MP3 Decoder Developer's Guide

DEVMP3-001
APRIL 07

Analog Devices Inc.
www.analog.com

Table of Contents

List of Tables

1 Introduction

This document describes how to use the MP3 decoder library.

1.1 Scope

This guide is intended for C-language developers proficient in the application of the Analog Devices Blackfin family of DSPs and related software tools. It is assumed that the reader is familiar with all aspects of these DSPs.

1.2 Target Platform

The MP3 decoder library was designed for the Analog Devices Blackfin DSPs.

1.3 Organisation of this Guide

Information supplied in this guide is organised in the following way:

- Section 2 provides the specifications of the MP3 decoder.
- Section 3 is a "Quick Start Guide" about how to integrate the MP3 decoder library.
- Section 4 describes each of the functions in the MP3 decoder library.
- Section 5 describes linking information, memory usage and performance of the MP3 decoder library..

1.4 Version Information

Note that this document refers to the following software versions:
MP3 Decoder: *ADI Version 2.0.x.*
Any information that accompanies the software libraries (in the form of 'readme' files, etc.) should be assumed to supersede the information contained in this document.

1.5 System Requirements

The MP3 decoder module was built and tested with VDSP++ 4.5 February 2007 update of the tools. The versions of the tools are listed in Table 1. To ensure the maximum level of compatibility, please use these versions of the tools.

Tool Name	Tool Description	Tool Version
ccblkfn	C/C++ compiler	7.2.4.3
easmblkfn	BlackFin assembler	2.6.16.5
linker	Linker	3.8.0.4

Table 1 Tool revisions

1.6 Acronyms

ADI	***A**nalog **D**evices **I**nc.*
API	***A**pplication **P**rogram **I**nterface*
CBR	***C**onstant **B**it **R**ate*
VBR	***V**ariable **B**it **R**ate*

1.7 References

[1] ISO/IEC JTC1/SC29/WG11 (MPEG), International Standard ISO/IEC 11172-3 "Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 3: Audio", 1993

[2] ISO/IEC JTC1/SC29/WG11 (MPEG), International Standard ISO/IEC 13818-3 "Information technology - Generic coding of moving pictures and associated audio information - Part 3: Audio", 1998

[3] ISO/IEC JTC1/SC29/WG11 (MPEG), International Standard ISO/IEC 11172-4 "Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s - Part 4: Compliance testing", 1992

[4] ISO/IEC JTC1/SC29/WG11 (MPEG), International Standard ISO/IEC 13818-4 "Information technology - Generic coding of moving pictures and associated audio information - Part 4: Compliance testing", 1994

[5] ID3 parser developer's guide; ADA document DEVMP3-002_ID3_Tag_Parser_Developer_Guide.doc; June 2006.

[6] MP3 Decoder For Blackfin Specification Sheet; ADA-171_MP3_Decoder_Spec_Sheet.doc; March 2007.

1.8 Additional Information

For more information on the latest ADI processors, silicon errata, code examples, development tools, system services and devices drivers, technical support and any other additional information, please visit our website at www.analog.com/processors.

2 Specifications

The Analog Devices MP3 decoder library provides the functionality of decoding MP3 bitstreams. The library was designed for the Blackfin 53x family of DSPs.

2.1 Decoder features

The decoder supports the following:

- All sampling frequencies as specified by ref[1] (ISO/IEC 11172-3, a.k.a. 'MPEG1 MP3', 32, 44.1, and 48kHz) and ref[2] (ISO/IEC 13818-3, a.k.a. 'MPEG2 MP3', 16kHz, 22.05kHz, and 24kHz)
- Support for sampling frequencies of 8kHz, 11.025kHz, and 12 kHz (for an unofficial standard, widely known as 'MPEG2.5 MP3')
- Support for Variable Bit Rates (VBR) of the MP3 standards
- Supports free format bit streams
- Supports mono, stereo and dual channel
- Fully re-entrant and multi-instancing capable
- Optional standalone ID3 tag parser (supports v1.0, v2 tags up to v2.3)

2.2 Input Format

The decoder supports MP3 bit streams as specified in ISO/IEC 11172- 3 and ISO/IEC 13818-3.

