MIDAS Digital Audio System

Programmer's Guide

Petteri Kangaslampi

March 1, 1998

Contents

Chapter 1

Introduction

1.1 Welcome

Welcome to the exciting world of digital audio! MIDAS Digital Audio System is the most comprehensive cross-platform digital audio system today. With features such as an unlimited number of digital channels on all supported platforms, simultaneous sample, module and stream playback, and seamless portability across operating systems, MIDAS is all you need for perfect sound in your applications.

This manual is the Programmer's Guide to the MIDAS Digital Audio System. It includes descriptions about all aspects of MIDAS, including initialization, configuration and usage of different system components. It does not attempt to document all functions and data structures available in MIDAS, but rather give a good overview on how you can use MIDAS in your own programs. For complete descriptions of all function and data structures, see the MIDAS API reference.

1.2 What is MIDAS?

What is MIDAS Digital Audio System anyway?

MIDAS Digital Audio System is a multichannel digital music and sound engine. It provides you with an unlimited number of channels of digital audio that you can use to play music, sound effects, speech or sound streams. MIDAS is portable across a wide range of operating systems, and provides an identical API in all supported environments, making it ideal for cross-platform software development.

MIDAS Digital Audio System is free for noncommercial usage, read the file `license.txt` included in the MIDAS distribution for a detailed license. Commercial licenses are also available.

Chapter 2

Getting started

Although MIDAS Digital Audio System is a very powerful sound engine, it is also extremely easy to use. This chapter contains all the information necessary to develop simple sound applications using MIDAS. It describes how to link MIDAS into your own programs, how to use the MIDAS API functions from your own code, and concludes with a simple module player program example.

2.1 Installing MIDAS

Installing MIDAS Digital Audio System is very simple: just create a separate directory for it, and decompress the distribution archive. MIDAS is normally distributed as one or several `.zip` files, and they all need to be decompressed in the same directory. If developing Win32 or Linux applications, use an unzip utility that handles long filenames, such as InfoZip unzip or WinZip, instead of MS-DOS pkunzip. Linux developers should decompress the files in Linux, as the archive may contain symbolic links for the Linux libraries.

Note! Make sure your unzip utility decompresses subdirectories correctly. InfoZip unzip and WinZip should do this by default, but pkunzip needs the "`-d`" option to do this.

2.2 Compiling with MIDAS

For applications using just the MIDAS Digital Audio System API, no special compilation options are necessary. All MIDAS API definitions are in the file `midasdll.h`, and the modules using MIDAS functions simply need to `#include` this file. No special macros need to be `#define`d, and the data structures are structure-packing neutral. `midasdll.h` is located in the `include/` subdirectory of the MIDAS distribution, and you may need to add that directory to your include file search path.

Under Windows NT/95, the MIDAS API functions use the `stdcall` calling convention, the same as used by the Win32 API. Under DOS, the functions use the `cdecl` calling convention, and under Linux the default calling convention used by GCC. This is done transparently to the user, however.

Delphi users can simply use the interface unit `midasdll.pas`, and access the MIDAS API functions through it. Although Delphi syntax is different from C, the function, structure and constant names are exactly the same, and all parameters are passed just like in the C versions. Therefore all information in this document and the API Reference is also valid for Delphi.

MS-DOS users with Watcom C will need to disable the "SS==DS" assumption from the modules that contain MIDAS callback functions. This can be done with the "`-zu`" command line option. Note that this is **not** necessary for code that just calls MIDAS functions.

2.3 Linking with MIDAS

If your program uses MIDAS Digital Audio System, you naturally need to link with the MIDAS libraries as well. This section describes how to do that on each platform supported.

All MIDAS Digital Audio System libraries are stored under the `lib/` subdirectory in the distribution. `lib/` contains a subdirectory for each supported platform, and those in turn contain directories for each supported compiler. The format of the compiler directory names is "¡compiler¿¡build¿", where ¡compiler¿ is a two-letter abbreviation for the compiler name, and ¡build¿ the library build type — retail or debug. Under most circumstances, you should use the retail versions of the libraries, as they contain better optimized code. Also, the debug libraries are not included in all releases.

For example, `lib/win32/vcretail/midas.lib` is the retail build of the Win32 Visual C/C++ static library.

2.3.1 Windows NT/95

Under the Win32 platform, applications can link with MIDAS Digital Audio System either statically or dynamically. Unless there is a specific need to link with MIDAS statically, dynamic linking is recommended. Delphi users need to use dynamic linking always.

When linking with MIDAS statically, simply link with the library file corresponding to your development platform. For Watcom C/C++, the library is `lib/win32/wcretail/midas.lib`, and for Visual C/C++ `lib/win32/vcretail/midas.lib`. Depending on your configuration, you may need to add the library directory to your "library paths" list. When MIDAS is linked into the application statically, the .exe is self-contained and no MIDAS `.dll` files are needed.

Dynamic linking is done by linking with the appropriate MIDAS import library instead of the static linking library. In addition, the MIDAS Dynamic Link Library (`midasXX.dll`) needs to be placed in a suitable directory — either to the same directory with the program executable, or in some directory in the user's PATH. The import libraries are stored in the same directory with the static libraries, but the file name is `midasdll.lib`. For example, Visual C users should link with `lib/win32/vcretail/midasdll.lib`. The MIDAS Dynamic Link Libraries are stored in `lib/win32/retail` and `lib/win32/debug`.

Delphi users do not need a separate import library — using the interface unit `midasdll.pas` adds the necessary references to the `.dll` automatically. Note that running the program under the Delphi IDE without the `.dll` available can cause strange error messages.

2.3.2 MS-DOS

As MS-DOS doesn't support dynamic linking, only a static link library is provided for MS-DOS. You'll simply need to link with the library from the appropriate subdirectory — usually `lib/dos/gcretail` for GCC (DJGPP) and `lib/dos/wcretail` for Watcom C. The executable is fully self-contained, and no additional files are needed. DJGPP users also need to link with the Allegro library, as MIDAS uses some of its functions for IRQ handling.

