ARC® Video Subsystem Codecs API

Application Program Interface

Reference

5189-004

ARC® Video Subsystem Codecs API Reference

ARC® International

European Headquarters
ARC International,
Verulam Point,
Station Way,
St Albans, Herts, AL1 5HE, UK
Tel. +44 (0) 1727 891400
Fax. +44 (0) 1727 891401

North American Headquarters
3590 N. First Street, Suite 200
San Jose, CA 95134 USA
Tel. +1 408.437.3400
Fax +1 408.437.3401

www.arc.com

Confidential Information

Notice

U.S. Government Restricted Rights Legend

Trademark Acknowledgments

5189-004 April 2008

Contents

List of Figures

Chapter 1 — Introduction

This document describes the operation, requirements and programming interface of the integrating the ARC Video Subsystem codecs for the ARC 700 processor core with The ARC Media Extensions.

The following video codecs are supported:

- H.264 Baseline Decoder
- MPEG-2 Main Profile and Main Level Decoder (MPEG-2 MP@ML)
- MPEG-4 Simple Profile Decoder (MPEG-4 SP)
- MPEG-4 Advanced Simple Profile Decoder, MPEG-4 ASP (no GMC)
- Windows Media Video 9 Simple Profile and Main Profile (Low and Medium) Decoder, WMV9/VC-1

Although each codec has subtly different requirements in terms of initialization, control and buffer management, an abstracted interface is provided whereby an external system (software or hardware) can integrate any combination of the codecs. The following sections describe this interface in more detail:

- Basic Codec Operation
- System Overview
- Codec Structure

The function reference contains further details on each function.

Each codec has specific function requirements that are documented in the following sections:

- H.264 Codec
- VC-1 Codec
- MPEG-2 and MPEG-4 Codecs

Basic Codec Operation

Each codec is a stand-alone library, and no assumptions are made with regard to the video stream to be decoded, other than it being in the correct, de-multiplexed format for the codec. It is assumed that each codec has access to all relevant packet data (e.g., for H.264 all NALs are provided to the codec with no pre-filtering by the external de-multiplexing system).

Each codec is controlled using a simple API based on five key operations: initialization (2 or 3 functions), decoding, control (trick play), decoded frame access, and cleanup. These operations encompass all input, decoding, output and buffer-management functions.

The codecs and their associated programming interface have been constructed to operate in both a low-level, single-process environment and under a high-level operating system that employs virtual addressing (for example, Linux). Each codec library includes the necessary primitives to support the required execution environment; the primitives can be enabled or disabled by selective compilation.

Key areas of operation that are covered by these primitives include buffer allocation/management and DMA control (as the ARC Media Extensions DMA unit uses physical addresses).

In order to reduce the possibility of namespace conflicts each codec exposes the operation functions via a structure, keeping the functions themselves declared to be source module local only (i.e. static). Thus it is possible to integrate one or more of the codecs into a single project without the risk of namespace conflict by using the abstracted function pointer interface provided by the interface structure.

System Overview

Each codec is controlled using an abstracted API based on the following key operations.

- Initialization - init(), set_buffer_addresses()
- Decoding - decode()
- Control (trick play) - control()
- Frame access - get_frame_data()
- Cleanup - uninit()

Depending on the codec, additional operations may be supported:

- Buffer setup - set_sizes()
- Input stream reading - read_callback()
- Get buffer size vc1_ext_vb_framesize()
- Get number of frames needed vc1_ext_vb_minframes()
- Set buffer pointers vc1_ext_vb_addframes()