2.3 Output Format

The output from the MP3 decoder is 16-bit PCM. If two audio channels are present, the PCM samples from the two channels are interleaved.

2.4 Validation

The MP3 decoder has passed the conformance tests as outlined in ref[3] and ref[4], and the validation is summarized in the following subsections.

2.4.1 Test Vectors Used

All test vectors relevant to layer 3 (and, with a maximum of two audio channels) of the MP3 standard, as specified in ref[3] and ref[4], were used to validate the decoder. Also, a number of bit streams created by various MP3 encoders (including LAME) were used in the validation to test for interoperability.

2.4.2 Compliance Definition

As quoted from ref[3] and similarly in ref[4]:

> To be called an ISO/IEC 11172-3 audio decoder, the decoder shall provide an output such that the rms level of the difference signal between the output of the decoder under test and the supplied reference output is less than 2-15/sqrt(12) for the supplied sine sweep (20Hz-1OkHz) with an amplitude of -2OdB relative to full scale. In addition, the difference signal shall have a maximum absolute value of at most 2-14 relative to full-scale.

This works out as follows:

1) max absolute peak error = 3 in 16-bit pcm sample
2) max RMS error = 8.8×10^{-6}

A diff tool called 'mp3diff' is available that can compare the outputs of the decoder under test and the reference decoder. If any of these limits are exceeded, the diff tool will report the test as a failure.

3 Quick Start

The purpose of this section is to provide a brief description of the MP3 decoder API. This is to aid developers to integrate the MP3 decoder module into their application code with minimal effort.

1. Section 3.1 provides a step by step usage guide for the MP3 decoder.
2. Section 3.2 describes how to create multiple instance of the MP3 decoder.
3. Section 3.3 discusses some system integration issues when using the MP3 decoder.

3.1 Step-by-step Guide

1. Include the *adi_mp3_dec.h* header. Declare 3 data arrays as follows. These data arrays are required for the correct operations of the decoder. Note that the array names are not required to be the names used below. These arrays are used by the decoder to hold instance data as well as used for scratch during decoding.

```
int OnchipBlock0[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK0];
int OnchipBlock1[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK1];

int OnchipBlock0Scratch[ADI_MP3_DEC_SCRATCH_ON_CHIP_BLOCK0];
```

2. Declare an input bit stream buffer and an output PCM buffer of the predefined lengths:

```
adi_mp3_dec_bitstream_data_t input_buffer[ADI_MP3_DEC_MIN_BITSTREAM_BUFSIZE];
adi_mp3_dec_audio_data_t output_buffer[ADI_MP3_DEC_MAX_OUTBUFSIZE];
```

3. Declare two *adi_mp3_dec_memory_pool_t* objects. These objects are used by the decoder for managing the data blocks declared in the previous step. For instance:

```
adi_mp3_dec_memory_pool_t  state;
adi_mp3_dec_memory_pool_t  scratch;
```

4. Initialize the two *adi_mp3_dec_memory_pool_t* objects declared in step 3. One object is used for managing the private memory used by the decoder, while the other object is used for managing the scratch memory, i.e.:

```
/* state memory */
state.pOnChipBlock0  = (int *)OnchipBlock0;
state.pOnChipBlock1  = (int *)OnchipBlock1;

/* scratch memory */
scratch.pOnChipBlock0   = (int *)OnchipBlock0Scratch;
```

5. Declare a pointer of type *adi_mp3_dec_instance_t*. This pointer will point to a newly created MP3 decoder instance.

6. Create a new MP3 decoder instance by calling the function: *adi_mp3_dec_create()*. If a new instance has been successfully created, the function returns the address of the new instance, otherwise a NULL address is returned. This instance creation only needs to be done once unless the instance is destroyed. The instance does not need to be destroyed and recreated for every new mp3 stream.

