DS Sound Engine v1.08

Print this manual for your own convenience and make sure you read it completely.
If you have read it before, make sure you at least read the 'History' section to see what has changed.

Introducing: DS Sound Engine v1.08

First of all we would like to thank you for using 'Engine Software's DS Sound Engine' for your projects! The DS Sound Engine consists of a *conversion utility*, a *stereo music replayer* (based on the XM format) and a *sound effects player* (based on the WAV PCM format). The conversion utility, called XM2DS, converts and optimizes the XM files and WAV files and packs them all together into a binary file to be used in your projects.

XM is a well known format and there are many freeware trackers available on various platforms which support this format. To name a few:

- MadTracker 2 (Windows)
- FastTracker 2 (DOS)

As many trackers support other formats besides XM (like: MOD, IT, S3M) it is fairly easy to use other formats as well; simply open them in your favorite tracker and save them as an XM file.

XM Restrictions

Currently the replayer has the following restrictions:
(missing features that are marked blue will be added to the engine sooner or later)

Instruments
- The first sample from the wavetable will be used for the whole key-range, so use one sample per instrument.
- Vibrato depth/rate/sweep not supported (alternative: use effect column's vibrato).

Samples
- PingPong looping is not supported (alternative: duplicate sample and mirror second half).

Volume Column
- Set vibrato speed/vibrato not supported (alternative: use effect column's vibrato).
- Tone Portamento not supported (alternative: use effect column's portamento).

Effects Column
- Some extended effects not supported: E3, E4, E6, E7, E9.
- Effect T (tremor) not supported.

Also keep in mind that the maximum number of available channels on the Nintendo DS is **16** (14 when you want to use post-processing effects).

☑ Make sure your composer knows these restrictions!

WAV Restrictions

XM2DS can only convert PCM waves, so WAV files which contain ADPCM data (or any other type which is not PCM) will result in an error. The frequency of the WAV file can be anything, but remember that using a samplerate higher than 32 KHz is overkill on the DS hardware.

Note: although you are not allowed to use ADPCM in the source WAV files, XM2DS will generate ADPCM samples for you if the quality level permits.

☑ Make sure your audio engineer knows these restrictions!

How to get Started

In the archive file you will find the following files:

All versions:

- **Manual.pdf**
  You're reading it right now… Read it 'till the end!
- **Example subdirectory**
  Contains simple, straightforward example sourcecode which sets up the engine and plays a song. Press keys to trigger sound effects.
- **DirectSound.h**
  This is the header you have to include in your project whenever you access the API. It contains the declaration of the interface functions and structures. There are also some defines and enumerations you can use.

Basic version:

- **LibDirectSound.final_size.a[.thumb]**
  The (C compatible) object data of the sound engine. Compiled as ARM or Thumb code (depending on the extension) and optimized for size. Link it together with the other objects in your project.
- **LibDirectSound.safe_size.a[.thumb]**
  Same, but compiled with 'SND_SAFE_MODE' enabled (outputs errors and warnings to the debug console).
- **LibDirectSound.final_speed.a[.thumb]**
  Similar to final_size, but optimized for speed instead of size.
- **LibDirectSound.safe_speed.a[.thumb]**
  Same, but compiled with 'SND_SAFE_MODE' enabled (outputs errors and warnings to the debug console).

Full version:

- **DirectSound.c / DirectSoundEx.h**
  The complete source code and an extra header. Gives complete control and allows you to configure and profile the sound engine to suit your project even more.

```
// (make sure you include the library or source code mentioned above in your project)
#include <nitro.h>
#include "DirectSound.h"

// symbolic constants generated by XM2DS
#include "data/sound/music.h"
#include "data/sound/sfx.h"

void NitroMain()
{
   // initialize NitroSDK
   OS_Init();
   FS_Init(2);
   // etc. (create a heap!)

   // initialize the sound system
   snd_sys_init(0, NULL, NULL);

   // initialize the song system by specifying the songset you created with XM2DS
   snd_sng_init_s("/sound/music.bin");

   // initialize the sound effect system by specifying the sfxset you created with XM2DS
   // as well as the amount of channels you want to reserve for sfx playback
   snd_sfx_init_s(4, "/sound/sfx.bin");

   // ready to make some noise!
   snd_sng_play(SNG_TITLE_SCORE);
   snd_sfx_play(SFX_WELCOME, 64, SND_PRI_NORMAL);

   // [insert complicated code here]

   // thanks for using our engine!
   snd_sys_exit();
}
```

Interface

Introduction

The sound engine's functions have been divided into several sub-classes. All interface functions have a '`snd_`' prefix. After that, a 3 character sub-class name - plus underscore - will follow:

`sys_` System related functions
`mix_` Mixer related functions (on the DS these encapsulate low-level ARM7 sound commands)
`sng_` Song related functions
`sfx_` Sound effect related functions

The most interesting functions will be described first. After that, a description of the more advanced functions will follow.

