ARM MP3 Encoder
Integration Guide
Copyright © 2001 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 March 2001</td>
<td>A</td>
<td>First draft</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is final (information on a developed product).

Web Address

http://www.arm.com
Contents

ARM MP3 Encoder Integration Guide

Preface
About this book ................................................................. vi
Feedback ........................................................................... ix

Chapter 1  Introduction
1.1  About MPEG audio coding ........................................... 1-2
1.2  About the ARM MP3 Encoder ..................................... 1-4

Chapter 2  The ARM MP3 Encoder API
2.1  Configuration options .................................................. 2-2
2.2  Data types ................................................................. 2-5
2.3  Constants ................................................................. 2-7
2.4  Functions ................................................................. 2-9

Chapter 3  Example Application
3.1  About the example application ..................................... 3-2
3.2  Command-line arguments .......................................... 3-3
3.3  Benchmarking .......................................................... 3-4

Glossary
Preface

This preface introduces the ARM MP3 Encoder. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.
Preface

About this book

This book provides user information for the ARM MP3 Encoder. It gives an overview of MPEG audio coding, and information on the MP3 Encoder and its API, and provides an application for reference.

Intended audience

This book is intended for all developers who are designing or implementing an ARM-based embedded system that uses the ARM MP3 Encoder.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Read this chapter for an introduction to the functionality of the ARM MP3 Encoder.

Chapter 2 The ARM MP3 Encoder API
Read this chapter for an introduction to the API, and details of its configuration options, data types, and functions.

Chapter 3 Example Application
Read this chapter for details on the use of the library, and simple functional tests and benchmarks. The distribution contains makefiles and Metrowerks CodeWarrior project files, that can be used with the ARM Developer Suite (ADS) to build the application.
Typographical conventions

The following typographical conventions are used in this book:

**typewriter** Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

**typewriter** Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

**typewriter italic**

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

**typewriter bold**

Denotes language keywords when used outside example code and ARM processor signal names.

*italic*

Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

**bold**

Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See http://www.arm.com for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at: http://www.arm.com/DevSupp/Sales+Support/faq.html

**ARM publications**

Refer to the following ARM publications for information on related subjects:

- *ARM MPEG-2 Audio Layer III Decoder* (ARM DUI 0121)
- *ARM MPEG - Advanced Audio Coding Decoder* (ARM DUI 0129)
Other publications

The publications listed below might also be useful:


For more information on Metrowerks, and the CodeWarrior IDE generally, visit the Metrowerks web site at [http://www.metrowerks.com](http://www.metrowerks.com).
Feedback

ARM Limited welcomes feedback on both the ARM MP3 Encoder and its documentation.

Feedback on the ARM MP3 Encoder

If you have any problems with the ARM MP3 Encoder, please contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any comments on this book, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.
Chapter 1
Introduction

This chapter introduces MPEG in general and the MP3 Encoder. It contains the following sections:

- About MPEG audio coding on page 1-2
- About the ARM MP3 Encoder on page 1-4.
1.1 About MPEG audio coding

MPEG Audio is an encoding scheme that compresses high quality audio by a factor of 10 (approximately) with little audible degradation. It defines three different compression methods, the most popular of which is Layer III, commonly known as MP3. This method is the most efficient and most computationally complex.

Like many compression methods, MP3 attempts to reduce the amount of information in the signal by:

- removing redundant information that can be reconstructed from what remains
- removing irrelevant information whose absence will not be noticed
- encoding the remaining information in an efficient manner.

In the specific case of MP3 coding, this is achieved as follows:

1. For a stereo signal, exploit similarities between the channels where possible. This is known as joint stereo coding.

2. Apply nonlinear quantization to a frequency spectrum of the signal, using a model of human acoustic perception to ensure that the noise introduced by this is masked as much as possible by the signal.

3. Encode the information required to reconstruct the signal in a bitstream format, using variable-length coding for the bulk of the data.