7. Declare two configuration objects for decoder configuration, i.e.:

```
adi_mp3_dec_config_t requested_config;
adi_mp3_dec_config_t actual_decoder_config;
```

   The first configuration object is for users to set configurations while the second configuration object will hold what the actual decoder configurations are after calling the initialization function. A frame property object also needs to be declared to hold the properties of each frame, i.e.:

```
adi_mp3_dec_frame_properties_t frame_properties;
```

   Set up the configuration object member variables as shown below. Note that the *use_circular_buffer* flag is set to 1 as the input buffering mode has to be circular (linear mode is not supported in this version). The *noMoreDataWithApp*

member variable is used by the application to inform the decoder that there is no more data available at the end of the bit stream. At initialization, this variable should be set to 0.

```
requested_config.noMoreDataWithApp = 0;

requested_config.input_buffer_length = ADI_MP3_DEC_MIN_BITSTREAM_BUFSIZE;
requested_config.input_buffer_base_addr = input_buffer;
requested_config.frame_properties = &frame_properties;
requested_config.max_bytes_to_search_for_sync = ADI_MP3_DEC_MAX_BYTES_TO_SEARCH_FOR_SYNC;
requested_config.use_circular_buffer = 1;

dec_output_status.frame_properties_ptr = &frame_properties;
```

8. The instance is initialized using *adi_mp3_dec_init*() using the configuration object from the previous step. This initialization has to be done before decoding each new mp3 stream.

9. The decoder is now ready to start decoding. Call *adi_mp3_decoder()* to decode the MP3 bit stream. Note that before start decoding, the input buffer does not need to be filled with mp3 data, so the first call to the decode function would result in the return of the error code ADI_MP3_DEC_NEED_MORE_DATA (see the next step).

10. The decode function returns an error code. The error code gives an indication of whether any errors have occurred during the decoding process. The user application should check the error code first to determine if the decoding process was successful. If not enough data was present in the input buffer, the decoding function would return an error code ADI_MP3_DEC_NEED_MORE_DATA, and the user would need to fill the input buffer with more data before calling the decoding function again (section 3.3.1 describes what the user application needs to do to manage the input buffer). All the status information for the current mp3 frame are returned in a structure of type *adi_mp3_dec_output_status_t*. Information such as the number of output PCM samples per channel and the number of output channels can be found in this structure. Repeat steps 9 and 10 until all the mp3 frames have been decoded, at which point the decoding function would return an error code of ADI_MP3_DEC_STREAM_COMPLETE.

3.2 Multi-instancing

By multi-instance:

1. A MP3 instance means calling *adi_mp3_dec_init()* per new input stream and repeated *adi_mp3_decoder()* calls until the end of the stream.
2. Each MP3 instance processes a different input stream (or can be the same stream, but treated as a separate one).
3. Each MP3 instance has its own scratch memory. The instances can share a common pool of scratch memory, however for re-entrant functionality, the following points need to be considered:
   - The scratch memory should be persistent for the complete frame decoding. It should not be over-written or corrupted by another instance within this frame decode.
   - If the DSP is controlled by an OS, a decoder thread should be interrupted by another only if the frame decode in that thread is completed.

   If complete re-entrancy is required, each instance should have its own dedicated scratch memory.

Further details about the usage of the MP3 library for multi-instance:

1. Include the adi_mp3_dec.h header. Declare 2 persistent data arrays per instance as follows. These data arrays are required for the correct operations of the decoder. Note that the array names are not required to be the names used below.
   For example:

```
int OnchipBlock0_Instance1[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK0];
int OnchipBlock1_Instance1[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK1];

for the first instance

and

int OnchipBlock0_Instance2[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK0];
int OnchipBlock1_Instance2[ADI_MP3_DEC_STATE_ON_CHIP_BLOCK1];

for the second instance and so on.
```

2. Declare 1 scratch data arrays per instance, if the instances are not sharing the scratch memory, as follows. These data arrays are required for the correct operations of the decoder. Note that the array names are not required to be the names used below. For example:

```
int OnchipBlock0Scratch_Instance1[ADI_MP3_DEC_SCRATCH_ON_CHIP_BLOCK0];

for the first instance

and

int OnchipBlock0Scratch_Instance2[ADI_MP3_DEC_SCRATCH_ON_CHIP_BLOCK0];

for the second instance and so on.
```

3. If the instances are to share the scratch memory, only one set of scratch data arrays needs to be declared. Note that the array names are not required to be the names used below. For example:

```
int OnchipBlock0Scratch[ADI_MP3_DEC_SCRATCH_ON_CHIP_BLOCK0];
```

4. Declare two *adi_mp3_dec_memory_pool_t* objects for each instance. These objects are used by the decoder for managing the data blocks declared in the previous step. For instance:

```
adi_mp3_dec_memory_pool_t  pPrivateMemory_Instance1;
adi_mp3_dec_memory_pool_t  pScratchMemory_Instance1; and so on.
```

5. Declare two *adi_mp3_dec_config_t* objects per instance, like *actual_decoder_config* and so on. These objects hold the decoder configuration information.

