SPIRIT DSP
Embedded Voice Experience

SPIRIT MP3 Decoder
User's Guide

Version 1.3

October 2010

Contents

About This Document

This document describes the API of Spirit MPEG audio (mp3) decoder. The decoder is available as:

- Portable floating-point ANSI C source code;
- Portable fixed-point ANSI C source code (full standard compliance);
- Portable fixed-point ANSI C source code (limited accuracy);
- Assembly optimized library for various DSP and RISC platforms.

Decoder Input Format

The MP3 (MPEG layer 3) audio decoder supports the following formats:

- MPEG-1, 2 or 2.5 formats.
- Layers 1, 2 and 3.
- VBR and Free-Format streams.
- Mono or stereo input streams.

Excluded features

By default, the MP3 decoder does not support the CRC code verification. The reason for that is the existence of some mp3 files with incorrectly encoded CRC code, so, if CRC check is enabled, such files cannot be decoded. This behavior matches the behavior of most PC-based decoders.

Decoder Output Format

- Stereo PCM signed 16-bit stereo.

Note that the output audio format does not depend on the input format. In case of mono input, both output channels will contain identical data. Stereo samples are stored in interleaved order; the left channel goes first. Note that the size of one output sample is always 32 bits (2 channels * 16 bits).

MP3 Introduction.

MP3 refers to the MPEG Layer 3 audio compression scheme that shrinks audio files with only a small sacrifice in sound quality. MP3 files can be compressed at different rates, but the more they are shrunk, the worse the sound quality. A standard MP3 compression is at a 10:1 ratio, and yields a file that is about 4 MB for a three-minute track.

The MPEG audio compression algorithm was developed by the Motion Picture Experts Group (MPEG), as a part of the International Organization for Standardization (ISO) standard for the high fidelity compression of digital audio. The MPEG-1 audio compression standard is one part of a multiple part standard that addresses the compression of video (ISO/IEC 11172-2), the compression of audio (ISO/IEC 11172-3), and the synchronization of the audio, video, and related data streams (ISO/IEC 11172-1). While the MPEG audio compression algorithm is lossy, it can often provide "transparent", perceptually lossless, compression.

The ISO/IEC-11172-3 standard specifies three similar formats, called “layers”, for the audio compression. All layers implements lossy compression of the digital audio for sampling rates 32, 44.1 and 48 KHz using sub-band coding with account for psychoacoustics principles. The main idea of these algorithms is to shape quantization noise in the frequency domain, so that it becomes undetectable by the average listener.

The further evolution of the standard is the MPEG-2 audio standard (ISO/IEC 13818-3). The MPEG-2 extends MPEG-1 for the sampling rates 16, 22.05 and 24 KHz. Also MPEG-2 introduces support for the multi-channel audio, however, this multi-channel extension is not considered in this document.

Also the unofficial extension of the Layer-3 audio for lower sampling frequencies (8, 11.025 and 12 KHz), called “MPEG 2.5 audio” is exists.

The widespread term “MP3” denotes the layer-3 audio (regardless of MPEG version), however, sometimes “MP3” used to denote all range of the MPEG audio algorithms. In this document we will use “MP3” with regard to all MPEG audio versions/layers.

MP3 format history

- **1987** the Fraunhofer Institut in Erlangen, Germany, started to work on perceptual audio coding in the framework of the EUREKA project EU147, Digital Audio Broadcasting (DAB).
- **1988** MPEG itself established, its full title Moving Picture Experts Group, not an organization in itself, but a subcommittee of the ISO/IEC (International Standards Organization/International Electrotechnical Commission).
- **1989** Fraunhofer Institut received a patent for MP3 in Germany.
- **1992** Fraunhofer’s algorithm was integrated in the emergent MPEG-1 standard.
- **1993** MPEG-1 (ISO/IEC 11172-3:1993) standard published.
- **1997** Since 1997, MP3 format was supported by several open-source projects
- **1998** MPEG-2 (ISO/IEC 13818-3:1998) standard published. The key difference with MPEG-1 standard is support for lower sampling frequencies (16-24 KHz). The original MPEG-1 supports frequencies from 32 to 48 KHz.
- **1998** WinAMP (free music player for Windows) appears.
- **1998** first portable MP3 player, Diamond Multimedia’s Rio 300 released
- **1999** The mp3 music distribution over the Internet becomes very popular. Many sites have emerged, allowing free music downloads. In 1999 the famous Napster (recently closed by RIAA lawsuit) appeared.

Currently, the MP3 format still remains the de-facto standard for audio compression. The key points of its success are:

- MP3 format provides good audio quality at high compression ratios (near-CD-quality for 10x compression).
- MP3 appeared at the “right time”, just as the demands for audio distribution over the Internet have risen.
- Due to its “open” nature, MP3 has support from open-source community. As a result, numerous free encoders and players (WinAmp, MusicMatch Jukebox, etc.) are available.

MP3 bitstream overview

The MP3 compressed bit stream can have one of several predefined fixed bit rates. The ranges for allowable bit rates are shown in the Table 1. Depending on the audio sampling rate, this translates to compression factors ranging from 2.3 to 48. It is allowed to encode different parts of the stream with different bit rates, producing so-called “Variable Bit Rate” (VBR) stream. In addition, the standard provides a "free-format” bit rate mode to support fixed bit rates other than the predefined rates. The coded bit stream supports an optional Cyclic Redundancy Check (CRC) error detection code.

MPEG version	Sampling rates (kHz)	Bitrate, kbps		
		Layer-1	Layer-2	Layer-3
MPEG-1 (ISO/IEC 11172-3)	32, 44.1, 48	32-448	32-384	32-320
MPEG-2 (ISO/IEC 13818-3)	16, 22.05, 24	32-256	8-160	8-160
MPEG-2.5 (Unofficial extension)	8, 11.025, 12	Nonexistent	Nonexistent	8-160

Table 1. Bitrates and sampling rates for the MP3 audio

The MP3 compressor encodes audio by the fixed size blocks. The block, called “frame” encodes predefined number of PCM samples (see Table 2). The decoding must start from the beginning of the frame, however it is possible to decode only a part of frame. The minimum decodable unit size is shown in the Table 2.