The bitstream consists of a sequence of frames, each containing the information required to reconstruct a fixed number of samples. Depending on the sample rate, a frame can represent either 576 or 1152 samples.

1.1.1 Variations in MPEG standards

The original MPEG Audio specification, MPEG-1, has had two extensions:

- MPEG-2 extends MPEG-1 to lower sampling frequencies to provide more efficient coding of low-quality signals.
- MPEG-2.5 is an unofficial extension of MPEG-2 to even lower sampling frequencies.
Table 1-1 summarizes the sample rates and encoded bit rates supported by the three variations.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Sampling frequencies (Hz)</th>
<th>Bit rates (kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-1</td>
<td>48000, 44100, 32000</td>
<td>32-320</td>
</tr>
<tr>
<td>MPEG-2</td>
<td>24000, 22050, 16000</td>
<td>8-160</td>
</tr>
<tr>
<td>MPEG-2.5</td>
<td>12000, 11025, 8000</td>
<td>8-64</td>
</tr>
</tbody>
</table>

1.1.2 Quality of sound

When encoding at high sample rates, a bit rate of 128kbps gives subjective quality similar to that of a compact disc recording, while at 64kbps the quality is similar to FM radio.
1.2 About the ARM MP3 Encoder

The ARM MP3 Encoder is an optimized software library, designed to efficiently encode MP3 on the ARM processor family. It was developed from the low complexity (LSI) version of the Encoder developed by the Fraunhofer Institute for Integrated Circuits.

The ARM MP3 Encoder is intended to be called from a high-level system-specific application, that is responsible for the following:

- configuring the Encoder
- allocating all workspace required by the Encoder
- providing the input data, in interleaved 16-bit linear PCM format, to the input buffer of the Encoder
- extracting the output bitstream from the output buffer of the Encoder.

The Encoder is distributed with an example application, that reads audio data from a file and writes the bitstream to another file. This is described in detail in Chapter 3 Example Application.

1.2.1 Supported features

The Encoder supports all features specified in MPEG-1, including joint stereo coding. It supports the lower sampling rates specified in MPEG-2 and MPEG-2.5, but does not support the multi-channel extensions of MPEG-2. The output bit rate can be either fixed or variable.

1.2.2 Supported ARM processors

The Encoder requires an ARM processor supporting v4 or higher of the ARM instruction set. All current ARM processors (including the ARM7TDMI, and all ARM9 and ARM10 variants) are supported. Please contact ARM Limited for details of the performance and memory requirements of the Encoder.
This chapter describes the *Application Programmer’s Interface* (API) provided by the ARM MP3 Encoder library. It contains the following sections:
- *Configuration options* on page 2-2
- *Data types* on page 2-5
- *Constants* on page 2-7
- *Functions* on page 2-9.
2.1 Configuration options

The API is specified in the include file mp3enc.h, that is distributed with the library. If there are any differences between the specification in the header file and the description in this chapter, the header file takes precedence.

The following configuration information must be provided by the application to set up the Encoder correctly:

- **Sample rate** on page 2-2
- **Bit rate** on page 2-2
- **Number of channels** on page 2-3
- **Bandwidth** on page 2-3
- **Intensity coding** on page 2-3
- **Huffman search mode** on page 2-3
- **Frame padding mode** on page 2-4
- **Protection mode** on page 2-4
- **Bitstream flags** on page 2-4.

2.1.1 Sample rate

You must specify the sample rate of the signal to encode it correctly. It is expressed in samples per second, and must take one of the values given in Table 1-1 on page 1-3.

2.1.2 Bit rate

The average bit rate of the output bitstream can be either fixed or variable:

**Fixed**

Specifies the target bit rate.

This is specified in bits per second, and must be between 8,000 and 160,000 per channel (that is, between 16,000 and 320,000 for a two-channel stereo signal). The bit rate must also be within the allowed range for the sample rate. See Table 1-1 on page 1-3 for the valid ranges.

**Variable**

Specifies the desired quality of the output.

