MP4 PARSER DEVELOPER'S GUIDE

ANALOG DEVICES, INC.

www.analog.com

KT-195 (REV 1.6, 2006-07-20)

Table of Contents

List of Figures

List of Tables

Copyright, Disclaimer & Trademark Statements

Copyright Information

Disclaimer

Trademark and Service Mark Notice

1 Introduction

This document describes how to use the MP4 file parser library.

1.1 Organisation of this Guide

- Section 1 this section contains the introduction.
- Section 2 lists the specifications of the MP4 Parser Library
- Section 3 is the quick start guide.
- Section 4 contains the programmer's reference.
- Section 5 lists the memory usage of the library.

1.2 Version Information

When this document refers to the mp4 parser library it is referring to the following version -:

MP4 File Parser: ADI version 1.2.x

Please note that any information supplied with the software library (in the form of 'readme' files etc), should be assumed to supersede information in this document.

1.3 System Requirements

This module was built and tested with VDSP++ 4.5 February 2007 update:

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 Tools revision (as reported using the –version command line option)

1.4 Acronyms

ADI	Analog Devices Inc.
API	Application Program Interface
DRM	Digital Rights Management
UTF-8	Unicode Transformation Format (8 bit character encoding)

1.5 References

[1] International Standard ISO/IEC 14496-12 "Information technology - Coding of audio-visual objects - Part 12: ISO base media file format", ISO/IEC JTC1/SC29 (MPEG-4), 2005

[2] International Standard ISO/IEC 14496-14 “Information technology - Coding of audio-visual objects - Part 14: MP4 file format”, ISO/IEC JTC1/SC29 (MPEG-4), 2003(E)

[3] International Standard ISO/IEC 14496-1 “Information technology - Coding of audio-visual objects - Part 1: Systems”, ISO/IEC JTC1/SC29 (MPEG-4), 2004(E)

[4] International Standard ISO/IEC 14496-3 “Information technology - Coding of audio-visual objects - Part 3: Audio”, ISO/IEC JTC1/SC29 (MPEG-4), 2005(E)

1.6 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 MP4 parser library parses MP4 file streams as specified in the ISO specification [1]. MPEG-4 files, 3GPP files and Quicktime files are all complying derivatives of this ISO base file format specification. The parser parses all required MP4 objects ('boxes') for the decoding of the media data (mdat) object. Due to the large number of MP4 object types, a limited set is supported. The parser handles only those containers that may possibly contain one of the supported objects. The following objects and containers are parsed by the MP4 parser:

Box	**Description**	**In container**
ftyp	Identifies the specifications to which this file complies	-
hdlr	Declares the process by which the media-data in the track is presented, and thus, the nature of the media in a track Only streams containing a single audio track are handled.	moov →trak →mdia [1]
stsd	The sample description table gives detailed information about the coding type used, and any initialization information needed for that coding Only a single audio coder is supported.	moov →trak →mdia →minf →stbl
esds	Contains configuration information about the audio data (sampling frequency, number of channels, etc.)	stsd →mp4a
ilst	Contains iTunes metadata (QuickTime format) The parser is capable of extracting the following bibliographical information (metadata): • Song title • Artist • Album • Year • Comments • Genre	udta →meta
dref	Checked to ensure that the 'mdat' box is present in the file	moov →trak →mdia →minf →dinf

[1] This means that hdlr must be in mdia in trak in moov.

mdhd	Contains timing related information for the audio track (timescale).	moov →trak →mdia
stts stss stsc stco stsz	Time-to-sample box Sync-sample box Sample-to-chunk box Chunk-offset box Sample-size box These boxes together contain information that allows the parser to seek to a given location and return file-access units.	moov →trak →mdia →minf →stbl
drms	The presence of this box indicates that Apple DRM Protection has been applied to the media content in the track. The parser is unable to decode the audio configuration information for such protected content.	moov →trak →mdia →minf →stbl →stsd

Table 2: Objects supported by the MP4 parser module

In addition, the parser module acts as a demultiplexer, in extracting the raw audio bitstream from the file for the AAC decoder to process. The audio data is arranged in chunks, which consist of one or more audio frames to be decoded. Audio chunks are contained within 'mdat' boxes, and may be interleaved with other multi-media data within the file.

Although it is possible for the parser to stop once all the required objects have been parsed, the parser is designed to consume all data input to it from the system. This is necessary because there is no standard order in where all the various boxes are located within a file, relative to each other. It also allows for the possibility of the parser simply identifying sections which the system itself can process, or for the user to extend the parser to process objects not listed in the table above.

3 Quick Start

The following is a step by step guide for integrating the MP4 library.

3.1 Data and variables

1. Include the *adi_mp4ad_parser.h* header file in the application.
2. The parser needs to use a block of memory to store state data. Declare a memory buffer of length ADI_MP4AD_FASTB0_PRIO0_STATE_MEM_NUMBYTES. This memory buffer should be of type *unsigned char* (i.e. 8-bit words).
3. Declare an object of type *adi_mp4ad_metadata_t*, and memory buffers for holding the bibliographical information. Each bibliographical field requires its own buffer. The size of each buffer is flexible and determined by the user application based on the amount of memory available in the system. For instance, to set up the song title object within the (*adi_mp4ad_metadata_t*) *text_fields* object:

   text_fields.song_title.tag_data = song_title;

   text_fields.song_title.max_num_bytes = 20;

   where *song_title* is the array declared in step 3 to hold the song title and 20 is the length of this *song_title* array in bytes. If a bibliographical field is not required to be extracted, then no memory array for that field is required and any pointers to bibliographical information should be set to NULL.
4. Declare an input data buffer of type *adi_mp4ad_bitstream_data_t*. The large indexing boxes, stts, stss, stsc, stco and stsz may each have only part of their data in the input buffer for parser processing and the parser will ask for more data when required. However for the remaining boxes, the current implementation requires the whole box to be in the input buffer if it is processed by the parser. It is recommended to set the size of the input buffer equal to the size required by these smaller boxes, typically 2 Kbytes should be enough for most files.
5. Declare an object of type *adi_mp4ad_config_t*, to be used to configure the parser.
6. Declare an object of type *adi_mp4ad_container_info_t.* This object will be used to hold the information parsed from the MP4 file container.
7. Declare a memory buffer for the storage of indexing information. The size of this buffer is set by the user and passed to the parser module (via the data field index_mem_num_bytes in the structure adi_mp4ad_config_t, see also 4.1.2). Typically for a 6 minute long track, the index tables can consume between 25 and 100 KB of memory.
8. Declare an object of type *adi_mp4ad_mem_t*. This object holds the parser state and scratch memory blocks.
9. Declare an instance pointer of type *adi_mp4ad_instance_t*. This will be used to hold the address of the parser instance.

