ARC® H.264 Encoder

ARC® Video Subsystem Family Encoders API

C Application Program Interface (API)

Programmer's Guide

5896-009

ARC® Video Subsystem Family Encoders API Programmer's Guide

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

5896-009 April 2008

Contents

Chapter 1 — About the API 8

Introduction 8
What This Document Covers 8
Who Should Read This Document 8
Overview of the API 9
The Base API Definition 9
Terminology 9
Organization 10
Function Calls 11
Exported Functions 11
Imported Functions 11
Return Values 12
Interfaces 12
Data Types 12
Fundamental Types 12
Custom Data Types 12
Management Functions 12
Usage Samples 13
A Sample Encoder API 13
Single Encoder Deployment 13
Multiple Encoder Deployment 14

Chapter 2 — API Reference 15

Overview 15
Data Formats 16
Fundamental Data Types 16
Custom Data Types 16
Using Custom Data Types 16
Core Custom Data Types 18
Exported Functions 24
Overview 24
init() 24
uninit() 26
encode() 27
control() 29
The Exported Interface Structure 30
Imported Functions 31

Overview 31
frames_ready() 31
data_needed() 33
allocate() 34
release() 36
attention() 37
Imported Interface Structure 38
Error Codes 39
Management Functions 40
create_encoder() 41
release_encoder() 41
create_argument() 42
destroy_argument() 42
init_arc_encoder_api() 43
uninit_arc_encoder_api() 43
init_mgmt_if() 44
uninit_mgmt_if() 44
Example Use of the Management Functions 45
Platform Support 46
Platform Dependencies 46
Porting the Encoders to a New Platform 46
What the API Does Not Cover 48

List of Figures

Figure 1 Encoding Application Organization 10
Figure 2 Memory Descriptor Flags Format 21

List of Tables

Table 1 Members of the EncArgument Data Type 17
Table 2 Members of the YUVData Data Type 18
Table 3 Members of the StreamDescriptor Data Type 19
Table 4 Members of the Frame Descriptor Data Type 19
Table 5 Members of the RawFrame Data Type 20
Table 6 Members of the EncPacket Data Type 20
Table 7 Members of the MemBlkDesc Data Type 21
Table 8 Memory Descriptor Flags 21
Table 9 Core API Data-Type Codes 23
Table 10 Parameters for the uninit Function 26
Table 11 Core API control Function Codes 29
Table 12 Core API Error Codes 39

List of Examples

Example 1 Use of a Single Encoder, Pseudo Code 13
Example 2 Use of an Encoder via an Interface, Pseudo Code 14
Example 3 Example usage of the Management Functions 45

Chapter 1 — About the API

Introduction

This document describes the ARC® Video Subsystem Family Encoders API, a C language interface through which an application can interact with an ARC Video Subsystem Family Encoders API. The API is designed to accommodate a variety of video encoders, with the aim of providing a simple interface that is consistent across all supported encoders without restricting functionality. To do this, the API provides the following:

- A set of function calls to enable application code to set up and control encoding sessions;
- A collection of data types to allow encoding setup parameters, control data and video data to be communicated to and from an encoder.

What This Document Covers

Coverage of the API is split into the common specification shared by all encoders, and the implementation details of the API instance provided with each encoder. This document is concerned with the common specification (base API definition) and contains the following:

- A summary of the API and its important features (this section);
- A detailed reference of the Base API covering the following:
  - — Data type conventions;
  - — Common data types featuring in the Base API;
  - — Function specifications;
  - — Platform considerations.

The material in this document serves a general reference for all encoders supported by the ARC Video Subsystem Family, and is intended to be used in combination with the detailed reference guides for each encoder type.

Who Should Read This Document

This document is aimed at software developers wishing to deploy an ARC Video Subsystem Family Encoders API within an embedded system.

Overview of the API

The Base API Definition

The ARC Video Subsystem Family Encoders API API is viewed from two perspectives: specification, and implementation.

- The specification takes the form of a base API definition describing in general terms how an encoder is interfaced to application code, and how the interface should behave;
- Each supported encoder is wrapped in its own API implementation conforming to the Base API Definition. The encoder-specific implementation is responsible for correctly managing the encoder in question, while respecting the behavior and semantics of the base API definition. Because each different encoder type has different setup and control requirements, the encoder-specific API needs to define a set of data types allowing these functions to be carried out (which must comply with the base definition).

