MIDAS Sound System

Programmer's Guide

Petteri Kangaslampi

May 24, 1997

Contents

Chapter 1

Introduction

1.1 Welcome

Welcome to the exciting world of digital audio! MIDAS Sound 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 Sound 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 MIDAS API Reference

1.2 What is MIDAS?

What is MIDAS Sound System anyway?

MIDAS 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 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 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 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 API, no special compilation options are necessary. All MIDAS API definitions are in the file `midasdll.h` [FIXME], 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 Sound 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 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 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 file `midas.lib` 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.

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, both dynamic and static libraries are provided. To link your program with MIDAS, add the proper MIDAS library directory (usually `lib/linux/gcretail`) to your library directory list (gcc option `-L`), and link the library in using the GCC option `-lmidas`. Depending on whether you are building a statically or dynamically linked program, GCC will automatically select the correct library.

2.4 Using MIDAS with Visual C Developer Studio

This section contains contains simple step-by-step instructions for using MIDAS Sound 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.

2.5 Using MIDAS with Watcom C IDE

This section contains contains simple step-by-step instructions for using MIDAS Sound 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.

2.6 A simple module player example

This section describes a very simple example program that uses MIDAS 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.

2.6.1 C module player example

```
#include <stdio.h>
#include <conio.h>
#include "midasdll.h"

int main(void)
{
    MIDASmodule module;

    MIDASstartup();
    MIDASinit();
    MIDASstartBackgroundPlay(0);

    module = MIDASloadModule("..\\data\\templsun.xm");
    MIDASplayModule(module, 0);

    puts("Playing - press any key");
    getch();

    MIDASstopModule(module);
    MIDASfreeModule(module);

    MIDASstopBackgroundPlay();
    MIDASclose();

    return 0;
}
```

2.6.2 Delphi module player example

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

2.6.3 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

9, 6 Resets the MIDAS internal state — This needs to be done before MIDAS is configured and initialized.

10, 7 Initializes MIDAS

11, 8 Starts playing sound in the background

13, 10 Loads the module file

14, 11 Starts playing the module we just loaded, leaving no channels available for sound effects.

16-17, 13-14 Simply waits for a keypress

19, 16 Stops playing the module

20, 17 Deallocates the module we loaded

22, 19 Stops playing sound in the background

23, 20 Uninitializes the MIDAS Sound System

Chapter 3

MIDAS concepts

This chapter describes basic MIDAS concepts and terminology. The description is done at a rather high level, and is kept rather informal — the intention is to document what the different terms mean from the user's point of view, and what can be done with them, rather than to give detailed information on their usage. Step-by-step instructions for using the different MIDAS features is available in the next chapter.

3.1 Samples

A sample in MIDAS Sound System includes the sample data for a sound sample, plus information on its data type, the sample length and looping information. It does not contain other attributes such as volume or panning — these are instrument properties. In MIDAS, samples are managed and stored by by Sound Devices. A sample is identified by a sample handle, and this sample handle can be used to play the sample or remove it from Sound Device storage.

3.2 Streams

In MIDAS, streams are continuous flows of sample data. Unlike samples, streams are not stored in the Sound Devices themselves, but are instead read from the system memory as they are played. Because of this, the complete data for the stream does not need to be available when playing starts, but can instead be read from disk, or even generated on the fly, as playback proceeds. The playback properties of the stream — sampling rate, volume and panning — can be changed in real time just like those of a sample. However, as the sample data for the stream is not stored anywhere in the Sound Device, each stream can only be played once.

The MIDAS API offers three ways for playing streams: file playback, polling playback and callbacks. For file playback, the application simply supplies the system the name of the file containing the sample data for the stream, and playback proceeds automatically. In polling playback mode, the application opens the stream and periodically feeds new sample data to be played. In callback mode, the application sets a callback function that is called each time the stream buffer loops, and can there fill the buffer with new sample data.