3.2 Initialization

10. Set up the memory objects declared in step 8. Only one field needs to be set up:
    - adi_mp4ad_state_mem_t.fastb0.prio0 – this should be set to the starting address of the state memory block declared in step 2.
11. Call the function *adi_mp4ad_create()* passing the memory object addresses in as the argument. This function creates a parser instance and returns the instance handle. If an error occurred during the creation process, a NULL pointer is returned.
12. Set up the configuration object declared in step 5. The member variables to be set up are:
    - use_circular_bitstream_buffer – set this flag to 1 to enable the parser to use the input buffer with circular addressing, otherwise 0 if linear input buffers are used.
    - circular_buffer_base – this is the address of the start of the circular input buffer declared in step 4. (only required if use_circular_bitstream_buffer set to 1)
    - circular_buffer_length – this is the length of the input buffer in bytes declared in step 4. (only required if use_circular_bitstream_buffer set to 1)
    - text_fields_ptr – pointer to structure adi_mp4ad_metadata_t (declared in step 3) to hold the text fields of metadata
    - container_info_ptr – pointer to structure adi_mp4ad_container_info_t (declared in step 6) to hold the information parsed from the stream
    - index_mem_ptr, index_mem_num_bytes – (optional) void pointer to a memory block that holds indexing and seeking information, and its length in bytes. This memory block must be provided if the audio data in the file is to be decoded.
13. Initialize the parser instance by calling the *adi_mp4ad_init()* function, passing in the instance handle and the configuration object address as arguments. At this point, the parser is ready to parse an MP4 file.

3.3 Parsing

14. Call the parsing function *adi_mp4ad_parser()*. This function parses whatever data is in the input buffer and returns an error code.
15. If ADI_MP4AD_UNKNOWN_OBJECT is returned to the user, then a box that is not handled by the parser is in the input buffer. Normally the user can safely skip over this box by adjusting the input pointer to start just after the unknown box, or by flushing the buffer and skipping the remaining bytes of the box in the file itself. The relevant fields from *adi_mp4ad_output_status_t* are:
    - unconsumed_input_data_ptr – points to the start of the unknown box in the input buffer
    - next_object_length_num_bytes – indicates the length of the unknown box in number of bytes

- next_object_id – the 4 byte identification code (box type) of the unknown box.

16. If ADI_MP4AD_NEED_MORE_INPUT is returned, a partial box has been encountered at the end of the input buffer and the user needs to ensure that the buffer is filled with the whole box (of length *next_object_length_num_bytes*) before the parser can proceed.

    However for the five indexing boxes, stts, stss, stsc, stco and stsz, the parser does not assume that the input buffer has the full remaining data and the parser will process what is available in the input buffer and request more data as required. In this case, the field *next_object_length_num_bytes* will reflect the number of bytes remaining in the box to be processed by the parser.

17. When the last of the box data is being input it is recommended that at least the next 8 bytes after the box to be processed is also input to the parser so it can check whether the next box needs to be processed or not before returning to the user.

18. The system should repeatedly go back to step 14 until either a fatal error is encountered or there is no more data from the file to process (i.e. the whole MP4 file must be parsed before decoded). All error codes are described in section 4.5.4 . In addition, possible non-fatal warning codes returned in *adi_mp4ad_output_status_t.warning_info* are listed in Table 12.

3.4 Second pass parsing

19. If the preprocessor macro SMALL_INDEX_MEMORY_MODEL is defined in the parser library source code when it is compiled, then it may be necessary to reparse a small section of the stream to complete the indexing table. If the function adi_mp4ad_is_index_table_complete() returns zero, then this is the case. See section 4.5.5 for more information

3.5 Decoding File-access units

Audio data are arranged into chunks which consist of a number of raw audio frames stored contiguously in the file. Chunks may have other data between them, which makes it necessary to use the indexing information transmitted in the file format to demultiplex the audio data from the file.

1. The module supplies the file offset and length of the next file-access unit (chunk) of audio data to be decoded. To achieve this, a call to *adi_mp4ad_get_next_audio_data_unit*() is made in the main decoding loop.
2. If ADI_MP4AD_SUCCESS is returned, the file may be 'sought' to the file offset *adi_mp4ad_output_status_t.data_unit_stream_offset*, at which point a chunk is to be found of length *adi_mp4ad_output_status_t.data_unit_length_bytes* bytes. The timing information of the audio frame at the start of the chunk is contained in *adi_mp4ad_output_status_t.data_unit_time_ms*.
3. Once *adi_mp4ad_output_status_t.data_unit_length_bytes* bytes have been processed by the decoder, to reconstruct a number of audio frames, the system should get the next file-access unit.

3.6 File seeking

1. The module can also be used to seek to an arbitrary random access position within the file. A call to *adi_mp4ad_seek*() is made in the main audio decoding loop.
2. If ADI_MP4AD_SEEK_IGNORED is returned, there is no need to seek to a new location. If the sought point is beyond the end of the audio bitstream, the last access unit will be pointed to instead.
3. If ADI_MP4AD_SUCCESS is returned, the file is 'seeked' to the file offset *adi_mp4ad_output_status_t.data_unit_stream_offset*, which corresponds to the presentation time *adi_mp4ad_output_status_t.data_unit_time_ms*, and the decoder input buffer is filled from this position, until *adi_mp4ad_output_status_t.data_unit_length_bytes* bytes have been read from the file.
4. Once *adi_mp4ad_output_status_t.data_unit_length_bytes* bytes have been processed by the decoder, then the system can proceed once more as described in section 3.5 to continue decoding the audio.