This is specified as a parameter between 0 and 100, where:

- 0 represents a low bit rate and poor quality
- 100 represents a high bit rate and higher quality.
2.1.3 Number of channels

You can encode either one or two channels. You can specify a different number of input and output channels, and the Encoder either *down-mixes* (combines one or more signals) or duplicates the signals, as appropriate.

--- **Note** ---

Duplicating a mono signal to produce a stereo signal is not recommended, because this is coded less efficiently than the mono signal.

2.1.4 Bandwidth

By default, the Encoder limits the bandwidth of the signal according to the bit rate. This avoids unpleasant high-frequency distortion at low bit rates. Bandwidth limitation is generally regarded as a less offensive distortion. You can override the bandwidth selected by the Encoder.

--- **Note** ---

The bandwidth is always limited to either half the sampling frequency, or 16kHz, whichever is lower.

2.1.5 Intensity coding

The Encoder can use a stereo coding technique known as *intensity coding*. You can exploit similarities between the channels of a stereo signal to increase coding efficiency, and so reduce quantization noise in the output. However, this can introduce artifacts in the stereo imaging, so its use is optional.

--- **Note** ---

Intensity coding is automatically disabled for bit rates above 96kbps.

2.1.6 Huffman search mode

This option specifies the complexity of the final variable-length coding stage. This affects the number of bits required to encode the signal, and this affects the amount of quantization noise. The standard search is fast and generally gives good results. The full search requires more CPU performance, and gives the best possible results.
2.1.7 Frame padding mode

The length of each frame of the bitstream is constrained to be a whole number of bytes. However, to achieve the desired bit rate exactly, the average frame length might have to be a fractional number of bytes. It is therefore necessary either to vary the frame sizes slightly, or to have a bit rate that does not exactly match the desired rate. This can be achieved using the following modes:

**ISO**
This mode inserts an extra byte in some frames to achieve exactly the desired average bit rate.

**NEVER**
This mode never inserts an extra byte, so that the average bit rate is slightly too low.

**ALWAYS**
This mode always inserts an extra byte, so that the average bit rate is slightly too high.

**NEAREST**
This mode selects either NEVER or ALWAYS, whichever gives a bit rate closer to the desired rate.

2.1.8 Protection mode

You can optionally add a *Cyclic Redundancy Check* (CRC) to each frame to allow error detection. This slightly reduces the bit rate available for the signal, and so can slightly degrade the quality at low bit rates.

2.1.9 Bitstream flags

Each MP3 frame contains three flags containing information about the material. The flags are:

**Private**
A general-purpose flag with no defined meaning.

**Copyright**
This is set if the material is copyright-protected.

**Original**
This is set if the material is original, or cleared if it is a copy.
2.2 Data types

The following data types are used by the ARM MP3 Encoder:

- **Internal data structure** on page 2-5
- **Encoder configuration structure** on page 2-5
- **Encoder information structure** on page 2-6.

### 2.2.1 Internal data structure

This is an opaque type containing all the internal read/write data used by the Encoder. This is separated into two blocks:

**State data**
This must be preserved by the application.

**Scratch data**
This might be overwritten by the application between calls to the Encoder.

The internal data structure is declared as follows:

```c
struct MP3_ENCODER_STATE;     /* Must be preserved by the application */
struct MP3_ENCODER_SCRATCH;   /* May be overwritten by the application */
                                /* between calls */

struct MP3_ENCODER
{
    struct MP3_ENCODER_STATE    *State;
    struct MP3_ENCODER_SCRATCH  *Scratch;
};
```

### 2.2.2 Encoder configuration structure

This structure must be provided by the application to configure the Encoder. See **Configuration options** on page 2-2 for a complete description of the options. The structure is declared as follows:

```c
struct MP3ENC_CONFIG
{
    int    sampleRate;      /* encoder sample rate */
    int    nChannelsIn;     /* number of channels on input (1,2) */
    int    nChannelsOut;    /* number of channels on output (1,2) */
    int    fVbrMode;        /* set to true to run variable bitrate */
    int    bitRate;         /* encoder bit rate in bits/sec */
    int    vbrQuality;      /* quality for variable bitrate 0: worst, 100: best */
    int    bandwidth;       /* targeted audio bandwidth in Hz - 0 for default */
    int    fIntensity;      /* allow usage of intensity stereo if set */
};
```
2.2.3 Encoder information structure