MPEG version	Frame size/minimum decodable unit, samples		
	Layer-1	Layer-2	Layer-3
MPEG-1 (ISO/IEC 11172-3)	384/32	1152/96	1152/576
MPEG-2 (ISO/IEC 13818-3)	384/32	1152/96	576/576
MPEG-2.5 (Unofficial extension)	Nonexistent	Nonexistent	576/576

Table 2. Frame sizes for the MP3 audio

The frame format is shown on Figure 1. The frame starts with a 32-bit frame header, which is used for stream synchronization and encodes basic audio information, such as sample rate, number of channels, bitrate, etc. The 16-bit CRC field is optional. Note that CRC is calculated not the entire frame, but only for the last 16 bits of the frame header and side info.

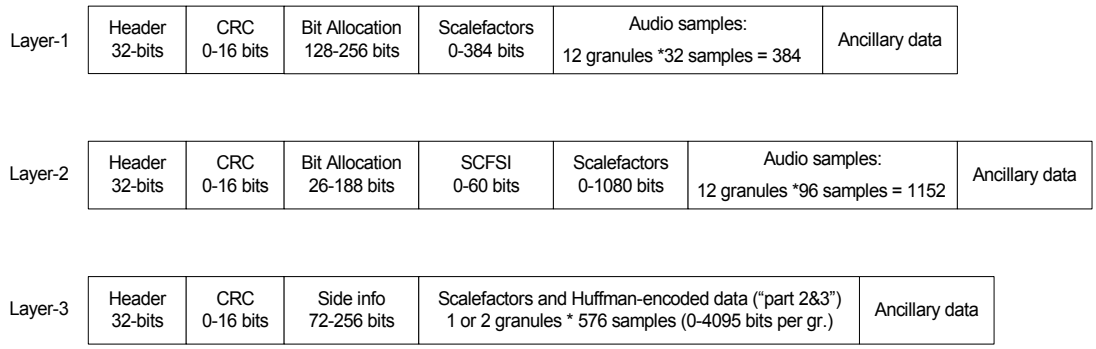

Figure 1. MP2 frame format.

There are no main file headers in MPEG audio files. An MPEG audio file is built up from a sequence of successive frames. MPEG stream cannot have any “gaps” between the frames. Frame size can be determined from the information contained in the frame header, so it is possible to determine position of the next frame header if the position of the previous one is known. The MPEG audio stream can contain ancillary data, which are used to keep the specified bit rate (i.e. frame size) when the bit reservoir cannot be increased or is not used.

In the Variable Bit Rate (VBR) stream, the frame size can differ across the stream. There are no special “flags” etc. to figure out whether the MPEG audio stream is VBR or not. The decoder should always assume that frame sizes may differ from frame to frame.

A rarely used feature of MPEG audio is the “Free format” mode. In this mode the frames in the stream can have arbitrary bitrate (possibly, different from the fixed set of bitrates specified in the standard). The free-format frame size cannot be calculated from the information contained in the frame header. In order to support the free-format mode, the decoder must figure out free-format frame size by searching several consecutive frame headers. The VBR mode is inadmissible when using free-format mode, and free-format frames cannot be mixed with fixed-bitrate frames.

In case of Layer I or Layer II, frames are totally independent entities: each frame contains all information required for its decoding, so the decoding process can start right after first frame header is found in the stream. However, in case of Layer III, frames are not always independent. The structure of the Layer-3 bit

stream is shown on Figure 2. The frame data can begin before the frame header (“bit reservoir”). The reference to the frame data (“main_data_begin”) is stored in the side info field of the frame. The bit reservoir technique allows to code the “hard to encode” audio fragments more efficiently. The size of the bit reservoir is limited to 511 bytes for MPEG1 and 255 bytes for MPEG1 and MPEG2. Note that in the extreme case the MPEG-2 frame data size can be only 1 byte (when decoding 8 kbps @ 24 KHz audio with CRC protection), so in this case up to 255 frames may be required before being able to decode one frame.

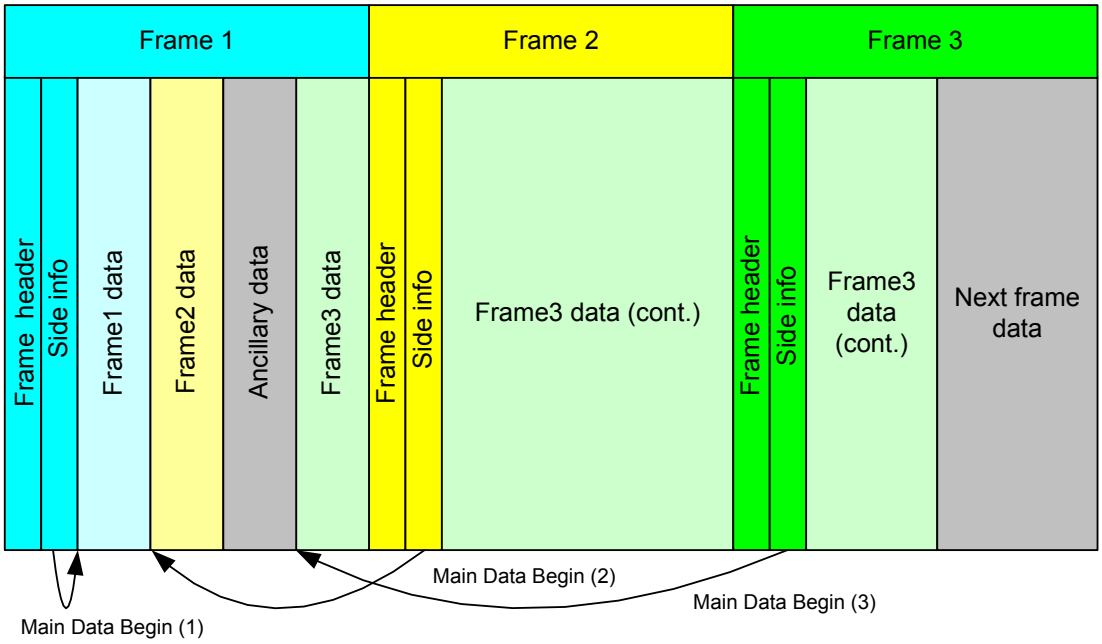

Figure 2. MP3 stream structure.