3.7 Accessing metadata

The MP4 boxes may contain specific information about the audio track in its 'meta' objects. An example of such bibliographic (or tag) information is the song title. Each tag is stored by the parser in a structure of type **`adi_mp4ad_tag_text_t`** which has three members.

*tag_data	a pointer to the tag data
max_num_bytes	the maximum number of bytes allocated for the tag
tag_length_num_bytes	the actual length of the tag within the file (this could be larger than max_num_bytes, in which case tag_data will only contain the first max_num_bytes-1 characters)

- The available tags are grouped together into a metadata structure of type **`adi_mp4ad_metadata_t`**.

3.7.1 Bibliographical Information

The current implementation of this module only supports the iTunes tagging format. A small set of tags are parsed, as listed in Table 3. All tags are represented using the UTF-8 character set.

Tag Name	Tag Location	Data Type
Song Title	adi_mp4ad_metadata_t.song_title	adi_mp4ad_tag_text_t
Artist	adi_mp4ad_metadata_t.artist	adi_mp4ad_tag_text_t
Album	adi_mp4ad_metadata_t.album	adi_mp4ad_tag_text_t
Year	adi_mp4ad_metadata_t.year	adi_mp4ad_tag_text_t
Comments	adi_mp4ad_metadata_t.comments	adi_mp4ad_tag_text_t
Genre	adi_mp4ad_metadata_t.genre	adi_mp4ad_tag_text_t

Table 3: Bibliographical Info

3.8 Input Buffering

The parser provides input buffer related information to the user in a structure of type **`adi_mp4ad_output_status_t`**. The following fields are used:

num_input_bytes_consumed	Number of bytes consumed during the previous call
unconsumed_input_data_ptr	Pointer to the next unread input data byte in the buffer
next_object_length_num_bytes	Number of bytes remaining to be processed in the next object to be parsed
next_object_id	The 4-byte id of the next MP4 box

Table 4: Data fields in the structure adi_mp4ad_output_status_t

After every call to the parser function, the parser returns the number of bytes consumed in the input buffer in the parsing process (**`num_input_bytes_consumed`**). The user application can use this information and the unconsumed byte pointer (**`unconsumed_input_data_ptr`**) to determine how much space is left in the input buffer, hence how many bytes of new data can be written into the input buffer before the next parser call. The argument **`num_input_bytes`** in the **`adi_mp4ad_parser()`** function always specifies the number of **unprocessed** bytes in the input buffer while the argument **`input_bitstream`** should always point to the next **unprocessed** byte in the buffer.

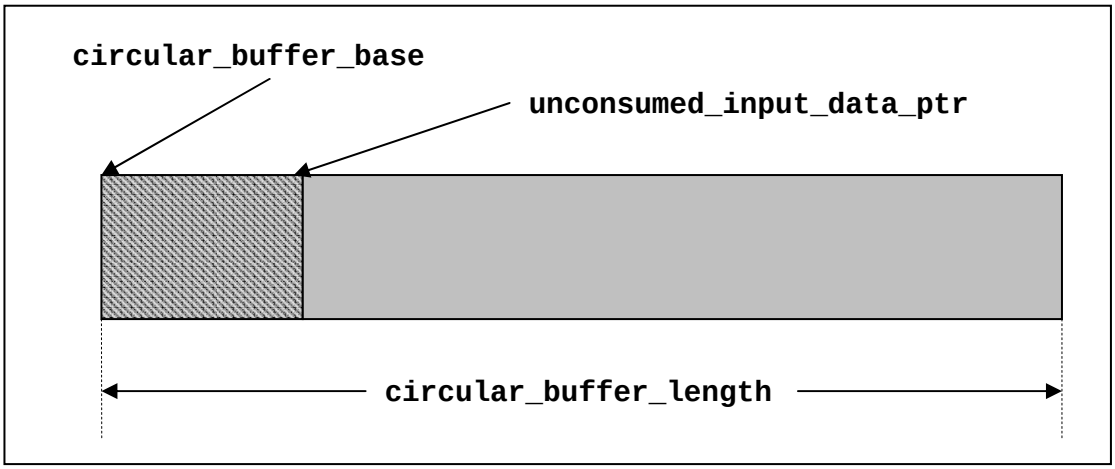

Figure 1: MP4 parser input buffer

3.9 AAC Audio Bitstream Configuration

The MPEG 4 File Format contains some information required to configure an AAC Decoder before the audio data can be correctly decoded. This information is contained in the 'esds' box, and corresponds to the Elementary Stream Descriptor syntax as defined in MPEG-4 Systems [3]. All information contained in such an 'esds' box for an audio track is made available to the user through the embedded data structure **`adi_mp4ad_esds_t esds`**, within **`adi_mp4ad_container_info_t`**.

The parser indicates that the MPEG 4 file contains AAC audio, by setting the following values:

`adi_mp4ad_container_info_t.esds.is_parsed = 1`

`adi_mp4ad_container_info_t.esds.audio_object_type = ADI_MP4AD_AACLC_MPEG4_OBJECT or ADI_MP4AD_SBR_MPEG4_OBJECT`

If the parser completes processing of the file but **`is_parsed`** is equal to 0, then the MPEG-4 audio configuration information contained in the 'esds' box was not found in the file. If the field **`audio_object_type`** is not equal to ADI_MP4AD_AACLC_MPEG4_OBJECT or ADI_MP4AD_SBR_MPEG4_OBJECT, then the file contains a bitstream syntax that will not be recognized by an AAC-LC or HE-AAC decoder.

The ADI AAC Decoder example code illustrates which audio configuration parameters parsed from the MPEG 4 file format are required to set up the ADI AAC Decoder.