This structure provides information about the Encoder that is only known after initialization.

```c
struct MP3ENC_INFO
{
    int bandwidth;       /* audio bandwidth in Hz */
    int bitRate;          /* actual bit rate */
    int delay;            /* encoder delay in units of sample frames */
    int bufferSizeMin;    /* minimum size of output buffer (bytes) */
};
```
2.3 Constants

The ARM MP3 Encoder defines the following constants, as shown in Table 2-1:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp3encEncoderStateSize</td>
<td>Defined at run time</td>
<td>Size of the internal state data structure, in bytes</td>
</tr>
<tr>
<td>mp3encEncoderScratchSize</td>
<td>Defined at run time</td>
<td>Size of the internal scratch data structure, in bytes</td>
</tr>
<tr>
<td>MP3ENC_BLOCKSIZE</td>
<td>576</td>
<td>Number of samples per channel to encode at a time</td>
</tr>
<tr>
<td>MP3ENC_OK</td>
<td>0</td>
<td>Function return code, indicating no error</td>
</tr>
<tr>
<td>MP3ENC_ERROR_INIT</td>
<td>Not 0</td>
<td>Illegal configuration parameters</td>
</tr>
<tr>
<td>MP3ENC_ERROR_NSAMPLES</td>
<td>Not 0</td>
<td>Too many samples in the input buffer</td>
</tr>
<tr>
<td>MP3ENC_ERROR_BUFSIZE</td>
<td>Not 0</td>
<td>Output buffer is too small</td>
</tr>
<tr>
<td>MP3ENC_ERROR_INTERNAL</td>
<td>Not 0</td>
<td>Internal Encoder error</td>
</tr>
<tr>
<td>MP3ENC_PADDING_ISO</td>
<td>N/A</td>
<td>Use ISO padding mode</td>
</tr>
<tr>
<td>MP3ENC_PADDING_NEVER</td>
<td>N/A</td>
<td>Use NEVER padding mode</td>
</tr>
<tr>
<td>MP3ENC_PADDING_ALWAYS</td>
<td>N/A</td>
<td>Use ALWAYS padding mode</td>
</tr>
<tr>
<td>MP3ENC_PADDING_NEAREST</td>
<td>N/A</td>
<td>Use NEAREST padding mode</td>
</tr>
</tbody>
</table>

2.3.1 Controlling the usage of writable memory

The constants mp3encEncoderStateSize and mp3encEncoderScratchSize are declared as run-time constants exported from the library. They are always the same for a particular version of the library, but might change between versions. They are provided to give the application control over all writable memory usage. You can allocate the memory for the Encoder data statically or dynamically, as in the following examples:

- Dynamic allocation on page 2-8
- Static allocation on page 2-8.
Dynamic allocation

To use dynamic memory allocation, you allocate the required number of bytes:

```c
struct MP3_ENCODER MP3_Encoder;

MP3_Encoder.State = (struct MP3_ENCODER_STATE *) malloc(mp3encEncoderStateSize );
MP3_Encoder.Scratch = (struct MP3_ENCODER_SCRATCH *) malloc(mp3encEncoderScratchSize );
```

Static allocation

To use static memory allocation, you first determine the size of the library by writing a small program to print the value of `mp3encEncoderStateSize` and `mp3encEncoderScratchSize`. A suitable program is as follows:

```c
#include <stdio.h>
#include "mp3enc.h"

int main(void)
{
    printf("Encoder state size is %d\n", mp3encEncoderStateSize);
    printf("Encoder scratch space is %d\n", mp3encEncoderScratchSize);
    return 0;
}
```