Frame header format

The first 32 bits (4 byte) of the audio frame are header information. Below, detailed description of these bits is given:

Syntax	No. of bits	Hex bitmask
header(){		
Syncword	11	FF E0 00 00
Idex	1	00 10 00 00
ID	1	00 08 00 00
Layer	2	00 06 00 00
protection_bit	1	00 01 00 00
bitrate_index	4	00 00 F0 00
sampling_frequency	2	00 00 0C 00
padding_bit	1	00 00 02 00
private_bit	1	00 00 01 00
Mode	2	00 00 00 C0
mode_extension	2	00 00 00 30
Copyright	1	00 00 00 08
original/copy	1	00 00 00 04
Emphasis	2	00 00 00 03
}		

Table 3 Header bitstream syntax

syncword (11 bits): The following bitstring: '1111 1111 111'.

IDex (1 bit): One bit to indicate the extended ID of the algorithm. It equals '1' for MPEG1 as well as for MPEG2 and '0' for MPEG2.5. Note: MPEG Version 2.5 is an unofficial extension of the MPEG 2 standard. It is an extension used for very low bitrate files, allowing the use of lower sampling frequencies.

ID (1 bit): One bit to indicate the ID of the algorithm. It equals '1' for MPEG1 and '0' for MPEG2 as well as for MPEG2.5 (i. e., for the lower sampling frequencies extension).

Idex	ID	Algorithm
0	0	MPEG25
0	1	Reserved
1	0	MPEG2
1	1	MPEG1

Table 4 Selection of decoded algorithms

layer (2 bits): Two bits to indicate which layer is used. For Layer 3, the bitstring is '**01**'.

layer	Algorithm
00	Reserved
01	Layer III
10	Layer II
11	Layer I

Table 5 Selection of MPEG audio layer

protection_bit (1 bit): One bit to indicate whether redundancy has been added in the audio bitstream to facilitate error detection and concealment. It equals '1' if no redundancy has been added and '0' if redundancy has been added (the 16-bit CRC field follows the header).

bitrate_index: Indicates the bitrate. The all-zero value indicates the 'free format' condition, in which any fixed bitrate, not limited to those present in the list, can be used. Fixed means that a bitstream frame contains either N or N+1 slots, depending on the value of the padding bit. The **`bitrate_index`** can be one of the values found in Table 6. It indicates the total bitrate irrespective of the mode (stereo, joint_stereo, dual_channel, single_channel).

Bitrate index	bitrate specified, kbps					
	MPEG1	MPEG1	MPEG2	MPEG1	MPEG2	MPEG2.5
	Layer 1	Layer 2	Layer 1	Layer 3	Layer 2 & 3	Layer 3
'0000'	Free	Free	Free	Free	Free	Free
'0001'	32	32	32	32	8	8
'0010'	64	48	48	40	16	16
'0011'	96	56	56	48	24	24
'0100'	128	64	64	56	32	32
'0101'	160	80	80	64	40	40
'0110'	192	96	96	80	48	48
'0111'	224	112	112	96	56	56
'1000'	256	128	128	112	64	64
'1001'	288	160	144	128	80	forbidden
'1010'	320	192	160	160	96	forbidden
'1011'	352	224	176	192	112	forbidden
'1100'	384	256	192	224	128	forbidden
'1101'	416	320	224	256	144	forbidden
'1110'	448	384	256	320	160	forbidden
'1111'	forbidden	forbidden	forbidden	forbidden	forbidden	forbidden

Table 6 Selection of bitrate by `bitrate_index`

Layer3 achieves variable bitrate by switching between the **`bitrate_index`** values. It can be used either to optimize storage requirements or to get any mean (interpolated) data rate by switching between adjacent **`bitrate_index`** values in the bitrate table. However, in free format, fixed bitrate is mandatory. The decoder is also not required to support bitrates higher than 320 kbps, 160 kbps or 64 kbps (depending on ID and IDex) in free format mode.

For Layer II there are some restricted combinations of bitrate and mode. The list of allowed combinations is found in the table below.

bitrate	single channel	stereo	intensity stereo	dual channel
free	yes	yes	yes	yes
32	yes	no	no	no
48	yes	no	no	no
56	yes	no	no	no
64	yes	yes	yes	yes

80	yes	no	no	no
96	yes	yes	yes	yes
112	yes	yes	yes	yes
128	yes	yes	yes	yes
160	yes	yes	yes	yes
192	yes	yes	yes	yes
224	no	yes	yes	yes
256	no	yes	yes	yes
320	no	yes	yes	yes
384	no	yes	yes	yes

Table 7 Allowed bitrate/channel combinations for Layer-2

sampling_frequency: Indicates the sampling frequency according to Table 8 below.

Bits	MPEG 1	MPEG 2	MPEG 2.5
00	44.1 kHz	22.05 kHz	11.025 kHz
01	48 kHz	24 kHz	12 kHz
10	32 kHz	16 kHz	8 kHz
11	reserved	reserved	reserved

Table 8 Sampling frequency selection

padding_bit: If this bit equals '1', the bitstream frame contains an additional slot to adjust the mean bitrate and get the exact sampling frequency, otherwise this bit will be '0'. For example: 128kbps 44.1kHz layer II uses a lot of 418 bytes long frames and a few 417 bytes long ones to get the exact 128k bitrate. For Layer I, a slot is 32 bits long, for Layer II and Layer III, it is 8 bits long. Padding is necessary with sampling frequencies of 11.025 kHz, 22.05 kHz and 44.1 kHz. Padding may also be required in free format.

private_bit: Bit for private use. This bit will not be used in the future by ISO/IEC.

mode: Indicates the mode according to Table 9. The joint stereo mode is intensity stereo and/or ms stereo.

Code	Stereo mode
00	Stereo
01	Joint stereo
10	Dual channel
11	Mono

Table 9 Stereo mode selection

mode_extension: These bits are used in joint stereo mode. For Layer-3 stream they indicate which type of joint stereo coding method is applied. The frequency ranges, over which the intensity_stereo and ms_stereo modes are applied, are implicit in the algorithm. For Layer-1 and Layer-2 streams, these two bits determine the frequency range (sub-bands) where intensity stereo is applied. There are 32 sub-bands in all.

The types of mode extensions are listed in Table 10.

Code	Layer-1 and 2	Layer-3	
	Joint stereo range	Intensity stereo	Ms stereo

00	bands 4 to 31	Off	Off
01	bands 8 to 31	On	Off
10	bands 12 to 31	On	On
11	bands 16 to 31	Off	On

Table 10 Selection of mode extension (only for joint stereo mode)

copyright: If this bit equals '0', there is no copyright on the Layer3 bitstream. '1' means that these data are copyright protected.

original/copy: This bit equals '0' if the bitstream is a copy and '1' if it is an original.

emphasis: Indicates the used type of emphasis. The emphasis is a technique which is sometimes used for audio transmission over band-limited (analog) channels. It pre-equalizes the audio contents by amplifying the high-frequency part of the signal.