Some useful pieces of information for the system integrator include the following:

`adi_mp4ad_container_info_t.esds.max_bitrate` – the maximum bitrate in bits per second within a time window of one second duration.

`adi_mp4ad_container_info_t.esds.avg_bitrate` – if zero then the track contains a VBR bitstream, otherwise it is the average bitrate in bits per second for the track.

`adi_mp4ad_container_info_t.mdhd.timescale` – the number of time units per second defined for the track.

`adi_mp4ad_container_info_t.mdhd.duration` – the length of the track in number of time units – dividing this value by the timescale parameter gives the length of the track in seconds.

3.10 Example Code

For example MP4 parser application pseudocode, see section 4.6 . Here, non-standard tags are skipped.

3.11 Detecting single or multiple audio tracks

The parser currently supports only streams containing a single audio track. To ensure this, the parser detects all handlers of the stream ('soun', 'vide', 'hint') and within the 'hdlr' box and aborts the parsing if multiple 'soun' handlers are detected.

This check must be modified in the source file hdlr.c for the parser to handle multiple audio track files. Track specific data also has to be kept separate for each track, and so code will need to be added to the parser so it knows which track a current box belongs to.

3.12 Using less memory for audio indexing

The indexing information for an audio track allows seeking and random access to an encoded *sample* (the amount of information supplied to the decoder each call). These samples are held in

chunks, which are the minimum unit read from a file. Often, long streams comprise many samples and the indexing information can require more than 1MB of memory for audio files longer than an hour. This can be alleviated by defining inside the parser library source code the preprocessor macro SMALL_INDEX_MEMORY_MODEL, which restricts random access to chunk-granularity (rather than to sample-granularity) by using a restricted set of indexing tables. See section 4.5.5 for more information.

4 Programmer's Reference

This section contains a detailed description of the MP4 parser API.

4.1 API Data Structures

4.1.1 adi_mp4ad_mem_t

void *	state.prio0	Pointer to block of memory for the parser. Block must contain at least ADI_MP4AD_FASTB0_PRIO0_STATE_MEM_NUMBYTES bytes

Table 5: Data field in the structure adi_mp4ad_mem_t

4.1.2 adi_mp4ad_config_t

adi_mp4ad_bitstream_data_t*	circular_buffer_base;	Pointer to lowest memory address of input bitstream buffer
int	circular_buffer_length	Length in bytes of the memory allocated in the input bitstream buffer
unsigned int	use_circular_bitstream_buffer	If nonzero, use circular input buffer.
adi_mp4ad_metadata_t*	text_fields_ptr	Pointer to structure containing pointers to strings for metadata parsing. Set pointer fields to NULL if metadata parsing is not required.
adi_mp4ad_container_info_t*	container_info_ptr	Pointer to structure to hold information parsed from the stream
void*	index_mem_ptr	Pointer to a memory block to hold indexing and seeking information. It is necessary to provide this storage if the parser is to be used to provide access to the raw audio file access units. This pointer can be set to NULL if the user only requires parsing of the file headers.
int	index_mem_num_bytes	The number of bytes in the memory block pointed to by index_mem_ptr. The minimum size is ADI_MP4AD_MIN_ INDEX_MEM_NUMBYTES.

Table 6: Data fields in the structure adi_mp4ad_config_t

4.1.3 adi_mp4ad_output_status_t

unsigned int	warning_info	Bitfield containing parsing warnings, as defined in adi_mp4ad_warning_info_bitfield_t
adi_mp4ad_bitstream_data_t*	unconsumed_input_data_ptr	Pointer to the first unconsumed byte from the bitstream
long int	num_input_bytes_consumed	The number of bytes parsed in the last call to adi_mp4ad_parser()
int	next_object_length_num_bytes	The number of bytes required to parse the next box (some boxes may still be able to be parsed even if it will not fit within the input buffer)
int	next_object_id	The 32-bit id of the next box to be parsed. Most box-ids can be interpreted as 4-byte unterminated ASCII strings, and is stored in this field in big-endian byte order
long int	data_unit_stream_offset	The stream byte offset returned when seeking or getting a data access unit
long int	data_unit_time_ms	The time in milliseconds returned when seeking or getting a data access unit
int	data_unit_length_bytes	The length in bytes of a particular data unit within the container format.
int	index_mem_used_bytes	The number of bytes that has been actually used to store indexing information parsed from the input MP4 file. In the case there is not enough memory allocated, this field will contains a negative number whose the absolute value represents the number of additional bytes needed to be provided to store all indexing tables in the MP4 container.

Table 7: Data fields in the structure adi_mp4ad_output_status_t

4.1.4 adi_mp4ad_container_info_t

adi_mp4ad_*xxxx*_t	*xxxx*	Embedded structure containing information parsed from the box *xxxx* (where *xxxx* is an MP4 box id). See the header file for full details of what information is available in each of these structures.

Table 8: Data fields in the structure adi_mp4ad_container_info_t

4.1.5 adi_mp4ad_module_info_t

unsigned int	version_num	The major-minor-patch version number of this library in the following format: `0xMMNNPP00`, where 0xMM is the major revision number in hex (MSB), 0xNN is the minor revision number in hex, 0xPP is the patch revision number in hex, and the LSB is reserved to be 0x00.
char*	version_text	A pointer to a NULL terminated string which identifies the specific module (and variant), and the version information of this library.

Table 9: Data fields in the structure adi_mp4ad_module_info_t

4.2 Enumerations

4.2.1 adi_mp4ad_return_code_t

ADI_MP4AD_SUCCESS	The parse operation was successful, no action needs to be taken.
ADI_MP4AD_TERMINAL_FAILURE	A catastrophic error has occurred. The user application should not call the parser again before initializing it.
ADI_MP4AD_NEED_MORE_INPUT	More data is required in the input buffer. The user application should fill the input buffer.
ADI_MP4AD_INVALID_CONFIG_PARAMS	The supplied configuration parameters were not incorrect and over-ridden by the parser.
ADI_MP4AD_OPERATION_FAILED	A requested operation has not competed successfully. The user application should not call the parser again before initializing it.
ADI_MP4AD_INVALID_BITSTREAM	An invalid MP4 stream is encountered, stop the parsing process.
ADI_MP4AD_STREAM_COMPLETE	Returned by the function adi_mp4ad_get_next_audio_data_unit() to indicate that there are no more file access units to process from the file.
ADI_MP4AD_UNKNOWN_OBJECT	An unsupported MP4 box was encountered. The user application must handle or skip this object.
ADI_MP4AD_SEEK_IGNORED	The seek operation was successful, but no action needs to be taken because the current location is close to the seek location.