6. Initialize the two *adi_mp3_dec_memory_pool_t* objects of all the instances declared in step 4. One object is used for managing the private memory used by the decoder, while the other object is used for managing the scratch memory:

```
/* state memory */
state_instanceX.pOnChipBlock0 = (int *)OnchipBlock0_instanceX;
state_instanceX.pOnChipBlock1 = (int *)OnchipBlock1_instanceX;

 /* scratch memory */
scratch_instanceX.pOnChipBlock0 = (int *)OnchipBlock0Scratch or
                                        OnchipBlock0Scratch_instanceX;
```

   Where instanceX is instance1 or instance2 that are declared in steps 1 and 2.

7. Declare a pointer of type *adi_mp3_dec_instance_t* per instance. These pointers will point to the newly created MP3 decoder instances. If a new instance has been successfully created, the set up function returns the address of the new instance, otherwise a NULL address is returned.

8. Create the new MP3 decoder instances by calling the initialization function: *adi_mp3_dec_init()* and pass the respective instance parameters as arguments.

   Ex:
```
int adi_mp3_dec_init(adi_mp3_dec_instance_t *decoder_instance,
             const adi_mp3_dec_config_t *req_dec_config,
             adi_mp3_dec_config_t *actual_decoder_config)
```

9. The decoders are now ready to start decoding. Call *adi_mp3_decoder()* for each instance, passing respective instance parameters as arguments to decode the given MP3 bit streams. Note that each decoder instance should have its own input and output buffers.

3.3 System Integration

This section highlights some system integration issues the system integrator needs to be aware of when integrating the MP3 decoder library.

3.3.1 Input Buffering

This section illustrates how to do input buffering with the MP3 decoder.

The MP3 decoder uses a circular input buffer to retrieve compressed audio data. The configuration object holds most of the information for the user application to do circular buffering. When filling the input buffer, the user application needs to determine:

1. Where the current read pointer is. This information can be retrieved from *dec_read_ptr* within the *adi_mp3_dec_output_status_t* structure. This pointer shows the current reading position of the decoder. Note that this pointer is read-only and the user application should not change this pointer.
2. How many bytes the decoder has consumed in the decoding call. This information is returned by the decoder in the variable *num_input_bytes_consumed* in the status output structure after every call.
3. Where the current write pointer is. The user application should declare a pointer variable to keep track of this.
4. The base address of the input buffer – This is the *input_buffer_base_addr* in the configuration object. This variable should not be changed after configuration.
5. The length of the input buffer – This is the *input_buffer_length* variable in the configuration object. This variable should not be changed after configuration.

With this information, the user application can calculate the amount of free space available in the buffer and copy appropriate amount of data into the correct location within the buffer. The function *User_fill_buf()* in the example code provided in the package illustrates one way of filling data. It always fills the input buffer provided there is enough new data left. Note that the size of the input buffer needs to be at least ADI_MP3_DEC_MIN_BITSTREAM_BUFSIZE bytes in size for the decoder to handle all valid bit rates up to 320kb/s.

Note that when the user application detects that there is no more data left to be processed, it should inform the decoder by using the function *adi_mp3_dec_reconfigure()* with the configuration item NOMORE_DATA_WITHAPP and a configuration value of 1. This configuration effectively sets the *noMoreDataWithApp* flag in the configuration to 1. Note that this configuration needs to be done before calling the decoder again.

4 MP3 Decoder Programmer's Reference

This section provides a detailed description of the MP3 decoder API structures and functions.