Note that some versions of the Watcom Linker are not case-sensitive by default, and you'll need to use case-sensitive linking with MIDAS. To do that, simply add `option caseexact` to your linker directives.

2.3.3 Linux

For Linux, only a static library is provided for the time being. To link your program with MIDAS Digital Audio System, add the proper library directory (usually `lib/linux/gcretail`) to your library directory list (gcc option `-L`), and link the library in using the GCC option `-lmidas`.

2.4 Using MIDAS from the IDE

This section contains step-by-step instructions on building applications that use MIDAS Digital Audio System with the Integrated Developent Environments of popular compilers.

2.4.1 Using MIDAS with Visual C Developer Studio

This section contains contains simple step-by-step instructions for using MIDAS Digital Audio System with Microsoft Developer Studio.

1. Begin the project as usual. If you already have an existing project, it should need no modifications.

2. Add some simple code for testing MIDAS — either copy the module player example below, or just add a call to *MIDASstartup* to the beginning of the program and `#include midasdll.h` at the beginning of the module.

3. Add the MIDAS include directories to the include search path: Open "Build/Project Settings" -dialog, choose the "C/C++" tab, select "Preprocessor" from the Category list, and add the MIDAS include directory to "Additional include directories". For example, if you installed MIDAS in `d:/midas`, add `d:/midas/include`.

4. Add a MIDAS library to the project. In most cases, you should use the retail import library, and thus link dynamically. Open "Insert/Files into Project" -dialog, and select the library file you want to use, typically `d:/midas/lib/win32/vcretail/midasdll.lib`.

Now you should be able to build the project normally. To be able to run the program, you must make sure that the MIDAS DLL is available either in the same directory with the produced executable, or in some directory in the system search path. You can simply copy the DLL from (for example) `d:/midas/lib/win32/retail` to the project directory.

2.4.2 Using MIDAS with Watcom C IDE

This section contains contains simple step-by-step instructions for using MIDAS Digital Audio System with Watcom C IDE.

1. Begin the project as usual. If you already have an existing project, it should need no modifications.

2. Add some simple code for testing MIDAS — either copy the module player example below, or just add a call to *MIDASstartup* to the beginning of the program and `#include midasdll.h` at the beginning of the module.

3. Add the MIDAS include directories to the include search path: Open "Options/C Compiler Switches" -dialog, choose "1. File Option Switches" from the switches list, and add the MIDAS include directory to "Include directories". For example, if you installed MIDAS in `d:/midas`, add `d:/midas/include`.

4. Add a MIDAS library to the project. In most cases, you should use the retail import library, and thus link dynamically. Open "Sources/New Source" -dialog, and select the library file you want to use, typically `d:/midas/lib/win32/wcretail/midasdll.lib`.

Now you should be able to build the project normally. To be able to run the program, you must make sure that the MIDAS DLL is available either in the same directory with the produced executable, or in some directory in the system search path. You can simply copy the DLL from (for example) `d:/midas/lib/win32/retail` to the project directory.

2.5 A simple module player example

This section describes a very simple example program that uses MIDAS Digital Audio System for playing music. First, the complete program source is given in both C and Delphi format, and after that the operation of the program is described line by line. To keep the program as short as possible, all error checking is omitted, and therefore it should not be used as a template for building real applications — the other example programs included in the MIDAS distribution are more suitable for that.

Both versions of the program should be compiled as console applications in the Win32 environment. Under MS-DOS and Linux the default compiler settings are fine.

2.5.1 C module player example

```
 1  #include <stdio.h>
 2  #include <conio.h>
 3  #include "midasdll.h"
 4
 5  int main(void)
 6  {
 7      MIDASmodule module;
 8      MIDASmodulePlayHandle playHandle;
 9
10      MIDASstartup();
11      MIDASinit();
12      MIDASstartBackgroundPlay(0);
13
14      module = MIDASloadModule("..\\data\\templsun.xm");
15      playHandle = MIDASplayModule(module, TRUE);
16
17      puts("Playing - press any key");
18      getch();
19
20      MIDASstopModule(playHandle);
21      MIDASfreeModule(module);
22
```

```
23      MIDASstopBackgroundPlay();
24      MIDASclose();
25
26      return 0;
27  }
```

2.5.2 Delphi module player example

```
 1  uses midasdll;
 2
 3  var module : MIDASmodule;
 4  var playHandle : MIDASmodulePlayHandle;
 5
 6  BEGIN
 7      MIDASstartup;
 8      MIDASinit;
 9      MIDASstartBackgroundPlay(0)
10
11      module := MIDASloadModule('..\data\templsun.xm');
12      playHandle MIDASplayModule(module, true);
13
14      WriteLn('Playing - Press Enter');
15      ReadLn;
16
17      MIDASstopModule(playHandle);
18      MIDASfreeModule(module);
19
20      MIDASstopBackgroundPlay;
21      MIDASclose;
22  END.
```

2.5.3 Module player example description

Apart from minor syntax differences, the C and Delphi versions of the program work nearly identically. This section describes the operation of the programs line by line. The line numbers below are given in pairs: first for C, second for Delphi.

1-3, 1 Includes necessary system and MIDAS definition files

7, 3 Defines a variable for the module that will be played

8, 4 Defines a variable for the module playing handle

10, 7 Resets the MIDAS internal state — This needs to be done before MIDAS is configured and initialized.

11, 8 Initializes MIDAS

12, 9 Starts playing sound in the background

14, 11 Loads the module file

15, 12 Starts playing the module we just loaded, looping it

17-18, 14-15 Simply waits for a keypress

20, 17 Stops playing the module

21, 18 Deallocates the module we loaded

23, 20 Stops playing sound in the background

24, 21 Uninitializes the MIDAS Digital Audio System

Chapter 3

Using MIDAS Digital Audio System

This chapter contains detailed step-by-step instructions on using MIDAS Digital Audio System. Complete descriptions of the functions, data structures and constants used here is available in the API reference.