The emphasis indication is here to tell the decoder that the file must be de-emphasized, i.e., the decoder must 're-equalize' the sound after a Dolby-like noise suppression. It is rarely used.

Emphasis	Emphasis specified
00	None
01	50/15 microseconds
10	Reserved
11	CCITT J.17

Table 11 Selection of emphasis type

In order to explain the meaning of the other fields of the frame format, the introduction to the encoding process should be made. The common part of the Layers 1-3 encoder is the analysis filter, which performs the sub-band decomposition of the input audio signal using cosine-modulated filter bank. The signal is decomposed into 32 sub-bands, and each sub band signal is decimated by a factor of 32, so for each 32 input PCM samples, the filter bank produces 1 sample for each sub-band. For an 1152 PCM samples-long frame, each sub-band filter produces 36 sub-band samples, as shown on Figure 3.

For Layer 1 and 2, the samples of each sub band are quantized. Each 12 consecutive samples in each sub-band are uniformly quantized using the same quantization step, which is encoded with the scalefactor.

For Layer-1 coding, the quantized samples are transmitted "as is", the number of bits for encoding of the quantized sample is constant for each sub-band in the frame, and transmitted in "bit allocation" frame field.

The Layer-2 scheme is similar to Layer-3. The only difference is that the frame contains 36 samples per sub-band rather than 12, but one scalefactor is still used for 12 consecutive sub-band samples only (this scheme allows more efficient scalefactors coding). Since some of the 3 sub-band scalefactors can have the same value, the redundant one can be left out of encoding, and data about the redundant scalefactors will then be encoded in the SCFSI (Scalefactors selector information) field.

Another difference between Layer 1 and Layer 3 is that the sub-band can use "group coding", when 3 consecutive samples in the sub-band are encoded by a single value. These operations result in grouping of sub-band samples, as shown on Figure 4. The information about samples encoding (number of bits per sample or triplet code information) is also encoded in the Bit Allocation field. The set of 3 consecutive samples from all sub-bands is called "granule".

In some sub-bands, all samples can be equal to 0 after quantization. Such sub-bands are not encoded; this is indicated in the Bit Allocation field. Note that the Layer-2 format encodes no more than 30 sub-bands: 2 high-frequency sub-bands are never encoded. (The Layer-1 format can encode the whole spectrum.)

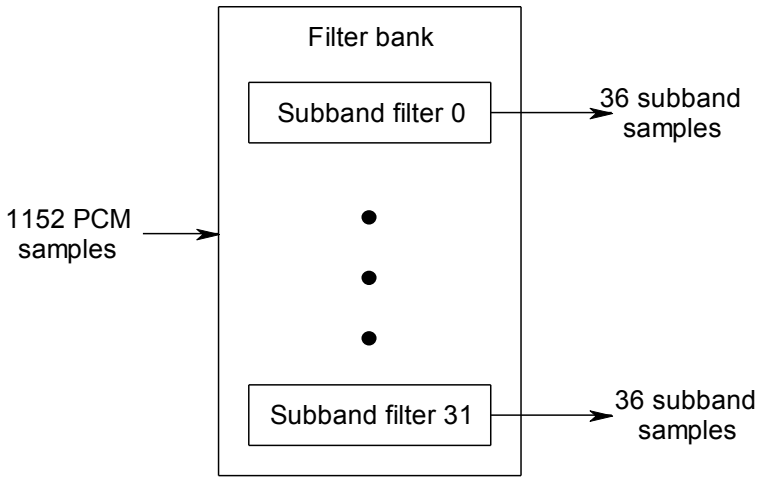

Figure 3. Sub-band decomposition of the MP2 audio block.

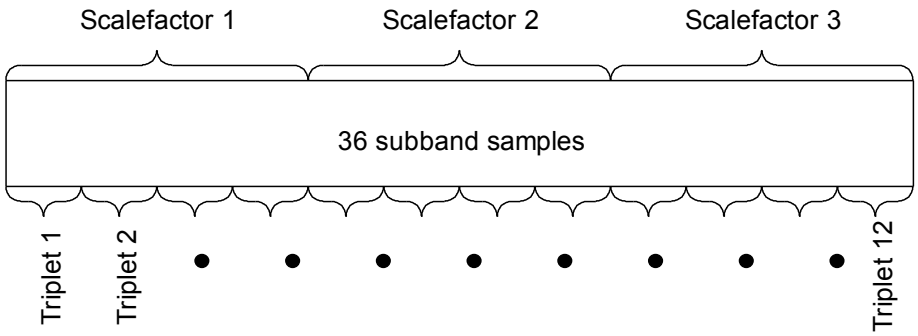

Figure 4. Grouping of sub-band samples for Layer-2.

The Layer-3 coding scheme is much more complex than that of Layer-1 and Layer-2. First of all, the Layer-3 coder uses “Hybrid” filter bank: the outputs of each sub-band of the analysis filter are processed using Modified Discrete Cosine Transform (MDCT), which is considered to be the second stage of hybrid filter bank and is used to increase frequency resolution.

The Layer-3 uses non-uniform quantizer (large values are quantized more roughly), which bears more likeness to the human audio perception mechanism. Finally, the Layer-3 uses lossless compression (Huffman encoding) of the quantized values.

Standard Compliance and Testing Procedure.

Numeric accuracy of the decoder was tested according to ISO/IEC 13818-4 and ISO/IEC 11172-4. The standard specifies that the maximum “RMS level” should be less than 8.809666e-006 for full accuracy decoding, and less than 1.409547e-004 for limited accuracy decoding. These ‘magic constants’ are explained as

8.809666e-006 = $2^{-} / \sqrt{12}$, which is equal to the quantization noise of 16-bit CD audio.

1.409546e-004 = $2^{-} / \sqrt{12}$, which is equal to the quantization noise of 12-bit audio.

Spirit provides four basic decoder configurations which differ by their quality:

- floating-point version;
- “96 dB” fixed-point version which uses 32-bit data path;
- “70 dB” fixed-point version which uses 16-bit data path;
- “90 dB” fixed-point version which uses mixed 32-16 bit data path.