4.1 API Functions

4.1.1 adi_mp3_dec_create()

4.1.1.1 Prototype

```
void* adi_mp3_dec_create(
        adi_mp3_dec_memory_pool_t *state,
        adi_mp3_dec_memory_pool_t *scratch);
```

4.1.1.2 Description

This function is to be used to create a single instance of the decoder.

4.1.1.3 Parameters

Name: state
Type: adi_mp3_dec_memory_pool_t*
Direction: Input
Description: Memory pool used to store decoder internal state information.

Name: scratch
Type: adi_mp3_dec_memory_pool_t*
Direction: Input
Description: Memory pool for storing decoder internal scratch information.

Return Value: The function returns a non-void pointer upon successful creation of the instance, otherwise a NULL pointer is returned. The latter is returned if the input pointers to the memory pools themselves are NULL, or if the user allocated buffers fail to be aligned correctly. Note that all user declared buffers for the state and scratch pools are to be aligned on a 32-bit boundary.

4.1.2 adi_mp3_dec_init()

4.1.2.1 Prototype

```
int adi_mp3_dec_init(
        adi_mp3_dec_instance_t *decoder_instance,
        const adi_mp3_dec_config_t *requested_dec_config,
        adi_mp3_dec_config_t *actual_dec_config_used);
```

4.1.2.2 Description

This function must be called to initialize the decoder instance created by the *adi_mp3_dec_create()* function. Part of this initialization uses a set of configuration parameters passed in by the application via the pointer *requested_dec_config*. The definition of its type **adi_mp3_dec_config_t** is given in section 4.3.4.

4.1.2.3 Parameters

Name: decoder_instance
Type: adi_mp3_dec_instance_t*
Direction: Input/output
Description: The decoder instance.

Name: requested_dec_config

Type: const adi_mp3_dec_config_t*
Direction: Input
Description: Structure containing configuration parameters.

Name: actual_dec_config_used
Type: adi_mp3_dec_config_t*
Direction: Output
Description: Structure containing the actual configuration parameters used in the initialization.

Return value: The function returns the code given below:
ADI_MP3_DEC_SUCCESS

See section 4.1.4.3 for description of the error codes.

4.1.3 adi_mp3_dec_resynch()

4.1.3.1 Prototype

```
void adi_mp3_dec_resynch(
        adi_mp3_dec_instance_t *decoder_instance);
```

4.1.3.2 Description

This function is to be used when the application needs to reset the decoder to a known state. Typically this will be required when fast-forward and rewind operations are executed. The decoder will be initialized to the same state as if you were to call the function **adi_mp3_dec_init()**, with one difference. That is application configuration parameters cannot be reset. The pre-existing configuration parameters are to be assumed. In the even that new configuration parameters are required, it can be done so using the supplied function **adi_mp3_dec_reconfigure()**. Note that the resynch function also resets the input buffer pointers, so fresh data should be added to the input buffer from the start of the buffer.

4.1.3.3 Parameters

Name: decoder_instance
Type: adi_mp3_dec_instance_t*
Direction: Input/output
Description: Takes as input the decoder instance and returns the updated instance.

Return value: None.

4.1.4 adi_mp3_dec_reconfigure()

4.1.4.1 Prototype

```
void adi_mp3_dec_reconfigure(
        adi_mp3_dec_instance_t *decoder_instance,
        const adi_mp3_dec_config_item_t config_item,
        const int32_t config_value,
        adi_mp3_dec_config_t *actual_dec_config_used);
```

4.1.4.2 Description

This function is to be used when on-the-fly run-time configuration of the decoder is desired. Note reconfiguration is only allowed on frame boundaries.

Note although this implies reconfiguration can be performed on every frame, it does not make sense to do so and the application must be careful about this. Reconfiguration is typical when switching from one bit stream to another, without the decoder needing

to be reinitialized by calling the function **adi_mp3_dec_init()**. In the case if configuration parameters need to be reset, this function is to be called.

Note not all configuration parameters can be configured on the fly (i.e. on a frame by frame basis). See the definition of **adi_mp3_dec_config_item_t** for all reconfigurable parameters.

When reconfiguration of a single parameter is desired, *config_item* must contain the enumerated name of the chosen parameter and its value specified in *config_value*.

Note that only NOMORE_DATA_WITHAPP configuration item is supported in this version of the decoder.

