation rule consists of two *expressions* separated by an '=' sign. Both expressions represent MIDI events. During the MIDI file translation process, **MIDILOG** compares each event in the **infile** with each event translation rule's *lvalue*, or the expression on the left side of its '=' sign. If no match is found, the **infile's** event is written, unchanged, to the **outfile**. However, if the event matches one of the logfile's lvalues, the event on the right side of the same event translation rule's '=' sign, called the *rvalue*, is written to the **outfile** instead.

Up to 128 event translation rules may appear in each logfile. Each expression used in an event translation rule must take one of the following forms. (Terms in [] brackets may appear zero or more times).

event([val,]): Any MIDI event.

meta_event(type, [val,]): A MIDI meta-event of type /type/, optionally containing zero or more bytes of associated data. For example, **meta_event(1, "Test")** is identical to **event($FF,$01,$04, "Test")**.

MT32_sysex_event(addr_MSB, addr, addr_LSB, data, [data,]): A type DT1 System Exclusive event capable of addressing Roland MT-32-compatible

synthesizers. No nibble masks or wildcards (see below) may be used. For example, **MT32_sysex_event($20,$00,$00, "This is a test")**, when written to a MIDI file as an rvalue, will cause the string 'This is a test' to appear on the MT-32's front-panel LCD display when the file is performed.

The values used within each expression must conform to the following rules:

$nn: A single hexadecimal byte, which may be 1 or 2 digits in length. The letter 'x' may be used to mask one of the nibbles in the byte, causing that nibble to be disregarded in event comparisons. The value of the last nibble masked in a rule's lvalue is restored to any masked nibbles in the rvalue, which is useful for preserving channel nibbles in replaced MIDI status bytes. Character case is not meaningful. Examples: $4, $69, $Bx.

nnn: A single decimal byte, which may range from 1 to 3 digits in length. Examples: 4, 105, 176.

'...' / "...": A string literal value. Either apostrophes or quotation marks may be used. Examples: 'Test', "Test".

? or $xx: Wildcard decimal byte value. The entire byte is masked (disregarded in event comparisons). The value of the last byte masked in a rule's lvalue is restored to any masked bytes in the rvalue, which is useful for preserving data bytes in replaced MIDI events.

Commas must be used to separate values within expressions. Up to 16 bytes may appear in each rule's lvalue expression, while as many as 128 bytes may appear in each rvalue expression.

All characters appearing on a line after a semicolon (';') are considered to be comments, and are ignored. All characters appearing before an equal sign ('=') are considered to be part of the rule's lvalue expression; all characters appearing after an equal sign are considered to be part of its rvalue expression. For example, the line:

```
event($Bx,120,0) = meta_event($01, "Test")   ;Control Change #120
```

contains a valid event translation rule which will trap **infile** events that set Controller 120 in any MIDI channel to a value of 0, replacing them in the **outfile** with Text meta-events containing the string "Test".

An empty rvalue in a rule (such as **event( )**) may be used to completely remove events which match the rule's lvalue. When an event is removed in this manner, however, its delta-time prefix is removed as well. This may cause loss of synchronization between tracks. To avoid this problem, it is best to replace the undesired event with an unrecognized Control Change or similar event, rather than simply preventing it from appearing in the **outfile**.

It is impossible to use **MIDILOG**'s advanced features effectively without a good understanding of MIDI events and the Standard MIDI File specification. However, an in-depth examination of these subjects is beyond the scope

of this documentation. To learn more about all aspects of MIDI, refer to a good general-purpose reference such as De Furia and Scacciaferro's *MIDI Programmer's Handbook*.

Below is a sample logfile for use with **MIDILOG**.

```
;
;SYSTEST.LOG
;Sample event translation logfile - turns SEQ.MID into APP.MID
;
;Usage: midilog seq.mid app.mid systest.log
;
     event($Bx,118,0) = meta_event($01, "Second measure")
     event($Bx,119,1) = meta_event($02, "Copyright (C) 1991")
     event($Bx,120,?) = meta_event($7E,?)                ;Generic meta
     event($Bx,117,?) = event($F0,$43,$12,$00,$07,$F7)   ;Generic sysex
     event($Bx,116,?) = MT32_sysex_event($10,$00,$01,1,7,7) ;MT-32 reverb
```

8.10 MIDIREC- Standard MIDI File Recorder

Discussion

The **MIDIREC** program receives incoming MIDI Channel Voice and System Exclusive messages via a Roland MPU-401-compatible MIDI interface, and records them to a Standard MIDI Format 0 file.

MIDIREC has the following command-line syntax:

```
MIDIREC filename [/MA:xx] [/MI:xx]
```

filename indicates the name of the Standard MIDI Format 0 file to create. Optional parameters include **/MA:xx and /MI:xx**, which specify the hexadecimal I/O port address and IRQ number, respectively, of the Roland-compatible MIDI interface used to receive the incoming data.

MIDIREC's advantage over many MIDI sequencers such as Cakewalk is its ability to record System Exclusive message transmissions, in addition to conventional MIDI Channel Voice messages. When used in conjunction with the Miles Sound Studio's intelligent Roland MT-32 System Exclusive disassembly feature, **MIDIREC** is a valuable diagnostic tool for the examination of a MIDI application's output stream, whether or not the stream originated from a Miles Sound System driver.

The **MIDIREC** output file conforms to the standards for a Format 0 MIDI file, and is recorded as though it is being played in 4/4 time with a division of 120 delta-time ticks per quarter note and a tempo of 500,000 microseconds per quarter note. If necessary, the file may be imported into a sequencer to adjust these parameters, but care should be taken not to inadvertently "lose" System Exclusive information in the process.