These operations are wrapped in a structure allowing external access via a unified function pointer interface for all codecs. The function pointer and structures are defined in the file `arc_codecs.h`.

```
typedef struct {
  unsigned char                *planes[3];
  int                           stride[3];
  int                           xOffset;
  int                           yOffset;
  int                           width;
  int                           height;
  int                           index;
} arc_frame_data;

typedef int (*fp_control)               (int, void *);
typedef int (*fp_init)                  (int, char *, int, int, int, int);
typedef int (*fp_set_buffer_addresses)  (unsigned long, unsigned long *,
                                         unsigned long *);
typedef int (*fp_uninit)                (void);
typedef int (*fp_decode)                (char *, int, int, int *);
typedef int (*fp_get_frame_data)        (arc_frame_data *);
typedef int (*fp_setsizes)              (unsigned long, unsigned long, int);
typedef int (*fp_read_callback)         (int, unsigned char *, int);
```

```
  fp_ext_vb_framesize            extvb_get_frame_size;
  fp_ext_vb_minframes            extvb_get_frame_min;
  fp_ext_vb_addframes            extvb_add_frame_ptrs;

typedef struct {
  fp_init                       init;
  fp_set_buffer_addresses       set_buffer_addresses;
  fp_control                    control;
  fp_decode                     decode;
  fp_get_frame_data             get_frame_data;
  fp_uninit                     uninit;
  fp_setsizes                   setsizes;      //Optional entry
  fp_read_callback              read_callback; //Only required for VC-1

  fp_ext_vb_framesize            extvb_get_frame_size;
  fp_ext_vb_minframes            extvb_get_frame_min;
  fp_ext_vb_addframes            extvb_add_frame_ptrs;

} arc_codec_functions;
```

Each codec declares and instance of the **arc_codec_functions** structure as a global (exported) object which can then be referenced by the controlling system. The name of each structure declaration is specified in the relevant section later in this document.

Any system that integrates with one or more of the ARC video codecs must perform the following operations:

- Determine the format of the data stream to be decoded.
- Extract relevant header information as required without discarding the stream header data – note if the host application is not able to parse the stream header information, it can optionally set a callback function **set_sizes** which the codec can call once the appropriate information has been parsed from the input stream (currently supported for H.264, MPEG-4 and VC-1 streams).
- Call codec initialization functions **set_buffer_addresses()** and **init()** to setup the necessary frame data buffers and to initialise the codec. If required (currently only for VC-1) initialize the **read_callback** member of the function pointer structure to a suitable stream reading function.
- Split all stream data into suitable packets (e.g., single NAL blocks for H.264) and provide them to the codec **decode()** function.
- After each call to the codec **decode()** function, depending on the returned decoded frame counter, call the codec **get_frame_data()** function to retrieve decoded frame data.
- On completion of all input stream data, call the codec **decode()** function with NULL data to flush any remaining decoded frames and call **get_frame_data()** to get access to them.
- Call the code **uninit()** function to remove any allocated data, and remove the codec.

During decoding, the codec **control()** function can be called to provide trick-play functionality.

NOTE The decoded frame data array (i.e. YUV data) is managed by the codec, which gives selected access to the display controller to define the next (current) frame to be displayed.

Codec Structure

The following figure shows an overview of the codecs structure and interaction with an external application.