The better decoder quality results in greater RAM usage. The floating-point and “96 dB” versions pass compliance test even with a 16-bit output. The “90 dB” and “70 dB” versions pass the “Limited accuracy” test.

Decoder version	RMS value	Decoder accuracy	PSNR on typical music files	RAM requirements
Floating-point	8.601812e-006	Full compliance	> 110 dB	12.4 Kbytes
“96 dB”	8.709599e-006	Full compliance	> 100 dB	12.4 Kbytes
“90 dB”	2.624353e-005	Limited accuracy	> 90 dB	10.3 Kbytes
“70 dB”	7.336634e-005	Limited accuracy	> 70 dB	6.6 Kbytes

The testing procedure includes:

- Comparison of decoder output with reference decoder output. As a reference decoder, the Adobe Audition™ MP3 decoder was used, since it is the only known PC decoder capable of decoding the latest MP3 test vectors from ISO.
- Running the decoder with non-MPEG streams.
- Positioning testing: verification that the decoder is able to start playback from arbitrary position in the stream.

The test suite includes test files from ISO/IEC 11172-4, Hannover University test suite, and a large set of mp3 files (more than 2000 items).

Library description

The library includes two files:

- `SpiritMP3Decoder.h` – C (C++) header file.
- `SpiritMP3Decoder.lib` – Library file (file name may vary for different target platforms)

The decoder uses the “pull” model of dataflow. During initialization, the user supplies the decoder with an `fread()`–like callback function. The decoder calls this function during decoding to obtain input data.

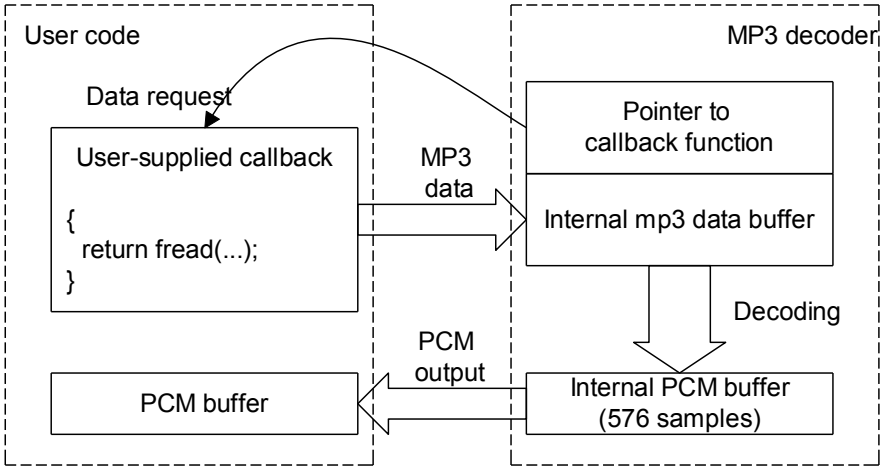

Figure 5 MP3 decoder data flow.

Description of Structures

TSpiritMP3Decoder

```
typedef struct
{
… Structure contents are omitted in the documentation.
} TSpiritMP3Decoder;
```

Description:

The MP3 decoder structure. All decoder variables are stored in this structure. The decoder does not use non-constant static or global variables so all functions are re-entrant. You can use several instances of the MP3 decoder by allocating one decoder structure per instance.

TSpiritMP3Info

```
typedef struct
{
  unsigned int nLayer;
  unsigned int nSampleRateHz;
  unsigned int nBitrateKbps;
  unsigned int nChannels;
  unsigned int IsGoodStream;
  unsigned int anCutOffFrq576[2];
  unsigned int nBitsReadAfterFrameHeader;
  unsigned int nSamplesPerFrame;
  unsigned int nSamplesLeftInFrame;
} TSpiritMP3Info;
```

Description:

The informational structure used with the `SpiritMP3Decode()` function and providing information about MP3 stream and audio parameters.

Fields Description:

Field	Description
`nLayer`	MPEG layer number. Possible valid values are 1, 2 or 3. Value 0 is used to indicate that other fields of this structure do not contain valid information.
`nSampleRateHz`	Sample rate value. Possible values: 32000, 44100, 48000 for MPEG 1 16000, 22050, 24000 for MPEG 2 8000, 11025, 12000 for MPEG 2.5
`nBitrateKbps`	Bitrate in kbits/s. For non-free format streams the values are in the range [8 –

	448]. In case of a variable bitrate this value equals to the bitrate of the last decoded frame. In case of free-format stream this value is equal to 0.
`nChannels`	Number of channels in MP3 stream. Note that the decoder always produces 2-channel output regardless of this value. Application can use this value to convert decoder output back to mono. Possible values: 1 for mono, 2 for stereo.
`IsGoodStream`	Flag indicates damaged stream. Equal to zero if stream error occurred just before the current frame was decoded, and has a non-zero value otherwise. Although damaged frames are not decoded, the frame immediately after the damaged one cannot be properly decoded due to the nature of MDCT, producing audible artifacts. The decoder automatically clears the output of such frames, but it is possible that the transition between audio frames will produce audible clicks. A player application can use this flag to provide suppression of additional clicks.
`anCutOffFrq[2]`	Cuts off frequencies for left and right channels. Valid only if `nLayer` is set to 3. Valid values are from 0 to 575, where 575 indicates full bandwidth (i.e., `nSampleRateHz`/2). Can be used for additional processing of decoded data, for example, with SPIRIT's Audio Post Processing algorithm.
`nBitsReadAfterFrameHeader`	Number of bits read via a callback function since the beginning of the current frame. Can be used to define position of the current frame in the stream.
`nSamplesPerFrame`	Number of audio samples per frame: Equals 384 for Layer 1, 1152 for Layer 2 and MPEG1 Layer 3, and 576 for MPEG2 & 2.5 Layer 3.
`nSamplesLeftInFrame`	Number of remaining audio samples still to decode in the current frame. Together with the `nBitsReadAfterFrameHeader` field, this one can be used for audio/video synchronization in a multiplexed MPEG steam.

Description of the Callback Functions

fnSpiritMP3ReadCallback ()

```
typedef unsigned int (fnSpiritMP3ReadCallback)(
    void         * pMP3CompressedData,  //Buffer to read data in
    unsigned int   nMP3DataSizeInChars, //Buffer size in 16-bit elements
    void         * token                //Application-supplied param
);
```

Description:

The typedef for the application-supplied callback function, which is used by the decoder to retrieve input data. The application must implement this function and initialize the decoder with a pointer on it. The function should work similar to the fread() function. The function can read less data than requested, however in this case decoder can pause decoding, and resume it only when input data will be available.