Commonly Used Interface Functions

With the following functions you can produce all sounds required. Whether you want to play a song or a sound effect: these functions will do it for you! There is an initialization function that must be called before you can use the engine. The other functions are used to start or stop playing songs or sound effects. There are also some function calls available which override the default behaviour of the sound engine.

System

```
void snd_sys_init (u32 channelMask, allocFuncPtr, freeFuncPtr);
```

This function should be called before you call any of the other sound engine functions.

- ***u32 channelMask***
  A bitmask where each bit represents a DS hardware-channel. Bit 0 represents channel #0, bit 15 represents channel #15.
  If you pass zero, the soundengine will use the default channel mask (check DirectSound.h to see what is the default).
  If you pass '0xFFFF' (or '65535'), all 16 hardware channels will be locked by the engine.
- ***allocFuncPtr***
  *A pointer to an allocation function which takes two parameters (**u32 size, u32 align**) and returns a **void *** to a block of memory that has enough room to store <size> bytes and is aligned to an <align> bytes boundary. It should never return NULL.*
  If you pass NULL, the sound engine will use the Nitro SDK when it requires dynamic memory allocations.
- ***freeFuncPtr***
  *A pointer to a deallocation function which takes one parameter (**void * pBlock**) and returns nothing. The soundengine will only pass pointers to this function which were obtained by calling 'allocFuncPtr'.*
  If you pass NULL, the sound engine will use the Nitro SDK when it requires dynamic memory allocations.

Note: if you want to use mixer post-processing effects, you should not include channel #1 and #3 in the channel mask. In other words: reset bit 1 and 3 of the channel mask. Pass 0xFFF5 for instance.

```
void snd_sys_exit (void)
```

Shuts down the soundengine and deallocates all dynamic memory that was in use by the engine. Call `snd_sys_init` when you feel like using it again.

```
void snd_sys_panic (void)
```

Call this when you have to stop all sound and timer interrupts used by the engine in a scenario where the system has become unstable (for instance: in an exception handler or error callback function).

Mixer

```
void snd_mix_setMasterVol (u32 volume);
```

Sets the hardware master volume.

- ***u32 volume***
  0 means: no output … 64 means: half output … 127 means: maximum output.
  The default mixer master volume is 127.

Note: This is the hardware master volume, so it also affects other libraries that use the sound hardware!

```
void snd_mix_stopAllSound (void);
```

Stops all hardware channels that are in use by the soundengine (both music and sound effects). Keep in mind that if a song is playing, it will soon start generating sound again on the stopped channels; if you want complete silence, be sure to stop song playback before calling this function.

```
void snd_mix_enableEffects (void *buffer, u32 size);
```

Locks hardware channel #1 and #3 for post-processing effects. After calling this function, you can start using the mixer effects.

- ***void *buffer***
  A buffer that will be used to capture the mixing output. The buffer must be aligned to a 32 byte boundary.
- ***u32 size***
  The size of the buffer. The length must be a multiple of 32 bytes. If you enable the reverb effect, the length of the buffer correlates to the delay of the reverb.

Note: In order to use post-processing effects, you must not allocate channel #1 and #3 for music and sound effect usage. In other words: ***the channel mask you pass to snd_sys_init must have bit 1 and 3 set to zero*** *(or the default channel mask in case you pass zero as a parameter).*

```
void snd_mix_disableEffects (void);
```

Unlocks hardware channel #1 and #3 and stops any active post-processing effects.

```
void snd_mix_setEffect (u32 effect, u32 volume);
```

Starts or changes the active mixer post-processing effect. The possible effects are enumerated in the `DirectSound.h` header. Currently the following effects are available:

SND_MIX_NORMAL	*normal output*
SND_MIX_REVERB	*the output will be captured and mixed together with the normal output, generating an echo. The size of the provided buffer (see snd_mix_enableEffects) determines the echo delay. Note that the captured output will also be captured again, so the effect volume also determines the amount of feedback.*

- ***u32 effect***
  The effect you would like to activate.
- ***u32 volume***
  The initial volume of the effect-output.

```
void snd_mix_setEffectVol (u32 volume);
```

Changes the active effect's output volume.

- ***u32 volume***
  The new effect-output volume.