To use an encoder-specific API implementation, the programmer should have an understanding of the base definition. Conversely, having got to grips with the base definition, the programmer is then in a strong position to rapidly master each encoder-specific API implementation.

Terminology

The following terms appear throughout this document:

Base API

The API specification described in this document.

Encoder-Specific API

An implementation of the API, which interfaces to a particular encoder type and provides a set of features as described in the Base API specification.

Application

The outer code controlling the encoder via the API.

Raw Frame

A frame of input data (generally assumed to be in YUV format) that is passed into an encoder for processing.

Input Stream

A sequence of input data. Since input data may be obtained from a variety of sources, ranging from byte-serial to frame-grabbing, this is viewed as a sequence of raw frames.

Data Packet

A unit of encoded data as output by the encoder.

Output Stream

A sequence of encoded data packets assembled in the order they are produced by the encoder.

Encoding Setup Parameters

A set of values affecting the encoding algorithm.

Organization

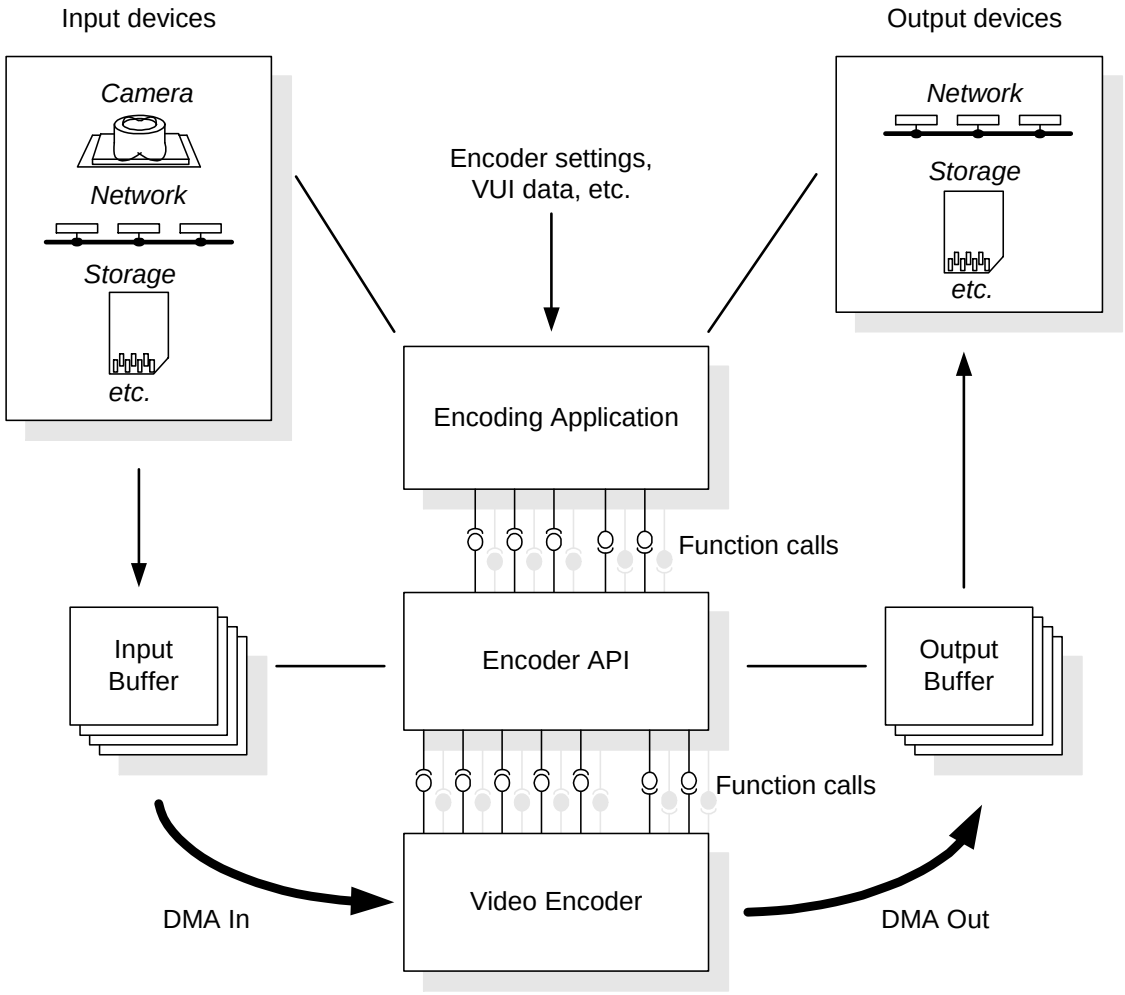


Figure 1 Encoding Application Organization

The arrangement of a typical encoder deployment is shown in Encoding Application . The deployment consists of an encoding application interacting with the encoder via the API. All interactions between the application and the encoder are mediated by the API, which consolidates the encoder's native interface into one that is compact and standardized.