3.1 Initializing and configuring MIDAS

MIDAS Digital Audio System initialization consists of five basic steps, which are outlined below. The last two steps are not necessary in all cases.

1. Call *MIDASstartup*. This should be done as early in the program as possible, preferably at the very beginning. *MIDASstartup* does not take a significant amount of time, and does not allocate any memory, it simply initializes MIDAS to a safe and stable state and resets all configration options. Calling *MIDASclose* is always safe after *MIDASstartup* has been called.

2. Configure MIDAS. This can be done by setting different MIDAS options with *MIDASsetOption*, or by calling *MIDASconfig* to prompt the user for the settings. The configuration can also be loaded from a file (with *MIDASloadConfig*) or registry (with *MIDASreadConfigRegistry*) at this point. Apart from the configuration functions, no MIDAS functions may be called yet.

3. Initialize MIDAS by calling *MIDASinit*. All MIDAS functions are usable after this, and the program can fully start using MIDAS. Most MIDAS configuration options can **not** be changed while MIDAS is initialized, so to change the options it is necessary to uninitialize MIDAS first.

4. Start background sound playback. Unless you want to poll MIDAS manually (with *MIDASpoll*), you should now call *MIDASstartBackgroundPlay* to start background sound playback. *MIDASstartBackgroundPlay* starts a playback thread (in multithreaded systems) or a timer, and polls MIDAS automatically from it.

5. Open sound channels with *MIDASopenChannels*. The number of open sound channel limits the number of sounds that can be played simultaneously — one sound channel can play one sound. The number of sound channels needed depends on the application, typical values might be 8–32 channels for a module, one channel per stream and 1–8 channels for sound effects. Having more channels open than necessary will not increase the CPU use, as inactive channels do not need CPU attention. However, some sound cards may place limitations on the maximum number of open channels. If you are only playing a single module, opening the channels manually is not necessary, as *MIDASplayModule* can open the needed channels automatically.

3.2 Uninitializing MIDAS

MIDAS Digital Audio System uninitialization is essentially the reverse process of initialization. Actually only the last step is (calling *MIDASclose*) is absolutely necessary, but it is good practise to uninitialize everything that has been initialized. In addition, all modules and samples that have been loaded should be deallocated before uninitializing MIDAS, as failing to do so may lead to memory leaks.

Basic MIDAS uninitialization consists of three steps:

1. If sound channels have been opened with *MIDASopenChannels*, close them by calling *MIDAScloseChannels*.

2. If background sound playback is used, stop it by calling *MIDASstopBackgroundPlay*.

3. Finally, uninitialize the rest of MIDAS Digital Audio System by calling *MIDASclose*.

3.3 Using modules

Digital music modules provide an easy to use and space-efficient method for storing music and more complicated sound effects. MIDAS Digital Audio System supports Protracker (.mod), Scream Tracker 3 (.s3m), FastTracker 2 (.xm) and Impulse Tracker (.it) modules. All types of modules are used through the same simple API functions described in the subsections below.

3.3.1 Loading and deallocating modules

Loading modules with MIDAS Digital Audio System is very simple, just call *MIDASloadModule*, giving as an argument the name of the module file. *MIDASloadModule* returns a module handle of type *MIDASmodule*, which can then be used to refer to the module. After the module is no longer used, it should be deallocated with *MIDASfreeModule*.

3.3.2 Playing modules

Modules that have been loaded into memory can be played by calling *MIDASplayModule*. *MIDASplayModule* takes as an argument the module handle for the module, and a boolean flag that indicates whether or not the module playback should loop or not. It returns a module playback handle of type *MIDASmodulePlayHandle* that can be used to refer to the module as it is being played.

MIDASplayModuleSection can be used to play just a portion of the module. A single module could, for example, contain several different songs, and *MIDASplayModuleSection* can be used to select which one of them to play.

MIDAS is also capable of playing several modules simultaneously, or even the same module several times from different positions. There are some limitations, however, see *MIDASplayModule* documentation for details. This can be useful for using module sections as sound effects, or fading between two modules.

Module playback can be stopped by calling *MIDASstopModule*, passing it as an argument the module playback handle returned by *MIDASplayModule*.

3.3.3 Controlling module playback

Although typically module playback just proceeds without user intervention, it is also possible to control the playback of the module. *MIDASsetPosition* can be used to change the current playing position, *MIDASsetMusicVolume* to set the master volume of the music, and *MIDASfadeMusicChannel* to fade individual music channels in or out. All of these functions require as their first argument the module playing handle from *MIDASplayModule*.

3.3.4 Getting module information

In MIDAS Digital Audio System, it is possible to query information about a module or the status of module playback. This information can be used to update an user interface, or synchronize the program operation to the music playback.

Basic information about the module, such as its name, is available by calling *MIDASgetModuleInfo*. The function requires as an argument a module handle returned by *MIDASloadModule*, and a pointer to a *MIDASmoduleInfo* structure, which it then will fill with the module information. A similar function, *MIDASgetInstrumentInfo*, is available for reading information about individual instruments in the module.

The current status of the playback of a module can be read with *MIDASgetPlayStatus*. It is passed a module playback handle from *MIDASplayModule*, and a pointer to a *MIDASplayStatus* structure, which it will then fill with the playback status information. The playback status

information includes the current module playback position, pattern number in that position and the current playback row, as well as the most recent music synchronization command infobyte.

MIDAS Digital Audio System also supports a music synchronization callback function, which will be called each time the player encounters a music synchronization command. The synchronization command is **Wxx** for Scream Tracker 3 and FastTracker 2 modules, and **Zxx** for Impulse Tracker modules. The callback can be set or removed with the function *MIDASsetMusicSyncCallback*. As the music synchronization callback is called in MIDAS player context, it should be kept as short and simple as possible, and it may not call MIDAS functions.

3.4 Using samples

Using samples is the easiest way to add sound effects and other miscellaneous sounds to a program with MIDAS Digital Audio System. This section describes how samples are used in MIDAS.