The most obvious use for streams is to play long sections of digital sound or music directly from disk, without needing to load everything into memory first. Stream playback could also be used to mix the output of a separate sound generator, such as a speech synthesizer, with the rest of the sound and music in the system.

Currently streams can only be played with software mixing Sound Devices. In addition, file playback is only possible under operating systems that support multithreading.

3.3 Channels

In MIDAS Sound System, channels are used to play the sound. Each channel can play one sample or stream at a time, either mono or stereo. When a new sound is played on a channel, the old one is removed. A channel is identified by a channel number, which range from zero upwards.

Before any sound can be played, a number of channels must be opened from the Sound Device. MIDAS supports an unlimited number of channels on all platforms, unless hardware mixing is used. Although the number of open channels has some impact on CPU usage, the amount of CPU power used depends mostly on the channels that actually play sound. Therefore it is much more important to ensure that no unnecessary sounds are left playing than it is to minimize the number of open channels.

3.4 Panning

Panning in MIDAS Sound System affects the apparent position of the sound in stereo environments. Sound panned to “left” will be played from the left speaker, “right” from the right one and “middle” from both. 64 different panning positions are available between “left” and “middle” plus “right” and “middle”, to give smooth control on the sound position.

In addition, surround sound is also considered a separate panning position. Sounds played with their panning position set to “surround” are played from the surround speakers if the computer is connected to a surround decoder. Even if surround equipment is not available, the sound appears to come from around the listener’s head, not from inside it as in “middle” panning position.

Note that not all Sound Devices necessarily support surround sound. Furthermore, if the computer is playing in stereo mode, but connected to mono equipment, all sounds played in “surround” will disappear!

3.5 Software and hardware mixing

Most sound cards supported by MIDAS Sound System are only capable of playing one digital sound channel at a time, but still MIDAS supports an unlimited number of channels with them. This is accomplished through software mixing of the sound — the sound channels are mixed digitally together before passing the sound to the sound card.

Mixing the sound in software is a complicated process, and, although MIDAS mixing routines are carefully optimized, can still take a considerable amount of CPU time. The CPU time used is determined by four factors: The number of channels active, the mixing rate, the output mode and the type of the samples played. The time used depends almost linearly on the mixing rate and the number of channels, and stereo output can take up to 50% more CPU time than mono. The sample type effect is almost the same - stereo samples can take up to 50% more CPU than mono ones, and 16-bit samples 50% more than 8-bit.

The opposite of software mixing is hardware mixing. Hardware mixing sound cards are capable of playing multiple digital sounds in hardware, and thus there is no need to mix the sound in software. This often uses much less CPU power, depending on the sound card, but as a tradeoff some flexibility is lost. One problem with hardware mixing cards is, that samples have to be placed in on-card memory. Very often the cards come with as little as 512kb of memory, and thus can store only a few samples. The second problem is, that playing streams with hardware mixing cards is usually impossible. This severely limits the usability of hardware mixing cards with MIDAS.

Chapter 4

The system

This chapter gives an overview on how MIDAS Sound System is actually built. It gives descriptions on all main system modules, information on how they use each other, and what their most important characteristics are. This information mainly applies for MIDAS usage below the API level, and most users do not need to be familiar with this information.

4.1 Overview of system architechture

[TBD: picture!]

4.2 Description of all modules

4.2.1 Sound Devices

Sound Devices are at the very heart of MIDAS. They provide the basic functionality for playing audio, they manage the samples loaded in the system, and provide a common programming interface for all supported sound hardware. Normally a user does not use the Sound Devices directly, but rather uses MIDAS upper level programming interfaces and libraries to play sound. However, internally MIDAS is very much centered around Sound Devices, and anyone doing any lower-level MIDAS programming or just examining the system source code can't avoid working with them.

A single Sound Device can contain information for one or more sound card. In that case the cards are usually different models of the same sound card - for example, the Sound Blaster series Sound Device includes five different sound card types: Sound Blaster 1.0, 1.5, 2.0, Pro and 16.

