MIDAS Sound System

Programmer's Guide

Petteri Kangaslampi

January 27, 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 application.

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 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.

2.2 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.

2.2.1 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 in the library file corresponding to your development platform. All MIDAS libraries are located in the `lib/` subdirectory of the distribution. Watcom C/C++ users need to link with the file `midasnt_watcom.lib`, and Visual C/C++ users with `midasnt_visualc.lib`. Depending on your configuration, you may need to add the `lib/` subdirectory into 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 library names correspond to the `.dll` name: `midasXX_visualc.lib` for Visual C/C++ and `midasXX_watcom.lib` for Watcom C/C++. (the XX stands for the MIDAS version number).

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.

2.2.2 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 `midasdos.lib` from the `lib/` subdirectory of the MIDAS distribution, and your program is ready to use MIDAS Sound System. 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.

2.2.3 Linux

For Linux, both dynamic and static libraries are provided. To link your program with MIDAS, add the MIDAS library directory 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.

2.3 An example program

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.

2.3.1 C program

```
#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;
}
```

2.3.2 Delphi program

```
uses midasdll;

var module : MIDASmodule;

BEGIN
    MIDASstartup;
    MIDASinit;
    MIDASstartBackgroundPlay(0)

    module := MIDASloadModule(’..\data\templsun.xm’);
    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.
```

2.3.3 Program 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.

[TODO:OS-Specific] **MS-DOS Note!** Code using MIDAS 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. Because of this, any pointers that you pass to MIDAS functions need to point to data segment variables — in other words, in the example above,

```
unsigned vol;
```

needs to be changed to

```
static unsigned vol;
```

to ensure the local variable gets placed in data segment. Pointers to global variables can naturally be used without any modifications.

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 MS-DOS timer screen synchronization

This section describes how to use the built-in MIDAS screen synchronized timer under MS-DOS. This information is not relevant in other operating systems.

5.1.1 Introduction

MIDAS Sound System timer includes built-in support for screen synchronization. This means that you can program the timer to call your own routines every frame, more exactly immediately before the Vertical Retrace, immediately after the Vertical Retrace has started and later during the Vertical Retrace. This enables you to synchronize your program to the screen update to get smooth animation, which otherwise would not be possible with a music system playing in the background. The routines can also be used for changing VGA hardware registers, such as display start address and scrolling, in correct moments, for triple buffering and for compensating for different machine speeds.

5.1.2 Using screen synchronization

If you wish to use timer screen synchronization, use the procedure outlined below:

1. BEFORE MIDAS Sound System is initialized, set up the display mode you are intending to use and get the timer screen synchronization value corresponding to that display mode using *tmrGetScrSync()*. If you are using several display modes with different refresh rates (in practise, different vertical resolutions, although in standard VGA only 240 or 480 scan line modes have different refresh rates) you must activate each of them in turn and get the synchronization values for each of them.

2. Initialize MIDAS Sound System etc.

3. Set up the display mode

4. When you need timer screen synchronization, start it using the function *tmrSyncScr()*, passing as arguments the timer screen synchronization value from step 1 and pointers to the routines you wish the timer to call. If you do not require support for some routine, pass a NULL pointer instead.

5. When timer screen synchronization is no longer required, stop it using *tmrStopScrSync()*.

If you change the display mode to one with a different refresh rate, you must first stop the screen synchronization, change the display mode, and after that re-synchronize the timer. 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.

5.1.3 Timer screen synchronized routines

tmrSyncScr() takes as arguments pointers to three separate functions — *preVR()*, *immVR()* and *inVR()* — that will be called at different points during the screen timer interrupt. Following is a brief description of each routine and what it is intended for.

preVR() is called immediately before Vertical Retrace, and must be as short as possible to avoid timer synchronization problems. It is intended mainly for changing the display start address register and updating counters.

immVR() is called immediately after Vertical Retrace has started, and must be as short as possible to avoid timer synchronization problems. It is intended mainly for changing VGA hardware registers that have to be modified during Vertical Retrace, such as pixel panning.

inVR() is called after *immVR()*, and may take a longer time if necessary. However, note that even though spending a long time in *inVR()* does not induce timer synchronization problems, it may cause problems in music tempo if it takes a too long time. Furthermore, the time spent in *inVR()* must not exceed one frame. *inVR()* is mainly intended for changing the palette or updating small portions of screen, such as drawing new characters to a start address scroller.