Music / Songs

```
void snd_sng_init (const void *pSongset);
```

Call this function every time you want to use a (different) songset which resides in RAM. The pointer you provide should point to the binary output created by XM2DS. Make sure it's aligned to a 4 byte boundary.

```
void snd_sng_init_s (const char *pSongsetFile);
```

Call this function every time you want to use a (different) songset in streaming mode. Instead of providing a pointer to the songset, you provide a pointer to the filename (and path) of the songset. Using a songset in this mode will reduce RAM usage considerably.

```
void snd_sng_setMasterVol (u32 volume);
```

Sets the music master volume.

- ***u32 volume***
  0 means: no output … 128 means: half output … 256 means: normal output.
  The default music master volume is 256.

Note: If fading is active when you call this function, the fading will stop! This is because the fade routine uses the music master volume to change the output level. The fade routine restores the music master volume when fading is no longer active.

void snd_sng_play (u32 songNr);

Call this funtion to start playing a song from the songset. A songset can contain more than one song. To start the first song in the set, use 0 as an argument. To start the second, use 1, etc. You can also use the symbolic constants which you can find in the header, generated by XM2DS.
If you want to play a song as a *jingle*, do a logical OR with the `SND_SNG_JINGLE` flag which is defined in `DirectSound.h`, like this:

```
snd_sng_play(0|SND_SNG_JINGLE); // play the first song in the set as a jingle
```

Note: a jingle is a song which pauses the active song, plays once (no looping) and resumes the song which was playing prior to the jingle. If no song was playing when the jingle started, the soundengine will simply stop any music from playing when the jingle ends.
To determine whether the active song is a jingle, call 'snd_sng_isJingle'.

Note 2: jingles do not work fully when the songset was initialized using 'snd_sng_init_s' . When you start a jingle in streaming mode, make sure that no song is playing (i.e.: you can only use the jingle feature to play a song 'one shot').

void snd_sng_stop (void);

Call this function to stop song playback. Sound effects will continue playing.

void snd_sng_fade (s32 speed);

Call this function to fade a song in or out. Use a negative speed to fade a song out and use a positive speed to fade a song in. The fade routine always fades from, or to the current music master volume. If you select a speed of –8, it will take `<master volume> / 8` XM rows to fade out the song. Use the function `snd_sng_isFading` to determine whether the song is still fading or not.
When you fade a song out, the song will be paused when the volume level reaches zero. When a fade stops, the music master volume is restored to what it was, before the fade started.

Here is an example:

```
// start with silence
snd_sng_stop();
// fade in (takes 8 rows, assuming a music master volume of 256)
snd_sng_fade(+32); // call the fade function before you play
snd_sng_play(0);
while (snd_sng_isFading()) { VblankIntrWait(); }

// fade out in 32 steps
snd_sng_fade(-8);
while (snd_sng_isFading()) { VblankIntrWait(); }

// fade in again (takes 16 rows this time)
snd_sng_fade(+16); // call the fade function before you resume
snd_sng_resume();
```

void snd_sng_pause (void);

Call this function to pause song playback. Call `snd_sng_resume` to resume song playback again.

void snd_sng_resume (void);

Call this function to resume song playback after a call to `snd_sng_pause`.

Note: a song which faded out can also be resumed with this function.

```
u32 snd_sng_isPlaying (void);
```

Call this function if you want to know whether a song is playing. The function returns a non-zero value when a song is playing. If no song is playing, zero will be returned.

Note: a song which faded out is also considered not to be playing, so this function will return zero in that case.

```
u32 snd_sng_isPaused (void);
```

Call this function if you want to determine whether a song is paused. The function returns a non-zero value when the song is paused. If a song is playing, or the previous song has been stopped, zero will be returned.

Note: when a song faded out, it will be paused as well.

```
u32 snd_sng_isFading (void);
```

Call this function if you want to know if the engine is fading the active song. The function returns a non-zero value when the current song is fading, otherwise zero will be returned.

```
u32 snd_sng_isJingle (void);
```

Call this function if you need to know whether the active song is a jingle or not (in other words: is the jingle I started still playing, or has it been replaced by the song I interrupted, or any other song?). The function returns a non-zero value when the song is a jingle, otherwise zero will be returned.

Note: a jingle is a song which pauses the active song, plays once (i.e.: no looping) and resumes the song which was previously playing. If no song is playing when starting a jingle, the sound engine will simply stop any music from playing when the jingle has finished.

Sound effects

```
void snd_sfx_init (u32 sfxChn, const void *pSfxSet);
```

This function initialises the sound effects system. The parameters are:

- ***u32 sfxChn***
  The number of sound effect channels. You can change this setting later, if you like, by calling 'snd_mix_setSfxChannels'.
- ***const void *pSfxSet***
  A pointer to the soundset (created by XM2DS) you want to use. Make sure it's in RAM and aligned to a 4 byte boundary.

```
void snd_sfx_init_s (u32 sfxChn, const void *pSfxSetFile);
```

This function initialises the sound effects system in streaming mode. Using a soundset in this mode will reduce RAM usage considerably, but requires more CPU power.