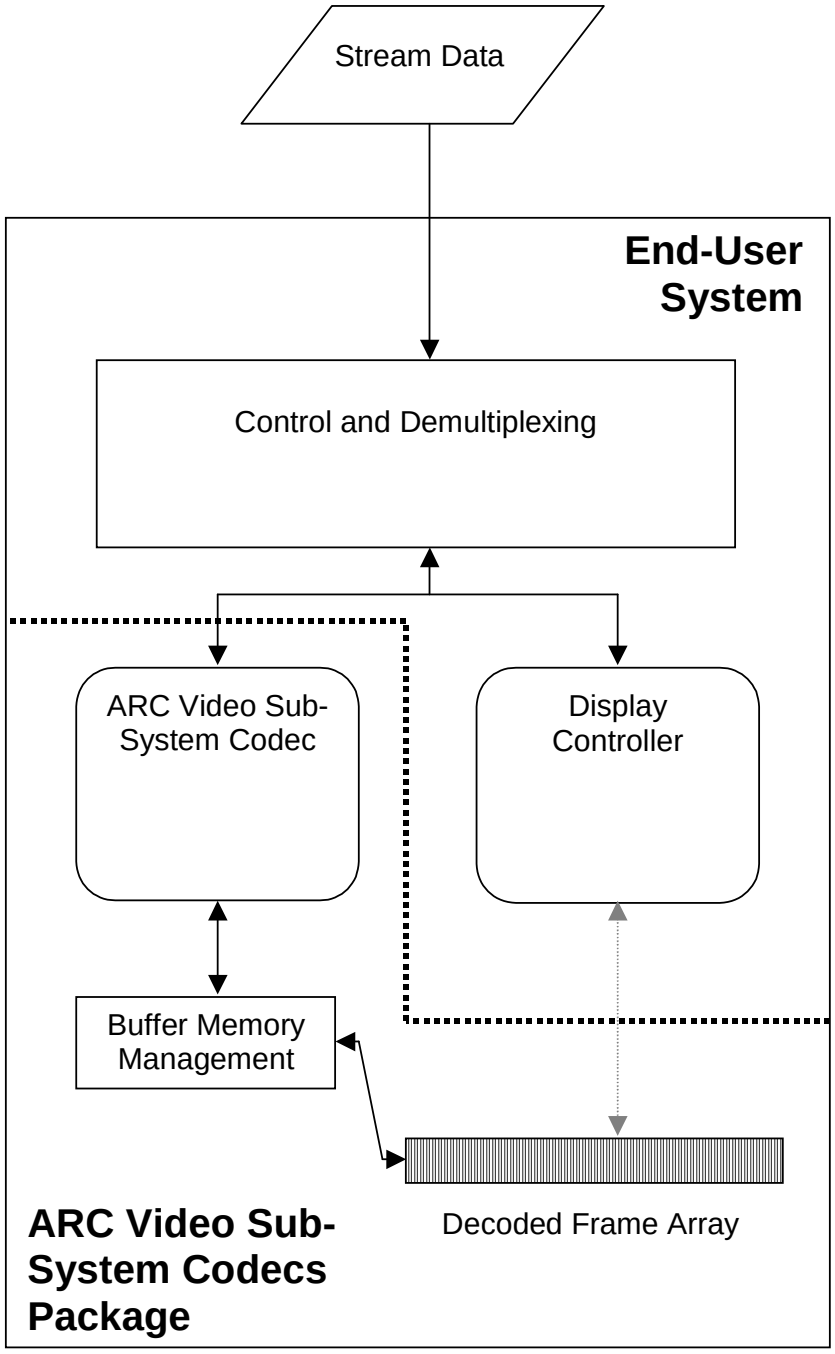

Figure 1 Overview of the Codecs Structure

Chapter 2 — Function Reference

The function listings are contained in the following sections:

Function Listing Format

The following is the general format a function in this reference.

function_name()

A short description of what **function_name()** does.

Synopsis

Provides a prototype for function **function_name()**.

```
<return_type> function_name(
  <type_1>  parameter_1,
  <type_2>  parameter_2,
  ...
  <type_n>  parameter_n)
```

Parameter	Definition
parameter_1	Definition of *parameter_1*
parameter_2	Definition of *parameter_2*
parameter_n	Definition of *parameter_n*

Description

Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- function blocks or might block under certain conditions
- function must be started as a task
- function creates a task
- function has pre-conditions that might not be obvious
- function has restrictions or special behavior

Return Value

Specifies any value or values returned by function **function_name()**.

See Also

Lists other functions or data types related to function **function_name()**.

Example

Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

init()

Create or initialize codec.

Scope

All codecs.