8.11 WAVELIB - Wave Synthesizer Librarian

Discussion

This utility should now be considered obsolete - all future digital synthesis should be performed with the new DLS functions! WAVELIB only exists for backwards compatibility now.

The **WAVELIB** program extracts raw PCM sample data from standard Creative Labs Sound Blaster Voice Editor-compatible .VOC files, as well as Microsoft Windows-compatible .WAV files, and stores it in a wave library file for use with XMIDI-sequenced digital sound effects.

WAVELIB has the following command-line syntax:

```
WAVELIB list_file.TXT
```

The **list file** is an unformatted ASCII text file which contains a list of .VOC, .WAV, and/or raw audio data files which are assigned to MIDI Program Change numbers and XMIDI Patch Bank Select values. Upon installing a wave library file with the AIL_create_wave_synthesizer() function, the instruments stored in the wave library will replace or add to the default set of MIDI instruments associated with the XMIDI driver in use. MIDI Program Change (and optionally XMIDI Patch Bank Select) messages in a sequence may be used to select a digital sound effect for triggering with subsequent MIDI Note On/Note Off messages. Pitch Bend, Panpot, and Part Volume messages are also emulated by the digital sound-effects system. Rhythm instruments assigned to XMIDI Patch Bank 127 may be mapped to various MIDI note numbers in channel 10, while melodic instruments assigned to other XMIDI Patch Banks (or none at all) are played normally in the other channels.

.VOC and .WAV files specified in the list file must fit into available EMS/ XMS memory, and should not contain "silence packing" blocks, infinite repeat loops, or voice blocks with varying sample rates or compression types. IMA ADPCM compression is not supported.

During the conversion process, "repeat blocks" in .VOC files are expanded into multiple instances of the repeated voice data blocks. The "extended block" type introduced with the Sound Blaster Pro developer kit is also supported by **WAVELIB**, as is the new 16-bit voice block type introduced with the Sound Blaster 16.

The following is a copy of the **WAVEDEMO.TXT** example list file which is provided with the MSS. Options supported by **WAVELIB** in this release include the features mentioned above. Later releases of **WAVELIB** may support additional options; check the **WAVEDEMO.TXT** file provided with subsequent releases for information on any new features or options.

```
;*****************************************************************************
;**                                                                         **
;** Example file list for software wave triggers                            **
;**                                                                         **
```

```
;** This file describes a set of digital audio samples to be played by the  **
;** digital audio system in response to MIDI Note On/Off events.            **
;**                                                                         **
;**                                                                         **
;**                                                                         **
;** At the beginning of the list file, use:                                 **
;**                                                                         **
;**    F=??? to declare the name of the wave library file to create         **
;**                                                                         **
;**          ??? represents a DOS filename in quotes; e.g. "demo.wvl"       **
;**                                                                         **
;**                                                                         **
;**                                                                         **
;** For each digital audio sample to be played as a MIDI instrument, use:   **
;**                                                                         **
;**    B=nnn to set XMIDI Patch Bank Select (controller 114) value          **
;**                                                                         **
;**          Set this value to 0 if your sequences do not contain Patch     **
;**          Bank Select controllers                                        **
;**                                                                         **
;**    P=nnn to set MIDI patch # for sound (B=0-126) or MIDI trigger key #  **
;**          for rhythm sound (B=127 only)                                  **
;**                                                                         **
;**    R=nnn to set root MIDI note # for sound (B=0-126)                     **
;**                                                                         **
;**          For example, middle C (C5) is note number 60; if R= parameter  **
;**          is not used, a value of 60 is assumed                          **
;**                                                                         **
;**          The R= parameter is not needed for bank 127 (rhythm) sounds    **
;**                                                                         **
;**                                                                         **
;**                                                                         **
;** The following parameters are used with raw data files ONLY:             **
;**                                                                         **
;**    S=nnn to set sample playback rate (11025, 22050, etc.)               **
;**                                                                         **
;**    C=nnn to set # of digital channels (1=mono, 2=stereo)                **
;**                                                                         **
;**    W=nnn to set sample word size (8 or 16 bits)                         **
;**                                                                         **
;**                                                                         **
;**                                                                         **
;** All .VOC/.WAV/raw source filenames must be enclosed in quotes ("")      **
;**                                                                         **
;** Up to 512 samples may appear in one list file.  Lines which begin with  **
;** semicolons are treated as comments.  Blank lines are permitted.         **
;**                                                                         **
;*****************************************************************************
F="demo.wvl"
B=127  , P=60  , S=44100 , C=1 , W=8 , "..\drivers\digital\welcome.raw"
B=127  , P=61  , S=44100 , C=1 , W=8 , "..\drivers\digital\flyby.raw"
B=127  , P=62  , S=44100 , C=1 , W=8 , "..\drivers\digital\she_bop.raw"
B=127  , P=63  , S=44100 , C=1 , W=8 , "..\drivers\digital\charge.raw"
B=127  , P=64  , S=44100 , C=1 , W=8 , "..\drivers\digital\klaxon.raw"
B=127  , P=65  , S=44100 , C=1 , W=8 , "..\drivers\digital\nirvana.raw"
B=127  , P=66  , S=44100 , C=1 , W=8 , "..\drivers\digital\thriller.raw"
B=127  , P=67  , S=44100 , C=1 , W=8 , "..\drivers\digital\nena.raw"
B=127  , P=68  ,                       "..\drivers\digital\glass.wav"
```

Index