Parameters:

`pMP3CompressedData`	Pointer to the internal decoder buffer to be filled.
`nMP3DataSizeInChars`	Number of MAU elements (i.e., elements with the `sizeof(char)` size) in the `pMP3CompressedData` buffer.
`token`	Optional application-supplied data passed by the decoder to this function.

Return Value:

A number of chars (16-bit or 8-bit elements depending on the architecture) placed into the `pMP3CompressedData` buffer.

fnSpiritMP3ProcessCallback ()

```
typedef void (fnSpiritMP3ProcessCallback)(
    int   *pMDCTcoeff, // [IN/OUT] MDCT coefficients
    int   isShort,     // Buffer size in 16-bit elements
    int   ch           // channel index
    void *token,       // Application-supplied param

);
```

Description:

The typedef for the optional application-supplied callback function, which can be used by the application to process coded data in the MDCT domain. The application must implement this function and initialize the decoder with a pointer on it. MDCT data processed in-place. This function called only for Layer-3 audio, but not for Layer-1 or Layer-2 audio. The MDCT coefficients represented by the 32-bit integers. It is not recommended to increase the value of the coefficients by more than 6 dB to avoid integer arithmetic overflow.

Parameters:

pMDCTcoeff	Pointer to the input/output data. 576 MDCT coefficients. Coefficient 0 correspond to lowest frequency (DC), Coefficient 575 correspond to the highest frequency (Nyquist)
isShort	Flag, indicating that MDCT coefficients correspond to the short window sequence (3 consecutive blocks, 192 samples in each block). In this case the MDCT coefficients are reordered as follows: **$W0_0$ $W0_1$ $W1_0$ $W1_1$ $W2_0$ $W2_1$ $W0_2$ $W0_3$ …** Where WN_k – coefficient k for window N. This means that coefficients for different windows are interleaved by pairs.
Ch	Channel index: 0 for left (or mono) channel, 1 for right channel.
token	Optional application-supplied data passed by the decoder to this function.

Return Value:

None.

Remarks

The one of the benefits of the use of *processCallback()* function is a low cost equalizer implementation. Any well defined spectrum envelop can be applied to signal by the cost of one multiplication per sample. The envelop can be controlled at 576 points (bands), because of only 576 frequency samples available. The use of short frames (*isShort* flag set) decreases this resolution to 192 bands. The following example illustrates how equalizer routine can be implemented:

```
void eq_apply(int *x, int isShort, int ch, void *token)
{
    const TAPP *app = token;
    const unsigned short *envelop = app->envelop;
    int i, j;

    if(app->lowRes) {           // isShort not required
        for(i = 0; i < 576; i+=6) {
            const unsigned short c = app->envelop[i];
            for(j = 0; j < 6; j++) {
                x[i+j] = mul (x[i+j], c);
            }
        }
    } else {
        if(!isShort) {          // 1 sample/band
            for(i = 0; i < 576; i++) {
                x[i] = mul (x[i], envelop[i]);
            }
        } else {                // 3 samples/band
            for(i = 0; i < 576; i+=6) {
                const unsigned short c0 = envelop[i+0]; // [i+1] [i+2] - not used
                const unsigned short c1 = envelop[i+3]; // [i+3] [i+4] - not used
                x[i+0] = mul (x[i+0], c0);
                x[i+2] = mul (x[i+2], c0);
                x[i+4] = mul (x[i+4], c0);

                x[i+1] = mul (x[i+1], c1);
                x[i+3] = mul (x[i+3], c1);
                x[i+5] = mul (x[i+5], c1);
            }
        }
    }
}
```

Library API Description

SpiritMP3DecoderInit ()

```
void SpiritMP3DecoderInit (
    TSpiritMP3Decoder * pDecoder,           //Pointer to decoder structure
    fnSpiritMP3ReadCallback * pCallbackFn,  //Pointer to data read callback
    fnSpiritMP3ProcessCallback *pProcessFn, //Pointer to post-process callback
    void * token                            //Optional callback parameter
);
```

Description:

Initialization function. It must be called before any other decoder functions.

Parameters:

`pDecoder`	Pointer to the decoder structure to be initialized.
`pCallbackFn`	Pointer to the application-supplied callback function. Decoder will use this function to retrieve input mp3 data.
`pProcessFn`	Pointer to the application-supplied callback function. Decoder will use this function to process MDCT coefficients when decoding layer-3 files.
`token`	Optional parameter that will be passed to the callback function.

Return Value:

None.

SpiritMP3Decode ()

```
unsigned int SpiritMP3Decode (
    TSpiritMP3Decoder *pDecoder,        //Pointer to decoder structure
    short *pPCMSamples,                 //Pointer to output buffer
    unsigned int nSamplesRequired,      //Number of 32-bit elements to decode
    TSpiritM3Info * pInfo,              //Pointer to information structure
);
```

Description:

Decodes a given number of samples and stores them in the `pPCMSamples` buffer.

Parameters:

pDecoder	Pointer to the initialized decoder structure.
pPCMSamples	Pointer to the output PCM buffer. The decoder always produces stereo output, regardless of the inputs.
nSamplesRequired	Number of audio samples to decode. NOTE that one audio sample contains 32 bits (two 16-bit words – for left and right channels). If this parameter value is zero, decoder does not process any data.
pInfo	Pointer to the information structure to be filled (can be NULL).

Return Value:

The number of decoded 32-bits elements.

Remarks

This function always decodes a given number of samples. In case of an mp3 stream error, damaged data are skipped and the next valid data portion is decoded. In case of the end of mp3 file, the function returns a value less than `nSamplesRequired`, otherwise it returns `nSamplesRequired`. Actually, samples are decoded first into the internal PCM buffer and the required number of samples is copied to `pPCMSamples` from the internal buffer. The maximum number of samples in the internal buffer is 576 for layer 3 and 384 for layers 1 and 2. If the number of samples in the internal buffer is greater than the requested number of samples, decoding is not performed, and previously decoded samples are copied from the internal buffer.

After successful decoding, the decoder fills the optional `pInfo` structure with information about the last decoded frame. If several frames were decoded, only information about the last frame is stored in the `pInfo` structure.