ADI_MP4AD_SEEK_TABLES_NOT_ALLOCATED	The seek or get next access unit operation was not successful, because the user did not supply enough memory to hold the indexing tables

Table 10: Possible values of the enumeration adi_mp4ad_return_code_t

4.2.2 adi_mp4ad_config_item_t

ADI_MP4AD_ALL_RECONFIG_PARAMS	Reconfigure all parameters from the supplied parameter set

Table 11: Possible value of the enumeration adi_mp4ad_config_item_t

4.2.3 adi_mp4ad_warning_info_bitfield_t

ADI_MP4AD_WARNING_NONE	No warnings present
ADI_MP4AD_WARNING_INDEX_MEMORY_TOO_SMALL	The memory allocated for storing the indexing information is not enough for the random access indexing information. It is highly recommended that the MP4 file should be re-parsed with a larger storage memory (for indexing tables) to prevent early termination of the decoding process later. The magnitude of the output status field index_mem_used_bytes indicates how much extra memory is needed in order to successfully store the index information. Typically for a 6 minute long track, the index tables can consume between 25 and 100 KB of memory
ADI_MP4AD_WARNING_BRANDS_NOT_LISTED	The list of compatible brands has not been stored due to the insufficient storage memory provided.
ADI_MP4AD_WARNING_DRM_PROTECTED	This flag indicates that the media data has been DRM protected. Currently only Apple DRM protected files are recognized. No DRM decryption is performed by the parser.

Table 12: Warning bits in adi_mp4ad_warning_info_bitfield_t

4.3 Definitions

4.3.1 ADI_MP4AD_FASTB0_PRIO0_STATE_MEM_NUMBYTES

The number of bytes to assign to the parser in the call to adi_mp4ad_create().

4.3.2 ADI_MP4AD_MAX_COMPATIBLE_BRANDS

The maximum number of compatible brands to parse from the 'ftyp' box.

4.3.3 ADI_MP4AD_HDLR_NAME_MAX_CHARS

The maximum number of characters to parse for the name field of the 'hdlr' box.

4.3.4 ADI_MP4AD_AACLC_MPEG4_OBJECT

The MPEG-4 audio object type identification number which signals that the audio data contained in the file conforms to the AAC-LC bitstream syntax.

4.3.5 ADI_MP4AD_SBR_MPEG4_OBJECT

The MPEG-4 audio object type identification number which signals that the audio data contained in the file conforms to the HE-AAC bitstream syntax.

4.4 Global constants

4.4.1 adi_mp4ad_module_info

extern const adi_mp4ad_module_info_t adi_mp4ad_module_info;

This constant global struct holds information about the library. See Table 9 for details of the struct.

4.5 API Functions

The following are the functions used to operate the parser.

4.5.1 adi_mp4ad_create

```
adi_mp4ad_instance_t* adi_mp4ad_create (
        adi_mp4ad_mem_t* per_instance_mem_blocks
)
```

Creates a new parser instance within the supplied memory blocks.

Parameter	per_instance_mem_blocks	Pointer to structure containing state and scratch memory. State memory is unique to each instance, and scratch memory is common for all instances
Return	NULL	If not enough memory is available for the instance
	<pointer to instance>	Otherwise

4.5.2 adi_mp4ad_init

```
adi_mp4ad_return_code_t adi_mp4ad_init (
        adi_mp4ad_instance_t *parser_instance,
        adi_mp4ad_config_t *config_params
)
```

Initialises the parser.
This function must be called after adi_mp4ad_create(), and before parsing starts on a new bitstream.

Parameter	*parser_instance	Pointer to the instance
	*config_params	Pointer to structure containing configuration parameters
Return	ADI_MP4AD_INVALID_CONFIG_PARAMS	If configuration parameters are not valid
	ADI_MP4AD_SUCCESS	Otherwise

4.5.3 adi_mp4ad_reconfigure

```
adi_mp4ad_return_code_t adi_mp4ad_reconfigure (
   adi_mp4ad_instance_t *instance_handle,
   adi_mp4ad_config_item_t config_item,
   adi_mp4ad_config_t *config_params
)
```

Reconfigures the parser after parsing has started.
NOT CURRENTLY IMPLEMENTED

Parameter	*instance_handle	Pointer to the instance
	config_item	The item to modify in the configuration parameter structure
	*config_params	Pointer to structure containing configuration parameters
Return	ADI_MP4AD_INVALID_CONFIG_PARAMS	
	ADI_MP4AD_SUCCESS	

4.5.4 adi_mp4ad_parser

```
adi_mp4ad_return_code_t adi_mp4ad_parser(
   adi_mp4ad_instance_t *instance_handle,
   int num_input_bytes,
   adi_mp4ad_bitstream_data_t *input_bitstream,
   adi_mp4ad_output_status_t *parser_status
)
```

Attempts to parse boxes from the bitstream.