5.1.4 Waiting for Vertical Retrace

When synchronizing your program to the screen update, instead of waiting for Vertical Retrace using the VGA hardware registers you must use the screen synchronized timer for this. This is because the music playing interrupt may occur just during the Vertical Retrace, causing you to miss one frame completely. To use the timer for this, set up a *preVR()* routine that increments a frame counter, and instead of waiting for Vertical Retrace bit wait for the frame counter to change. For example:

```
preVR:
        frameCount = frameCount + 1;

main:
        ...
        tmrSyncScr(scrSync, &preVR, NULL, NULL)
        ...
        oldCnt = frameCount;
        while ( oldCnt == frameCount );
```

Note that you must declare frameCount as **volatile**, or otherwise ensure that the compiler will not optimize the frame waiting loop to an infinite one, waiting for a register variable to change.

5.1.5 Speed compensation

The timer screen synchronization can also be used to compensate for different speeds on different computers. The following pseudo code should illustrate the point:

```
main loop:
        Wait for frameCount to change
        skipFrames = oldFrameCount - frameCount
        oldFrameCount = frameCount
        for i = 1 to skipFrames do
                MoveEverything
        DrawEverything
```

5.1.6 Implementation notes

Unlike in some old MIDAS versions, the current timer code no longer crashes under Windows 95. However, proper screen sychronization is simply impossible under Windows 95, due to inadequacies in Windows 95 timer hardware emulation, and MIDAS will just set up a steady timer of 70Hz (or what ever refresh rate you set) and call the display synchronization routines there.

For most purposes, this should present no problems to you. The display update might not be as smooth as under plain DOS, but no DOS programs run absolutely smoothly under Win95 anyway and the users should be used to that. However, if you use the screen synchronized timer for page flipping, you'll need to change from double to triple buffering. The reason for this is, that you can no longer be sure that the new page is indeed displayed after the timer interrupt that changed the start address returns. VGA will make use of the start address only at the next Vertical Retrace, and there is no guarantee that the timer comes at Vertical Retrace time (in fact it usually doesn't).

If MIDAS is unable to determine the correct display sync value (ie. the frame rate), it will use the frame rate set in the variable *mDefaultFramerate* (in 100*Hz =¿ 70Hz becomes 7000). This variable is initially set to 7000 (70Hz), but if you know the refresh rate of your display mode is different you can change this before calling *mGetScrSync()*. If your application uses several display modes with different frame rates, just set mDefaultFramerate before reading the sync value for each of them as follows:

```
SetFirstMode();
mDefaultFrameRate = 6000; /* 60 Hz */
mGetScrSync(...)
SetSecondMode()
mDefaultFrameRate = 5000; /* 50 Hz */
mGetScrSync(...)
```

To check whether MIDAS was able to synchronize to the screen correctly, check the variable *mSyncScreen* after calling *mGetScrSync()*. If the variable is 0, MIDAS was unable to determine the frame rate, took it from *mDefaultFramerate*, and is now running in Win95 compatibility mode. You'll probably want warn the user if that is the case:

```
SetMode();
mGetScrSync(...)
SetTextMode();
if ( mSyncScreen != 1 )
{
    printf("Warning! Unable to synchronize the program to "
           "screen update!\n\n"
           "This normally only happens when running under "
           "Windows 95 or similar system,\nplease consider"
           " running this program in MS-DOS mode. The "
           "program will work,\n but the screen update "
           "will not be as smooth as possible, and there "
           "may be\nproblems with music playback.\n\n"
           "Press any key to continue.\n");
    getch();
}
```

To force Win95 compatibility mode, set mSyncScreen to 0 yourself before calling mGetScrSync the first time. This allows you to set the timer run at a constant rate, independent of the display refresh rate.

5.2 Linking with Watcom C

Important note on linking MIDAS with Watcom C: Some Watcom linker versions are case insensitive by default. To be able to link MIDAS correctly, make sure you use the linker option

```
option caseexat
```