Input video data is collected from some input source (such as a camera, or a storage device, etc.), placed into one of a set of input buffers and fed into the encoder via the API. Compressed video data is fed into a set of output buffers by the encoder, and passed over the API to the application where it is written to an output sink, such as a storage device or network stream.

The application is responsible for arranging the filling of input buffers and the emptying of output buffers. The encoder performs direct memory accesses to read from the input buffers and write to the output buffers. Allocation and management of the buffers is coordinated via the API, and can be tailored to meet the requirements of the application and/or operating platform.

Function Calls

The API definition provides for a number of function calls. These divide into two groups, as follows:

- An exported set of functions (these are provided by the API to be called by the containing application)
- An imported set of functions provided by the containing application (or support code) and are called by the API.

The exported functions comprise conventional encoder behavior (initialization, encoding, etc.) while the imported functions allow the API to trigger functions in the application.

Exported Functions

The exported functions provide the application-initiated behavior of the API:

- **init()** Initialize an encoding session
- **uninit()** Tear down the encoder
- **encode()** Process a set of input data
- **control()** Perform management functions on the encoder

Each encoder-specific API implementation is required to implement a function matching each of these functions.

NOTE The function names listed above are generic names. Each encoder API implements a set of functions that match those listed above, but which are uniquely named. For example, an interface to an MPEG-1 encoder might implement the following functions: **mpeg1_init()**, **mpeg1_uninit()**, **mpeg1_encode()**, **mpeg1_control()**. The encoder reference manuals provide the actual function names used for implementation of the functions specified in this document.

Imported Functions

The imported functions provide a means for the encoder to notify the containing application of some event that requires servicing:

- **frames_ready()** Notify the application that encoded data is available
- **data_needed()** Request more input data from the application
- **allocate()** Request use of a resource
- **release()** Relinquish a resource
- **attention()** Catchall means of requesting intervention from the application

The need to provide these functions depends on the encoder type; not all encoders need to use these functions. The individual encoder reference manuals provide details on which imported functions are required.

Return Values

API function calls return an integer value reporting either success or failure. A zero return value indicates success. Non-zero return values combine the indication of failure with the cause of failure. The convention has been adopted that negative return values indicate API-level errors, and that positive values indicate encoder-level errors.

Interfaces

To simplify multiple-encoder deployments, the API definition provides the ability to use interfaces. The interfaces are data structures made up of function pointers. Each API implementation is required to support an exported interface, consisting of pointers to each of the exported functions listed in Exported Functions.

In addition to the exported interface, imported interfaces are supported. These interfaces are structures of function pointers, in this case provided by the application and accessing the set of the imported functions listed in Imported Functions.

Data Types

The API provides a set of type definitions to enable the exchange of data between the application and the encoder. The types be divided into two groups: fundamental and custom data types.

Fundamental Types

Fundamental or non-complex data types map directly onto native C language types. The API takes the step of providing these as typedefs to be explicit about the storage size and range of values represented by these types.

Custom Data Types

Both the encoders and the API itself have complex data requirements. The API encapsulates these into a set of data structures. A number of these structures serve a fundamental role in describing items such as raw input data, chunks of output data, etc., and are provided as part of the base API definition. Individual encoders have their own unique data requirements in terms of encoding setup parameters, etc., and these requirements are met by the API implementation specific to each encoder.

In order to keep the API consistent across all implementations while allowing for encoder-specific data types, the API definition provides a means of exchanging structured data in a type-neutral manner.

Management Functions

In order to simplify use of the API, a small set of functions are provided to perform repetitive tasks related to managing API objects These functions allow interfaces and data structures to be created and initialized with minimal programmer intervention. These functions form a management interface that is extended at run time by each encoder type in use, allowing the core API features to maintain independence from the encoder-specific implementations.

Usage Samples

To give an idea of how an encoder is used via the API in practice, this section provides several pseudo-code examples section. Read the examples with the preceding discussion of API organization and features in mind.