You can then define compile-time constants with these values. It is advisable to include a run-time check to ensure that the values are still valid:

```c
/* statically allocate the space using the compile-time constants */
static char        MP3_EncoderState   [ MP3ENC_ENCODER_STATE_SIZE ];
static char        MP3_EncoderScratch [ MP3ENC_ENCODER_SCRATCH_SIZE ];

struct MP3_ENCODER MP3_Encoder = {MP3_EncoderState, MP3_EncoderScratch};

/* ensure the size is correct */
assert( MP3ENC_ENCODER_STATE_SIZE   == mp3encEncoderStateSize );
assert( MP3ENC_ENCODER_SCRATCH_SIZE == mp3encEncoderScratchSize );
```
2.4 Functions

The following functions are provided by the ARM MP3 Encoder:
- `mp3encInit` on page 2-9
- `mp3encGetInfo` on page 2-10
- `mp3encEncode` on page 2-11.

2.4.1 mp3encInit

The `mp3encInit()` function initializes a new Encoder instance. The application must allocate the internal data structure and then call this function before attempting to encode any data.

**Syntax**

```c
int mp3encInit(struct MP3_ENCODER* Mp3Enc, const struct MP3ENC_CONFIG* Config);
```

*where:*

- `Mp3Enc` Is a pointer to the internal data structure of the ARM MP3 Encoder.
- `Config` Is a pointer to a pre-initialized configuration structure.

**Return values**

The function returns the following values:

- `MP3ENC_OK` The initialization was successful.
- `MP3ENC_ERROR_INIT` The configuration structure contained illegal options.
- `MP3ENC_ERROR_INTERNAL` An internal error occurred.
2.4.2 mp3encGetInfo

The mp3encGetInfo() function returns information about the Encoder. It can be called any time after mp3encInit().

Syntax

int mp3encGetInfo(const struct MP3_ENCODER* Mp3Enc, struct MP3ENC_INFO* Info);

where:

Mp3Enc Is a pointer to the internal data structure of the Encoder.
Info Is a pointer to an information structure that the function fills out.

Return values

The function returns the following values:

MP3ENC_OK The information was returned successfully.
MP3ENC_ERROR_INTERNAL
 An internal error occurred.

Returned information

The function returns the following information:

- The bandwidth of the output signal.
  Unless specified otherwise, the bandwidth of the output signal is restricted to avoid unpleasant high-frequency distortion.

- The delay introduced by encoding and decoding a signal.
  This is caused by the delay lines on the filter banks, and also by the need to look ahead in the signal to respond to rapid changes. The result is that, when comparing the original signal with the reconstructed output, a delay of typically about 20-200ms is observed, depending on the sample rate.

- The required size of the output buffer.
  This is the largest amount of bitstream data that can ever be produced when encoding a single block of input data.
2.4.3 mp3encEncode

The mp3encEncode function places a new block of input samples in the Encoder, and collects any new output data.

**Syntax**

```c
int mp3encEncode(struct MP3_ENCODER *Mp3Enc, const short *InputBuffer,
                  int nSamples, unsigned char *OutputBuffer, int nOutputSize,
                  int *nOut);
```

where:

- `Mp3Enc` is a pointer to the internal data structure of the Encoder.
- `InputBuffer` is a buffer of 16-bit linear PCM audio samples, interleaved.
- `nSamples` is the number of valid samples in the input buffer.
- `OutputBuffer` is the buffer where complete frames of bitstream data are written.
- `nOutputSize` is the size of the output buffer, in bytes.
- `nOut` is the total number of bytes written to the output buffer.

**Return values**

The function returns the following values:

- `MP3ENC_OK` The function completed successfully.
- `MP3ENC_ERROR_NSAMPLES` There were too many samples to encode.
- `MP3ENC_ERROR_BUFSIZE` The output buffer might be too small.
- `MP3ENC_ERROR_INTERNAL` An internal error occurred.
Notes

This function must normally be called with nSamples set to MP3ENC_BLOCKSIZE for a mono signal, or twice that for a stereo signal. It is an error to give a higher value.