Synopsis

```
int init(
   int type,
   char *header_data,
   int header_size,
   int header_offset,
   int width,
   int height);
```

Parameter	Definition	
Type	Codec type	
	1 = ARC_H264	H.264 Codec
	2 = ARC_MPEG1	MPEG 2-4 codec
	3 = ARC_MPEG2	MPEG 2-4 codec
	4 = ARC_MPEG4	MPEG 2-4 codec
	5 = ARC_H263RAW	MPEG 2-4 codec, H263 raw files
	6 = ARC_MJPEG	MPEG 2-4 coded, MJPEG files
	7 = ARC_VC1	VC-1 codec, ASF files
	8 = ARC_RCV	VC-1 codec, RCV files

	9 = ARC_WCV1	Not implemented
	10 = ARCWMVA	Not implemented
header_data	Pointer to buffer containing header data; use NULL if this parameter is not required by your target codec.	
header_size	Size of the buffer pointed to by *header_data*, use 0 if this parameter is not required by the target codec.	
header_offset	File offset in bytes of the start of the header data: use 0 if this parameter is not required by the target codec.	
width	Image width in pixels: use 0 if this parameter is not required by the target codec.	
height	Image height in pixels: use 0 if this parameter is not required by the target codec.	

See the following for codec-specific requirements and error codes:

- H.264 Codec Parameters and Error Codes for init()
- VC-1 Codec Parameters and Error Codes for init()
- MPEG Codec Parameters and Error Codes for init()

Description

Initialize or create a codec of the specified *type* and, if necessary, use it to parse the provided header data (see each specific codec section for requirements).

Function **init()** is required to select and initialize the required video codec. All codecs require certain header information to successfully complete initialization. However, no single method exists for header parsing. See the specific section for each codec for details on the header information required by that codec. The image *width* and *height* values must be provided from the parsed header information, if required by the target codec.

Each codec has a different requirement for the maximum number of system YUV buffers that are required during decoding. These buffers are allocated during initialization based on the stream type and the decoded picture size.

Return Value

This function returns 0 on success. System-level failure codes are defined as follows:

1: invalid codec type

2: memory allocation failure

3: memory mapping error (Linux VM)

other: Codec-specific error number

For codec-specific error numbers, see the section for the specific codec.

Example

```
int isz = ((imageHeight * imageWidth) * 3 / 2);
void *ptr = malloc(isz*num_buffers);
if (ptr == NULL)
{
  printf("\nFailed on malloc of frame buffers [%s]",strerror(errno));
  return (2);
}
for (i=0; i<5; i++)
{
```

```
    vAddresses[i] = (unsigned long)ptr + (i*isz);
  }
  if (!mpg_functions.set_buffer_addresses((unsigned long)SDMBase,vAddresses,NULL)
      || !mpg_functions.init (inputType,NULL,0,0,imageWidth,imageHeight))
  {
    printf("\nFailed to construct the decoder");
    return (2);
  }
```

set_buffer_addresses()

Initialize codec frame buffer addresses

Scope

All codecs except VC-1.

Synopsis

```
int set_buffer_addresses(
   unsigned long SDMBase,
   unsigned long *vAddresses,
   unsigned long *pAddresses);
```

Parameter	Definition
SDMBase	Base address of the SIMD SDM memory, either the physical address when not using Linux, or the virtual mapped address with using Linux.
vAddresses	Array of start addresses of YUV buffers for use by the codec. If running under Linux the user should ensure that these areas are contiguous and are non-cached.
pAddresses	Optional array of physical start addresses of YUV buffers for use by the codec. Only required if running under Linux, otherwise should be NULL.

Function **set_buffer_addresses()** is a generic function and has no codec-specific functionality.

Description

The codec requires access to the SIMD unit's SDM buffer and enough working YUV buffers to complete a successful decode. This function initializes the codec to set the correct addresses according to the execution environment. For example if running under Linux, the SDM must be mapped into virtual address space using the **mmap()** function for the codec to be able to access it. Also if running under Linux, the codec requires access at both virtual and physical levels to the frame buffers, hence the calling application should provide complementary arrays of virtual and physical addresses for the frame buffers.

Each codec has a different requirement for the maximum number of system YUV buffers that are required during decoding.

For H.264 and MPEG 2/4 these buffers are allocated during initialization based on the stream type and the decoded picture size.

For VC-1 the driving application is responsible for allocating buffers.