A Sample Encoder API

The usage samples rely on an example encoder API implementation providing the following functions:

- **my_encoder_init()**
- **my_encoder_deinit()**
- **my_encoder_encode()**
- **my_encoder_control()**

Each of these functions provides an encoder-specific implementation of the Exported Functions.

Additionally, the API implementation provides an example of the exported Interfaces, through which all of the above functions can be invoked.

Single Encoder Deployment

The pseudo-code sample in Use of a Single Encoder, Pseudo Code illustrates the use of an encoder via the API. For clarity, the API functions are shown in *emphasis*; all other functions should be provided by the application.

Example 1 Use of a Single Encoder, Pseudo Code

```
init_arc_encoder_api()
# Get encoding parameters, frame dims, etc., and pass to encoder
config = get_encoder_configuration()
my_encoder_init(config)
# Open input source and output sink
instream = create_input_stream()
outstream = create_output_stream()
# Encoding loop
while (input_available(instream))
       # Get raw data and encode it
       rawdata = read_yuv_frame()
       outdata = my_encoder_encode(rawdata)
       # Output encoded data, packet by packet
       for (index in 0 … number_of_elements(outdata))
              write_output_packet(outstream, outdata[index])
# Encoding finished, tear everything down
my_encoder_deinit()
close_stream(instream)
close_stream(outstream)
```

This example shows the simplest form of deployment. A single encoder is used, and the application links directly to the encoder-specific API functions (**my_encoder_init()**, etc.).

Multiple Encoder Deployment

The use of the exported interface structure allows an application to handle a set of different encoders in much the same way as a single encoder, as shown in Use of an Encoder via an Interface, Pseudo Code. The API functions are shown in *emphasis*; all other functions should be provided by the application.

Example 2 Use of an Encoder via an Interface, Pseudo Code

```
init_arc_encoder_api()
# For each encoder in use, initialize its API
init_all_encoder_management_interfaces()
do
	# Select an encoder for this session
	encoder_type = choose_encoder()
	interface = get_encoder_interface(encoder_type)
	# Configuration parameters are specific to encoder type
	config = get_encoder_configuration(encoder_type)
	# Initialize using the relevant function in the interface
	interface.init(config)
	# Process as normal
	instream = create_input_stream()
	outstream = create_output_stream()
	while (input_available(instream))
		rawdata = read_yuv_frame()
		outdata = interface.encode(rawdata)
		for (index in 0 … number_of_elements(outdata))
			write_output_packet(outstream, outdata[index])
		end
	interface.deinit()
	close_stream(instream)
	close_stream(outstream)
while (true)
```

Chapter 2 — API Reference

Overview

This section provides a detailed reference for the base API specification, describing features and behavior common to all API implementations.

- Data Formats
- Exported Functions
- Imported Functions
- Error Codes
- Management Functions
- Platform Support

Data Formats

The API provides a set of data type definitions, which can be classed into two groups:

- Fundamental type definitions, provided to isolate the API from compiler-specific variations in fundamental data types;
- Complex data types, provided to enable sets of related data to be aggregated together.

In order to keep the API simple, a system of data polymorphism is introduced. This allows the API functions definitions to be data-type neutral.

Fundamental Data Types

Integer data types are described throughout the API implementation and documentation using the following convention:

- A letter denoting whether the item is signed (**s**) or unsigned (**u**) followed by;
- A number indicating the length of the item in bits.

Therefore **u32** denotes an unsigned 32-bit quantity, **s8** an 8-bit signed quantity.

Boolean data is handled by a `boolean` typedef, implemented as an integer. False is zero.

The API definition makes no provision for floating-point types, as they should be avoided in time-constrained code. Their use is occasionally warranted for test purposes, such as generating encoding statistics. Individual encoder references indicate where floating-point arithmetic is used, and how to disable it.

Custom Data Types

A number of data types are provided as part of the Base API Specification, and are detailed in this section. Each encoder-specific API implementation provides a further set of data structures for its own use - the contents and format of these being determined by the demands of the encoder in question. These encoder-specific data types are described in the individual encoder reference manuals.

Using Custom Data Types

In order to balance the need to provide a consistent API with that of accommodating the varying data requirements of different encoders, a simple form of data polymorphism is provided. This allows instances of custom data types to be passed over the API in the form of a descriptor that allows type identification and correct typecasting of the object at the destination.