The parameters are:

- ***u32 sfxChn***
  The number of sound effect channels. You can change this setting later, if you like, by calling 'snd_mix_setSfxChannels'.
- ***const char *pSfxSetFile***
  A pointer to the soundset file (created by XM2DS) you want to use.

```
void snd_sfx_setMasterVol (u32 volume);
```

Changes the sound effects' master volume.

- ***u32 volume***
  0 means: no output … 128 means: half output … 256 means: normal output. The default sound effects' master volume is 256.

```
void snd_sfx_play (u32 fx, u32 vol, u32 pri);
```

Call this function whenever you want to play a sound effect. The parameters are:

- ***u32 fx***
  The number (identification) of the effect you want to play. 0 means: the first effect from the current soundset, 1 means: the second effect from the current soundset, etc.
- ***u32 vol***
  The volume of the effect. 1 means: very soft … 64 means: very loud.
- ***u32 pri***
  The priority of the effect. The higher the number, the higher the priority of the effect. If - for example - all sound channels are occupied and you want to play another effect, the sound engine will decide whether it should stop an effect with a lower priority to make room for the new effect, or whether it should just ignore your request to play the effect.

Note: The effects will be placed in the centre of the spectrum. If you want to control the panning position of the sound use 'snd_sfx_playEx' instead.

```
void snd_sfx_pauseAll (void);
```

This function pauses all sound effect channels. To resume, call `snd_sfx_resumeAll`.

```
void snd_sfx_resumeAll (void);
```

This function resumes all sound effect channels, including the ones that were paused with `snd_sfx_pauseEx`.

```
void snd_sfx_stopAll (void);
```

This function stops all sound effect channels. If channels were paused, they cannot be resumed anymore!

u32 snd_sfx_isPlaying (u32 fx);

Call this function if you want to determine whether a sound effect is (still) playing. The parameter:

- ***u32 fx***
  The number (identification) of the effect you want to test. 0 means: the first effect from the current soundset, 1 means: the second effect from the current soundset, etc.

The function will return zero if the sound effect is not playing. Otherwise a non-zero number will be returned. The function will also work for sound effects which were triggered using `snd_sfx_playEx`, although you can use the `SND_SFX_EX.status` field as well for extended sound effects. See the next page to learn more about extended sound effects.

u32 snd_sfx_count (void);

Call this function if you want to know how many sound effects are stored in the active sfxset.

Extended sound effects – the Structure

To make use of the so-called 'extended sound effects'-interface, you have to create one or more instances of the following structure (as defined in `DirectSound.h`):

```
// --- extended sound effects structure
typedef struct
{
   u16 status;       // + 0 status (read only)
   u16 fx;           // + 2 effect number
   u16 pri;          // + 4 priority
   u8  vol;          // + 6 volume (1..64)
   s8  pan;          // + 7 panning (-64..+64)
   u32 freq;         // + 8 playback frequency in Hz (0 = use default)
   s32 loopStart;    // +12 start of loop (offset in samples, negative means no loop)
                     //(+16)
} SND_SFX_EX; // 16 bytes
```

All interface functions, related to extended sound effects, require a pointer to this structure. The pointer will be used as an identifier internally (so the pointer should point to static memory space if you want to use certain extended sound effect functions).

Some extra information about the fields in this structure:

- ***u16 status***
  You don't have to set this field. The soundengine will update this for you during each extended sound effect related interface-call. You can use it to check whether a sound effect is playing, stopped or paused. Use the enumeration in DirectSound.h to access the individual bits:

  ```
  enum _snd_sfx_status
  {
     SND_SFX_MUTE   = 0x01,
     SND_SFX_PAUSE  = 0x02
  };
  ```