Return Value

This function returns 1 on success. System-level failure codes are defined as follows:

0: internal error

Example

```
int isz = ((imageHeight * imageWidth) * 3 / 2);
void *ptr = malloc(isz*num_buffers);
if (ptr == NULL)
{
  printf("\nFailed on malloc of frame buffers [%s]",strerror(errno));
  return (2);
}
for (i=0; i<5; i++)
{
  vAddresses[i] = (unsigned long)ptr + (i*isz);
}
if (!mpg_functions.set_buffer_addresses((unsigned long)SDMBase,vAddresses,NULL)
    || !mpg_functions.init (inputType,NULL,0,0,imageWidth,imageHeight))
{
  printf("\nFailed to construct the decoder");
  return (2);
}
```

decode()

Decode the specified data buffer.

Scope

All codecs.

Synopsis

```
int decode(
   char *data,
   int size,
   int offset,
   int *num_frames);
```

Parameter	Definition
data	Pointer to buffer of read-in frame data; should be NULL when no more data are available (end of stream).
size	Size of the buffer pointed to by data; should be 0 for end of stream.
offset	File offset of the start of the buffer; use 0 if this parameter is not required for your target codec.
num_frames	Address of variable to indicate the number of available decoded frames.

See the following for codec-specific requirements and error codes:

- H.264 Codec Parameters and Error Codes for decode()
- VC-1 Codec Parameters and Error Codes for decode()
- MPEG Codec Parameters and Error Codes for decode()

Description

Function **decode()** uses the codec type specified in **init()** to decode the specified *data* buffer, returning the number of decoded frames ready for display.

Note that this function does not return any decoded data. Instead it returns the number of decoded frames waiting to be displayed (set in the *num_frames* parameter).

CAUTION If you call **decode()** before the decoded frames have been displayed, the previously decoded data might be lost due to buffer recovery. If the number returned in *num_frames* is not 0, call **get_next_frame()** *num_frames* times to get the set of frames to be displayed *before* calling **decode()** again.

Function **decode()** is the central interface to the codec, providing the codec with demultiplexed data packets of the correct format. Each codec has a specific requirement for the way that the demultiplexer must provide data to it; see the section for the specific target codec for details.

Return Value

0: Success

1: Invalid or missing codec

other: Codec-specific error number

For codec-specific error numbers, see the section for the specific codec.

The number of decoded frames ready for display is returned to *num_frames*.

Example

```
arc_frame_data frameData;
memset(&frameData,0,sizeof(arc_frame_data));

int size = GetNextNAL(&inputData);
if (!mpg_functions.decode (inputData,size,0,&numFrames))
{
  fprintf(stderr,"\nDecode failed\n");
  return (2);
}
if (numFrames == 1)
  mpg_functions.get_frame_data (&frameData);
```

control()

Control the operation of the codec to enable trick-play operations.

Scope

All codecs.

Synopsis

```
int control (
    int command
    void *args);
```

Parameter	Definition
command	Trick-play command:
	8: Resynchronize
args	Optional additional arguments to the codec

See the following for codec-specific requirements and error codes:

- H.264 Codec Parameters and Error Codes for control()
- VC-1 Codec Parameters and Error Codes for control()
- MPEG Codec Parameters and Error Codes for control()

Description

The control function is used to inform the codec that there is a discontinuity in the stream data, and a resynchronization is required. This is typically used when trick-play features are used to seek forwards and backwards through the video sequence.

Return Value

0: Success

1: Codec specific error

Example

```
//seek forward 10 seconds in the decoded stream
demux_seek_forwards(10);
//tell the codec to resync when decoding resumes
h264_functions.control(8,NULL);
//continue decoding
```

get_frame_data()

Get the frame data of the next frame to be displayed.

Scope

All codecs.