This descriptor is embedded as the first member of each data type that can be passed over the API in this way, and obtaining the descriptor is simply a matter of accessing the relevant member of the data structure. The descriptor in turn holds a pointer to the base address of the containing object. Each data type is allocated a unique integer type code, so that the code on the receiving side of an API call can identify the type of an argument and interpret the data contained within it correctly. The definition of the descriptor is:

```
typedef struct
{
  u32 type_id;
  u32 size;
```

```
  u32 magic;
  void *owner;
} EncArgument;
```

The data members of this type are shown in the table Members of the EncArgument Data Type.

Table 1 Members of the EncArgument Data Type

Member	Notes
type_id	Type code of the object being described. Each type is allocated a unique integer code to identify it.
size	In-memory size in bytes of the described object, including the embedded descriptor.
magic	Reserved for API-internal use.
owner	A pointer to the object being described. This is by convention the containing object.

CAUTION	The **magic** field is used to identify objects allocated and managed by the API management functions. Objects created and managed by application code should always initialize this field to zero. Do not modify the **magic** field. Doing so leads to memory leaks.

Each data type using this data type should embed it as its first member (i.e., occupying the lowest address):

```
typedef struct
{
  EncArgument typeinfo;
  char *value;
} ArgCharstar;
```

The API provides a macro **IS_ARGTYPE** to embed the descriptor. This macro is used throughout the API documentation:

```
typedef struct
{
  IS_ARGTYPE;
  char *value;
} ArgCharstar;
```

You must initialize a descriptor before passing the associated data item via the API. For example:

```
// Assume this…
#define ENC_ARGTYPE_CHARSTAR    3

// Then to set up a data structure
ArgCharstar myarg;
myarg.typeinfo.type_id = ENC_ARGTYPE_CHARSTAR;
myarg.typeinfo.size = sizeof(ArgCharstar);
myarg.typeinfo.magic = 0;
myarg.typeinfo.owner = &myarg;
```

The core API provides a set of macros to simplify this task:

```
SETUP_ENCARGUMENT(myarg, ENC_ARGTYPE_CHARSTAR, sizeof(ArgCharstar);
```

Alternately, a function is provided in Management Functions to create a data type instance and correctly initialize its type information structure:

```
ENC_ARGTYPE_CHARSTAR myargptr;
myargptr = create_argument(ENC_ARGTYPE_CHARSTAR);
```

Objects created using the `create_argument` function should be released as follows:

```
destroy_argument(myargptr);
```

It is important to correctly initialize type descriptors for data passed over the API. An inability to correctly determine the type of an argument might cause the invoked function to fail.

Where an API function is takes an *EncArgument* parameter, pass a pointer to the descriptor structure:

```
result = control(1, MYFN, &(myarg.typeinfo));
result = control(1, MYFN, &(myargptr->typeinfo));
```

Or alternatively, using the API macros:

```
Result = control(1, MYFN, &(ENC_ARG (myarg));
Result = control(1, MYFN, &(ENC_PTRARG (myargptr));
```

Core Custom Data Types

YUV Data

This is used to describe a frame of raw video data in Y,U,V form. Since the API is intended for use on a variety of platforms, addresses are specified in both in-process (virtual) and physical forms.

```
typedef struct
{
  unsigned char* Y_base;
  unsigned char* U_base;
  unsigned char* V_base)
  unsigned char* Y_phys;
  unsigned char* U_phys;
  unsigned char* V_phys;
} YUVData;
```

The data members of this type are shown in the table Members of the YUVData Data Type.

Table 2 Members of the YUVData Data Type

Member	Description
Y_base	In-process address of the input frame Y component
U_base	In-process address of the input frame U component
V_base	In-process address of the input frame V component data
Y_phys	Physical address of the input frame Y component
U_phys	Physical address of the input frame U component
V_phys	Physical address of the input frame V component

Stream Descriptor

This structure is used to provide information about the stream being encoded.

```
typedef struct
{
  IS_ARGTYPE;
  u16 width;
  u16 height;
  u32 frames;
  u32 current_frame;
  bool interlaced;
} StreamDescriptor;
```

The data members are shown in the table Members of the StreamDescriptor Data Type.