If the `nSamplesRequired` parameter is 0, but the `pInfo` parameter is not NULL, and no frames have been decoded yet, the decoder decodes the data in the internal buffer and fills the `pInfo` structure, but does not copy the data into `pPCMSamples`.

Source Samples

Sample 1: MP3 to PCM file decoder.

```
// This sample code decodes file "myfile.mp3" into PCM file "myfile.PCM"

#include "SpiritMP3Decoder.h"
#include <stdio.h>

// Optimal size of output buffer is a multiple of
// MP3 granule size (576 samples)
#define MY_FRAME_SIZE_IN_SAMPLES 576

// Output PCM buffer: size multiplied by two to account for stereo signal
short     g_aPCMSamples [2*MY_FRAME_SIZE_IN_SAMPLES];

// Decoder structure
TSpiritMP3Decoder g_MP3Decoder;

// Callback function, reads MP3 data from the file, calling fread().
// Assumes that token is a MP3 FILE handle;
unsigned int RetrieveMP3Data(void * pMP3CompressedData,
                             unsigned int nMP3DataSizeInChars,
                             void * token)
{
  return fread(pMP3CompressedData, sizeof(char), nMP3DataSizeInChars, (FILE*)token);
}

void main()
{
  unsigned int nSamples;

  // Open mp3 and PCM files. Error check is omitted from sample code.
  FILE * pMP3file = fopen ("myfile.mp3","rb");
  FILE * pPCMfile = fopen ("myfile.pcm","wb");

  // Initialize decoder, set user-defined parameter to mp3 FILE handle
  SpiritMP3DecoderInit(
    &g_MP3Decoder,       // MP3 decoder object
    RetrieveMP3Data,     // Input callback function
    NULL,               // No post-process callback
    pMP3file             // Callback parameter
    );

  // Decode loop
  do
  {
    nSamples = SpiritMP3Decode(
      &g_MP3Decoder,              // MP3 decoder object
      g_aPCMSamples,              // Output PCM buffer
      MY_FRAME_SIZE_IN_SAMPLES,   // PCM buffer size in samples (1 sample = 2ch*16 bits)
      NULL                    // [OUT] MP3 stream info structure - optional, not used in example.
      );
    fwrite(g_aPCMSamples, 2*sizeof(short), nSamples, pPCMFile);
  } while(nSamples);

  // Cleanup
  fclose(pPCMfile);
  fclose(pMP3file);
}
```

Frequently Asked Questions

1. I want to evaluate the MP3 decoder quality.

You can use following items, which are available from Spirit free of charge:

- Demo version of the decoder library for the target platform.
- PC model of the decoder library (DLL).
- PC demo realization of the MP3 decoder.

2. I have obtained a demo library from SPIRIT. What are the differences between the demo and the release version?

There are no differences between the demo and the commercial version except bitrate restrictions. (Same folders structure, same API, same file names, etc). The demo version supports only the highest bit rates, i.e., 320 kbps for Layer-3, 384 kbps for Layer-2, and 448 kbps for Layer-1. As a consequence, the demo version cannot support Variable Bit Rate (VBR) files.

3. I have decoded the MP3 file to WAV file using the sample application supplied with the MP3 decoder library. But the resulting .WAV file is playing wrong (too fast or too slow). What is the problem?

The reason for this problem is that the sampling rate is encoded only once in the header of the .WAV file, while the .mp3 stream encodes the sampling rate in each mp3 frame (i.e., for every 1152 samples), and in some rare cases the sampling rate of the mp3 frames can change across the stream. As you can see from the source code, the sample application sets the .WAV sample rate according to the sampling rate of the LAST decoded .mp3 frame. So, the described problem is possible in the following cases:

- The input .mp3 file is a concatenation of two MP3 streams with different sampling rates (most probably).
- The .mp3 file has some inconsistent data at the end of the file, but the decoder accepts these data as a valid MP3 stream.

Please note that there is no good solution for such problems: if you are going to support such invalid .mp3 streams, your application should not expect that the sampling rate and number of channels will remain the same over all frames of the mp3 stream.

For example, the problem with your file could be fixed if the WAV sample rate is set according to the sampling rate of the FIRST (and not the last) decoded frame, but we have an example of the .mp3 which has the first frame encoded with one sampling rate, while the rest of the file encoded with a different one – so we will get the problem in this case (the file "mp3Err02.mp3", described in the test report).

4. I'm going to implement a file list decoding, and want to minimize the decoding latency between audio tracks. Can I feed the decoder with different MP3 streams without resetting the decoder?

You can use different approaches. Even simple concatenation of different mp3 streams will work: you can feed streams continuously without any decoder re-initialization. However, in this case your application should be aware that the stream sampling rate can change (as mentioned in the previous question); besides, you can lose 1 frame in the beginning of the new stream if it has a different sampling rate (due to the decoder's internal error protection), but this should not be perceptually significant.

The recommended (and slightly more complicated) way of switching decoding between streams is:

1. Detect that the decoder has finished the stream decoding: it is finished when the decoder does not produce any data AND does not consume any data. Note that for small bitrates, several MP3 frames can be buffered in the internal decoder input buffer, and the decoder can produce PCM data even when the input callback function returns 0, so you should check both conditions: NOT produce AND NOT consume.
2. After the decoding is finished, re-initialize the decoding with `SpiritMP3DecoderInit()`.

3. When the decoder calls the callback function next time, feed the new stream data.

In any case, the decoder re-initialization does not require any computation-intensive steps, and is much faster than frame decoding. Whichever way you choose, there will be no additional decoding latency.

5. How must the callback function be implemented?

The decoder can accept data from any sources (FLASH, mass storage device, etc). The decoder itself does not use any file-IO functions, nor any other C RTL functions. Input MP3 data are supplied by the application via the callback function. The function prototype is as follows:

```
typedef unsigned int (
     fnSpiritMP3ReadCallback)(void * pMP3CompressedData,
     unsigned int nMP3DataSizeInChars,
     void * token);
```

where `pMP3CompressedData` is the internal decoder buffer to be filled with mp3 data; `nMP3DataSizeInChars` is the buffer size, and `token` is a token supplied by the application during decoder initialization, which is transparent to the decoder. The function returns the number of Minimum Access Units (MAU's) or chars, placed into the decoder's internal buffer.