Synopsis

`int get_frame_data (arc_frame_data *`*frameData*`);`

Parameter	Definition
frameData	Structure containing relevant frame data returned by the codec: `frameData.planes[0]` = pointer to Y data `frameData.planes[1]` = pointer to U data `frameData.planes[2]` = pointer to V data `frameData.stride[0]` = Y stride `frameData.stride[0]` = U stride `frameData.stride[0]` = V stride `frameData.xOffset` = horizontal pixel offset of image to be displayed from the top-left of the returned image (normally `0` unless the image is to be displayed in a cropped format) `frameData.yOffset` = vertical pixel offset of image to be displayed from the top-left of the returned image (normally `0` unless the image is to be displayed in a cropped format) `frameData.width` = horizontal size of the image to display `frameData.height` = vertical size of the image to display `frameData.index` = not used

This is a generic function and has no codec-specific functionality.

Description

This function provides the frame-access operation. Use it to get the frame data of the next frame to be displayed after having successfully completed a decoding operation that returned one or more decoded frames.

The codec interface allows for some codecs to provide more that one frame of data to display after a single call to the **decode()** function; this typically occurs during decoding of initial frames (for H.264) and at the end of the sequence.

Call this function iteratively to return the next frame data to be displayed for each time slice until the next decoding operation needs to be performed.

The returned data are in YUV420 format.

Return Value

0: End of sequence

1: No frame data available

2: Success

The address of the next frame data to be displayed is returned in *frame_data*.

Example

```
arc_frame_data frameData;
memset(&frameData,0,sizeof(arc_frame_data));

int size = GetNextNAL(&inputData);
if (!mpg_functions.decode (inputData,size,0,&numFrames))
{
  fprintf(stderr,"\nDecode failed\n");
  return (2);
}
if (numFrames == 1)
  mpg_functions.get_frame_data (&frameData);
```

uninit()

Uninitialize (destroy) the current target codec.

Scope

All codecs.

Synopsis

```
int uninit (void);
```

Description

This function provides the cleanup operation, releasing any allocated memory and other resources to the system.

Call **uninit()** at the *end of each sequence*, as stream specific information is stored internally by the codec.

Return Value

0: Success

1: Invalid codec

Example

```
mpg_functions.uninit();
```

set_sizes()

Callback function provided by the host application for the codec to supply image size information.

Scope

All codecs.

Synopsis

```
int set_sizes (unsigned long image_width, unsigned long image_height,
               int num_buffers);
```

Parameter	Definition
image_width	The width in pixels of the image being decoded (before any cropping)
image_height	The height in pixels of the image being decoded (before any cropping)
num_buffers	The maximum number of buffers required by the codec to be available to decode the input stream

Description

This function provides a mechanism for a codec to supply information on the decoded stream back to the host, in the case that the host does not include sufficient header decode capabilities.

The function is called by the codec, if present, and can be used by the host to initialize the required array of frame buffers to the correct size. When used in this way it is typical for the codec initialization to be performed using NULL pointers for the virtual and physical addresses, and to call the **set_buffer_addresses()** function again with the correct virtual and physical addresses within the **set_sizes()** callback.

Return Value

0: Initialization failed

1: Success

Example

```
int set_sizes (unsigned long w, unsigned long h)
{
  int i;
  int isz = w * h * 3 / 2;
  imageWidth = w;
  imageHeight = h;
  unsigned long memBase = (unsigned long)malloc(isz * num_buffers);
  if (memBase == 0)
    fatal("H264: Failed malloc for buffers");
  for (i=0; i < num_buffers+1; i++)
  {
    vAddresses[i] = (unsigned long)memBase;
    pAddresses[i] = vAddresses[i];
    memBase += isz;
  }
```

```
    h264_functions.set_buffer_addresses((unsigned long) SDMBase,
                                        vAddresses,pAddresses);
    return (1);
  }

  int main (int argc, char *argv[])
  {
    //initialize the function pointer
    h264_functions.set_sizes = set_sizes;

    //continue with the rest of the application

  }
```

read_callback()

Callback function provided by the host application for the codec to read stream data.

Scope

VC-1 only.

Synopsis

`int read_callback (int` *file_offset*`, unsigned char *`*buffer*`, int` *size*`);`

Parameter	Definition
file_offset	The offset, in bytes, from the start of the stream to start reading from
buffer	The data buffer to write the read data into
size	The number of bytes to read from the input stream

Description

This function provides a mechanism to ask the host system to provide data from the input stream.

