MikMod Sound Library

Documentation edition 1.3
December 2013

Miodrag Vallat
(miod@mikmod.org)

1 Introduction

The MikMod sound library is an excellent way for a programmer to add music and sound effects to an application. It is a powerful and flexible library, with a simple and easy-to-learn API.

Besides, the library is very portable and runs under a lot of Unices, as well as under OS/2, MacOS and Windows. Third party individuals also maintain ports on other systems, including MS-DOS, and BeOS.

MikMod is able to play a wide range of module formats, as well as digital sound files. It can take advantage of particular features of your system, such as sound redirection over the network. And due to its modular nature, the library can be extended to support more sound or module formats, as well as new hardware or other sound output capabilities, as they appear.

2 Tutorial

This chapter will describe how to quickly incorporate MikMod's power into your programs. It doesn't cover everything, but that's a start and I hope it will help you understand the library philosophy.

If you have a real tutorial to put here, you're welcome ! Please send it to me. . .

2.1 MikMod Concepts

MikMod's sound output is composed of several sound *voices* which are mixed, either in software or in hardware, depending of your hardware configuration. Simple sounds, like sound effects, use only one voice, whereas sound modules, which are complex arrangements of sound effects, use several voices.

MikMod's functions operate either globally, or at the voice level. Differences in the handling of sound effects and modules are kept minimal, at least for the programmer.

The sound playback is done by a *sound driver*. MikMod provides several sound drivers: different hardware drivers, and some software drivers to redirect sound in a file, or over the network. You can even add your own driver, register it to make it known by the library, and select it (this is exactly what the module plugin of xmms does).

2.2 A Skeleton Program

To use MikMod in your program, there are a few steps required:

- Include '`mikmod.h`' in your program.
- Register the MikMod drivers you need.
- Initialize the library with MikMod_Init() before using any other MikMod function.
- Give up resources with MikMod_Exit() at the end of your program, or before when MikMod is not needed anymore.
- Link your application with the MikMod sound library.

Here's a program which meets all those conditions:

```
/* MikMod Sound Library example program: a skeleton */

#include <mikmod.h>

main()
{
/* register all the drivers */
MikMod_RegisterAllDrivers();

/* initialize the library */
MikMod_Init("");

/* we could play some sound here... */

/* give up */
```

```
MikMod_Exit();
}
```

This program would be compiled with the following command line: `cc -o example example.c ‘libmikmod-config --cflags‘ ‘libmikmod-config --libs‘`

Although this programs produces no useful result, many things happen when you run it. The call to `MikMod_RegisterAllDrivers` registers all the drivers embedded in the MikMod library. Then, `MikMod_Init` chooses the more adequate driver and initializes it. The program is now ready to produce sound. When sound is not needed any more, `MikMod_Exit` is used to relinquish memory and let other programs have access to the sound hardware.

2.3 Playing Modules

Our program is not really useful if it doesn't produce sound. Let's suppose you've got this good old module, "Beyond music", in the file '`beyond music.mod`'. How about playing it ?

To do this, we'll use the following code:

```
/* MikMod Sound Library example program: a simple module player */

#include <unistd.h>
#include <mikmod.h>

main()
{
    MODULE *module;

    /* register all the drivers */
    MikMod_RegisterAllDrivers();

    /* register all the module loaders */
    MikMod_RegisterAllLoaders();

    /* initialize the library */
    md_mode |= DMODE_SOFT_MUSIC;
    if (MikMod_Init("")) {
        fprintf(stderr, "Could not initialize sound, reason: %s\n",
                MikMod_strerror(MikMod_errno));
        return;
    }

    /* load module */
    module = Player_Load("beyond music.mod", 64, 0);
    if (module) {
        /* start module */
        Player_Start(module);

        while (Player_Active()) {
```

```
            /* we're playing */
            usleep(10000);
            MikMod_Update();
        }

        Player_Stop();
        Player_Free(module);
    } else
        fprintf(stderr, "Could not load module, reason: %s\n",
                MikMod_strerror(MikMod_errno));

    /* give up */
    MikMod_Exit();
}
```

What's new here ? First, we've not only registered MikMod's device driver, but also the module loaders. MikMod comes with a large choice of module loaders, each one for a different module type. Since *every* loader is called to determine the type of the module when we try to load them, you may want to register only a few of them to save time. In our case, we don't matter, so we happily register every module loader.

Then, there's an extra line before calling `MikMod_Init`. We change the value of MikMod's variable `md_mode` to tell the library that we want the module to be processed by the software. If you're the happy owner of a GUS-type card, you could use the specific hardware driver for this card, but in this case you should not set the `DMODE_SOFT_MUSIC` flag.

We'll ensure that `MikMod_Init` was successful. Note that, in case of error, MikMod provides the variable `MikMod_errno`, an equivalent of the C library `errno` for MikMod errors, and the function `MikMod_strerror`, an equivalent to `strerror`.

Now onto serious business ! The module is loaded with the `Player_Load` function, which takes the name of the module file, and the number of voices afforded to the module. In this case, the module has only 4 channels, so 4 voices, but complex Impulse Tracker modules can have a lot of voices (as they can have as many as 256 virtual channels with so-called "new note actions"). Since empty voices don't cost time to be processed, it is safe to use a big value, such as 64 or 128. The third parameter is the "curiosity" of the loader: if nonzero, the loader will search for hidden parts in the module. However, only a few module formats can embed hidden or non played parts, so we'll use 0 here.

Now that the module is ready to play, let's play it. We inform the player that the current module is `module` with `Player_Start`. Playback starts, but we have to update it on a regular basis. So there's a loop on the result of the `Player_Active` function, which will tell us if the module has finished. To update the sound, we simply call `MikMod_Update`.

After the module has finished, we tell the player its job is done with `Player_Stop`, and we free the module with `Player_Free`.

2.4 Playing Sound Effects

MikMod is not limited to playing modules, it can also play sound effects, that is, module samples. It's a bit more complex than playing a module, because the module player does

a lot of things for us, but here we'll get more control over what is actually played by the program. Let's look at an example:

```
/* MikMod Sound Library example program: sound effects */

#include <unistd.h>
#include <mikmod.h>

main()
{
    int i;
    /* sound effects */
    SAMPLE *sfx1, *sfx2;
    /* voices */
    int v1, v2;

    /* register all the drivers */
    MikMod_RegisterAllDrivers();

    /* initialize the library */
    md_mode |= DMODE_SOFT_SNDFX;
    if (MikMod_Init("")) {
        fprintf(stderr, "Could not initialize sound, reason: %s\n",
                MikMod_strerror(MikMod_errno));
        return;
    }

    /* load samples */
    sfx1 = Sample_Load("first.wav");
    if (!sfx1) {
        MikMod_Exit();
        fprintf(stderr, "Could not load the first sound, reason: %s\n",
                MikMod_strerror(MikMod_errno));
        return;
    }
    sfx2 = Sample_Load("second.wav");
    if (!sfx2) {
        Sample_Free(sfx1);
        MikMod_Exit();
        fprintf(stderr, "Could not load the second sound, reason: %s\n",
                MikMod_strerror(MikMod_errno));
        return;
    }

    /* reserve 2 voices for sound effects */
    MikMod_SetNumVoices(-1, 2);
```

```
    /* get ready to play */
    MikMod_EnableOutput();

    /* play first sample */
    v1 = Sample_Play(sfx1, 0, 0);
    for(i = 0; i < 5; i++) {
        MikMod_Update();
        usleep(100000);
    }

    /* half a second later, play second sample */
    v2 = Sample_Play(sfx2, 0, 0);
    do {
        MikMod_Update();
        usleep(100000);
    } while (!Voice_Stopped(v2));

    MikMod_DisableOutput();

    Sample_Free(sfx2);
    Sample_Free(sfx1);

    MikMod_Exit();
}
```

As in the previous example, we begin by registering the sound drivers and initializing the library. We also ask for software mixing by modifying the variable `md_mode`.

It's time to load our files, with the `Sample_Load` function. Don't forget to test the return value — it looks ugly here on such a small example, but it's a good practice. . .

Since we want to play two samples, we have to use at least two voices for this, so we reserve them with a `MikMod_SetNumVoices` call. The first parameter sets the number of module voices, and the second parameter the number of sound effect voices. We don't want to set the number of module voices here (it's part of the module player's duty), so we use the value `-1` to keep the current value, and we reserve two sound effect voices.

Now we're ready to play, so we call `MikMod_EnableOutput` to make the driver ready. Sound effects are played by the `Sample_Play` function. You just have to specify which sample you want to play, the offset from which you want to start, and the playback flags. More on this later. The function returns the number of the voice associated to the sample.

We play the first sample for half a second, then we start to play the second sample. Since we've reserved two channels, both samples play simultaneously. We use the `Voice_Stopped` function to stop the playback: it returns the current status of the voice argument, which is zero when the sample plays and nonzero when it has finished. So the `do` loop will stop exactly when the second sample is finished, regardless of the length of the first sample.

To finish, we get rid of the samples with `Sample_Free`.

2.5 More Sound Effects

Sound effects have some attributes that can be affected to control the playback. These are speed, panning, and volume. Given a voice number, you can affect these attributes with the `Voice_SetFrequency`, `Voice_SetPanning` and `Voice_SetVolume` functions.

In the previous example, we'll replace the actual sound code, located between the calls to `MikMod_EnableOutput` and `MikMod_DisableOutput`, with the following code:

```
Sample_Play(sfx1, 0, 0);
for(i = 0; i < 5; i++) {
    MikMod_Update();
    usleep(100000);
}
v2 = Sample_Play(sfx2, 0, SFX_CRITICAL);
i = 0;
do {
    MikMod_Update();
    usleep(100000);
    v1 = Sample_Play(sfx1, 0, 0);
    Voice_SetVolume(v1, 160);
    Voice_SetFrequency(v1, (sfx1->speed * (100 + i)) / 100);
    Voice_SetPanning(v2, (i++ & 1) ? PAN_LEFT : PAN_RIGHT);
} while (!Voice_Stopped(v2));
```

The first thing you'll notice, is the `SFX_CRITICAL` flag used to play the second sample. Since the `do` loop will add another sample every 100 milliseconds, and we reserved only two voices, the oldest voice will be cut each time this is necessary. Doing this would cut the second sample in the second iteration of the loop. However, since we flagged this sound as "critical", it won't be cut until it is finished or we stop it with a `Voice_Stop` call. So the second sample will play fine, whereas the first sample will be stopped every loop iteration.