- ***u16 fx***
  The sound effect you want to play. 0 means: the first effect from the current soundset, 1 means: the second effect from the current soundset, etc. (preferably use the header, generated by XM2DS).
  This field will be accessed in calls to 'snd_sfx_playEx' and 'snd_sfx_updateEx'.
- ***u16 pri***
  The priority of the effect; the higher the number, the higher the priority of the effect. If, for example, all sound channels are occupied and you want to play another effect, the sound engine will decide whether it should stop an effect with a lower priority to make room for the new effect or whether it should just ignore your request to play the effect.
  This field will only be accessed when calling 'snd_sfx_playEx'.
- ***u8 vol***
  The volume of the sound effect. 1 means: very soft … 64 means: loud.
  This field will be accessed when calling 'snd_sfx_playEx' and 'snd_sfx_updateEx'.
- ***s8 pan***
  The panning position of the sound effect. -64 means: furthest left … 0 means: center … +64 means: furthest right.
  This field will be accessed when calling 'snd_sfx_playEx' and 'snd_sfx_updateEx'.
- ***u32 freq***
  The 'playback-frequency' or 'pitch' of the sound effect in Hz. 0 means: default frequency … 16000 means: 16 KHz … 22000 means: 22 KHz … etc.
  This field will be accessed when calling snd_sfx_playEx and snd_sfx_updateEx. If the value of this field is 0, the soundengine will replace it with the actual (default) playback frequency value as stored in the original WAV file.
- ***s32 loopStart***
  *The position in the waveform (in samples) where playback should restart when the end of the waveform has been reached. A negative number means: no looping. A positive number means: loop the sound effect starting at: <sample **loopStart**>.*
  This field will only be accessed when calling 'snd_sfx_playEx'.

Extended sound effects – the Functions

The following functions allow you to use the structure, described on previous page.

`void snd_sfx_playEx (SND_SFX_EX *pFx);`

Call this function whenever you want to play or retrigger an extended sound effect. Fill in a `SND_SFX_EX` structure and pass its pointer. You should keep this structure in memory if you want to update, stop or pause the effect later on (the pointer to the structure has to be the same, so it should be in static memory space during its life-time).

`void snd_sfx_updateEx (SND_SFX_EX *pFx);`

Call this function whenever you want to change the volume, panning or pitch (frequency) of an extended sound effect which is already playing.
When you just want to update the status-field of the effect you're playing, you can call this function as well.
Make sure the pointer passed is the same as used in the call to `snd_sfx_playEx`. Also the `fx` field should be left unchanged when updating extended sound effects (weird things might happen!)

`void snd_sfx_pauseEx (SND_SFX_EX *pFx);`

Call this function whenever you want to pause an extended sound effect which is playing. Make sure the pointer passed is the same one as used in the call to `snd_sfx_playEx`.

`void snd_sfx_resumeEx (SND_SFX_EX *pFx);`

Call this function whenever you want to resume a previously paused sound effect. Make sure the pointer passed is the same as used in the call to `snd_sfx_pauseEx`.

`void snd_sfx_stopEx (SND_SFX_EX *pFx);`

Call this function whenever you want to stop a sound effect. Make sure the pointer passed is the same one as used in the call to `snd_sfx_playEx`.

Advanced Interface Functions

The functions described here can be handy in certain situations, but they're not required in most projects.

```
void snd_sng_setChannels (u32 musChn);
```

Call this function if you want to change the number of music channels. It can be called at any time (even during music playback). This function will be automatically called whenever you start playing a song; the sound engine will determine how many music channels are needed by the song and it will initialize the mixer accordingly. It's very dangerous to make this number larger than the actual number of channels of the song. Making it smaller won't do any harm, apart from the fact that the last (few) channel(s) won't play. It will give you a bit of extra CPU time.

```
void snd_sfx_setChannels (u32 sfxChn);
```

Call this function if you want to change the number of sound effect channels. It can be called at any time (even while sound effects are playing). This function will be called for you by `snd_sfx_init` and whenever a song needs the channels reserved for sound effects. If you want to change the number of sound effects yourself for some reason, then feel free to do so.

```
void snd_sng_setEffHandler (void(*pHandler)(u32 eff, u32 dat));
```

Call this function if you want to (un)install a handler for unhandled XM effects; every time the music replayer encounters an unknown or unhandled effect, it will call this handler with 2 parameters:

- the effect number
- the effect parameter

You can use this to receive signals from the songs in your project. Timing to the music, music-based games, effects synced to the beat, etc. are all possible by using this functionality.
Pass `NULL` if you want to uninstall a previously installed handler.

```
u32 snd_sng_getInfo (u32 type);
```

Call this function if you want to retrieve specific song related information. The following types are currently supported (and more can be added on request):

SND_INFO_ROW
Current row.

SND_INFO_POS
Current position.

SND_INFO_PAT
Current pattern.

SND_INFO_BPM
Current tempo in beats per minute.

SND_INFO_LOOPS
Amount of loops performed.

```
void snd_sng_setInfo (u32 type, u32 data);
```

Call this function if you want to change song-playback behaviour. The following types are currently supported (more can be added on request):

SND_INFO_ROW
Change current row. Will take effect immediately.
Parameter: row number.

SND_INFO_POS
Change current position. Will take effect when current pattern ends.
Parameter: position number.