At the end of a piece of music, if there are fewer than MP3ENC_BLOCKSIZE samples per channel, call the function with nSamples set to the number of remaining samples. The Encoder automatically pads the input with silence.

When the last samples have been encoded, the end of the bitstream might still be buffered inside the Encoder. To extract the final frame(s), call this function with nSamples set to zero as many times necessary until it returns no data (that is, it returns with nOut set to zero). In other words, the application must stop calling this function when the input data is exhausted and the function returns with nOut set to zero.

While the input stream is being read, it is normal for this function to occasionally return zero data, because there is often not enough data to make a whole frame. The partial frame is buffered within the Encoder, and is output as soon as it is complete. When the input stream is exhausted, this function returns data every time it is called until encoding is complete.

On average, there is:

• one frame per input block for sample rates less than 32kHz
• one frame for every two blocks for higher sample rates.

However, this function occasionally returns two blocks, and so the output buffer must be large enough to contain both. The function returns with an error if the buffer is too small.
Chapter 3
Example Application

This chapter describes the example application supplied with the ARM MP3 Encoder library. It contains the following sections:

- *About the example application* on page 3-2
- *Command-line arguments* on page 3-3
- *Benchmarking* on page 3-4.
3.1 About the example application

The ARM MP3 Encoder library is distributed with an example command-line application that:

- demonstrates the use of the library
- allows simple functional tests and benchmarks.

The distribution contains makefiles and Metrowerks CodeWarrior project files, that can be used with the ARM Developer Suite (ADS) to build the application. See the documentation for the ARM Developer Suite for details of building and running ARM applications.

The application runs on any ARM-based system (real or emulated) that supports semihosted file operations.

The example application uses another library, AudioIO.a, to read the data from a variety of common audio file formats. This library is not optimized, and must be linked with the ARM C++ library.

--- Note ---

The AudioIO.a library is intended only for testing purposes, and its use in an embedded application is strongly discouraged.
3.2 Command-line arguments

The example application recognizes the following arguments.

Note

All the numeric arguments have default values if they are omitted. Only the file names are mandatory.

- `b N` Sets the output bit rate in N bits per second (use zero for a variable rate).
- `s N` Sets the input sample rate in N samples per second.
- `w N` Sets the bandwidth in Hz.
- `q N` Sets the quality if you are using a variable bit rate, where N is in the range 0-100.
- `-i` Allows intensity stereo coding.
- `-d` Down-mixes stereo to mono.
- `-v` Sets verbose mode. The output progress information is sent to stderr.
- `-t` Sends the output benchmarking information (timing and memory allocation) to stdout.

`infile` Names the input file in one of the following formats:

- RIFF/WAV
- AIFF
- SND
- raw PCM.

`outfile` Names the output file. This is an MP3 bitstream.
3.3 Benchmarking

The example application can generate basic information about memory and CPU usage.

--- Note ---

The statistics are of limited accuracy, and are intended only as a guide. You must measure the performance characteristics with more accurate methods when designing a system using the library.

---

To generate the information, do the following:

1. Set up the ARMulator with an emulated clock speed of 100kHz, and a core and memory map to match your system.

2. If the system does not have zero wait-state memory, scale the memory access times to match the emulated clock speed. For example, for a 100MHz system with 20ns memory, scale the memory access time to 20µs.

--- Note ---

Beware that this might cause further inaccuracy in the timing information.

---

3. Run the ARM MP3 Encoder with the -t argument for a range of configurations and inputs. This displays:

   • The amount of memory allocated for the buffers and working data during initialization.