Then, we choose to play the first sample a bit lower, with `Voice_SetVolume`. Volume voices range from 0 (silence) to 256. In this case we play the sample at 160. To make the sound look weird, we also change its frequency with `Voice_SetFrequency`. The computation in the example code makes the frequency more and more high (starting from the sample frequency and then increasing from 1% each iteration).

And to demonstrate the `Voice_SetPanning` function, we change the panning of the second sample at each iteration from the left to the right. The argument can be one of the standard panning `PAN_LEFT`, `PAN_RIGHT`, `PAN_CENTER` and `PAN_SURROUND`[1], or a numeric value between 0 (`PAN_LEFT`) and 255 (`PAN_RIGHT`).

[1] `PAN_SURROUND` will be mapped to `PAN_CENTER` if the library is initialized without surround sound, that is, if the variable `md_mode` doesn't have the bit `DMODE_SURROUND` set.

3 Using the Library

This chapter describes the various parts of the library and their uses.

3.1 Library Version

If your program is dynamically linked with the MikMod library, you should check which version of the library you're working with. To do this, the library defines a few constants and a function to help you determine if the current library is adequate for your needs or if it has to be upgraded.

When your program includes `mikmod.h`, the following constants are defined:

- `LIBMIKMOD_VERSION_MAJOR` is equal to the major version number of the library.
- `LIBMIKMOD_VERSION_MINOR` is equal to the minor version number of the library.
- `LIBMIKMOD_REVISION` is equal to the revision number of the library.
- `LIBMIKMOD_VERSION` is the sum of `LIBMIKMOD_VERSION_MAJOR` shifted 16 times, `LIBMIKMOD_VERSION_MINOR` shifted 8 times, and `LIBMIKMOD_REVISION`.

So your program can tell with which version of the library it has been compiled this way:

```
printf("Compiled with MikMod Sound Library version %ld.%ld.%ld\n",
       LIBMIKMOD_VERSION_MAJOR,
       LIBMIKMOD_VERSION_MINOR,
       LIBMIKMOD_REVISION);
```

The library defines the function `MikMod_GetVersion` which returns the value of LIBMIKMOD_VERSION for the library. If this value is greater than or equal to the value of LIBMIKMOD_VERSION for your program, your program will work; otherwise, you'll have to inform the user that he has to upgrade the library:

```
{
    long engineversion = MikMod_GetVersion();

    if (engineversion < LIBMIKMOD_VERSION) {
        printf("MikMod library version (%ld.%ld.%ld) is too old.\n",
               (engineversion >> 16) & 255,
               (engineversion >> 8) & 255,
               (engineversion) & 255);
        printf("This programs requires at least version %ld.%ld.%ld\n",
               LIBMIKMOD_VERSION_MAJOR,
               LIBMIKMOD_VERSION_MINOR,
               LIBMIKMOD_REVISION);
        puts("Please upgrade your MikMod library.");
        exit(1);
    }
}
```

3.2 Type Definitions

MikMod defines several data types to deal with modules and sample data. These types have the same memory size on every platform MikMod has been ported to.

These types are:

- `CHAR` is a printable character. For now it is the same as the `char` type, but in the future it may be wide char (Unicode) on some platforms.
- `SBYTE` is a signed 8 bit number (can range from -128 to 127).
- `UBYTE` is an unsigned 8 bit number (can range from 0 to 255).
- `SWORD` is a signed 16 bit number (can range from -32768 to 32767).
- `UWORD` is an unsigned 16 bit number (can range from 0 to 65535).
- `SLONG` is a signed 32 bit number (can range from -2.147.483.648 to 2.147.483.647).
- `ULONG` is an unsigned 32 bit number (can range from 0 to 4.294.967.296).
- `BOOL` is a boolean value. A value of 0 means false, any other value means true.

3.3 Error Handling

Although MikMod does its best to do its work, there are times where it can't. For example, if you're trying to play a corrupted file, well, it can't.

A lot of MikMod functions return pointers or `BOOL` values. If the pointer is `NULL` or the `BOOL` is 0 (false), an error has occurred.

MikMod errors are returned in the variable `MikMod_errno`. Each possible error has a symbolic error code, beginning with `MMERR_`. For example, if MikMod can't open a file, `MikMod_errno` will receive the value `MMERR_OPENING_FILE`.

You can get an appropriate error message to display from the function `MikMod_strerror`.

There is a second error variable named `MikMod_critical`. As its name suggests, it is only set if the error lets the library in an unstable state. This variable can only be set by the functions `MikMod_Init`, `MikMod_SetNumVoices` and `MikMod_EnableOutput`. If one of these functions return an error and `MikMod_critical` is set, the library is left in the uninitialized state (i.e. it was not initialized, or `MikMod_Exit` was called).

If you prefer, you can use a callback function to get notified of errors. This function must be prototyped as `void MyFunction(void)`. Then, call `MikMod_RegisterHandler` with your function as argument to have it notified when an error occurs. There can only be one callback function registered, but `MikMod_RegisterHandler` will return you the previous handler, so you can chain handlers if you want to.

3.4 Library Initialization and Core Functions

To initialize the library, you must register some sound drivers first. You can either register all the drivers embedded in the library for your platform with `MikMod_RegisterAllDrivers`, or register only some of them with `MikMod_RegisterDriver`. If you choose to register the drivers manually, you must be careful in their order, since `MikMod_Init` will try them in the order you registered them. The `MikMod_RegisterAllDrivers` function registers the network drivers first (for playing sound over the network), then the hardware drivers, then the disk writers, and in last resort, the nosound driver. Registering the nosound driver first would not be a very good idea. . .

You can get some printable information regarding the registered drivers with `MikMod_InfoDriver`; don't forget to call `free` on the returned string when you don't need it anymore.

After you've registered your drivers, you can initialize the sound playback with `MikMod_Init`, passing specific information to the driver if necessary. If you set the variable `md_device` to zero, which is its default value, the driver will be autodetected, that is, the first driver in the list that is available on the system will be used; otherwise only the driver whose order in the list of the registered drivers is equal to `md_device` will be tried. If your playback settings, in the variables `md_mixfreq` and `md_mode`, are not supported by the device, `MikMod_Init` will fail.

You can then choose the number of voices you need with `MikMod_SetNumVoices`, and activate the playback with `MikMod_EnableOutput`.

Don't forget to call `MikMod_Update` as often as possible to process the sound mixing. If necessary, fork a dedicated process to do this, or if the library is thread-safe on your system, use a dedicated thread.

If you want to change playback settings, most of them can't be changed on the fly. You'll need to stop the playback and reinitialize the driver. Use `MikMod_Active` to check if there is still sound playing; in this case, call `MikMod_DisableOutput` to end playback. Then, change your settings and call `MikMod_Reset`. You're now ready to select your number of voices and restart playback.

When your program ends, don't forget to stop playback and call `MikMod_Exit` to leave the sound hardware in a coherent state.

On systems that have pthreads, libmikmod is thread-safe[1]. You can check this in your programs with the `MikMod_InitThreads` function. If this function returns 1, the library is thread-safe.

The main benefit of thread-safety is that `MikMod_Update` can be called from a separate thread, which often makes application design easier. However, several libmikmod global variables are accessible from all your threads, so when more than one thread need to access libmikmod variables, you'll have to protect these access with the `MikMod_Lock` and `MikMod_Unlock` functions. If libmikmod is not thread-safe, these functions are no-ops.

3.5 Samples and Voice Control

Currently, MikMod only supports uncompressed mono WAV files as samples. You can load a sample by calling `Sample_Load` with a filename, or by calling `Sample_LoadFP` with an open `FILE*` pointer. These functions return a pointer to a `SAMPLE` structure, or `NULL` in case of error.

The `SAMPLE` structure has a few interesting fields:

- `speed` contains the default frequency of the sample.
- `volume` contains the default volume of the sample, ranging from 0 (silence) to 64.
- `panning` contains the default panning position of the sample.

Altering one of those fields will affect all voices currently playing the sample. You can achieve the same result on a single voice with the functions `Voice_SetFrequency`,

[1] Unless you explicitely choose to create a non thread-safe version of libmikmod at compile-time.

`Voice_SetVolume` and `Voice_SetPanning`. Since the same sample can be played with different frequency, volume and panning parameters on each voice, you can get voice specific information with `Voice_GetFrequency`, `Voice_GetVolume` and `Voice_GetPanning`.

You can also make your sample loop by setting the fields `loopstart` and `loopend` and or'ing `flags` with `SF_LOOP`. To compute your loop values, the field `length` will be useful. However, you must know that all the sample length are expressed in samples, i.e. 8 bits for an 8 bit sample, and 16 bit for a 16 bit sample... Test `flags` for the value `SF_16BITS` to know this.

Speaking of flags, if you're curious and want to know the original format of the sample on disk (since libmikmod does some work on the sample data internally), refer to the `inflags` field.

If the common forward loop isn't enough, you can play with some other flags: `SF_BIDI` will make your sample loop "ping pong" (back and forth), and `SF_REVERSE` will make it play backwards.

To play your sample, use the `Sample_Play` function. This function will return a voice number which enable you to use the `Voice_xx` functions.

The sample will play until another sample takes over its voice (when you play more samples than you reserved sound effect voices), unless it has been flagged as `SFX_CRITICAL`. You can force it to stop with `Voice_Stop`, or you can force another sample to take over this voice with `Voice_Play`; however `Voice_Play` doesn't let you flag the new sample as critical.

Non looping samples will free their voice channel as soon as they are finished; you can know the current playback position of your sample with `Voice_GetPosition`. If it is zero, either the sample has finished playing or it is just beginning; use `Voice_Stopped` to know.