SND_INFO_POS_ROW
Change current position and row. Will take effect immediately.
Parameter: position in hi 16 bits, row in lo 16 bits.

SND_INFO_BPM
Change current tempo. Will take effect immediately.
Parameter: new tempo in beats per minute (BPM).

SND_INFO_MUTE
Pauses (and thus mutes) certain channels. Will take effect immediately.
Parameter: The channels. Each bit represents a channel. b0 = channel 1, b1 = channel 2, and so on.

Note: resuming the song (calling 'snd_sng_resume') will unmute all channels.

SND_INFO_UNMUTE
Resumes (and thus unmutes) certain channels. Will take effect immediately.
Parameter: The channels. Each bit represents a channel. b0 = channel 1, b1 = channel 2, etc.

XM2DS Tool

The XM2DS tool was designed to run as a Win32 console application. It can be used to convert XM files to a songset (the music format used by the sound engine, which can contain up to 256 songs). Apart from that, it can also convert several WAV files into a soundset (the sound effect-format used by the sound engine).

Converting a lot of data at once can be achieved by using scripts (see the next page for an explanation about scripts). If you prefer to convert using so-called 'makefiles', instead of a script, the options below probably better suit your task:

Converting XM files
To convert your set of XM files into a binary, usable by the sound engine, is not a hard thing to do; the syntax is as follows:

```
XM2DS [-q<quality] -x<output name> <xm file> [<xm file> [<xm file> […]]]
```

Example:

```
XM2DS -xPACMAN_MUSIC.BIN INTRO.XM UI.XM GAME.XM GAMEOVER.XM
```

This will convert INTRO.XM, UI.XM, GAME.XM and GAMEOVER.XM and store those in PACMAN_MUSIC.BIN. Each song can have a different amount of channels and a different set of instruments. INTRO.XM will be song 0, UI.XM will be song 1, and so on.
XM2DS will also generate a header 'PACMAN_MUSIC.H', which contains symbolic constants for the songs. Your composer should be very interested in the file 'warnings.txt'; it will allow him to improve the quality of the converted sound.

Example of the instrument bank generation
Let's say you have two XM files. Song 0 contains 2 instruments (1:piano and 2:bass), Song 1 contains 4 instruments (1:piano, 2:bassdrum, 3:snaredrum and 4:bass). After converting these XM's to a songset, the songset will contain an instrument bank with the following instruments:

1:piano	
2:bass	(instrument 4 in song 1 will be remapped to instrument 2)
3:bassdrum	(instrument 2 in song 1 will be remapped to instrument 3)
4:snaredrum	(instrument 3 in song 1 will be remapped to instrument 4)

There is simply no need to worry about the instrument bank! The composer *should* keep in mind that an instrument will only be discarded if it is exactly the same as another instrument in any song included in the songset (same waveform, same settings, same envelopes).

Note that XM2DS ignores sample names and instrument names, etc.; it compares only the data that is actually used during sound processing.

Converting WAV files
To convert your set of WAV files to a binary, do this:

```
XM2DS [-q<quality>] -w<output name> <wav file> [<wav file> [<wav file> […]]]
```

Example:

```
XM2DS -wPACMAN_SFX.BIN WACKOWACKO.WAV BIGPILL.WAV EATINGGHOST.WAV DEAD.WAV
```

This will convert the waves WACKOWACKO.WAV, BIGPILL.WAV, EATINGGHOST.WAV and DEAD.WAV to the appropriate data (depending on quality level) and store them into PACMAN_SFX.BIN. WACKOWACKO.WAV will be sound effect 0, BIGPILL.WAV will be sound effect 1, etc.
XM2DS will also generate a header 'PACMAN_SFX.H', which contains symbolic constants for the sound effects. Your composer should be very interested in the file 'warnings.txt'; it will allow him to improve the quality of the converted sound.

Using scripts

This feature is especially useful when converting large amounts of XM- and/or WAV-files. The syntax is as follows:

```
XM2DS –s<script file>
```