3.4.1 Loading samples

Before samples can be used, they naturally need to be loaded into memory. MIDAS Digital Audio System currently supports samples in two formats: raw audio data files and RIFF WAVE files. Raw sample files can be loaded with *MIDASloadRawSample*, and RIFF WAVE samples with *MIDASloadWaveSample*. Both functions require as arguments the name of the sample file, and sample looping flag — 1 if the sample should be looped, 0 if not. *MIDASloadRawSample* also requires the sample type as an argument, *MIDASloadWaveSample* determines it from the file header. Both functions return a sample handle of type *MIDASsample* that can be used to refer to the sample.

After the samples are no longer used, they should be deallocated with *MIDASfreeSample*. You need to make sure, however, that the sample is no longer being played on any sound channel when it is deallocated.

3.4.2 Playing samples

Samples that have been loaded into memory can be played with the function *MIDASplaySample*. It takes as arguments the sample handle, playback channel number, and initial values for sample priority, playback rate, volume and panning. The function returns a MIDAS sample playback handle of type *MIDASsamplePlayHandle* that can be used to refer to the sample as it is being played. A single sample can be played any number of times simultaneously.

Sample playback can be stopped with *MIDASstopSample*, but this is not necessary before the sample will be deallocated — a new sample can simply be played on the same channel, and it

will then replace the previous one. The sample handles will be recycled as necessary, so there is no danger of memory leaks.

3.4.3 Samples and channels

One sound channel can be used to play a single sample, and therefore *MIDASplaySample* requires the number of the channel that is used to play the sample as an argument. The channel number can be set manually, or MIDAS Digital Audio System can handle the channel selection automatically.

If the channel number for the sample is set manually, the channel used should be allocated with *MIDASallocateChannel* to ensure that the channel is not being used for other purposes. If a free channel is available, the function will return its number that can then be used with *MIDASplaySample*. After the channel is no longer used, it should be deallocated with *MIDASfreeChannel*.

Another possibility is to let MIDAS select the channel automatically. A number of channels can be allocated for use as automatic effect channels with *MIDASallocAutoEffectChannels*. *MIDASplaySample* can then be passed **MIDAS_CHANNEL_AUTO** as the channel number, and it will automatically select the channel that will be used to play the effect. After the automatic effect channels are no longer used, they should be deallocated with *MIDASfreeAutoEffectChannels*.

3.4.4 Controlling sample playback

Most sample playback properties can be changed while it is being played. *MIDASsetSampleRate* can be used to change its playback rate, *MIDASsetSampleVolume* its volume, *MIDASsetSamplePanning* its panning position and *MIDASsetSamplePriority* its playback priority. All of these functions take as an argument the sample playback handle from *MIDASplaySample*, and the new value for the playback property.

The sample playback properties can be changed at any point after the sample playback has been started until the sample playback is stopped with *MIDASstopSample*. If the sample has reached its end, or has been replaced by another sample with automatic channel selection, the function call is simply ignored.

3.4.5 Getting sample information

The sample playback status can be monitored with *MIDASgetSamplePlayStatus*. It takes as an argument the sample playback handle, and returns the current sample playback status. The playback status information can be used to determine whether or not a sample has already reached its end.

3.5 Using streams

In MIDAS Digital Audio System, streams are continuous flows of sample data. Unlike samples, they do not need to be loaded completely into memory before they can be played, but can rather be loaded from disk or generated as necessary. This section describes how streams are used in MIDAS.

3.6 Playing streams

There are two different ways of playing streams in MIDAS Digital Audio System: stream file playback and polling stream playback. In stream file playback, MIDAS creates a new thread that will read the stream data from a given file and plays it automatically on the background. In polling stream playback the user needs to read or generate the stream data, and feed it to MIDAS.

Stream files are played with *MIDASplayStreamFile* and *MIDASplayStreamWaveFile*. Both functions require as arguments the stream file name, stream playback buffer length and stream looping flag. The files played with *MIDASplayStreamFile* should contain only raw sample data, and the function will therefore require as arguments also the stream data sample type and playback rate. *MIDASplayStreamWaveFile* plays RIFF WAVE files, and can read the information from the file header. The playback functions will return a stream handle that can be used to refer to the stream.

Polling stream playback is started with *MIDASplayStreamPolling*. It requires as its arguments the stream sample type, playback rate and buffer length. The actual stream playback will not start, however, until some stream data is fed to the system with *MIDASfeedStreamData*. After the playback has starter, *MIDASfeedStreamData* needs to be called periodically to feed the system new stream data to play, otherwise the system will run out of stream data and stop playback.

The stream playback buffer length controls the amount of stream data buffered for stream playback. The longer the buffer is, the longer the system can play the stream when new data is not available until the playback needs to be stopped. Running out of stream data will result in irritating breaks in the sound and should be avoided. Longer playback buffers will, however, add delay to the stream playback, and consume more memory. For stream file playback, a stream buffer length of 500ms should be suitable. For polling stream playback, the buffer length should be at least twice the longest expected delay between two calls to *MIDASfeedStreamData*.

Stream playback is stopped with *MIDASstopStream*, regardless of the stream type.

3.6.1 Controlling stream playback

The stream playback properties can be changed while it is being played. *MIDASsetStreamRate* can be used to change its playback rate, *MIDASsetStreamVolume* its volume and *MIDASsetStreamPanning* its panning position. All of these functions take as an argument the stream handle from the stream playback function, and the new value for the playback property.

The playback of the stream can also be paused with *MIDASpauseStream*, and resumed after pausing with *MIDASresumeStream*. This can be useful if your application knows it will run out of stream data soon, and wishes to fade the stream out and pause it until more data is available. Stream data reading and feeding can continue while the stream is paused until the stream buffer is full.

3.6.2 Getting stream information

The amount of data currently in the stream buffer can be monitored with *MIDASgetStreamBytesBuffered*. The information could be used to determine how soon new stream data is needed to continue playback, or whether or not enough space exists in the stream buffer for a complete new block of data.