When you don't need a sample anymore, don't forget to free its memory with `Sample_Free`.

3.6 Modules and Player Control

As for the sound drivers, you have to register the module loaders you want to use for MikMod to be able to load modules. You can either register all the module loaders with `MikMod_RegisterAllLoaders`, or only a few of them with `MikMod_RegisterLoader`. Be careful if you choose this solution, as the 15 instrument MOD loader has to be registered last, since loaders are called in the order they were register to identify modules, and the detection of this format is not fully reliable, so other modules might be mistaken as 15 instrument MOD files.

You can get some printable information regarding the registered loaders with `MikMod_InfoLoader`; don't forget to call `free` on the returned string when you don't need it anymore.

Note that, contrary to the sound drivers, you can register module loaders at any time, it doesn't matter.

For playlists, you might be interested in knowing the module title first, and `Player_LoadTitle` will give you this information. Don't forget to `free` the returned text when you don't need it anymore.

You can load a module either with `Player_Load` and the name of the module, or with `Player_LoadFP` and an open `FILE*` pointer. These functions also expect a maximal number of voices, and a curiosity flag. Unless you have excellent reasons not to do so, choose a big limit, such as 64 or even 128 for complex Impulse Tracker modules. Both functions return a pointer to an `MODULE` structure, or `NULL` if an error occurs.

You'll find some useful information in this structure:

- `numchn` contains the number of module "real" channels.
- `numvoices` contains the number of voices reserved by the player for the real channels and the virtual channels (NNA).
- `numpas` and `numpat` contain the number of song positions and song patterns.
- `numins` and `numsmp` contain the number of instruments and samples.
- `songname` contains the song title.
- `modtype` contains the name of the tracker used to create the song.
- `comment` contains the song comment, if it has one.
- `sngtime` contains the time elapsed in the module, in $2^{-}10$ seconds (not exactly a millisecond).
- `sngspd` and `bpm` contain the song speed and tempo.
- `realchn` contains the actual number of active channels.
- `totalchn` contains the actual number of active virtual channels, i.e. the sum of `realchn` and the number of NNA virtual channels.

Now that the module is loaded, you need to tell the module player that you want to play this particular module with `Player_Start` (the player can only play one module, but you can have several modules in memory). The playback begins. Should you forget which module is playing, `Player_GetModule` will return it to you.

You can change the current song position with the functions `Player_NextPosition`, `Player_PrevPosition` and `Player_SetPosition`, the speed with `Player_SetSpeed` and `Player_SetTempo`, and the volume (ranging from 0 to 128) with `Player_SetVolume`.

Playback can be paused or resumed with `Player_TogglePause`. Be sure to check with `Player_Paused` that it isn't already in the state you want !

Fine player control is achieved by the functions `Player_Mute`, `Player_UnMute` and `Player_ToggleMute` which can silence or resume a set of module channels. The function `Player_Muted` will return the state of a given channel. And if you want even more control, you can get the voice corresponding to a module channel with `Player_GetChannelVoice` and act directly on the voice.

Modules play only once, but can loop indefinitely if they are designed to do so. You can change this behavior with the `wrap` and `loop` of the `MODULE` structure; the first one, if set, will make the module restart when it's finished, and the second one, if set, will prevent the module from jumping backwards.

You can test if the module is still playing with `Player_Active`, and you can stop it at any time with `Player_Stop`. When the module isn't needed anymore, get rid of it with `Player_Free`.

3.7 Loading Data from Memory

If you need to load modules or sound effects from other places than plain files, you can use the `MREADER` and `MWRITER` objects to achieve this.

The `MREADER` and `MWRITER` structures contain a list of function pointers, which emulate the behaviour of a regular `FILE *` object. In fact, all functions which take filenames or `FILE *` as arguments are only wrappers to a real function which takes an `MREADER` or an `MWRITER` argument.

So, if you need to load a module from memory, or for a multi-file archive, for example, all you need is to build an adequate `MREADER` object, and use `Player_LoadGeneric` instead of `Player_Load` or `Player_LoadFP`. For samples, use `Sample_LoadGeneric` instead of `Sample_Load` or `Sample_LoadFP`.

4 Library Reference

This chapter describes in more detail all the functions and variables provided by the library. See Section 3.2 [Type Definitions], page 9, for the basic type reference.

4.1 Variable Reference

4.1.1 Error Variables

The following variables are set by the library to return error information.

`int MikMod_errno`
: When an error occurs, this variable contains the error code. See Section 4.3 [Error Reference], page 21, for more information.

`BOOL MikMod_critical`
: When an error occurs, this variable informs of the severity of the error. Its value has sense only if the value of `MikMod_errno` is different from zero. If the value of `MikMod_critical` is zero, the error wasn't fatal and the library is in a stable state. However, if it is nonzero, then the library can't be used and has reseted itself to the uninitialized state. This often means that the mixing parameters you choose were not supported by the driver, or that it doesn't has enough voices for your needs if you called `MikMod_SetNumVoices`.

4.1.2 Sound Settings

The following variables control the sound output parameters and their changes take effect immediately.

`UBYTE md_musicvolume`
: Volume of the module. Allowed values range from 0 to 128. The default value is 128.

`UBYTE md_pansep`
: Stereo channels separation. Allowed values range from 0 (no separation, thus mono sound) to 128 (full channel separation). The default value is 128.

`UBYTE md_reverb`
: Amount of sound reverberation. Allowed values range from 0 (no reverberation) to 15 (a rough estimate for chaos...). The default value is 0.

`UBYTE md_sndfxvolume`
: Volume of the sound effects. Allowed values range from 0 to 128. The default value is 128.

`UBYTE md_volume`
: Overall sound volume. Allowed values range from 0 to 128. The default value is 128.

4.1.3 Driver Settings

The following variables control more in-depth sound output parameters. Except for some `md_mode` flags, their changes do not have any effect until you call `MikMod_Init` or `MikMod_Reset`.

`UWORD md_device`

This variable contains the order, in the list of the registered drivers, of the sound driver which will be used for sound playback. This order is one-based; if this variable is set to zero, the driver is autodetected, which means the list is tested until a driver is present on the system. The default value is 0, thus driver is autodetected.

`MDRIVER* md_driver`

This variable points to the driver which is being used for sound playback, and is undefined when the library is uninitialized (before `MikMod_Init` and after `MikMod_Exit`). This variable is for information only, you should never attempt to change its value. Use `md_driver` and `MikMod_Init` (or `MikMod_Reset`) instead.

`UWORD md_mixfreq`

Sound playback frequency, in hertz. High values yield high sound quality, but need more computing power than lower values. The default value is 44100 Hz, which is compact disc quality. Other common values are 22100 Hz (radio quality), 11025 Hz (phone quality), and 8000 Hz (mu-law quality).

`UWORD md_mode`

This variable is a combination of several flags, to select which output mode to select. The following flags have a direct action to the sound output (i.e. changes take effect immediately):

'`DMODE_INTERP`'

This flag, if set, enables the interpolated mixers. Interpolated mixing gives better sound but takes a bit more time than standard mixing. If the library is built with the high quality mixer, interpolated mixing is always enabled, regardless of this flag.

'`DMODE_REVERSE`'

This flag, if set, exchanges the left and right stereo channels.

'`DMODE_SURROUND`'

This flag, if set, enables the surround mixers. Since surround mixing works only for stereo sound, this flag has no effect if the sound playback is in mono.

The following flags aren't taken in account until the sound driver is changed or reset:

'`DMODE_16BIT`'

This flag, if set, selects 16 bit sound mode. This mode yields better sound quality, but needs twice more mixing time.

'`DMODE_HQMIXER`'

This flag, if set, selects the high-quality software mixer. This mode yields better sound quality, but needs more mixing time. Of course, this flag has no effect if no `DMODE_SOFT_xx` flag is set.

'`DMODE_SOFT_MUSIC`'

This flag, if set, selects software mixing of the module.

'DMODE_SOFT_SNDFX'
: This flag, if set, selects software mixing of the sound effects.

'DMODE_STEREO'
: This flag, if set, selects stereo sound.

The default value of this variable is 'DMODE_STEREO | DMODE_SURROUND | DMODE_16BITS | DMODE_SOFT_MUSIC | DMODE_SOFT_SNDFX'.

4.2 Structure Reference

Only the useful fields are described here; if a structure field is not described, you must assume that it's an internal field which must not be modified.

4.2.1 Drivers

The MDRIVER structure is not meant to be used by anything else than the core of the library, but its first four fields contain useful information for your programs:

CHAR* Name
: Name of the driver, usually never more than 20 characters.

CHAR* Description
: Description of the driver, usually never more than 50 characters.

UBYTE HardVoiceLimit
: Maximum number of hardware voices for this driver, 0 if the driver has no hardware mixing support.

UBYTE SoftVoiceLimit
: Maximum number of software voices for this driver, 0 if the driver has no software mixing support.

CHAR* Alias
: A short name for the driver, without spaces, usually never more than 10 characters.

4.2.2 Modules

The MODULE structure gathers all the necessary information needed to play a module file, regardless of its initial format.

4.2.2.1 General Module Information

The fields described in this section contain general information about the module and should not be modified.

CHAR* songname
: Name of the module.

CHAR* modtype
: Type of the module (which tracker format).

CHAR* comment
: Either the module comments, or NULL if the module doesn't have comments.

`UWORD flags`
: Several module flags or'ed together.

  '`UF_ARPMEM`'
  : If set, arpeggio effects have memory.

  '`UF_BGSLIDES`'
  : If set, volume slide effects continue until a new note or a new effect is played.

  '`UF_HIGHBPM`'
  : If set, the module is allowed to have its tempo value (bpm) over 255.

  '`UF_INST`'
  : If set, the module has instruments and samples; otherwise, the module has only samples.

  '`UF_LINEAR`'
  : If set, slide periods are linear; otherwise, they are logarithmic.

  '`UF_NNA`'
  : If set, module uses new note actions (NNA) and the `numvoices` field is valid.

  '`UF_NOWRAP`'
  : If set, pattern break on the last pattern does not continue to the first pattern.