4.1.4.3 Parameters

Name: decoder_instance
Type: adi_mp3_dec_instance_t*
Direction: Input/output
Description: Decoder instance.

Name: config_item
Type: const adi_mp3_dec_config_item_t
Direction: Input
Description: Configuration parameter. Enumerated type.

Name: config_value
Type: const int
Direction: Input
Description: The value of the specified configuration item to be set. For all configuration parameters this must be the pointer to the configuration structure, typecasted to an integer value.

Name: actual_dec_config_used
Type: adi_mp3_dec_config_t*
Direction: Output
Description: Structure containing the actual configuration parameters used in the reconfiguration.

Return value: None

4.1.5 adi_mp3_decoder()

4.1.5.1 Prototype

```
int adi_mp3_decoder(
        adi_mp3_dec_instance_t *decoder_instance,
        const int32_t num_input_bytes,
        const adi_mp3_dec_bitstream_data_t *input_bitstream,
        adi_mp3_dec_output_status_t *dec_output_status,
        adi_mp3_dec_audio_data_t *output_pcm_channels[]);
```

4.1.5.2 Description

This function is the main decoder routine, producing PCM data for each decoded channel. The decoder is designed to be called with an arbitrary number of bytes in the input buffer, leaving the decision making of whether enough input data is available to the decoder.

The following usage conditions must be satisfied for the correct usage of the decoder:-

1. For every frame to be decoded, the first time the decoder is called with some input data, *input_bitstream* must be pointing to the beginning of the frame. It is the user's responsibility to determine the start of the data within the buffer.

Frames provided are placed in the input buffer contiguously, the start of subsequent frames in the bit stream does not need to be determined as the decoder returns the pointer to the next byte to be consumed.

Note *input_bitstream* should not be confused with the base address of the application buffer that is passed as part of the configuration parameters in the function **adi_mp3_dec_init()**. This parameter, passed to the decoder shall always be pointing to the next byte to be consumed by the decoder.

2. For every frame, the decoder needs the entire frame (the size of each frame dependents on the bit rate of the MP3 stream) in the input buffered to perform decoding. If this is not satisfied, the decoder returns to the application with the value ADI_MP3_DEC_NEED_MORE_DATA, asking for more input data.

4.1.5.3 Parameters

Name: dec_instance
Type: void*
Direction: Input/Output
Description: Decoder instance address.

Name: num_input_bytes
Type: const int
Direction: Input
Description: The number of unprocessed bytes in the input buffer. The application must not assume the decoder will process (consume) this many words from the input buffer. Depending on the state of the decoder it may consume, nil, some or all of the data in the input buffer.

The number of bytes consumed as well as the pointer to the next consumed word is outputted by the decoder in its output status structure called of type **adi_mp3_dec_output_status_t**.

Name: input_bitstream
Type: const adi_mp3_dec_bitstream_data_t*
Direction: Input
Description: The starting address of the unprocessed MP3 compressed data within the input buffer. The input buffer must contain at least *num_input_bytes*. The decoder will ignore additional data in this buffer.

Name: dec_output_status
Type: adi_mp3_dec_output_status_t*
Direction: Output
Description: Output status information returned from the decoder. See definition in section 4.3.6.

Name: output_pcm_channels[]
Type: adi_mp3_dec_audio_data_t*
Direction: Output
Description: Must be an array of at least one element. Additional elements in the array will be ignored. Each element must contain the pointer to where the decoder shall write the 1152 output PCM samples of each channel. Each PCM sample is a 16-bit word. For the MP3 decoder, only one output data array of length ADI_MP3_DEC_MAX_OUTBUFSIZE words is required. For stereo output, the left and right channel samples are interleaved.

Note only for specific return values from the decoder will the decoder have written PCM data to the channel buffers. The definition of each return value in section 4.2 defines whether data is written to these buffers or not.

Return value: The function returns one of the following error codes:

- ADI_MP3_DEC_SUCCESS
- ADI_MP3_DEC_NEED_MORE_DATA
- ADI_MP3_DEC_STREAM_COMPLETE
- ADI_MP3_DEC_INVALID_FILE

See section 4.2 for a description of these error codes.