Note that with ADPCM streams the stream buffer contains the decompressed data, as 16-bit samples, instead of the compressed ADPCM data.

3.7 Using echo effects

The MIDAS Digital Audio System Echo Effects Engine can be used to add different echo and reverb effects to the sound output. These effects can range from simple echoes and filtering effects to heavy hall reverbs and stereo enhancements. This section describes how the echo effects are used

3.7.1 Adding echo effects

Echo effects are added to the sound output with *MIDASaddEchoEffect*. It takes as an argument a pointer to a filled *MIDASechoSet* structure (described below), and returns and echo handle that can then be used to refer to the effect. The echo set structure is not used after *MIDASaddEchoEffect* returns, and may be deallocated. Any number of echo effects can be active simultaneously.

After the echo effect is no longer wanted, it can be removed from the output with *MIDASremoveEchoEffect*. Note that modifying the *MIDASechoSet* structure of an echo effect that is already being used has no effect.

3.7.2 The echo effect structure

A MIDAS Digital Audio System echo effect is described by a *MIDASechoSet* structure. The echo set contains three common fields plus one or more echoes. The **feedback** field controls the amount of feedback in the echo set, **gain** the echo effect total gain, and **numEchoes** simply the number of echoes in the echo set. See the *MIDASechoSet* description in the API Reference for more details.

The echoes of an echo set are described by an array of *MIDASecho* structures. Each echo has fields that describe its delay, gain, filtering and channel reverse status. The delay of an echo controls how far back from the echo effect delay line the data for the echo is taken — the greater the delay, the longer the echo is. Gain controls the strength of the echo, the echo data is essentially multiplied by the gain. Each echo can optionally be low-pass or high-pass filtered, in a system with more than a couple of echoes, low-pass filtering can reduce the build-up of high-frequency noise. Finally, the left and right channels of the echo can be reversed, producing an interesting stereo effect in some cases.

Chapter 4

Advanced topics

This chapter includes advanced programming information about MIDAS Digital Audio System. Although the information contained here is not necessary for being able to use MIDAS, it should be read by everybody who wishes to get everything out of it. Topics covered here include performance optimization, sound quality optimization, and working with compressed sample types.

4.1 Optimizing MIDAS performance

Although MIDAS Digital Audio System has been carefully optimized for maximum performance, playing multichannel digital audio can still be fairly time-consuming. To get around the limitations of current PC sound cards, MIDAS needs to mix the sound channels in software, and this mixing process accounts for most of CPU usage caused by MIDAS. In many cases tradeoffs can be made between sound quality and CPU usage, although it is also possible in some cases to lower CPU usage dramatically without affecting sound quality.

In some cases the opposite is also true: It can be possible to get better sound quality out of MIDAS without using much additional CPU power. This section therefore describes how you can get best possible sound quality of MIDAS while using as little CPU time as possible, and how to find the right balance between sound quality and CPU usage for your application.

4.1.1 The number of channels

First and foremost, the CPU time used for mixing depends on the number of active sound channels in use. Each channel needs to mixed to the output separately, and thus requires CPU time. Note, however, that the total number of open channels is not very significant, only the

number of channels that are actually playing sound. Also, sounds played at zero volume take very little CPU time.

On a computer with a 90MHz Pentium CPU, with the default sound quality settings, the CPU time used is roughly 0.6%–0.9% of the total CPU time per channel. The number can vary greatly depending on the type of samples used and other factors, but can be used as a guideline in deciding how many sound channels to use. Regardless of the output mode settings, however, minimizing the number of channels used is an easy way to increase MIDAS performance.

Using fewer sound channels does not necessarily mean sacrificing sound quality or richness. Music modules do not necessarily need to have over 10 channels to sound good — talented musicians have composed stunning songs with the Amiga Protracker program which only supports 4 channels. Unnecessary or very quiet sound effects could be eliminated, making way for more important ones. Finally, instead of playing two or more sounds simultaneously, the sounds could be mixed beforehand with a sample editor into one.

4.1.2 Sample types

Another factor that has a great effect on MIDAS Digital Audio System CPU usage is the type of the samples played. The simplest sample type, and therefore the fastest to play, is a 8-bit mono sample. 16-bit samples take typically 50% more CPU time to play than 8-bit samples, and stereo samples more than 50% than corresponding mono samples. In addition, ADPCM streams take additional CPU time for decompressing the ADPCM data to 16-bit before it can be played.

However, 16-bit samples do sound much better than 8-bit ones. A good compromise is to use *mu*-law samples instead. *mu*-law samples have almost the same sound quality as 16-bit samples, while being as fast to mix as 8-bit samples. In addition, they only take as much space as 8-bit samples, and thus lower the memory requirements of the program as well. In some mixing modes MIDAS will actually automatically convert 16-bit samples to *mu*-law if it is beneficial. The option *MIDAS_OPTION_16BIT_ULAW_AUTOCONVERT* can be used to control this behaviour.

ADPCM compression yields a 1:4 compression on 16-bit sound data, using effectively only 4 bits per sample, while maintaining sound quality better than 8-bit samples. As ADPCM sample data cannot easily be played without decompressing it first, however, MIDAS only supports ADPCM sample data in streams. Although ADPCM streams take somewhat more CPU time to play than 16-bit streams, as the data needs to be decompressed, they can still turn out to be faster in practise, as the amount of data that needs to be read from disk is much smaller.

4.1.3 Output settings

Finally perhaps the most important factor in determining MIDAS CPU usage and sound quality: output sound quality settings. Several different sound quality settings can be adjusted, and each can be used to adjust the balance of sound quality and CPU usage.

The most important of all output quality settings is the mixing mode. By default, MIDAS Digital Audio System uses normal mixing mode, which has very good performance. High-quality mixing mode with sound interpolation is available, and it greatly enhances the sound quality in some cases, but also requires much more CPU power. High-quality mixing can use two to three times as much CPU time as normal quality mixing, and should be reserved for applications that only use a few sound channels or do not require the CPU time for other uses. The mixing mode settings can be changed by setting the option *MIDAS_OPTION_MIXING_MODE* with *MIDASsetOption*.