  '`UF_S3MSLIDES`'
  : If set, module uses old-S3M style volume slides (slides processed every tick); otherwise, it uses the standard style (slides processed every tick except the first).

  '`UF_XMPERIODS`'
  : If set, module uses XM-type periods; otherwise, it uses Amiga periods.

  '`UF_FT2QUIRKS`'
  : If set, module player will reproduce some FastTracker 2 quirks during playback.

  '`UF_PANNING`'
  : If set, module use panning commands.

`UBYTE numchn`
: The number of channels in the module.

`UBYTE numvoices`
: If the module uses NNA, and this variable is not zero, it contains the limit of module voices; otherwise, the limit is set to the `maxchan` parameter of the `Player_Loadxx` functions.

`UWORD numpos`
: The number of sound positions in the module.

`UWORD numpat`
: The number of patterns.

`UWORD numins`
: The number of instruments.

`UWORD numsmp`
: The number of samples.

`INSTRUMENT* instruments`
: Points to an array of instrument structures.

`SAMPLE* samples`
: Points to an array of sample structures.

`UBYTE realchn`
: During playback, this variable contains the number of active channels (not counting NNA channels).

`UBYTE totalchn`
: During playback, this variable contains the total number of channels (including NNA channels).

`ULONG sngtime`
: Elapsed song time, in $2^{-}10$ seconds units (not exactly a millisecond). To convert this value to seconds, divide by 1024, not 1000 !

4.2.2.2 Playback Settings

The fields described here control the module playback and can be modified at any time, unless otherwise specified.

`UBYTE initspeed`
: The initial speed of the module (Protracker compatible). Valid range is 1-32.

`UBYTE inittempo`
: The initial tempo of the module (Protracker compatible). Valid range is 32-255.

`UBYTE initvolume`
: The initial overall volume of the module. Valid range is 0-128.

`UWORD panning[]`
: The current channel panning positions. Only the first `numchn` values are defined.

`UBYTE chanvol[]`
: The current channel volumes. Only the first `numchn` values are defined.

`UWORD bpm` The current tempo of the module. Use `Player_SetTempo` to change its value.

`UBYTE sngspd`
: The current speed of the module. Use `Player_SetSpeed` to change its value.

`UBYTE volume`
: The current overall volume of the module, in range 0-128. Use `Player_SetVolume` to change its value.

`BOOL extspd`
: If zero, Protracker extended speed effect (in-module tempo modification) is not processed. The default value is 1, which causes this effect to be processed. However, some old modules might not play correctly if this effect is not neutralized.

`BOOL panflag`
: If zero, panning effects are not processed. The default value is 1, which cause all panning effects to be processed. However, some old modules might not play correctly if panning is not neutralized.

`BOOL wrap` If nonzero, module wraps to its restart position when it is finished, to play continuously. Default value is zero (play only once).

`UBYTE reppos`
: The restart position of the module, when it wraps.

`BOOL loop` If nonzero, all in-module loops are processed; otherwise, backward loops which decrease the current position are not processed (i.e. only forward loops, and backward loops in the same pattern, are processed). This ensures that the module never loops endlessly. The default value is 1 (all loops are processed).

`BOOL fadeout`
: If nonzero, volume fades out during when last position of the module is being played. Default value us zero (no fadeout).

`UWORD patpos`
: Current position (row) in the pattern being played. Must not be changed.

`SWORD sngpos`
: Current song position. Do not change this variable directly, use `Player_NextPosition`, `Player_PrevPosition` or `Player_SetPosition` instead.

`SWORD relspd`
: Relative playback speed. The value of this variable is added to the module tempo to define the actual playback speed. The default value is 0, which make modules play at their intended speed.

4.2.3 Module Instruments

Although the `INSTRUMENT` structure is intended for internal use, you might need to know its name:

`CHAR* insname`
: The instrument text, theoretically its name, but often a message line.

4.2.4 Samples

The `SAMPLE` structure is used for sound effects and module samples as well. You can play with the following fields:

`SWORD panning`
: Panning value of the sample. Valid values range from PAN_LEFT (0) to PAN_RIGHT (255), or PAN_SURROUND.

`ULONG speed`
: Playing frequency of the sample, it hertz.

`UBYTE volume`
: Sample volume. Valid range is 0-64.

`UWORD flags`
: Several format flags or'ed together describing the format of the sample in memory.

  Format flags:

  '`SF_16BITS`'
  : If set, sample data is 16 bit wide; otherwise, it is 8 bit wide.

  '`SF_BIG_ENDIAN`'
  : If set, sample data is in big–endian (Motorola) format; otherwise, it is in little–endian (Intel) format.

  '`SF_DELTA`'
  : If set, sample is stored as delta values (differences between two consecutive samples); otherwise, sample is stored as sample values.

  '`SF_ITPACKED`'
  : If set, sample data is packed with Impulse Tracker's compression method; otherwise, sample is not packed.

  '`SF_SIGNED`'
  : If set, sample data is made of signed values; otherwise, it is made of unsigned values.

  '`SF_STEREO`'
  : If set, sample data is stereo (two channels); otherwise, it is mono.

  Playback flags:

  '`SF_BIDI`' If set, sample loops "ping pong" (back and forth).

  '`SF_LOOP`' If set, sample loops forward.

  '`SF_REVERSE`'
  : If set, sample plays backwards.

`UWORD inflags`
: Same as "flags", but describing the format of the sample on disk.

`ULONG length`
: Length of the sample, in *samples*. The length of a sample is 8 bits (1 byte) for a 8 bit sample, and 16 bits (2 bytes) for a 16 bit sample.

`ULONG loopstart`
: Loop starting position, relative to the start of the sample, in samples.

`ULONG loopend`
: Loop ending position, relative to the start of the sample, in samples.

4.2.5 MREADER

The `MREADER` contains the following function pointers:

`int (*Seek)(struct MREADER*, long offset, int whence)`
: This function should have the same behaviour as `fseek`, with offset 0 meaning the start of the object (module, sample) being loaded.

`long (*Tell)(struct MREADER*)`
: This function should have the same behaviour as `ftell`, with offset 0 meaning the start of the object being loaded.

`BOOL (*Read)(struct MREADER*, void *dest, size_t length)`
: This function should copy `length` bytes of data into `dest`, and return zero if an error occured, and any nonzero value otherwise. Note that an end-of-file condition will not be considered as an error in this case.

`int (*Get)(struct MREADER*)`
: This function should have the same behaviour as `fgetc`.

`BOOL (*Eof)(struct MREADER*)`
: This function should have the same behaviour as `feof`.

For an example of how to build an `MREADER` object, please refer to the `MFILEREADER` object in file `mmio/mmio.c` in the library sources.

4.2.6 MWRITER

The `MWRITER` contains the following function pointers:

`int (*Seek)(struct MWRITER*, long offset, int whence);`
: This function should have the same behaviour as `fseek`, with offset 0 meaning the start of the object being written.

`long (*Tell)(struct MWRITER*);`
: This function should have the same behaviour as `ftell`, with offset 0 meaning the start of the object being written.

`BOOL (*Write)(struct MWRITER*, void *src, size_t length);`
: This function should copy `length` bytes of data from `src`, and return zero if an error occured, and any nonzero value otherwise.

`int (*Put)(struct MWRITER*, int data);`
: This function should have the same behaviour as `fputc`.

For an example of how to build an `MWRITER` object, please refer to the `MFILEWRITER` object in file `mmio/mmio.c` in the library sources.

4.3 Error Reference

The following errors are currently defined:

4.3.1 General Errors

'`MMERR_DYNAMIC_LINKING`'
: This error occurs when a specific driver was requested, but the support shared library couldn't be loaded. Currently, the only drivers which can yield this error are the ALSA, EsounD and Ultra drivers.

'`MMERR_OPENING_FILE`'
: This error occurs when a file can not be opened, either for read access from a `xx_Loadxx` function, or for write access from the disk writer drivers.

'MMERR_OUT_OF_MEMORY'
: This error occurs when there is not enough virtual memory available to complete the operation, or there is enough memory but the calling process would exceed its memory limit. MikMod does not do any resource tuning, your program has to use the `setrlimit` function to do this if it needs to load very huge samples.

4.3.2 Sample Errors

'MMERR_SAMPLE_TOO_BIG'
: This error occurs when the memory allocation of the sample data yields the error MMERR_OUT_OF_MEMORY.

'MMERR_OUT_OF_HANDLES'
: This error occurs when your program reaches the limit of loaded samples, currently defined as 384, which should be sufficient for most cases.

'MMERR_UNKNOWN_WAVE_TYPE'
: This error occurs when you're trying to load a sample which format is not recognized.

4.3.3 Module Errors

'MMERR_ITPACK_INVALID_DATA'
: This error occurs when a compressed module sample is corrupt.

'MMERR_LOADING_HEADER'
: This error occurs when you're trying to load a module which has a corrupted header, or is truncated.

'MMERR_LOADING_PATTERN'
: This error occurs when you're trying to load a module which has corrupted pattern data, or is truncated.

'MMERR_LOADING_SAMPLEINFO'
: This error occurs when you're trying to load a module which has corrupted sample information, or is truncated.

'MMERR_LOADING_TRACK'
: This error occurs when you're trying to load a module which has corrupted track data, or is truncated.

'MMERR_MED_SYNTHSAMPLES'
: This error occurs when you're trying to load a MED module which has synthsounds samples, which are currently not supported.[1]

'MMERR_NOT_A_MODULE'
: This error occurs when you're trying to load a module which format is not recognized.

'MMERR_NOT_A_STREAM'
: This error occurs when you're trying to load a sample with a sample which format is not recognized.

[1] You can force libmikmod to load the module (without the synthsounds, of course) by setting the `curious` parameter to `1` when invoking `Player_Loadxx`.

4.3.4 Driver Errors

Generic driver errors

'MMERR_16BIT_ONLY'
: This error occurs when the sound device doesn't support non-16 bit linear sound output, which are the requested settings.