Parameter	*instance_handle	Pointer to the instance
	num_input_bytes	The number of input bytes available from the input bitstream pointer

	*input_bitstream	Pointer to the start of the unconsumed bitstream data.
	*parser_status	Pointer to a structure containing status information that changes between calls
Return	ADI_MP4AD_INVALID_BITSTREAM	An invalid MP4 stream is encountered, stop the parsing process.
	ADI_MP4AD_NEED_MORE_INPUT	More data is required in the input buffer. The user application should fill the input buffer.
	ADI_MP4AD_UNKNOWN_OBJECT	An unsupported MP4 box was encountered. The user application must handle or skip this object.
	ADI_MP4AD_TERMINAL_FAILURE	A catastrophic error has occurred. The user application should not call the parser again before initializing it.
	ADI_MP4AD_STREAM_COMPLETE	The parser has competed the second pass of the parsing when using a small index table. See section 4.5.5

4.5.5 adi_mp4ad_is_index_table_complete

```
int adi_mp4ad_is_index_table_complete (
        adi_mp4ad_instance_t *instance_handle,
        int      *remaining_index_offset,
        int      *remaining_index_length
)
```

Returns a boolean indicating if the entire indexing information has been parsed.

Parameter	*instance_handle	Pointer to the instance
	*remaining_index_offset	Pointer to a variable to return the file offset of the remaining index information.
	*remaining_index_length	Pointer to a variable to return the length of the remaining index information.
Return	0	More indexing information must be parsed from the file.
	1	All indexing information has been parsed from the file.

If the preprocessor macro SMALL_INDEX_MEMORY_MODEL is defined in the parser library source code, the parser will use chunk boundaries for seeking and random access, instead of using sample boundaries. For streams using one sample per chunk, this will have little effect, however longer streams often use many samples per chunk and so a considerable memory saving is possible. In the project file provided with the MP4 file parser library source code, this macro would be defined in the DPJ compiler pre-processor settings.

If SMALL_INDEX_MEMORY_MODEL is defined, the parser ignores the stsz.entry_size table that holds the lengths of each sample and instead creates a stco.chunk_length table to hold the length of each chunk. To do this, it is necessary that both stsc and stco boxes are parsed before the stsz box. If these boxes are not contained in this order in the file, it is necessary to parse the stsz box

in a second parsing pass. A return value of 0 from function adi_mp4ad_is_index_table_complete() when called after the initial parsing loop indicates that this is the case, and the application should:

- Seek the input file to the file location remaining_index_offset
- Read the number of bytes given by remaining_index_length
- Call adi_mp4ad_parser() to complete the parsing operation. Several calls may be required for streams with a large stsz box. When remaining_index_length bytes have been parsed, the parser returns ADI_MP4AD_STREAM_COMPLETE

Subsequent access and seeking of the stream will then be at chunk boundaries. Note that the function adi_mp4ad_seek() will return slightly different file offsets and lengths compared to the case where the macro SMALL_INDEX_MEMORY_MODEL is not defined.

4.5.6 adi_mp4ad_seek

```
adi_mp4ad_return_code_t adi_mp4ad_seek (
        adi_mp4ad_instance_t *parser,
        int seek_time_ms,
        int curr_time_ms,
        int curr_stream_byte_offset,
        adi_mp4ad_output_status_t *status
)
```

Returns the file offset and length of the file access unit immediately before the required seek time

Parameter	*parser	Pointer to the instance
	seek_time_ms	The seek time in milliseconds
	curr_time_ms	The current time of the stream, in milliseconds
	curr_stream_byte_offset	The current file offset of the stream
	*status	Pointer to a structure containing the file offset and length of the file access unit immediately before the required seek time
Return	ADI_MP4AD_OPERATION_FAILED	It is not possible to seek to the desired seek time
	ADI_MP4AD_SEEK_IGNORED	The seek operation was successful, but no action needs to be taken because the current location is close to the seek location.
	ADI_MP4AD_SEEK_TABLES_NOT_ALLOCATED	It is not possible to seek to the desired seek time because the user did not supply a memory block to store the seek tables, at the time of initialising the parser.
	ADI_MP4AD_SUCCESS	The desired seek time has been found and its index/length returned. If the sought point is beyond the end of the audio

		bitstream, the last access unit will be returned instead.

4.5.7 adi_mp4ad_get_next_audio_data_unit

adi_mp4ad_return_code_t adi_mp4ad_get_next_audio_data_unit (
    adi_mp4ad_instance_t *parser,
    adi_mp4ad_output_status_t *status
)
Returns the file offset and length of the next file access unit in the stream

Parameter	*parser	Pointer to the instance
	*status	Pointer to a structure containing the file offset and length of the next file access unit
Return	ADI_MP4AD_OPERATION_FAILED	It is not possible to progress to the next file access unit
	ADI_MP4AD_SEEK_TABLES_NOT_ALLOCATED	It is not possible to progress to the next file access unit because the user did not supply a memory block to store the indexing tables, at the time of initialising the parser.
	ADI_MP4AD_SUCCESS	The next file access unit has been found and its index/length returned
	ADI_MP4AD_STREAM_COMPLETE	The final file access unit has already been returned, and there are no more access units remaining to return