   • The peak and average CPU usages at regular intervals. The overall peak CPU usage is given when encoding is complete.
## Glossary

**Bandwidth**
The highest frequency component of a signal.

**Down-mixing**
Combining two or more signals to produce a smaller number of signals.

**kbps**
Kilobits per second.

**Lossy**
A compression scheme that does not allow exact reconstruction of the input.

**Motion Pictures Expert Group**
This is a body responsible for developing standards for coding audio-visual data.

**MPEG**
See Motion Pictures Expert Group.

**MP3**
Is either MPEG-1 or MPEG-2 Audio Layer III, a digital audio compression scheme.

**Opaque type**
A datatype whose contents are not directly accessible.

**PCM**
See Pulse Code Modulation.

**Pulse Code Modulation**
This is a representation of uncompressed digital audio data.

**Quantization**
Restricting a signal to take values from a discrete set of allowed levels.

**Spectrum**
A representation of a signal in terms of its frequency components.
Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A
AIFF format 3-3
ALWAYS frame padding mode 2-4
API 2-2
Application Programmer’s Interface. See API
ARM C++ library 3-2
ARM MP3 Encoder features 1-4
overview 1-4
ARM processors for MP3 1-4
AudI0.a file 3-2

B
Bandwidth 2-3, 3-3
Benchmarking 3-3
generating data 3-4
Bit rate 2-2
adjusting 2-4
Bitstream flags 2-4

C
Channels 2-3
Coding efficiency 2-3
Command-line arguments 3-3
Compression methods 1-2
Configuration options
bandwidth 2-3
bit rate 2-2
bitstream flags 2-4
frame padding 2-4
Huffman search mode 2-3
intensity coding 2-3
number of channels 2-3
protection mode 2-4
sample rate 2-2
Constants 2-7
controlling writable memory 2-7
mp3encEncoderScratchSize 2-7
mp3encEncoderStateSize 2-7
Copyright bitstream flag 2-4
Cyclic Redundancy checks 2-4

D
Data types
Encoder configuration structure 2-5
Encoder information structure 2-6
internal data structure 2-5
Distortion, high frequency 2-3
Down-mixing 2-3, 3-3
Dynamic memory allocation 2-8

E
Encoder
returned information on 2-10
Encoder configuration structure 2-5
Encoder information structure 2-6
Encoding a signal 2-3
Error detection 2-4
Index

F
Files
  AudIo.a 3-2
  mp3enc.h 2-2
Fixed bit rate 2-2
Frame padding mode 2-4
Functions
  mp3encEncode() 2-11
  mp3encGetInfo() 2-10
  mp3encInit() 2-9

H
High-frequency distortion 2-3
Huffman search mode 2-3

I
Initializing a new Encoder instance 2-9
Input file format 3-3
Input sample rate 3-3
Input samples 2-11
Intensity coding 2-3, 3-3
Internal data structure 2-5
ISO frame padding mode 2-4

J
Joint stereo coding 1-4

L
Layer III compression 1-2

M
Memory allocation 3-3
dynamic 2-8
static 2-8
Mono to stereo signals 2-3
MPEG overview 1-2
MPEG standards
  MPEG-1 1-2
  MPEG-2 1-2
  MPEG-2.5 1-2
mp3encEncoderScratchSize constant 2-7
mp3encEncoderStateSize constant 2-7
mp3encEncode() function 2-11
mp3encGetInfo() function 2-10
mp3encInit() function 2-9
mp3enc.h file 2-2

N
NEAREST frame padding mode 2-4
NEVER frame padding mode 2-4

O
Original bitstream flag 2-4
Output
  bit rate 1-4
  data 2-11
  file 3-3
  information 3-3
  rate 3-3

P
Private bitstream flag 2-4
Processors for MP3 1-4

Q
Quality of sound 1-3, 2-2, 2-4, 3-3

R
Raw PCM format 3-3
RIFF/WAV format 3-3

S
Sample rate 2-2
Scratch data 2-5

T
Timing information 3-3

V
Variable bit rate 2-2
Verbose mode 3-3

W
Writable memory 2-7

SND format 3-3
Sound quality 1-3, 2-2, 2-4, 3-3
State data 2-5
Static memory allocation 2-8