Please note that for some 16-bit DSPs, 1 MAU = 16 bit = 2 bytes = 1 word! In this case the mp3 file data should be read as little-endian words: it is expected that the first mp3 byte goes to the least significant half of the DSP word. There is no restriction for word-alignment of mp3 data; however, the mp3 data must be byte-aligned. In all other cases the data should be placed into the buffer preserving the file order (without -endian conversion).

It is expected that the callback implementation should be similar to the `fread()` function behavior: i.e., it is assumed that the application can provide as much data as requested by the decoder, except the end-of-file situation. So, under normal conditions, the callback function should fill `nMP3DataSizeInChars` words of data and return the `nMP3DataSizeInChars` value. If for some reasons this requirement cannot be fulfilled, the decoder just pauses the decoding, and the decoding function returns 0. Decoding can be resumed as soon as the input data are available; this does not introduce any side effects.

The maximum size of requested data is 578 bytes. Please note that the requested data size varies from call to call from several words to the maximum value. Please also note that the internal buffer size is smaller than the maximum MP3 frame size, so in case of higher bitrates the callback function can be called several times during the decoding of one mp3 frame.

Keep in mind that if the MP3 decoder is fed with invalid data, it will continuously search the input data for mp3 sync points, either until a valid mp3 frame is found, or end-of-file is reached. In this case the decoder function does not return! So, when feeding the codec with MP3 decoded data, please do not expect that the codec buffer will always be filled with data from the decoder, and please verify your application with some invalid data (any big binary file...).

If you need to gracefully abort the decoding, you can simulate the end-of file by returning 0 from the callback function.

6. It would be nice to have some post-processing (equalizer, speed change, etc).

The "base" variant of the MP3 decoder does not have any port-processing functions, however, some of them can be easily implemented in the decoder.

1. Double-speed/Half-speed playback: The pitch of audio can be changed in the frequency domain, so the decoder can provide double of half speed playback without changing the pitch of the audio (so called "time stretch" effect).
2. The Decoder can provide audio equalization with minimum MIPS requirements. However, the fixed-point implementation introduces a limitation on the maximum amplification factor (depends on the

target platform, but in most cases it is no more than +6 dB). For a high-quality equalizer, it is recommended to use an external equalization algorithm, which is also available from SPIRIT.

Also you can use SPIRIT's proprietary algorithm for the reconstruction of missing high-frequency part in coded mp3 audio.

7. *Decoder is crashing when running on the target platform, but the same MP3 file is decoded fine using the supplied sample application.*

Most likely, you have overfilled the decoder memory. Please check that input callback function does not overrun the decoder's internal buffer. Also please check that the output PCM buffer has enough space.

8. *Decoder cannot decode some Layer-2 files (48 kbps stereo, 384 kbps mono, etc).*

The MPEG standard has some "forbidden" bitrate/sample rate combination for Layer-2 (please see Table 7. above). These combinations are forbidden because standard does not specify some essential decoding tables for them. But the reference code ("dist10") does support these combinations: actually it seems to be a bug in the reference code rather than a feature...

The Spirit decoder behavior can be changed in compile-time to either follow ISO documentation or dist10 code. See embedded Excel table below for technical details. Please contact technical support, if you have further questions about this topic.

This file contains analysis for Layer-2 decoder implementation. The ISO standard specifies that some Layer-2 bitrates are forbidden, however, some decoders (including reference dist10 code) allows such bitrates. The main problem to support illegal bitrate is the choice of so called "bit allocation table", which depends on bitrate and sample rate. There are exists 4 such tables (numbering from 0). This file represents table selection for some decoders implementations.

	ISO allowed bitrates per ch		ISO documentation table selection (Annex B)			dist10 code table selection (see COMMON.C::pick_table())			Spirit Decoder when MP3CFG_STRICT_ISO_STANDARD disabled			MAD decoder (see layer12.c::mad_layer_II())		
			kHz			kHz			kHz			kHz		
	kbps	allowed?	48	44.1	32	48	44.1	32	48	44.1	32	48	44.1	32
free-format		yes	0	1	1	2	2	3	0	1	1	0	1	1
	16	no	n/a	n/a	n/a	2	2	3	2	2	3	0	1	1
	24	no	n/a	n/a	n/a	2	2	3	2	2	3	0	1	1
	28	no	n/a	n/a	n/a	2	2	3	2	2	3	0	1	1
	32	yes	2	2	3	2	2	3	2	2	3	2	2	3
	40	no	n/a	n/a	n/a	2	2	3	2	2	3	0	1	1
	48	yes	2	2	3	2	2	3	2	2	3	2	2	3
	56	yes	0	0	0	0	0	0	0	0	0	0	0	0
	64	yes	0	0	0	0	0	0	0	0	0	0	0	0
	80	yes	0	0	0	0	0	0	0	0	0	0	0	0
	96	yes	0	1	1	0	1	1	0	1	1	0	1	1
	112	yes	0	1	1	0	1	1	0	1	1	0	1	1
	128	yes	0	1	1	0	1	1	0	1	1	0	1	1
	160	yes	0	1	1	0	1	1	0	1	1	0	1	1
	192	yes	0	1	1	0	1	1	0	1	1	0	1	1
	224	no	n/a	n/a	n/a	0	1	1	0	1	1	0	1	1
	256	no	n/a	n/a	n/a	0	1	1	0	1	1	0	1	1
	320	no	n/a	n/a	n/a	0	1	1	0	1	1	0	1	1
	384	no	n/a	n/a	n/a	0	1	1	0	1	1	0	1	1

9. *Having implemented my own mp3-splitter I mentioned that sometimes the stream properties (layer, sampling rate, etc.) reported by splitter and decoder are different. How to explain it?*

The mp3 frame header is always byte-aligned and starts with a start-code (syncword). The only valid syncwords are:

- b1111 1111 1111 - MPEG1&2
- b1111 1111 1110 001 - MPEG2.5 layer 3

The syncword is not a unique bit-sequence within a stream. The following parameters remain unchanged within a valid stream and must be also taken into account to detect headers correctly:

- MPEG version
- Layer number
- Sampling rate
- Bitstream type (free-format or non-freeformat)

In order to detect mp3-header application should search for a syncword and accept header only if next consecutive header is found. Both headers must be equal. Please note this is not prohibited to channels number to change across the stream.

The splitter implementation based on syncword detection approach only is the most common mistake. The syncwords approach works perfect for video codecs but not applicable for mp3.