Example:

```
XM2DS –sSCRIPT.TXT
```

This will instruct XM2AGB to parse SCRIPT.TXT. Below you'll find an example script.

```
# This is an example script – ©2003-©2007 by Engine Software BV
#
# (if a line starts with '#', it will be ignored… perfect for remarks!)

# set the quality level
-q0

# creating a songset:
# -x<name of output file>
# <xm file 1>
# <xm file 2>
# .
# .
# <xm file n>

-xsongset1.bin
songs\demo1\demo1.xm
songs\demo1\demo2.xm
songs\demo1\demo3.xm

# Let's create another songset…
# reasons for making different songsets are: need extra RAM space, or
#                                            more than 255 instruments or songs required

-xsongset2.bin
songs\demo2\demo1.xm

# creating an sfxset:
# -w<name of output file>
# <wav file 1>
# <wav file 2>
# .
# .
# <wav file n>

-wsfxset1.bin
wav\sfx1.wav
wav\sfx2.wav
wav\sfx3.wav

# multiple soundsets can be used for translations:
# set #1 contains speach for english… set #2 for japanese or dutch!
# Same indices… Easy programming!
# Also if level #1 requires different effects than level #2, it might be
# worthwhile to generate multiple sfxsets (to save RAM in non-streaming mode).

-wsfxset2.bin
wav\sfx4.wav
wav\sfx5.wav
wav\sfx6.wav

# end of script
```

More information about quality levels

Quality level #4	*no samples will be converted to ADPCM, only loop positions will be moved when necessary.*
Quality level #3	*samples that don't loop or have their loop positions aligned to 8 samples are converted to ADPCM.*
Quality level #2	*samples that don't loop or have their loop original loop positions misaligned are converted to ADPCM.*
Quality level #1	*all samples will be converted to ADPCM, except 8-bit samples with potential looping issues.*
Quality level #0	*all samples are converted to ADPCM and composer hints are ignored.*

Tips & Tricks

Some tips & tricks for all of you:

For the programmer:

- The music engine uses ARM7 alarm #0 to synchronize the 'ticks' of the music. ARM7 alarm #1 is used for certain mixer post-processing effects. All other 'sound alarms' are available for your own use.
- The XM2DS converter can be set to generate less/more optimal sample data. Use '`-q<quality level>`' (minimum quality: '`-q0`', maximum quality: '`-q4`', default: '`-q1`').
- Combine as many songs as you can (preferably the ones that share instruments) in a single songset. This will improve compression ratios and makes using the songs in your project easier too. The only reason to use multiple songsets would be: memory issues, localization, the 'instrument limit' (max: 255), or any other use you can find for it.
- You can use the looping-count information (`snd_sng_getInfo`) to stop a song after a certain amount of repeats, but if you want to play the song only once you can play the song as a jingle. Make sure you stop the previous song before you start the 'jingle', otherwise it will continue playing after the 'jingle' has ended.
- Jingles currently don't work when using streaming mode. Behaviour is undefined when a jingle is started while another song is playing! It's okay to start a jingle when no other song is playing, though.
- Do not use SND_SAFE_MODE in the final ROM. This saves some code space and CPU power.
- The TCM (Tightly Coupled Memory) usage of the engine is configurable in DirectSound.h.

For the composer/audio engineer:

- The Nintendo DS is very strict when it comes to sample length and loop positions. Here's a table with the restrictions:

8 bit PCM	Both length & loop position must be a multiple of 4 samples
16 bit PCM	Both length & loop position must be a multiple of 2 samples
ADPCM	Both length & loop position must be a multiple of 8 samples

- Using samples with a sample-rate higher than 32kHz is a waste of memory.
- Switch of Linear Interpolation and Volume Ramping in your tracker when composing for DS (if you want to hear how things will sound on final hardware, that is).
- Use a linear frequency table (not an Amiga frequency table, because it will result in a slightly different sounding portamento/vibrato).
- If you use the same instruments in several songs, make sure that the settings are exactly the same (apart from names and such). If not, they will be duplicated.
- Only use the 'G' and 'H' effect when the programmer knows about it, as it affects the master music volume.
- Clean up commands that don't do anything (i.e.: a note delay without specifying a note) as they still consume memory and CPU power. The XM2DS converter will get rid of some 'junk', but only when it knows it is safe to do so.
- You can add certain tags to your instruments (before, or after), namely:
  - PCM8 force this instrument's sample conversion to 8-bit PCM
  - PCM16 force this instrument's sample conversion to 16-bit PCM
  - ADPCM force this instrument's sample conversion to ADPCM
  - *No tag* programmer uses quality value (0..4) to determine conversion type

Simplified Specifications

Static size of the sound engine:

Data + bss section:	~31kB	
Code section (when optimized for speed):	~11kB	(compiled as Thumb-code)
	~17kB	(compiled as ARM-code)
Code section (when optimized for size):	~8kB	(compiled as Thumb-code)
	~13kB	(compiled as ARM-code)

(these are all worst case values with all features enabled – the engine can be stripped considerably)

Some facts:

- The DIV/SQRT co-processors are not used, so problems related to context switching are impossible.
- ARM7 timers #0 and #1 are reserved for the soundengine.
- NitroSDK is used for all ARM7 communication.
- NitroSDK's power management functionality is used when mixer effects are enabled.
- NitroSDK's filesystem is used in streaming mode.