'MMERR_8BIT_ONLY'
: This error occurs when the sound device doesn't support non-8 bit linear sound output, which are the requested settings.

'MMERR_DETECTING_DEVICE'
: This error occurs when the driver's sound device has not been detected.

'MMERR_INITIALIZING_MIXER'
: This error occurs when MikMod's internal software mixer could not be initialized properly.

'MMERR_INVALID_DEVICE'
: This error occurs when the driver number (in `md_device`) is out of range.

'MMERR_NON_BLOCK'
: This error occurs when the driver is unable to set the audio device in non blocking mode.

'MMERR_OPENING_AUDIO'
: This error occurs when the driver can not open sound device.

'MMERR_STEREO_ONLY'
: This error occurs when the sound device doesn't support mono sound output, which is the requested setting.

'MMERR_ULAW'
: This error occurs when the sound device only supports uLaw output (which implies mono, 8 bit, and 8000 Hz sampling rate), which isn't the requested setting.

AudioFile driver specific error

'MMERR_AF_AUDIO_PORT'
: This error occurs when the AudioFile driver can not find a suitable AudioFile port.

AIX driver specific errors

'MMERR_AIX_CONFIG_CONTROL'
: This error occurs when the "Control" step of the device configuration has failed.

'MMERR_AIX_CONFIG_INIT'
: This error occurs when the "Init" step of the device configuration has failed.

'MMERR_AIX_CONFIG_START'
: This error occurs when the "Start" step of the device configuration has failed.

Ultra driver specific errors

'MMERR_GUS_RESET'
: This error occurs when the sound device couldn't be reset.

'MMERR_GUS_SETTINGS'
: This error occurs because the sound device only works in 16 bit linear stereo sound at 44100 Hz, which is not the requested settings.

'MMERR_GUS_TIMER'
: This error occurs when the ultra driver could not setup the playback timer.

HP-UX driver specific errors

'MMERR_HP_AUDIO_DESC'
: This error occurs when the HP driver can not get the audio hardware description.

'MMERR_HP_AUDIO_OUTPUT'
: This error occurs when the HP driver can not select the audio output.

'MMERR_HP_BUFFERSIZE'
: This error occurs when the HP driver can not set the transmission buffer size.

'MMERR_HP_CHANNELS'
: This error occurs when the HP driver can not set the requested number of channels.

'MMERR_HP_SETSAMPLESIZE'
: This error occurs when the HP driver can not set the requested sample size.

'MMERR_HP_SETSPEED'
: This error occurs when the HP driver can not set the requested sample rate.

ALSA driver specific errors

'MMERR_ALSA_NOCONFIG'
: This error occurs when no ALSA playback configuration is available.

'MMERR_ALSA_SETPARAMS'
: This error occurs when the ALSA driver can not set the requested sample format, sample rate, number of channels, access type, or latency values.

'MMERR_ALSA_SETFORMAT'
: This error occurs when the ALSA driver can not set the requested sample format.

'MMERR_ALSA_SETRATE'
: This error occurs when the ALSA driver does not support the requested sample rate.

'MMERR_ALSA_SETCHANNELS'
: This error occurs when the ALSA driver does not support the requested number of channels.

'MMERR_ALSA_BUFFERSIZE'
: This error occurs when the ALSA driver can not retrieve the buffer or period size.

'MMERR_ALSA_PCM_START'
: This error occurs when the ALSA driver can not start the pcm playback.

'MMERR_ALSA_PCM_WRITE'
: This error occurs when the ALSA driver encounters a write error.

'MMERR_ALSA_PCM_RECOVER'
: This error occurs when the ALSA driver encounters an error recovery failure.

Open Sound System driver specific errors

'MMERR_OSS_SETFRAGMENT'
: This error occurs when the OSS driver can not set audio fragment size.

'MMERR_OSS_SETSAMPLESIZE'
: This error occurs when the OSS driver can not set the requested sample size.

'MMERR_OSS_SETSPEED'
: This error occurs when the OSS driver can not set the requested sample rate.

'MMERR_OSS_SETSTEREO'
: This error occurs when the OSS driver can not set the requested number of channels.

SGI driver specific errors

'MMERR_SGI_MONO'
: This error occurs when the hardware only supports stereo sound.

'MMERR_SGI_SPEED'
: This error occurs when the hardware does not support the requested sample rate.

'MMERR_SGI_STEREO'
: This error occurs when the hardware only supports mono sound.

'MMERR_SGI_16BIT'
: This error occurs when the hardware only supports 16 bit sound.

'MMERR_SGI_8BIT'
: This error occurs when the hardware only supports 8 bit sound.

Sun driver specific error

'MMERR_SUN_INIT'
: This error occurs when the sound device initialization failed.

OS/2 driver specific errors

'MMERR_OS2_MIXSETUP'
: This error occurs when the DART driver can not set the mixing parameters.

'MMERR_OS2_SEMAPHORE'
: This error occurs when the MMPM/2 driver can not create the semaphores needed for playback.

'MMERR_OS2_THREAD'
: This error occurs when the MMPM/2 driver can not create the thread needed for playback.

'MMERR_OS2_TIMER'
: This error occurs when the MMPM/2 driver can not create the timer needed for playback.

DirectX Driver Specific Errors

'MMERR_DS_BUFFER'
: This error occurs when the DirectX driver can not allocate the playback buffers.

'MMERR_DS_EVENT'
: This error occurs when the DirectX driver can not register the playback event.

'MMERR_DS_FORMAT'
: This error occurs when the DirectX driver can not set the playback format.

'MMERR_DS_NOTIFY'
: This error occurs when the DirectX driver can not register the playback callback.

'MMERR_DS_PRIORITY'
: This error occurs when the DirectX driver can not set the playback priority.

'MMERR_DS_THREAD'
: This error occurs when the DirectX driver can not create the playback thread.

'MMERR_DS_UPDATE'
: This error occurs when the DirectX driver can not initialize the playback thread.

Windows Multimedia API Driver Specific Errors

'MMERR_WINMM_ALLOCATED'
: This error occurs when the playback resource is already allocated by another application.

'MMERR_WINMM_DEVICEID'
: This error occurs when the Multimedia API Driver is given an invalid audio device identificator.

'MMERR_WINMM_FORMAT'
: This error occurs when the playback output format is not supported by the audio device.

'MMERR_WINMM_HANDLE'
: This error occurs when the Multimedia API Driver is given an invalid handle.

'MMERR_WINMM_UNKNOWN'
: This error should not occur ! If you get this error, please contact the libmikmod development mailing list.

MacOS Driver Specific Errors

'MMERR_MAC_SPEED'
: This error occurs when the playback speed is not supported by the audio device.

'MMERR_MAC_START'
: This error occurs when the MacOS driver can not start playback.

4.4 Function Reference

4.4.1 Library Core Functions

`BOOL MikMod_Active(void)`

Description
: This function returns whether sound output is enabled or not.

Result

0 Sound output is disabled.

1 Sound output is enabled.

Notes Calls to `MikMod_Update` will be ignored when sound output is disabled.

See also `MikMod_DisableOutput`, `MikMod_EnableOutput`.

`void MikMod_DisableOutput(void)`

Description
: This function stops the sound mixing.

Notes Calls to `MikMod_Update` will be ignored when sound output is disabled.

See also `MikMod_Active`, `MikMod_EnableOutput`.

`int MikMod_EnableOutput(void)`

Description
: This function starts the sound mixing.

Result

0 Sound mixing is ready.

nonzero An error occurred during the operation.

Notes Calls to `MikMod_Update` will be ignored when sound output is disabled.

See also `MikMod_Active`, `MikMod_DisableOutput`.

`void MikMod_Exit(void)`

Description
: This function deinitializes the sound hardware and frees all the memory and resources used by MikMod.

See also `MikMod_Init`, `MikMod_Reset`.

`long MikMod_GetVersion(void)`

Description
: This function returns the version number of the library.

Result The version number, encoded as follows: `(maj<<16)|(min<<8)|(rev)`, where '`maj`' is the major version number, '`min`' is the minor version number, and '`rev`' is the revision number.

`CHAR* MikMod_InfoDriver(void)`

Description
This function returns a formatted list of the registered drivers in a buffer.

Result A pointer to a text buffer, or `NULL` if no drivers are registered.

Notes The buffer is created with `malloc`; the caller must free it when it is no longer necessary.

See also `MikMod_RegisterDriver`, `MikMod_RegisterAllDrivers`.

`CHAR* MikMod_InfoLoader(void)`

Description
This function returns a formatted list of the registered module loaders in a buffer.

Result A pointer to a text buffer, or `NULL` if no loaders are registered.

Notes The buffer is created with `malloc`; the caller must free it when it is no longer necessary.

See also `MikMod_RegisterLoader`, `MikMod_RegisterAllLoaders`.

`int MikMod_Init(CHAR *parameters)`

Description
This function performs the library initialization, including the sound driver choice and configuration, and all the necessary memory allocations.

Parameters
parameters
Optional parameters given to the sound driver. These parameters are ignored if the value of `md_device` is zero (autodetection).

Result
0 Initialization was successful.

nonzero An error occurred during initialization.

Notes When the initialization fails, the library uses the nosound sound driver to let other the other MikMod functions work without crashing the application.

See also `MikMod_Exit`, `MikMod_InitThreads`, `MikMod_Reset`.

`BOOL MikMod_InitThreads(void)`

Description
This function returns whether libmikmod is thread-safe.

Result
0 The library is not thread-safe.

1 The library is thread-safe.

Notes This function should be called before any call to `MikMod_Lock` or `MikMod_Unlock` is made.

See also `MikMod_Lock`, `MikMod_Unlock`.

`void MikMod_Lock(void)`

Description

This function obtains exclusive access to libmikmod's variables.

Notes This function locks an internal mutex. If the mutex is already locked, it will block the calling thread until the mutex is unlocked.
Every `MikMod_Unlock` call should be associated to a `MikMod_Lock` call. To be sure this is the case, we advise you to define and use the following macros:
`#define MIKMOD_LOCK MikMod_Lock();{`
`#define MIKMOD_UNLOCK }MikMod_Unlock();`
The function `MikMod_InitThreads` must have been invoked before any call to `MikMod_Lock` in made.