4.6 Operation

Pseudo-code for operating the parser is shown below.
The parsing loop terminates when required by the calling application.

```
// Create instance
adi_mp4ad_instance_t* adi_mp4ad_parser_instance = adi_mp4ad_create (&parser_memory);
if (adi_mp4ad_parser_instance == NULL)
       Exit()  // Cannot create instance

// Input buffer initialisation for parser instance
requested_parser_config.circular_buffer_base = parser_input_buffer;
requested_parser_config.circular_buffer_length = INPUT_BUFFER_LENGTH;
requested_parser_config.use_circular_bitstream_buffer = 1;

// Set memory pointers for parsed metadata
PARSER_STATE_text_fields.song_title.tag_data = PARSER_METADATA_SONGTITLE;
PARSER_STATE_text_fields.artist.tag_data = PARSER_METADATA_ARTIST;
PARSER_STATE_text_fields.album.tag_data = PARSER_METADATA_ALBUM;
PARSER_STATE_text_fields.year.tag_data = PARSER_METADATA_YEAR;
PARSER_STATE_text_fields.genre.tag_data = PARSER_METADATA_GENRE;
PARSER_STATE_text_fields.comments.tag_data = PARSER_METADATA_COMMENTS;
// Set maximum string lengths for parsed metadata
```

```
PARSER_STATE_text_fields.song_title.max_num_bytes = ADI_MP4AD_METADATA_SONGTITLE_NUMBYTES;
PARSER_STATE_text_fields.artist.max_num_bytes = ADI_MP4AD_METADATA_ARTIST_NUMBYTES;
PARSER_STATE_text_fields.album.max_num_bytes = ADI_MP4AD_METADATA_ALBUM_NUMBYTES;
PARSER_STATE_text_fields.year.max_num_bytes = ADI_MP4AD_METADATA_YEAR_NUMBYTES;
PARSER_STATE_text_fields.genre.max_num_bytes = ADI_MP4AD_METADATA_GENRE_NUMBYTES;
PARSER_STATE_text_fields.comments.max_num_bytes = ADI_MP4AD_METADATA_COMMENTS_NUMBYTES;

requested_parser_config.text_fields_ptr = &PARSER_STATE_text_fields;

// set pointers to data struct and memory block for information parsed from the file
requested_parser_config.container_info_ptr = &PARSER_STATE_atoms;
requested_parser_config.index_mem_ptr = &PARSER_STATE_atoms_indexing;
requested_parser_config.index_mem_num_bytes = PARSER_INDEXING_SIZE_BYTES;

parser_error = adi_mp4ad_init (      adi_mp4ad_parser_instance, &requested_parser_config);
switch (parser_error)
{
       case ADI_MP4AD_SUCCESS:
       {
              // successful initialisation of parser
              break;
       }
       case ADI_MP4AD_INVALID_CONFIG_PARAMS:
       {
              printf("Parser config params were changed - continuing on\n");
              break;
       }
       case ADI_MP4AD_TERMINAL_FAILURE:
       {
              printf("Cannot initialise the parser with invalid configuration parameters \n");
              exit (1);
       }
}

// Parse bitstream
fill_input_buffer();
while (not end of input file)
{

       // Parse the buffer
       parser_error = adi_mp4ad_parser (    adi_mp4ad_parser_instance,
                                            parser_available_bytes,
                                            input_buffer_ptr,
                                            &parser_output_status);
       switch (parser_error)
       {
              case ADI_MP4AD_NEED_MORE_INPUT:
                     if (input_buffer_is_full)
                            Exit() // (fatal). Buffer too small to parse this required box.
                     fill_input_buffer();
                     break;
              case ADI_MP4AD_INVALID_BITSTREAM:
                     Exit.  // Invalid bitstream
              case ADI_MP4AD_TERMINAL_FAILURE:
                     Exit.  // Fatal
              case ADI_MP4AD_UNKNOWN_OBJECT:
                     // The id of the unknown box is in parser_output_status.next_object_id
                     // this code assumes all unknown objects can be simply skipped
                     if (parser_output_status.next_object_length_num_bytes > bytes_left_in_buffer)
                            // Skip (next_object_length_num_bytes - bytes_left_in_buffer) bytes
                            // in file, and flush input buffer
                     Else
                            // Skip next_object_length_num_bytes in the input buffer
                     fill_input_buffer();
                     break;
       }
}

// Is the index table complete?
```

```
if ( !adi_mp4ad_is_index_table_complete(
               adi_mp4ad_parser_instance,
               &remaining_index_offset,
               &remaining_index_length) )
{

       // No, index table is incomplete
       // Reparse from the start of the remainder of the index table

       // Seek the input file to the remaining index table
       fseek (infile, remaining_index_offset, SEEK_SET);
       flush_input_buffer();

       // Reparse from the start of the remaining box(es)
       while (parser_error != ADI_MP4AD_STREAM_COMPLETE)
       {
              // Read up to remaining_index_length bytes from the file
              num_bytes_added  = fill_input_buffer();
              remaining_index_length -= num_bytes_added;
              if (num_bytes_added == 0)
              {
                   printf("Could not read more data before index tables could be finalised.\n");
                   Exit;
               }

              // Parse the buffer
              parser_error = adi_mp4ad_parser (    adi_mp4ad_parser_instance,
                                          parser_available_bytes,
                                          input_buffer_ptr,
                                          &parser_output_status);

              // check for parser errors
              if (    (parser_error != ADI_MP4AD_NEED_MORE_INPUT)
                   && (parser_error != ADI_MP4AD_STREAM_COMPLETE) )
              {
                     printf("MP4 file parsing error while finalising index tables\n”);
                     Exit;
              }

       }
}
```