Another setting that greatly affects CPU usage and sound quality is the output mixing rate. CPU usage depends almost linearly on the mixing rate, with higher mixing rates using more CPU power but also sounding better. The default mixing rate is 44100Hz, but in many cases it can be lowered to 32000Hz or 22050Hz without too great sound quality loss. In addition, if all of your sounds are played at the same rate (eg. 22050Hz), the mixing rate should be set at the same rate — using a higher rate would not bring any better sound quality and could actually increase noise in the output.

Experimenting with different mixing rate and mode combinations can also be worthwhile, as in some cases using a lower mixing rate with high-quality mixing can sound better than a higher mixing rate with normal quality.

The last setting to consider is the output mode setting. The output mode should normally be set to a 16-bit mode, as using 8-bit modes does not decrease CPU usage, only sound quality. If the sound card used doesn't support 16-bit output, MIDAS will use 8-bit output automatically. Using mono output instead of stereo, however, can decrease CPU usage by up to 50%. Therefore, if your application does not use stereo effects in its sound output, consider using mono output mode instead. The output mode can be changed by setting the option *MIDAS_OPTION_OUTPUTMODE* with *MIDASsetOption*.

4.2 Using -law compression

In MIDAS Digital Audio System, -law samples and streams can be used as an effective compromise between CPU and space usage and sound quality, as they provide sound quality almost equivalent to 16-bit samples while using only as much CPU time and space as 8-bit samples. This section describes how -law samples are encoded and used with MIDAS.

4.2.1 Encoding -law samples

Encoding -law samples is simple. The `tools/` directory in the MIDAS distribution contains directories for each supported platform, and these directories contain a program called `ulaw`. This program can be used to encode 16-bit samples into -law samples, and decode -law samples back to 16-bit ones. The syntax is simple. To encode a 16-bit sample file to -law, use:

```
ulaw e input-file-name output-file-name
```

And to decode a -law file to a 16-bit one, use:

```
ulaw d input-file-name output-file-name
```

The files should contain just raw sample data, with no headers. Stereo and mono samples and streams are processed exactly the same way.

4.2.2 Using -law samples

-law samples and streams are used just like any other samples and streams in MIDAS Digital Audio System. Simply pass the playback function *MIDAS_SAMPLE_ULAW_MONO* or *MIDAS_SAMPLE_ULAW_STEREO* as the sample type, and everything will work normally. -law sample data can be used for both samples and streams.

4.3 Using ADPCM compression

ADPCM streams provide a space-effective way of storing long sections of audio with a fairly good sound quality. Although ADPCM streams have lower sound quality than uncompressed 16-bit ones, they do sound better than 8-bit ones, and, as they only use effectively 4-bit samples, they provide 1:4 compression to the sound. This section descibes how ADPCM streams are encoded and used with MIDAS Digital Audio System.

4.3.1 Encoding ADPCM streams

Encoding ADPCM streams is fairly simple. The `tools/` directory in the MIDAS distribution contains directories for each supported platform, and these directories contain a program called `adpcm`. This program can be used to encode 16-bit streams into ADPCM ones, and decode ADPCM streams back to 16-bit. The syntax is similar to the -law encoder, although a bit more complicated. To encode a 16-bit stream into ADPCM, use:

```
adpcm e input-file-name output-file-name channels frame-length
```

Where **channels** is the number of channels in the stream (1 for mono, 2 for stereo) and **frame-length** the ADPCM frame length in bytes, including the frame header. As ADPCM sample data is adaptative delta encoded, it is normally impossible to start decoding an ADPCM stream from the middle. To get around this problem, MIDAS divides the ADPCM stream into "frames", and is able to start decoding from the beginning of any frame.

The frame length you should use depends on the needs of your application. If your application will always play the streams from beginning to end, any value will do — 1024 is a reasonable choice. However, if stream playback can start from the middle of the stream, you should consider how the stream is accessed. In particular, if the stream is read in blocks of a set number of bytes, the frame length should be equal to the block size.

If you wish to make the ADPCM frames of a given length of time, remember that each ADPCM sample is 4 bits. Therefore, one byte of ADPCM data will contain data for two mono samples or one stereo sample. The ADPCM frame header is 9 bytes long for mono streams and 12 bytes long for stereo ones. Therefore, to get 59ms long frames for a stereo stream played at 22050Hz, the frame length should be 453 bytes.

To decode an ADPCM stream back to a 16-bit one, use:

```
adpcm d input-file-name output-file-name channels frame-length
```

Like with the -law encoder/decoder, the files should contain just raw sample data, with no headers.

4.3.2 Using ADPCM streams

ADPCM streams are used just like other streams in MIDAS Digital Audio System. Simply pass *MIDAS_SAMPLE_ADPCM_MONO* or *MIDAS_SAMPLE_ADPCM_STEREO* as the sample type to the MIDAS stream playback functions, and everything will work normally. ADPCM sample data can only be used for streams. If you are feeding the stream data manually, however, remember that playback can only start from the beginning of an ADPCM frame.

Chapter 5

Operating system specific information

Although the normal MIDAS Digital Audio System APIs are indentical in all supported platforms, there are some operating system specific points that should be noted. In particular, the limitations of the MS-DOS operating system make it somewhat difficult to program under.

5.1 Using DirectSound

Beginning from version 0.7, MIDAS Digital Audio System supports DirectSound for sound output. Although most of the time this is done completely transparently to the user, there are some decisions that need to be made in the initialization phase.

5.1.1 Initialization

By default, DirectSound support in MIDAS Digital Audio System is disabled. To enable it, set *MIDAS_OPTION_DSOUND_MODE* to a value other than *MIDAS_DSOUND_DISABLED*. The DirectSound mode you choose depends on the needs of your application, and the available modes are described in detail in the next section.