See also `MikMod_InitThreads`, `MikMod_Unlock`.

`void MikMod_RegisterAllDrivers(void)`

Description

This function registers all the available drivers.

See also `MikMod_InfoDriver`, `MikMod_RegisterDriver`.

`void MikMod_RegisterAllLoaders(void)`

Description

This function registers all the available module loaders.

See also `MikMod_InfoLoader`, `MikMod_RegisterLoader`.

`void MikMod_RegisterDriver(struct MDRIVER* newdriver)`

Description

This function adds the specified driver to the internal list of usable drivers.

Parameters

newdriver A pointer to the `MDRIVER` structure identifying the driver.

Notes It is safe to register the same driver several times, although it will not be duplicated in the list.
You should register all the drivers you need before calling `MikMod_Init`. If you want to register all the available drivers, use `MikMod_RegisterAllDrivers` instead.

See also `MikMod_InfoDriver`, `MikMod_RegisterAllDrivers`.

`MikMod_handler_t MikMod_RegisterErrorHandler(MikMod_handler_t newhandler)`

Description

This function selects the function which should be called in case of error.

Parameters

newhandler

The new error callback function.

Result The previous error callback function, or `NULL` if there was none.

Notes `MikMod_handler_t` is defined as `void(*function)(void)`, this means your error function has the following prototype: `void MyErrorHandler(void)`
The error callback function is called when errors are detected, but not always immediately (the library has to resume to a stable state before calling your callback).

`void MikMod_RegisterLoader(struct MLOADER* newloader)`

Description
This function adds the specified module loader to the internal list of usable module loaders.

Parameters
newloader A pointer to the `MLOADER` structure identifying the loader.

Notes It is safe to register the same loader several times, although it will not be duplicated in the list.
You should register all the loaders you need before calling `Player_Load` or `Player_LoadFP`. If you want to register all the available module loaders, use `MikMod_RegisterAllLoaders` instead.
The 15 instrument module loader (`load_m15`) should always be registered last.

See also `MikMod_InfoLoader`, `MikMod_RegisterAllLoaders`.

`MikMod_player_t MikMod_RegisterPlayer(MikMod_player_t newplayer)`

Description
This function selects the function which should be used to process module playback.

Parameters
newplayer The new playback function

Result The previous playback function.

Notes `MikMod_player_t` is defined as `void(*function)(void)`, this means your player function has the following prototype: `void MyPlayer(void)`
The player function is called every module tick to process module playback. The rate at which the player function is called is controlled by the sound driver, and is computed by the following equation:
$(bpm * 50)/125$ calls per second, which means every $125000/(bpm * 50)$ milliseconds. The `bpm` value is the tempo of the module and can change from its initial value when requested by the module.
When changing the playback function, you should make sure that you chain to the default MikMod playback function, otherwise you won't get module sound anymore...

Example

```
MikMod_player_t oldroutine;

void MyPlayer(void)
{
```

```
    oldroutine();
    /* your stuff here */
    ...
}

main()
{
    ...
    /* Register our player */
    oldroutine = MikMod_RegisterPlayer(MyPlayer);
    ...
}
```

`int MikMod_Reset(CHAR *parameters)`

Description

This function resets MikMod and reinitialize the sound hardware.

Parameters

parameters

Optional parameters given to the sound driver. If you set the value of `md_device` to zero (autodetect), these parameters are ignored.

Result

0 Reinitialization was successful.

nonzero An error occurred during reinitialization.

Notes Use this function when you have changed the global configuration variables: `md_device` and `md_mixfreq`, or one of the `md_mode` flags which require sound reinitialization. Sound playback will continue as soon as the driver is ready.

See also `MikMod_Exit`, `MikMod_Init`.

`int MikMod_SetNumVoices(int musicvoices, int samplevoices)`

Description

This function sets the number of mixed voices which can be used for music and sound effects playback.

Parameters

musicvoices

The number of voices to reserve for music playback.

samplevoices

The number of voices to reserve for sound effects.

Result

0 Initialization was successful.

nonzero An error occurred during initialization.

Notes A value of `-1` for any of the parameters will retain the current number of reserved voices.

The maximum number of voices vary from driver to driver (hardware drivers often have a limit of 32 to 64 voices, whereas the software drivers handle 255 voices). If your settings exceed the driver's limit, they will be truncated.

See also `MikMod_Init`, `MikMod_Reset`.

`void MikMod_Unlock(void)`

Description

This function relinquishes exclusive access to libmikmod's variables.

Notes This function unlocks an internal mutex, so that other threads waiting for the lock can be resumed.
Every `MikMod_Unlock` call should be associated to a `MikMod_Lock` call. To be sure this is the case, we advise you to define and use the following macros:
`#define MIKMOD_LOCK MikMod_Lock();{`
`#define MIKMOD_UNLOCK }MikMod_Unlock();`
The function `MikMod_InitThreads` must have been invoked before any call to `MikMod_Unlock` in made.

See also `MikMod_InitThreads`, `MikMod_Lock`.

`void MikMod_Update(void)`

Description

This routine should be called on a regular basis to update the sound.

Notes The sound output buffer is filled each time this function is called; if you use a large buffer, you don't need to call this routine as frequently as with a smaller buffer, but you get a bigger shift between the sound being played and the reported state of the player, since the player is always a buffer ahead of the playback.
If you play low quality sound (for example, mono 8 bit 11 kHz sound), you only need to call this routine a few times per second. However, for high quality sound (stereo 16 bit 44 kHz), this rate increases to a few hundred times per second, but never more, due to the minimal buffer size constraint imposed to the sound drivers.
If you plan on modifying voice information with the `Voice_xx` functions, you should do this before calling `MikMod_Update`.

`char* MikMod_strerror(int errnum)`

Description

This function associates a descriptive message to an error code.

Parameters

errnum The MikMod error code.

Result A pointer to a string describing the error.

4.4.2 Module Player Functions

`BOOL Player_Active(void)`

Description

This function returns whether the module player is active or not.

Result

0 No module is playing.

nonzero A module is currently playing.

Notes This function will still report that the player is active if the playing module is paused.

See also `Player_Paused`, `Player_TogglePause`, `Player_Start`, `Player_Stop`

`void Player_Free(MODULE* module)`

Description
This function stops the module if it is playing and unloads it from memory.

Parameters
module The module to free.

See also `Player_Load`, `Player_LoadFP`.

`int Player_GetChannelVoice(UBYTE channel)`

Description
This function determines the voice corresponding to the specified module channel.

Parameters
channel The module channel to use.

Result The number of the voice corresponding to the module channel.

Notes If the module channel has NNAs, the number will correspond to the main channel voice.

See also `Voice_SetPanning`, `Voice_SetVolume`, `Player_Mute`, `Player_ToggleMute`, `Player_Unmute`.

`MODULE* Player_GetModule(void)`

Description
This function returns the module currently being played.

Result A pointer to the `MODULE` being played, or `NULL` if no module is playing.

See also `Player_Stop`, `Player_Start`.

`MODULE* Player_Load(CHAR* filename, int maxchan, BOOL curious)`

Description
This function loads a music module.

Parameters
filename The name of the module file.

maxchan The maximum number of channels the song is allowed to request from the mixer.

curious The curiosity level to use.

Result A pointer to a `MODULE` structure, or `NULL` if an error occurs.

Notes If the curiosity level is set to zero, the module will be loaded normally. However, if it is nonzero, the following things occur:

- pattern positions occurring after the "end of song" marker in S3M and IT modules are loaded, and the end of song is set to the last position.
- hidden extra patterns are searched in MOD modules, and if found, played after the last "official" pattern.
- MED modules with synthsounds are loaded without causing the `MMERR_MED_SYNTHSAMPLES`, and synthsounds are mapped to an empty sample.

See also `Player_Free`, `Player_LoadFP`, `Player_LoadTitle`, `Player_LoadTitleFP`, `Player_Start`.

`MODULE* Player_LoadFP(FILE* file, int maxchan, BOOL curious)`

Description

This function loads a music module.

Parameters

file An open file, at the position where the module starts.

maxchan The maximum number of channels the song is allowed to request from the mixer.

curious The curiosity level to use.

Result A pointer to a `MODULE` structure, or `NULL` if an error occurs.

Notes The file is left open, at the same position as before the function call.
If the curiosity level is set to zero, the module will be loaded normally. However, if it is nonzero, the following things occur:

- pattern positions occurring after the "end of song" marker in S3M and IT modules are loaded, and the end of song is set to the last position.
- hidden extra patterns are searched in MOD modules, and if found, played after the last "official" pattern.
- MED modules with synthsounds are loaded without causing the `MMERR_MED_SYNTHSAMPLES`, and synthsounds are mapped to an empty sample.

See also `Player_Free`, `Player_Load`, `Player_LoadTitle`, `Player_LoadTitleFP`, `Player_Start`.

`MODULE* Player_LoadTitle(CHAR* filename)`

Description

This function retrieves the title of a module file.

Parameters

filename The name of the module file.

Result A pointer to the song title, or `NULL` if either the module has no title or an error has occurred.

Notes The title buffer is created with `malloc`; the caller must free it when it is no longer necessary.

See also `Player_Load`, `Player_LoadFP`, `Player_LoadTitleFP`.

`MODULE* Player_LoadTitleFP(FILE* file)`

Description
This function retrieves the title of a module file.

Parameters
file An open file, at the position where the module starts.

Result A pointer to the song title, or `NULL` if either the module has no title or an error has occurred.

Notes The title buffer is created with `malloc`; the caller must free it when it is no longer necessary.

See also `Player_Load`, `Player_LoadFP`, `Player_LoadTitle`.

`void Player_Mute(SLONG operation, ...)`

Description
This function mutes a single module channel, or a range of module channels.

Parameters
operation Either the number of a module channel to mute (starting from zero), or an operation code. In the latter case, two extra parameters are needed to determine the range of channels.