The function is called by the codec and should be implemented by the host application for those codecs that require this functionality (currently only VC-1).

Return Value

0: Read failed

1: Success

Example

```
  static int read_callback (int fileOffset, unsigned char *buffer, int minSize)
  {
    int sz;
    fseek (inputFile,fileOffset, SEEK_SET);
    sz = fread(buffer,1,minSize,inputFile);
    if (sz == minSize)
      return (1);
    else
      return (0);
  }

  int main (int argc, char *argv[])
```

```
{
  //initialize the function pointer
  vc1_functions.read_callback = read_callback;

  //continue with the rest of the application

}
```

vc1_ext_vb_framesize()

Scope

VC-1 only.

Synopsis

```
int vc1_ext_vb_get_framesize (void)
```

Description

Get the buffer's size (in bytes) needed for one frame decoding.

Return Value

Frame size (in bytes)

vc1_ext_vb_minframes()

Scope

VC-1 only.

Synopsis

```
int vc1_ext_vb_get_minframes (void)
```

Description

Get the minimum number of frames needs to stream decoding.

Return Value

Number of frames

vc1_ext_vb_addframes()

Scope

VC-1 only.

Synopsis

`int vc1_ext_vb_addframes (void** *ppListFrame*, int *nFrame*)`

Parameter	Definition
ppListFrame	pointer to array of external allocated video buffer pointers
nFrame	Number of external video buffers

Description

Set buffer pointers. Buffers allocated by the driving application based on the stream information (**vc1_ext_vb_get_framesize()** and **vc1_ext_vb_get_minframes()** functions).

Return Value

0: Error

1: Success

Chapter 3 — H.264 Codec

The H.264 codec conforms to the baseline profile of the H.264 standard and supports frame bumping. This often results in multiple calls to the **decode()** function returning no available decoded frames for display while the internal frame array is populated. It is therefore likely that the enclosing controller must enforce a startup delay while the decoded frame buffers are being filled. The situation is not usually repeated after bumping starts. Conversely, note that at the end of the sequence, the stored frames are flushed, providing multiple frames of data to the controlling application. The controlling application is therefore expected to manage time synchronization by successive calls to the **get_next_frame()** function at each time slice.

When using the bumping model the number of YUV buffers required is dynamically determined by the codec and returned in a call to the `set_sizes` function from the codec. The number of buffers required is determined by reference to the H.264 standard, Annex A, table A-1, and is a function of the decoded image dimensions and the profile level that the sequence was encoded to. For example, a level 3 encoded file with D1 resolution will require 5 frames storage for decode according to the standard whereas a QCIF resolution file encoded to the same level will require 16 buffers. Note the number of buffers required by the codec will always be two (2) above that defined by the standard to allow for optimal internal buffer management, thus the range of returned buffer count values is 7 to 18.

The function interface **struct** is declared as **h264_functions**.

The following sections show the codec parameters and error codes for this codec:

Parameters and Error Codes for init()

Type	`H264` = 1
header_data	`NULL`, none required; pass all NALs, including headers, to decode().
header_size	`0`
header_offset	`0`
width	`0` (none required)
height	`0` (none required)
Error codes	`4`: SIMD-unit initialization error.

Because the demultiplexer might decode some header NALs to extract required information prior to initializing this codec, we recommend buffering the NAL data for efficient supply to the **decode()** function. Note that as the **decode()** function is used to decode *all* NALs, the initialization phase of the system might include a number of calls to the **decode()** function.

Parameters and Error Codes for decode()

data Complete NAL block with header leading bytes (0x00,0x00,0x01) removed

Size Size of the data buffer in bytes.

offset 0 (not required)

num_frames The number of decoded frames ready for display.

Error codes 4: invalid bitstream.

The input data format is single, complete NAL units with delimiter-codes removed.

All NALs contained in the input stream must be provided to **decode()**. Each NAL should have delimiter data removed before it is passed to **decode()**; provide only single NALs for each invocation.