4.2 Error Codes

The following table lists all the valid error codes for the decoder library.

Error code	Description
ADI_MP3_DEC_SUCCESS	Decoding was successful, there may be PCM output
ADI_MP3_DEC_TIMEOUT_REACHED	Time out has been reached, this error code is only valid in the evaluation build of the MP3 decoder library.
ADI_MP3_DEC_NEED_MORE_DATA	Decoder needs more input data, no PCM output
ADI_MP3_DEC_STREAM_COMPLETE	Decoding of the entire stream is complete, there may be PCM output
ADI_MP3_DEC_INVALID_CONFIG_PARAMS	Invalid configuration parameter(s) specified
ADI_MP3_DEC_INVALID_FILE	The bit stream being decoded is an invalid MP3 bit stream, no PCM output
ADI_MP3_DEC_FREE_FORMAT	No longer used. Retained for backward compatibility.

Table 2 Error codes

4.3 API Data Structures

4.3.1 adi_mp3_dec_bitstream_data_t

typedef char adi_mp3_dec_bitstream_data_t;

4.3.1.1 Description

This type is to be used to represent the input bitstream to the decoder.

4.3.2 adi_mp3_dec_audio_data_t

typedef short adi_mp3_dec_audio_data_t;

4.3.2.1 Description

This type is to be used to represent PCM samples output from the decoder. Each PCM sample produced is a 16-bit integer.

4.3.3 adi_mp3_dec_config_item_t

```
typedef enum {
    NOMORE_DATA_WITHAPP     = 1,
    USE_CIRCULAR_BUFFER     = 2,
    INPUT_BUFFER_LENGTH     = 3,
    ALL_CONFIG_PARAMS       = 4
} adi_mp3_dec_config_item_t;
```

4.3.3.1 Description

This enumerated type contains all the configuration parameters can be reconfigured on the fly (i.e. on a frame by frame basis) as well as the value ALL_RECONFIG_PARAMS to be used when all reconfiguration parameters needs to be reconfigured. Note that the current version of the library only supports the configuration item NOMORE_DATA_WITHAPP. The other configuration items are ignored.

Enumeration	Description
NOMORE_DATA_WITHAPP	Configure whether there is still more data to be processed
USE_CIRCULAR_BUFFER	Configure input buffering mode
INPUT_BUFFER_LENGTH	Configure the input buffer length
ALL_CONFIG_PARAMS	Configure all reconfigure parameters

Table 3 Configuration items

4.3.4 adi_mp3_dec_config_t

```
typedef struct {
    int  noMoreDataWithApp;
    int  use_circular_buffer;
    int  input_buffer_length;
    int  max_bytes_to_search_for_sync;
    adi_mp3_dec_bitstream_data_t *input_buffer_base_addr;
    adi_mp3_dec_frame_properties_t *frame_properties;
} adi_mp3_dec_config_t;
```

4.3.4.1 Description

This structure is to be used to store all configuration parameters.

Variable	Description
noMoreDataWithApp	The incoming MP3 data stream has reached its end. This needs to be set to 1 when the incoming data is exhausted
use_circular_buffer	Use input circular buffering. Should always be set to 1.
input_buffer_length	Input buffer length for circular buffering in bytes.
max_bytes_to_search_for_sync	Maximum number of bytes into the bit stream for searching a valid MP3 sync word before giving up.
input_buffer_base_addr	Input buffer base address, used for circular buffering.
frame_properties	Pointer pointing to the frame properties object.

Table 4 Configuration structure members

The *max_bytes_to_search_for_sync* variable is used to impose a search range for finding the frame sync. This search range is to be specified in number of bytes. If the decoder fails to find two consecutive valid frame syncs within this specified range, the bit stream is deemed to be an invalid MP3 bit stream. If the MP3 bit stream to be decoded contains ID3 v2 tags, it is recommended to skip these tag information and feed the start of the audio data to the decoder (if the ID3 tag data is fed into the decoder, they are ignored). This way the decoder would not mistakenly declare the bit stream as an invalid MP3 stream if the amount of ID3 data exceeded the search range.