In addition to the DirectSound mode, you also need to set the window handle that MIDAS will in turn pass to DirectSound. DirectSound uses this window handle to determine the active window, as only the sound played by the active application will be heard. To set the window handle, simply call

```
MIDASsetOption(MIDAS_OPTION_DSOUND_HWND, (DWORD) hwnd);
```

where **hwnd** is the window handle of your application's main window.

5.1.2 DirectSound modes

Apart from *MIDAS_DSOUND_DISABLED*, three different DirectSound modes are available in MIDAS Digital Audio System. This section describes them in detail.

MIDAS_DSOUND_STREAM: DirectSound stream mode. MIDAS will play its sound to a DirectSound stream buffer, which will then be mixed to the primary buffer by DirectSound. If the DirectSound object hasn't explicitly been set, MIDAS will initialize DirectSound and set the primary buffer format to the same as MIDAS output format. This mode allows arbitrary buffer length, and possibly smoother playback than primary buffer mode, but has a larger CPU overhead.

MIDAS_DSOUND_PRIMARY: DirectSound primary buffer mode. The sound data will be played directly to the DirectSound primary buffer. This mode has the smallest CPU overhead of all available DirectSound modes, and provides smallest possible latency, but is not without its drawbacks: The primary buffer size is set by the driver, and cannot be changed, so the buffer size may be limited.

MIDAS_DSOUND_FORCE_STREAM: This mode behaves exactly like the stream mode, except that DirectSound usage is forced. Normally, MIDAS will not use DirectSound if it is running in emulation mode (as the standard Windows WAVE output device will provide better performance), so this mode must be used to force DirectSound usage. Forcing MIDAS to use DirectSound in stream mode will also the applications to use DirectSound themselves simultaneously.

By default, MIDAS an automatical fallback mechanism for DirectSound modes: If DirectSound support is set to primary mode, but primary buffer is not available for writing, MIDAS will use stream mode instead. And, if DirectSound is running in emulation mode, MIDAS will automatically use normal Win32 audio services instead. This way it is possible to simply set the desired DirectSound mode, and let MIDAS decide the best of the alternatives available.

5.1.3 Buffer sizes

When MIDAS Digital Audio System is using DirectSound with proper drivers (ie. not in emulation mode), much smaller buffer sizes can be used than normal. Because of this, the DirectSound buffer size is set with a different option setting — *MIDAS_OPTION_DSOUND_BUFLEN* — from the normal mixing buffer length. When playing in emulation mode, MIDAS will use the normal mixing buffer length, as smaller buffers can't be used as reliably.

Selecting the correct buffer size is a compromise between sound latency and reliability: the longer the buffer is, the greater latency the sound has, and the longer it takes the sound to actually reach output, but the smaller the buffer is made, the more often the music player needs to be called. To ensure that there are no breaks in sound playback, the music player needs to be

called at least twice, preferably four times, during each buffer length: for a 100ms buffer, for example, the sound player needs to be called at least every 50ms, or 20 times per second.

Although the calling frequency requirements don't seem to be very severe, in practise trying to guarantee that a function gets called even 20 times per second can be difficult. The realtime capabilities of the Win32 platform, especially Windows 95, leave a lot to be desired: A 16-bit program or system service can easily block the system for long periods of time. By default, MIDAS uses a separate thread for background playback, but although this thread runs at a higher priority than the rest of the program, you may find that using manual polling will help you get more consistent and reliable sound playback.

Unfortunately there is no single buffer size that works for everybody, so some experimentation will be needed. The default MIDAS DirectSound buffer size is 100ms, which should be a reasonable compromise for most applications, but, depending on your applications, buffer sizes at 50ms or below should be usable.

5.1.4 Using other DirectSound services with MIDAS

If necessary, it is also possible to use other DirectSound services simultaneously with MIDAS Digital Audio System. In this case, MIDAS should be set to use DirectSound in forced stream mode, and the DirectSound object needs to be explicitly given to MIDAS before initialization:

```
MIDASsetOption{MIDAS_OPTION_DSOUND_OBJECT, (DWORD) ds);
```

Where **ds** is a pointer to the DirectSound object used, returned by *DirectSoundCreate()*. The user is also responsible for setting DirectSound cooperative level and primary buffer format.

Although this DirectSound usage is not recommended, it can be used, for example, to play music with MIDAS while using the DirectSound services directly for playing sound effects.

5.1.5 When to use DirectSound?

Although DirectSound provides a smaller latency than the normal Windows sound devices, and possibly smaller CPU overhead, it is not suitable of all applications. This section gives a quick overview on what applications should use DirectSound, what shouldn't, and which DirectSound mode is most appropriate.

The most important drawback of DirectSound is, that only the active application gets its sound played. While this can be useful with games that run fullscreen, it makes DirectSound completely unusable for applications such as music players, as background playback is impossible with DirectSound. Therefore standalone music player programs should never use DirectSound.

Also, if your application does not benefit from the reduced latency that DirectSound provides, it is safer not to use DirectSound. The DirectSound drivers currently available are not very mature, and the DirectX setup included in the DirectX SDK is far from trouble-free. In addition, programs using DirectSound need to distribute the DirectX runtime with them, making them considerably larger.

However, if you are writing an interactive high-performance application, where strict graphics and sound synchronization is essential, DirectSound is clearly the way to go. For these kind of applications, DirectSound primary buffer should be the best solution, unless there are clear reasons for using stream mode.

5.1.6 DirectSound and multiple windows

When the application uses DirectSound for sound output, only the sound from the active window is played. Therefore DirectSound requires a window handle to be able to determine which window is active. If the application has multiple windows that it needs to activate separately, however, this can cause problems. DirectSound provides no documented way to change the window handle on the fly.