(More details will follow… everything here is subject to change.)

History

Programmers: read the red lines
Composers: read the blue lines

1.08

- Fixed a bug which caused the DS to hang when the game card was pulled out in slumber mode (the WiFi sleep mode).
- Fixed a memory corruption bug which would occur when sound effects were released while their data was still being loaded asynchronously.
- Removed real-time instrument streaming mode from the engine (snd_sng_play_s). **Update your sources!**
- XM2DS: improved XM and instrument optimization. This also fixes a bug where loops were playing back incorrectly when an instrument contained (unused) sample data behind the loop position. **Update your datafiles!**
- XM2DS: improved error handling (for example: XM files with multiple samples in instruments will still convert and play, but they might play back incorrectly)
- XM2DS now also exports a C source file for each set, containing an array of pointers to the original filename strings.
- Improved memory usage and peak performance.
- Various minor bugfixes.
- New convenience functions: 'snd_sng_count', 'snd_sfx_count' (which return the number of songs/sfx in the active set) and 'snd_sys_panic' (which stops all sound and timer interrupts used by the engine without shutting down). See the function overview for more information.
- More checks performed in safe mode.
- New compile time option SND_USE_ITCM, which allows source licensees to determine how much TCM is used by the sound engine to gain extra performance.
- Improved XM playback compatibility (effect '9' [sample offset], effect 'F' [set speed/BPM], and various continious effects, like vibrato and tremolo).

1.02

- Fixed a bug which caused the least significant (most simple) row in the songset to be played as an empty row.
- Fixed a bug in XM2DS which caused some instruments to be duplicated (instruments were padded with random data, so your songsets might become smaller now).
- Improved the XM conversion speed of XM2DS.
- A value of '0' in the frequency field of extended sound effects will now always be updated to the 'default frequency stored in the original WAV file' when calling snd_sfx_updateEx.

1.01 (beta)

- Added a new song play function which streams the instrument samples during song playback.
- You can now provide heap management functions to the soundengine. **Update your sources!**
- Resolved all 'implicit conversion' warnings for the source licensees.
- Extra checks added to SND_SAFE_MODE.
- XM2DS now generates a warning text file for the composer.

1.00c

- Added the ability to switch between blocking and async sound effect streaming.
- XM2DS now allows you to put more than 255 sound effects in one set (max. 4096).

1.00b (beta)

- Improved instrument optimization in the XM2DS converter.
- Added streaming functionality to reduce RAM usage (beta). **Update your datafiles!**
- Effect '9' (sample offset) was broken.

0.99d (beta)

- Added mixer post-processing effects (reverb).
- Now using ARM7 timer instead of ARM9 timer for song processing.
- The loop position of extended sound effects is now interpreted as 'samples' instead of 'dwords'.
- When triggering sound effects quickly after one another, only the last one would play. This has been fixed.
- Improved ADPCM support in the XM2DS converter.
- XM2DS now removes redundant commands.
- Added more checks and messages to SND_SAFE_MODE.
- Effect 'ED' (note delay) was broken.
- Added support for effect 'EC' (note cut).
- Added support for effect 'P' (panning slide).

0.99b (beta)

- Added three new sound effect related functions: snd_sfx_pauseAll, snd_sfx_resumeAll & snd_sfx_stopAll.
- Improved sound effect handling (improved speed and accuracy).
- Fixed a bug: snd_sfx_updateEx did not flush the ARM7 commands it generated.
- Changed the naming of the priority enumeration: SND_FX_xxx -> SND_PRI_xxx. **Update your sourcefiles!**

0.99a (beta)

- Fixed a bug in the envelope processing (the panning/volume of instruments using envelopes was incorrect at times)
- Enabling 'SND_STRIP_FREQUENCIES' would change the pitch of the song. Fixed.
- Added some extra checks when 'SND_SAFE_MODE' is enabled.
- Fixed bugs in sound effect handling.
- Some other minor changes.

F.A.Q.

#0
***Q**: Why is this page so empty?*
***A**: I'm hoping to add some sort of question-database here.*

Watch this space!

Questions, Requests and Bug Reports

If you have general questions or requests, then don't hestitate to send an E-mail to *Ruud van de Moosdijk*. His E-mail address is: ruud@enginesoftware.nl.

If you have programming related questions or when you want to report a bug, send an E-mail to the programmer of the sound engine and XM2DS tool, *Jan-Lieuwe Koopmans*.
The E-mail address is: jan-lieuwe@enginesoftware.nl. Please mention the version you're using when you do so.

Please note that questions, already answered in this manual, will not be entertained. Make sure that you read this manual carefully!