Figure 2 Pseudocode for parsing operation

This module is also used to direct the decoding operation by finding the file locations and lengths of the stream's file access units. Pseudocode for the decoding operation is shown below.

```
// Decode loop
is_decode_continue = 1;
while (is_decode_continue)
{
       // Seek to a new time if required
       if (seek to a new location)
       {
              parser_error = adi_mp4ad_seek (      adi_mp4ad_parser_instance,
                                                   seek_time_millisecs,
                                                   curr_time_millisecs,
                                                   curr_stream_offset,
                                                   &parser_output_status);
              switch (parser_error)
              {
                     case ADI_MP4AD_SUCCESS:
                     {
                            // Seek completed
                            curr_time_millisecs = parser_output_status.data_unit_time_ms;
                            curr_stream_offset = parser_output_status.data_unit_stream_offset;

                            // Seek the input file to the desired stream offset
```

```
                        fseek (infile, curr_stream_offset, SEEK_SET);

                        // Fill the decoder's input buffer with the new data at
                        // the desired location
                        flush_buffer();
                        fill_input_buffer(parser_output_status.data_unit_length_bytes);

                        // let the decoder know that the system has caused a jump
                        // to a new location in the bitstream
                        decoder_resynch();

                        break;
                }
                ADI_MP4AD_SEEK_IGNORED:
                {
                        // Ignore seek, move on
                        break;
                }
                case ADI_MP4AD_OPERATION_FAILED:
                {
                        printf ("WARNING: Not able to seek to time %d milliseconds.\n",
                                        seek_time_millisecs);
                        break;
                }
                case ADI_MP4AD_SEEK_TABLES_NOT_ALLOCATED:
                {
                        printf ("WARNING: Cannot perform seeking without allocation of memory
                           for indexing tables.\n",
                        break;
                }
                default:
                {
                        printf ("MP4A seek failed.\n");
                }
        }
}

// Get the next file access unit if necessary
Else if (bytes remaining in current data unit > 0)
{
        fill_buffer (bytes remaining in current data unit);
}

Else // seek was not performed and current chunk has been fully processed
{
        parser_error = adi_mp4ad_get_next_audio_data_unit (
                                adi_mp4ad_parser_instance,
                                &parser_output_status);
        switch (parser_error)
        {
                case ADI_MP4AD_SUCCESS:
                {
                        curr_stream_offset = parser_output_status.data_unit_stream_offset;
                        // Seek to the next chunk
                        fseek (infile, curr_stream_offset, SEEK_SET);
                        // Fill the decoder's input buffer from the desired location
                        flush_buffer ();
                        fill_input_buffer(parser_output_status.data_unit_length_bytes);
                        break;
                }
                case ADI_MP4AD_STREAM_COMPLETE:
                {
                        printf("Decoder has processed the whole song from the Audio File.\n");
                        is_decode_continue = 0;
                        continue;
                }
                case ADI_MP4AD_SEEK_TABLES_NOT_ALLOCATED:
                {
```

```
                        printf ("WARNING: Cannot perform seeking without allocation of memory
                           for indexzing tables.\n",
                        break;
                }
                default:
                {
                        is_decode_continue = 0;
                        printf("Parser has failed to return the audio file-access chunk\n");
                        continue;
                }
        }
}

// Decode
decoder_result = decoder ();

// Update current position
curr_stream_offset += Number of bytes consumed by decoder;
curr_time_millisecs +=  ((1000.0 * number of samples produced by the decoder) /
                                sample_rate);

}
```

Figure 3 Pseudocode for decoding operation

4.7 Modifying the parser

New boxes may be parsed by modifying the parser to add new box handlers. The following sections discuss how to do this using a template skeleton box handler.

4.7.1 Adding box handlers

The parser consists of a state machine that controls the parsing of individual boxes. Each box has an associated handler function, which performs the actual parsing of the data from the bitstream. If a box handler is not available for a box, the parser returns ADI_MP4AD_UNKNOWN_OBJECT and expects the application to manually skip the box.
A distinction is made between *box* and *containers*. A box contains information that may be parsed, and a container contains one or more boxes, and allows boxes to be constructed into hierarchies. A simple implementation of a box handler requires all bytes of a box to be contained in the input buffer, or ADI_MP4AD_NEED_MORE_INPUT is returned. A more complex implementation which requires an internal state machine in the box handler is possible if the user needs to minimise the size of the parser buffer. An example of this type of implementation can be found in the code handling the indexing information boxes. Container headers are merely skipped and their component containers/boxes are parsed directly.

The steps required to add a new box handler are shown below. The box's id is assumed to be ASCII 'xxxx':

- If object is a container
  - Add { 'xxxx', 1, parser_function_container } to global variable boxtype_parameters in adi_mp4ad_parser_internal.h
- Else if object is a box
  - Add { 'xxxx', 0, parser_function_xxxx } to global variable boxtype_parameters in adi_mp4ad_parser_internal.h
  - Add parser code to parser_function_xxxx() in xxxx.c. See below for a skeleton for parser_function_xxxx().

- Add xxxx.c to the VisualDSP parser project
- Add a prototype for parser_function_xxxx() to adi_mp4ad_parser_internal.h:

  adi_mp4ad_return_code_t parser_function_xxxx(adi_mp4ad_instance_t *parser);

- Add structure adi_mp4ad_xxxx_t in adi_mp4_parser.h
- Embed structure adi_mp4ad_xxxx_t in adi_mp4ad_container_info_t in adi_mp4_parser.h

4.7.2 Skeleton box handler function

A skeleton parser function is shown below for hypothetical box ‘xxxx’. Bits are parsed from the bitstream using the adi_mp4ad_parser_get_bits () function, where a maximum of 32-bits can be returned per call.

On completion the parser function must return ADI_MP4AD_SUCCESS for parsing to continue. For fully self-contained parser functions, the parser state should be set to ADI_MP4AD_BOXTYPE_OBJECT to indicate that the next bits encountered belong to an atomic object. Other states may be defined if required.

```
#include "adi_mp4ad_parser_internal.h"

FAST_CODE_SECTION
adi_mp4ad_return_code_t parser_function_xxxx(adi_mp4ad_instance_t *parser)
{
	adi_mp4ad_return_code_t result = ADI_MP4AD_SUCCESS;

	// Parse data from bitstream
	parser->atoms.xxxx.field[0] = adi_mp4ad_parser_get_bits(parser, 32);
	parser->atoms.xxxx.field[1] = adi_mp4ad_parser_get_bits(parser, 32);
	...

	// Atom complete
	parser->state.current = ADI_MP4AD_BOXTYPE_OBJECT;

	return result;
}
```

5 Memory & Performance

5.1 Sections & Linking

The following table lists the pre-defined code and data sections for the parser and the ideal placement in memory. Note that because parsing is relatively fast, memory placement is less critical.

Section name	Code or Data	Size (Bytes)	Placement
adi_fast_prio0_code (code)	Code	9470	L1/L2/L3
program	Code	1694	L1/L2/L3
<PRIO 0>	Data[State]	300	L1/L2/L3
constdata	Data[Static]	520	L1/L2/L3

5.2 Memory Usage

The parser instance data memory usage is held in the define ADI_MP4AD_FASTB0_PRIO0_STATE_MEM_NUMBYTES. If fewer bytes are allocated in the call to API function adi_mp4ad_create(), it will return NULL.

Memory for the storage of metadata information parsed from the file depends directly on the size of the type definitions adi_mp4ad_container_info_t and adi_mp4ad_metadata_t, and on the length of the string buffers allocated by the user to the text metadata fields.

Memory for the storage of indexing information is defined by the user.