Note: if the DMA-assist function is enabled (part of CAVLC hardware extensions), then the input NAL data must be stored in uncached memory, or alternatively the data cache must be flushed prior to calling **decode()**.

Parameters and Error Codes for control()

command 8: resync on next reference frame.

No additional commands or functionality are required for this codec.

Chapter 4 — VC-1 Codec

The VC-1 codec conforms to the Microsoft WMV/VC-1 standard. The codec employs a non-linear access model to the input data, requiring that input buffered data be stored with a byte offset from the start of the stream or file. This information is required for both the **init()** and **decode()** functions.

Note: the VC-1 codec does not include an integrated ASF parser. The controlling application must perfrom the necessary parsing and demultiplexing..

The number of YUV buffers required is 5.

The function interface **struct** is declared as **vc1_functions**.

The following sections show the codec parameters and error codes for this codec:

Codec Parameters and Error Codes for init()

Type	ARC_RCV = 8
header_data	Required for all header packets.
header_offset	Required.
width	0 (not required)
height	0 (not required)
Error codes	4: SIMD-unit initialization error.

The codec should be set with a type of ARC_RCV for RCV format files.

Parameters and Error Codes for decode()

data	NULL
Size	0
offset	0 (not required)
num_frames	The number of decoded frames ready for display.
Error codes	4: invalid bitstream.

No input data is required. The codec relies on the host providing a stream reader callback function, which is referenced via the **vc1_functions.read_callback()** entry.

Before initialization, the driving application must ensure that the bitstream is either held in uncached memory or a data cache flush should be performed.

Parameters and Error Codes for control()

command	8: resync on next reference frame.
**args*	The value of the *args* parameter, cast as a pointer to a 64-bit signed integer
	Optional 64-bit time offset from the present time.

When performing trick play, it is expected that the external host determines the location of the next elementary video packet to decode (which contains a payload with a keyframe of zero offset). This information is provided to the **decode()** function to enable it to determine where in the stream to resync to. The **control()** function is used to signal to the codec that a resync operation will be required from the next call to the decode function.

Chapter 5 — MPEG-2 and MPEG-4 Codecs

The MPEG codec supports decoding of MPEG-2, MPEG-4 and H263 raw encoded stream and is based on the open-source FFMPEG project.

FFMPEG is distributed under the GNU Lesser General Public License (LGPL), a copy of which is available at http://www.gnu.org/copyleft/lesser.html.

The number of YUV buffers required is 5.

The function interface **struct** is declared as **mpg_functions**.

The following sections show the codec parameters and error codes for this codec:

Parameters and Error Codes for init()

Type	`ARC_MPEG1` = 2(not fully compliant with the MPEG1 standard)
	`ARC_MPEG2` = 3
	`ARC_MPEG4` = 4
	`ARC_H263RAW` = 5
header_data	`NULL` (not required)
header_size	0
header_offset	0
width	Required width in pixels.
height	Required height in pixels.
Error codes	4: SIMD unit initialization error.

As the MPEG codec sees all stream data, no specific header data is required during initialization. However it is expected that some simple parsing is done on the input data to determine the image size (and is also required to set or create the YUV buffers).

Parameters and Error Codes for decode()

data	Set of packets required to decode a single frame.
size	Size of the data buffer in bytes.
offset	0 (not required)
num_frames	The number of decoded frames ready for display.

Error codes 4: invalid bitstream.

The input data should be provided as the complete set of packets required to decode one frame of data. This corresponds to all non-frame data packets (i.e., codes less than 0x101 or greater than 0x1B0 for MPEG-1 and MPEG-2), followed by all frame data packets until the next non-frame data packet. Note: Unlike H.264, the MPEG data must include the initial leading packet delimiter bytes.

If the DMA-assist function is enabled (part of VLC hardware extensions), then the input NAL data must be stored in uncached memory, or alternatively the data cache must be flushed prior to calling **decode()**.

MPEG Codec Parameters and Error Codes for control()

command 8, resync on next reference frame

No additional commands or functionality are required for this codec.