Notes If the operation is `MUTE_INCLUSIVE`, the two channel numbers delimit the range and are part of the range ; otherwise, if the operation is `MUTE_EXCLUSIVE`, they are outside of the range.

Example

```
/* mute channel 10 */
Player_Mute(10);
/* mute channels 2 to 5 */
Player_Mute(MUTE_INCLUSIVE, 2, 5);
/* mute channels 7 to 9 */
Player_Mute(MUTE_EXCLUSIVE, 6, 10);
```

See also `Player_Muted`, `Player_ToggleMute`, `Player_Unmute`.

`BOOL Player_Muted(UBYTE channel)`

Description
This function determines whether a module channel is muted or not.

Parameters
channel The module channel to test (starting from zero).

Result
0 The channel is not muted.

1 The channel is muted.

See also `Player_Mute`, `Player_ToggleMute`, `Player_Unmute`.

`void Player_NextPosition(void)`

Description
This function jumps to the next position in the module.

Notes All playing samples and active song voices are cut to avoid hanging notes.

See also `Player_PrevPosition`, `Player_SetPosition`.

`BOOL Player_Paused(void)`

Description
This function determines whether the module is paused or not.

Result
0 The module is not paused.

1 The module is paused.

See also `Player_TogglePause`.

`void Player_PrevPosition(void)`

Description
This function jumps to the previous position in the module.

Notes All playing samples and active song voices are cut to avoid hanging notes.

See also `Player_NextPosition`, `Player_SetPosition`.

`void Player_SetPosition(UWORD position)`

Description
This function jumps to the specified position in the module.

Parameters
position The pattern position to jump to.

Notes All playing samples and active song voices are cut to avoid hanging notes.

See also `Player_NextPosition`, `Player_PrevPosition`.

`void Player_SetSpeed(UWORD speed)`

Description
This function sets the module speed.

Parameters
speed The new module speed, in the range 1-32.

See also `Player_SetTempo`.

`void Player_SetTempo(UWORD tempo)`

Description
This function sets the module tempo.

Parameters
tempo The new module tempo, in the range 32-255.

See also `Player_SetSpeed`.

`void Player_SetVolume(SWORD volume)`

Description
This function sets the module volume.

Parameters

volume The new overall module playback volume, in the range 0-128.

`void Player_Start(MODULE* module)`

Description

This function starts the specified module playback.

Parameters

module The module to play.

Notes If another module is playing, it will be stopped and the new module will play.

See also `Player_Stop`.

`void Player_Stop(void)`

Description

This function stops the currently playing module.

See also `Player_Start`.

`void Player_ToggleMute(SLONG operation, ...)`

Description

This function changes the muted status of a single module channel, or a range of module channels.

Parameters

operation Either the number of a module channel to work on (starting from zero), or an operation code. In the latter case, two extra parameters are needed to determine the range of channels.

Notes If the operation is `MUTE_INCLUSIVE`, the two channel numbers delimit the range and are part of the range ; otherwise, if the operation is `MUTE_EXCLUSIVE`, they are outside of the range.

Example

```
/* toggle mute on channel 10 */
Player_ToggleMute(10);
/* toggle mute on channels 2 to 5 */
Player_ToggleMute(MUTE_INCLUSIVE, 2, 5);
/* toggle mute on channels 7 to 9 */
Player_ToggleMute(MUTE_EXCLUSIVE, 6, 10);
```

See also `Player_Mute`, `Player_Muted`, `Player_Unmute`.

`void Player_TogglePause(void)`

Description

This function toggles the playing/paused status of the module.

Notes Calls to `Player_xx` functions still have effect when the module is paused.

See also `Player_Paused`, `Player_Start`, `Player_Stop`.

`void Player_Unmute(SLONG operation, ...)`

Description

This function unmutes a single module channel, or a range of module channels.

Parameters

operation Either the number of a module channel to unmute (starting from zero), or an operation code. In the latter case, two extra parameters are needed to determine the range of channels.

Notes If the operation is `MUTE_INCLUSIVE`, the two channel numbers delimit the range and are part of the range ; otherwise, if the operation is `MUTE_EXCLUSIVE`, they are outside of the range.

Example

```
/* unmute channel 10 */
Player_Unmute(10);
/* unmute channels 2 to 5 */
Player_Unmute(MUTE_INCLUSIVE, 2, 5);
/* unmute channels 7 to 9 */
Player_Unmute(MUTE_EXCLUSIVE, 6, 10);
```

See also `Player_Mute`, `Player_Muted`, `Player_ToggleMute`.

4.4.3 Sample Functions

`void Sample_Free(SAMPLE* sample)`

Description

This function unloads a sample from memory.

Parameters

sample The sample to free.

See also `Sample_Load`, `Sample_LoadFP`.

`SAMPLE* Sample_Load(CHAR* filename)`

Description

This function loads a sample.

Parameters

filename The sample filename.

Result A pointer to a `SAMPLE` structure, or `NULL` if an error has occurred.

See also `Sample_Free`, `Sample_LoadFP`.

`SAMPLE* Sample_LoadFP(FILE* file)`

Description

This function loads a sample.

Parameters

file An open file, at the position where the sample starts.

Result A pointer to a `SAMPLE` structure, or `NULL` if an error has occurred.

Notes The file is left open, at the same position as before the function call.

See also `Sample_Free`, `Sample_Load`.

`SBYTE Sample_Play(SAMPLE* sample, ULONG start, UBYTE flags)`

Description
This function plays a sample as a sound effect.

Parameters
sample The sample to play.

start The starting position (in samples).

flags Either zero, for normal sound effects, or `SFX_CRITICAL`, for critical sound effects which must not be interrupted.

Result The voice number corresponding to the voice which will play the sample.

Notes Each new sound effect is played on a new voice. When all voices are taken, the oldest sample which was not marked as critical is cut and its voice is used for the new sample. Critical samples are not cut unless all the voices are taken with critical samples and you attempt to play yet another critical sample. Use `Voice_Stop` to force the end of a critical sample.

See also `MikMod_SetNumVoices`, `Voice_Play`, `Voice_SetFrequency`, `Voice_SetPanning`, `Voice_SetVolume`, `Voice_Stop`.

4.4.4 Voice Functions

`ULONG Voice_GetFrequency(SBYTE voice)`

Description
This function returns the frequency of the sample currently playing on the specified voice.

Parameters
voice The number of the voice to get frequency.

Result The current frequency of the sample playing on the specified voice, or zero if no sample is currently playing on the voice.

See also `Voice_SetFrequency`.

`ULONG Voice_GetPanning(SBYTE voice)`

Description
This function returns the panning position of the sample currently playing on the specified voice.

Parameters
voice The number of the voice to get panning position.

Result The current panning position of the sample playing on the specified voice, or `PAN_CENTER` if no sample is currently playing on the voice.

See also `Voice_SetPanning`.

`SLONG Voice_GetPosition(SBYTE voice)`

Description
This function returns the sample position (in samples) of the sample currently playing on the specified voice.

Parameters

voice The number of the voice to get sample position (starting from zero).

Result The current play location of the sample playing on the specified voice, or zero if the position can not be determined or if no sample is currently playing on the voice.

Notes This function may not work with some drivers (especially for hardware mixed voices). In this case, it returns always `-1`.

See also `Sample_Play`, `Voice_Play`.

```
UWORD Voice_GetVolume(SBYTE voice)
```

Description

This function returns the volume of the sample currently playing on the specified voice.

Parameters

voice The number of the voice to get volume.

Result The current volume of the sample playing on the specified voice, or zero if no sample is currently playing on the voice.

See also `Voice_RealVolume`, `Voice_SetVolume`.

```
void Voice_Play(SBYTE voice, SAMPLE* sample, ULONG start)
```

Description

Start a new sample in the specified voice.

Parameters

voice The number of the voice to be processed (starting from zero).

sample The sample to play.

start The starting position (in samples).

Notes The sample will be played at the volume, panning and frequency settings of the voice, regardless or the sample characteristics.
The sample played this way gets the same "critical" status as the sample which was previously played on this voice.

See also `Sample_Play`, `Voice_SetFrequency`, `Voice_SetPanning`, `Voice_SetVolume`.

```
ULONG Voice_RealVolume(SBYTE voice)
```

Description

This function returns the actual playing volume of the specified voice.

Parameters

voice The number of the voice to analyze (starting from zero).

Result The real volume of the voice when the function was called, in the range 0-65535, not related to the volume constraint specified with `Voice_SetVolume`.

Notes This function may not work with some drivers (especially for hardware mixed voices). In this case, it always returns zero.
Also note that the real volume computation is not a trivial process and takes some CPU time.

See also `Sample_Play`, `Voice_GetVolume`, `Voice_Play`, `Voice_SetVolume`.

`void Voice_SetFrequency(SBYTE voice, ULONG frequency)`

Description
This function sets the frequency (pitch) of the specified voice.

Parameters
voice The number of the voice to be processed (starting from zero).

frequency The new frequency of the voice, in hertz.

See also `Sample_Play`, `Voice_GetFrequency`, `Voice_Play`, `Voice_SetPanning`, `Voice_SetVolume`, `Voice_Stop`.

`void Voice_SetPanning(SBYTE voice, ULONG panning)`

Description
This function sets the panning position of the specified voice.

Parameters
voice The number of the voice to be processed (starting from zero).

panning The new panning position of the voice.

Notes Panning can vary between 0 (`PAN_LEFT`) and 255 (`PAN_RIGHT`). Center is 127 (`PAN_CENTER`. Surround sound can be enabled by specifying the special value `PAN_SURROUND`.

See also `Sample_Play`, `Voice_GetPanning`, `Voice_Play`, `Voice_SetFrequency`, `Voice_SetVolume`, `Voice_Stop`.

`void Voice_SetVolume(SBYTE voice, UWORD volume)`

Description
This function sets the volume of the specified voice.

Parameters
voice The number of the voice to be processed (starting from zero).

volume The new volume of the voice, in the range 0-256.

See also `Sample_Play`, `Voice_GetVolume`, `Voice_Play`, `Voice_SetFrequency`, `Voice_SetPanning`, `Voice_Stop`.