To get around this problem, MIDAS Digital Audio System provides two functions to suspend and resume playback: *MIDASsuspend* and *MIDASresume*. *MIDASsuspend* stops all sound playback, uninitializes the sound output device, and returns it to the operating system. *MIDASresume* in turn resumes sound playback after suspension. These functions can be used to change the DirectSound window handle on the fly: First call *MIDASsuspend*, set the new window handle, and call *MIDASresume* to resume playback. This will cause a break to the sound, and the sound data currently buffered to the sound output device will be lost.

Depending on the application, it may also be possible to get around the DirectSound multiple window problem by creating a hidden parent window for all windows that will be used, and pass the window handle of that parent window to DirectSound.

5.2 MS-DOS timer callbacks

This section describes how MIDAS Digital Audio System uses the MS-DOS system timer, and how to install user timer callbacks. This information is not relevant in other operating systems.

5.2.1 Introduction

To be able to play music in the background in MS-DOS, and to keep proper tempo with all sound cards, MIDAS needs to use the system timer (IRQ 0, interrupt 8) for music playback.

Because of this, user programs may not access the timer directly, as this would cause conflicts with MIDAS music playback. As the system timer is often used for controlling program speed, and running some tasks in the background, MIDAS provides a separate user timer callback for these purposes. This callback should used instead of accessing the timer hardware directly.

The callbacks can be ran at any speed, and can optionally be synchronized to display refresh.

5.2.2 Using timer callbacks

Basic MIDAS timer callback usage is very simple: Simply call *MIDASsetTimerCallbacks*, passing it the desired callback rate and pointers to the callback functions. After that, the callback functions will be called periodically until *MIDASremoveTimerCallbacks* is called. *MIDASsetTimerCallbacks* takes the callback rate in milliHertz (one thousandth of a Hertz) units, so to get a callback that will be called 70 times per second, set the rate to 70000. The callback functions need to use **MIDAS_CALL** calling convention (`__cdecl` for Watcom C, empty for DJGPP), take no arguments and return no value.

For example, this code snippet will use the timer to increment a variable **tickCount** 100 times per second:

```
void MIDAS_CALL TimerCallback(void)
{
    tickCount++;
}
...
MIDASinit();
...
MIDASsetTimerCallbacks(100000, FALSE, &TimerCallback, NULL, NULL);
...
```

5.2.3 Synchronizing to display refresh

The MIDAS timer supports synchronizing the user callbacks to display refresh under some circumstances. Display synchronization does not work when running under Windows 95 and similar systems, and may fail in SVGA modes with many SVGA cards. As display synchronization is somewhat unreliable, and also more difficult to use than normal callbacks, using it is not recommended if a normal callback is sufficient.

To synchronize the timer callbacks to screen refresh, use the following procedure:

1. **BEFORE** MIDAS Digital Audio System is initialized, set up the display mode you are going to use, and get the display refresh rate corresponding to that mode using *MIDASgetDisplayRefreshRate*. If your application uses several different display modes, you will need to set up each

of them in turn and read the refresh rate for each separately. If *MIDASgetDisplayRefreshRate* returns zero, it was unable to determine the display refresh rate, and you should use some default value instead. Display refresh rates, like timer callback rates, are specified in milliHertz (1000*Hz), so 70Hz refresh rate becomes 70000.

2. Initialize MIDAS Digital Audio System etc.

3. Set up the display mode

4. Start the timer callbacks by calling *MIDASsetTimerCallbacks*. The first argument is the refresh rate from step 1, second argument should be set to TRUE (to enable display synchronization), and the remaining three arguments are pointers to the **preVR**, **immVR** and **inVR** callback functions (see descriptions below).

5. When the callbacks are no longer used, remove them by calling *MIDASremoveTimerCallbacks*.

When you are changing display modes, you must first remove the existing timer callbacks, change the display modes, and restart the callbacks with the correct rate. Please note that synchronizing the timer to the screen update takes a while, and as the timer is disabled for that time it may introduce breaks in the music. Therefore we suggest you handle the timer screen synchronization before you start playing music.

If MIDAS is unable to synchronize the timer to display refresh, it will simply run the callbacks like normal user callbacks. Therefore there is no guarantee that the callbacks will actually get synchronized to display, and your program should not depend on that. For example, you should not use the timer callbacks for double buffering the display, as **preVR** might not be called at the correct time — use triple buffering instead to prevent possible flicker.

5.2.4 The callback functions

MIDASsetTimerCallbacks takes as its three last arguments three pointers to the timer callback functions. These functions are:

preVR() — if the callbacks are synchronized to display refresh, this function is called immediately **before** Vertical Retrace starts. It should be kept as short as possible, and can be used for changing a couple of hardware registers (in particular the display start address) or updating a counter.

immVR() — if the callbacks are synchronized to display refresh, this function is called immediately after Vertical Retrace starts. As *preVR()*, it should be kept as short as possible.

inVR() — if the callbacks are synchronized to display refresh, this function is called some time later during Vertical Retrace. It can take a longer time than the two previous functions, and can be used, for example, for setting the VGA palette. It should not take longer than a quarter of the time between callbacks though.

If the callbacks are not synchronized to display refresh, the functions are simply called one after another. The same timing requirements still hold though.

5.2.5 Framerate control

DOS programs typically control their framerate by checking the Vertical Retrace from the VGA hardware. If MIDAS is playing music in the background, this is not a good idea, since the music player can cause the program to miss retraces. Instead, the program should set up a timer callback, possibly synchronize it to display refresh, use that callback to increment a counter, and wait until the counter changes.

For example:

```
volatile unsigned frameCount;
...
void MIDAS_CALL PreVR(void)
{
    frameCount++;
}
...
MIDASsetTimerCallbacks(70000, FALSE, &PreVR, NULL, NULL);
...
while ( !quit )
{
   DoFrame();
   oldCount = frameCount;
   while ( frameCount == oldCount );
}
```

Note that **frameCount** needs to be declared **volatile**, otherwise the compiler might optimize the wait completely away.

A similar strategy can be used to keep the program run at the same speed on different computers. You can use the frame counter to determine how many frames rendering the display takes, and run the movements for all those frames before rendering the next display.