Internally, a Sound Device is represented by one large data structure. The structure contains a set of variables for configuring the Sound Device, plus a number of function pointers. These function pointers are then used to call the Sound Device functions. This ensures that any code working with MIDAS Sound Devices only has to maintain a pointer to the current Sound Device structure in use, and can use any supported Sound Device without code modification.

4.2.2 Generic Module Player

Generic Module Player, or GMPlayer for short, takes care of all module playing in MIDAS. Modules in different formats are converted to GMPlayer internal *gmpModule* structure format when loaded to memory, so that a single module player can handle all formats. To maintain best possible compatibility with all trackers, however, GMPlayer implements some commands and features differently depending on the original module format. This ensures that playback quality is not compromised.

GMPlayer is not limited for playing a single module only, in fact it can be used to play an unlimited number of songs simultaneously. This way sound effects can be composed as small song fragments, to eliminate the need for long samples.

GMPlayer is not dependent on the underlying sound hardware — it operates as easily with no sound output at all as it does with an advanced wavetable sound card. At initialization phase, GMPlayer is passed a pointer to the Sound Device that should be used for music playback, and afterwards it directs all sounds to that Sound Device.

4.2.3 MIDAS kernel

The MIDAS ”kernel” consists of smaller modules used by all system components. This includes memory management functions, error handling routines, file management functions and common utility functions. These kernel functions ensure that MIDAS does not depend on the underlying operating system any C runtime libraries — all dependencies are encapsulated in single modules.

4.3 Calling conventions

All internal MIDAS Sound System functions use the same calling conventions and a similar method for returning data. This simplifies programming, as all functions behave consistently, and ensures that error codes get handled properly.

4.3.1 Error codes

All functions in MIDAS Sound System return an error code. There is no exception to this, and all other data is returned using pointers (see below). This simplifies error handling, and makes sure error codes always get handled properly. In addition, when MIDAS is compiled with DEBUG mode on, the error handling functions can supply detailed information about the errors that happened, such as which function originally caused the error and what functions called it.

4.3.2 Returning data

As the return value of the function is always reserved for the error code, all other data will need to be returned by using pointers. All MIDAS functions that return data will accept as their last arguments pointers to the variables that will hold the return values. Make sure you **always** pass these functions legal pointers as the return variables even if you don't need the return values — memory corruption may occur otherwise.

For example, to query the current volume on channel 3 in the default Sound Device, you can use:

```
unsigned vol;
...
midasSD->GetVolume(3, &vol);
...
```

Note that in a real-life program you also need to handle the returned error code.

MS-DOS Note! Code that will be called from the MIDAS interrupts needs to have the compiler ”SS==DS” -assumption disabled. This is because the stack used by the timer interrupt (and thus most of MIDAS code plus any user callbacks) might be different from the main program stack, and the compiler should therefore not generate code that uses data segment variables via the stack segment.

Note that this does **not** apply to all code that calls MIDAS, only code inside MIDAS plus any timer or module player callbacks.

4.4 Error handling

The section above already gave a brief introduction to MIDAS error handling, but this section will describe it in detail.

MIDAS uses a common method for handling errors throughout the system. All functions return an error code, and it is up to the caller to decide how to handle the error condition. Most functions simply pass the error code to their caller, but it is recommended that the function does as much clean-up as possible before doing so. In particular, all allocated memory should be deallocated and all open files closed.

When MIDAS is compiled in DEBUG mode, the error handling system has additional functionality. It will build up a list of all errors that occurred, with the name of the function that raised the error. When errors are then passed upwards in the function call stack, all functions are added to the list. Thus the error exit not only reports what error occurred, but also what function caused the error and where it was called.

Chapter 5

Operating system specific information

Although the normal MIDAS 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 now 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 is disabled. To enable DirectSound support, 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. 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 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 using 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. 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 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 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 Sound 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 Sound 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.