`void Voice_Stop(SBYTE voice)`

Description
This function stops the playing sample of the specified voice.

Parameters
voice The number of the voice to be processed (starting from zero).

Notes After the call to `Voice_Stop`, the function `Voice_Stopped` will return nonzero (true) for the voice. If you want to silence the voice without stopping the playback, use `Voice_SetVolume(voice, 0)` instead.

See also `Sample_Play`, `Voice_Play`, `Voice_SetFrequency`, `Voice_SetPanning`, `Voice_SetVolume`.

```
BOOL Voice_Stopped(SBYTE voice)
```

Description
This function returns whether the voice is active or not.

Parameters
voice The number of the voice to be checked (starting from zero).

Result
0 The voice is stopped or has no sample assigned.

nonzero The voice is playing a sample.

Notes This function may not work with some drivers (especially for hardware mixed voices). In this case, its return value is undefined.

See also `Voice_Stop`.

4.5 Loader Reference

4.5.1 Module Loaders

MikMod presents a large choice of module loaders, for the most common formats as well as for some less-known exotic formats.

`load_669` This loader recognizes "Composer 669" and "Unis 669" modules. The 669 and "Extended 669" formats were among the first PC module formats. They do not have a wide range of effects, but support up to 32 channels.
"Composer 669" was written by Tran of Renaissance, a.k.a. Tomasz Pytel and released in 1992. "Unis 669 Composer" was written by Jason Nunn and released in 1994.

`load_amf` This loader recognizes the "Advanced Module Format", which is the internal module format of the "DOS Sound and Music Interface" (DSMI) library. This format has the same limitations as the S3M format. The most famous DSMI application was DMP, the Dual Module Player.
DMP and the DSMI library were written by Otto Chrons. DSMI was first released in 1993.

`load_dsm` This loader recognizes the internal DSIK format, which is the internal module format of the "Digital Sound Interface Kit" (DSIK) library, the ancester of the SEAL library. This format has the same limitations as the S3M format.
The DSIK library was written by Carlos Hasan and released in 1994.

`load_far` This loader recognizes "Farandole" modules. These modules can be up to 16 channels and have Protracker comparable effects.
The Farandole composer was written by Daniel Potter and released in 1994.

`load_gdm` This loader recognizes the "General DigiMusic" format, which is the internal format of the "Bells, Whistles and Sound Boards" library. This format has the same limitations as the S3M format.
The BWSB library was written by Edward Schlunder and first released in 1993.

`load_imf` This loader recognizes "Imago Orpheus" modules. This format is roughly equivalent to the XM format, but with two effects columns instead of a volume column and an effect column.
Imago Orpheus was written by Lutz Roeder and released in 1994.

`load_it` This loader recognizes "Impulse Tracker" modules, currently the most powerful format. These modules support up to 64 real channels, and up to 256 virtual channels with the "New Note Action" feature. Besides, it has the widest range of effects, and supports 16 bit samples as well as surround sound.
"Impulse Tracker" was written by Jeffrey Lim and released in 1996.

`load_med` This loader recognizes "OctaMED" modules. These modules are comparable to Protracker modules, but can embed "synthsounds", which are midi-like instruments.
"MED" and later "OctaMED" were written by Teijo Kinnunen. "MED" was released in 1989, and "OctaMED" was released in 1992.

`load_m15` This loader recognizes the old 15 instrument modules, created by "Ultimate Soundtracker", "Soundtracker" and the first versions of "Protracker".
Since this format was one of the first module formats, developed in 1987, it does not have any signature field, which makes it hard to detect reliably, because of its similarities with later module formats.

`load_mod` This loader recognizes the standard 31 instrument modules, created by "Protracker" or Protracker-compatible programs. The original Protracker format was limited to 4 channels, but other trackers like "TakeTracker", "StarTracker" or "Oktalyzer" afforded more channels.
Although it is now technically obsolete, this format is still widely used, due to its playback simplicity (on the adequate hardware, the Amiga).

`load_mtm` This loader recognizes the "MultiTracker Module Editor" modules. The MTM format has up to 32 channels, and protracker comparable effects. It was intended to replace "Composer 669". The "MultiTracker Module Editor" was written by Starscream of Renaissance, a.k.a. Daniel Goldstein and released in late 1993.

`load_okt` This loader recognizes the "Amiga Oktalyzer" modules. The OKT format has up to 8 channels, and a few protracker compatible effects, as well as other OKT-specific effects, of which only a few are currently supported by libmikmod.
"Oktalyzer" was written by Armin Sander and released in 1990.

`load_stm` This loader recognizes "ScreamTracker" modules. "ScreamTracker" was the first PC tracker, as well as the first PC module format. Loosely inspired by the "SoundTracker" format, it does not have as many effects as Protracker, although it supports 31 instruments and 4 channels.
"ScreamTracker" was written by PSI of Future Crew, a.k.a. Sami Tammilehto.

`load_stx` This loader recognizes "STMIK 0.2" modules. "STMIK" (the Scream Tracker Music Interface Kit) was a module playing library distributed by Future Crew to play Scream Tracker module in games and demos. It uses an intermediate format between STM and S3M and comes with a tool converting STM modules

to STX.
"STMIK" was written by PSI of Future Crew, a.k.a. Sami Tammilehto.

`load_s3m` This loader recognizes "ScreamTracker 3" modules. This version was a huge improvement over the original "ScreamTracker". It supported 32 channels, up to 99 instruments, and a large choice of effects.
"ScreamTracker 3" was written by PSI of Future Crew, a.k.a. Sami Tammilehto, and released in 1994.

`load_ult` This loader recognizes "UltraTracker" modules. They are mostly similar to Protracker modules, but support two effects per channel.
"UltraTracker" was written by MAS of Prophecy, a.k.a. Marc Andre Schallehn, and released in 1993.

`load_uni` This loader recognizes "UNIMOD" modules. This is the internal format used by MikMod and APlayer. Use of this format is discouraged, this loader being provided for completeness.

`load_xm` This loader recognizes "FastTracker 2" modules. This format was designed from scratch, instead of creating yet another Protracker variation. It was the first format using instruments as well as samples, and envelopes for finer effects.
FastTracker 2 was written by Fredrik Huss and Magnus Hogdahl, and released in 1994.

4.5.2 Sample Loaders

Currently, the only file type than can be loaded as a sample is the RIFF WAVE file. Stereo or compressed WAVE files are not supported yet.

4.6 Driver Reference

4.6.1 Network Drivers

These drivers send the generated sound over the network to a server program, which sends the sound to the real sound hardware. The server program can be on the same machine than your program, but MikMod does not have access to the hardware. Network drivers only support software mixing.

`drv_AF` This driver works with the "Digital AudioFile" library.
Start the server on the machine you want, set its hostname in the '`AUDIOFILE`' environment variable, and MikMod is ready to send it sound.

`drv_esd` This driver works with the "Enlightened Sound Daemon".
Start the esd daemon on the machine you want, set its hostname in the '`ESPEAKER`' environment variable, and MikMod is ready to send it sound.

4.6.2 Hardware Drivers

These drivers access to the sound hardware of the machine they run on. Depending on your Unix flavor, you'll end with one or more drivers from this list:

`drv_aix` This driver is only available under AIX, and access its audio device.
This driver only supports software mixing.

`drv_alsa` This driver is only available under Linux, and requires the ALSA driver to be compiled for your current kernel.
This driver only supports software mixing, but a future version of the driver might be able to use the hardware capabilities of some sound cards.

`drv_dart` This driver is only available under OS/2 version 3 and higher (Warp), and uses the "Direct Audio Real-Time" interface.
This driver only supports software mixing.

`drv_ds` This driver is only available under WIN32, and uses the "DirectSound" api.
This driver only supports software mixing.

`drv_hp` This driver is only available under HP-UX, and access its audio device.
This driver only supports software mixing.

`drv_os2` This driver is only available under OS/2 version 3 and higher (Warp), and OS/2 2.x with MMPM/2.
This driver only supports software mixing.

`drv_oss` This driver is available under any Unix with the Open Sound System drivers installed. Linux and FreeBSD also come with the OSS/Lite driver (the non-commercial version of OSS) and can make use of this driver.
This driver only supports software mixing.

`drv_sam9407`
This driver is only available under Linux, and requires the Linux sam9407 driver to be compiled for your current kernel.
This driver only supports hardware mixing.

`drv_sgi` This driver is only available under IRIX, and uses the SGI audio library.
This driver only supports software mixing.

`drv_sun` This driver is only available under Unices which implement SunOS-like audio device interfaces, that is, SunOS, Solaris, NetBSD and OpenBSD.
This driver only supports software mixing.

`drv_ultra`
This driver is only available under Linux, and requires the Linux Ultrasound driver (the ancestor of ALSA) to be compiled for your current kernel.
This driver only supports hardware mixing.

`drv_win` This driver is only available under WIN32, and uses the "multimedia API".
This driver only supports software mixing.

4.6.3 Disk Writer Drivers

These drivers work on any machine, since the generated sound is not sent to hardware, but written in a file. Disk writer drivers only support software mixing.

`drv_raw` This driver outputs the sound data in a file by default named '`music.raw`' in the current directory. The file has no header and only contains the sound output.

`drv_wav` This driver outputs the sound data in a RIFF WAVE file by default named '`music.wav`' in the current directory.

4.6.4 Other Drivers

These drivers are of little interest, but are handy sometimes.

`drv_stdout`
: This driver outputs the sound data to the program's standard output. To avoid inconvenience, the data will not be output if the standard output is a terminal, thus you have to pipe it through another command or to redirect it to a file. Using this driver and redirecting to a file is equivalent to using the `drv_raw` disk writer.
This driver only supports software mixing.

`drv_pipe`
: This driver pipes the sound data to a command (which must be given in the driver commandline, via `MikMod_Init`).
This driver only supports software mixing.

`drv_nos`
: This driver doesn't produce sound at all, and will work on any machine.
Since it does not have to produce sound, it supports both hardware and software mixing, with as many hardware voices as you like.

Function Index

M

P

S

V

Type and Variable Index

Table of Contents