4.3.5 adi_mp3_dec_memory_pool_t

```
typedef struct {
        void *on_chip_block0;
        void *on_chip_block1;
        void *on_chip;
        void *off_chip;
} adi_mp3_dec_memory_pool_t;
```

4.3.5.1 Description

This structure contains pointers to memory pools that need to be passed to the decoder via the function *adi_mp3_dec_create()*.

4.3.6 adi_mp3_dec_output_status_t

```
typedef struct {
    int32_t frame_completed;
```

```
    int32_t num_output_channels;
    int32_t num_output_samples_per_channel;
    adi_mp3_dec_bitstream_data_t *dec_read_ptr;
    adi_mp3_dec_frame_properties_t *frame_properties_ptr;
    int32_t num_input_bytes_consumed;
} adi_mp3_dec_output_status_t;
```

4.3.6.1 Description

This structure is returned by the call to the main decoder function **adi_mp3_decoder()**. Most of its members are currently not used and these variables will be defined in the future release of the decoder library.

Variable	Description
frame_completed	Set to 1 if a frame has been decoded
num_output_channels	Number of output audio channels
num_output_samples_per_channel	Number of PCM samples in the output buffer for each audio channel
dec_read_ptr	Current decoder read pointer (for input)
num_input_bytes_consumed	Number of input data bytes consumed by the decoder in the last decoding call
frame_properties_ptr	Address of the frame properties object

Table 5 Output status members

4.3.7 adi_mp3_dec_frame_properties_t

```
typedef struct {
    int32_t nSampleRate;
    int32_t iBitRate;
    int32_t nMpegType;
    int32_t nLayerNum;
    int32_t num_output_channels;
    int32_t isvbr;
    uint32_t iFrameDuration;
    uint32_t iCurrentFrameSize;
    uint32_t avg_bitrate_in_bytes_persec;
} adi_mp3_dec_frame_properties_t;
```

4.3.7.1 Description

This structure contains information about the bit stream being decoded. The decoder fills the properties in this structure once it has enough input data to determine these properties. The values are updated after each MP3 frame has been completely decoded.

Variable	Description
nSampleRate	Sampling rate (Hz)
iBitRate	Bit rate (bits/s)
nMpegType	MPEG type (1 for MPEG-1, 2 for MPEG-2, 3 for MPEG-2.5)
nLayerNum	Layer number (i.e. 3 for L3)
num_output_channels	Number of output channels (either 1 or 2)
isvbr	Flag set if the MP3 stream is VBR, cleared for CBR. Note that it takes several frames for this flag to be correctly set once decoding has commenced
iFrameDuration	Current frame duration in ms
iCurrentFrameSize	Current frame size in bytes
avg_bitrate_in_bytes_persec	Average bit rate in unit of bytes/sec. Note that it takes several frames before this value is updated.

Table 6 Frame properties members

5 Sections & Linking

This section describes the memory sections in the MP3 decoder library targeting the Blackfin processor. Some code and data sections should be placed in L1 memory whenever possible to achieve best performance. Table below lists the code and data sections and their ideal placement in memory. For optimal performance, the instruction and data caches should be enabled.

Section Name	Code or Data	Ideal Memory Location for Placement
MP3_code	Code	L1 (On-chip)
MP3_const0	Data	L1 (On-chip)
MP3_const1	Data	L1 (On-chip)

Table 7 Memory sections

The decoder utilizes the state and scratch memory blocks allocated by the application. Where these arrays are allocated in memory has significant impact on performance. For optimal performance, all of these arrays (2 state and 1 scratch) should be placed in L1 data memory. If there is insufficient L1 data memory to accommodate all of these buffers, then these buffers should be selectively placed into the available L1 data memory in the following priority:

1. Scratch block 0
2. State block 1
3. State block 0

There are instances where the user application wishes to place all MP3 code and data in L3 SDRAM. This is often the case when using Blackfin DSPs which have small on-chip memories such as BF-531. In those cases, it is recommended to enable both instruction and data caches. The data cache is ideally configured in write-back mode for the memory regions where the 2 state and 1 scratch buffers (as described previously) reside.

5.1 MIPS & Memory Footprints

For details on MIPS and memory footprints of the MP3 decoder, please refer to [6].