Table 3 Members of the StreamDescriptor Data Type

Member	Description
width	Width of frames in the stream
height	Full height of frames in the stream
frames	Running count of raw frames
current_frame	Running count of processed frames
interlaced	If true, indicates that the input data is interlaced

Frame Descriptor

Provides information about an individual frame in a stream, serving to hold data common to both raw and encoded versions of the frame.

```
typedef struct
{
  IS_ARGTYPE;
  u32 frame_num;
  u32 capture_time;
  StreamDescriptor *strDesc;
} FrameDescriptor;
```

Table 4 Members of the Frame Descriptor Data Type

Member	Description
frame_num	Serial number of this frame
capture_time	Time stamp at capture of the corresponding input frame
strDesc	A pointer to the descriptor for the stream to which this frame belongs

Input Data Frame

Provides information about a single frame of raw input data.

```
typedef struct
{
  IS_ARGTYPE;
  FrameDescriptor info;
  bool unused;
  u32 tag;
```

```
} RawFrame;
```

Table 5 Members of the RawFrame Data Type

Member	Description
info	Descriptor for this frame
unused	Indicates whether the frame contains data awaiting consumption by the encoder. If this member is set to false (i.e. in use), the frame contains data awaiting consumption. If this member is true, the frame is available for writing data into.
tag	System-specific frame identity (e.g. Index into frame-buffer array)

Output Data Packet

Describes a unit of encoded data output by an encoder. The granularity of the packet is determined by a combination of encoder type and parameter values.

```
typedef struct
{
  IS_ARGTYPE;
  u32 packet_no;
  UNCACHED_PTR (data);
  u32 packet_len;
  FrameDescriptor info;
  bool unused;
  u8 num_unused_bits_first;
  u8 num_unused_bits_last;
  RawFrame *decoded_picture;
  char *stats_buf;
  u32 stats_buf_len;
} EncPacket;
```

Table 6 Members of the EncPacket Data Type

Member	Description
packet_no	Packet serial number
data	Base address of encoded data
packet_len	Number of bytes of encoded data
info	The descriptor for the associated frame of data
unused	Indicates whether the packet contains data awaiting output by the application. Data packets returned by the encoder will have this member set to false. Once the application has consumed the encoded data, it should set this member to true.
num_unused_bits_first	Offset in bits to the start of valid data in the first byte of the packet
num_unused_bits_last	Number of bits after the end of valid data in the last byte of the packet
decoded_picture	If reconstruction of input frames is available and enabled, the frames will be referenced here.
stats_buf	If statistics are enabled, this points to a formatted string of statistics data

Member	Description
stats_buf_len	Length of the statistics buffer

Memory Block Descriptor

Used to pass references to special-purpose memory over the API. Both virtual and physical addresses for the block are used.

Table 7 Members of the MemBlkDesc Data Type

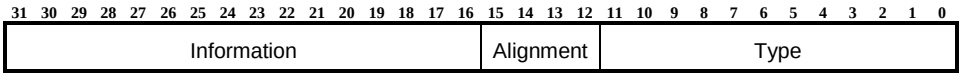
```
typedef struct
{
  IS_ARGTYPE;
  void *phys;
  void *virt;
  u32 size;
  u32 flags;
  u32 tag;
} MemBlkDesc;
```

Member	Description
phys	Base physical address of the block
virt	Base virtual (in-process) address of the block
size	Length in bytes of the block
flags	A set of flags giving details about the block. See Memory Descriptor Flags Format
tag	Optional system-dependent identity of the block

Memory Description Flags

The format of the flags word used to describe memory blocks is shown in ***Memory Descriptor Flags Format***.

Figure 2 Memory Descriptor Flags Format



The fields and their currently allocated values are shown in the table Memory Descriptor Flags.

Table 8 Memory Descriptor Flags

Field	Description
Type	This field describes the purpose for which the block is used. The values listed below may not be used in combination. • ENC_MEM_OPSTREAM, Encoded data destination • ENC_MEM_RECONYUV, Reconstructed YUV data destination • ENC_MEM_INPUTYUV, Input YUV data source

Field	Description
Alignment	This field describes the minimum address alignment in bytes of the block. • ENC_MEM_ALIGN_4, 4 byte alignment (default) • ENC_MEM_ALIGN_8, 8 byte alignment • ENC_MEM_ALIGN_16, 16 byte alignment • ENC_MEM_ALIGN_32, 32 byte alignment • ENC_MEM_ALIGN_64, 64 byte alignment • ENC_MEM_ALIGN_128, 128 byte alignment
Information	This field provides additional allocation information and descriptions of specialized attributes. • ENC_MEM_DC_PAD, The block should be placed or padded so that it does not share any data cache line with any other object • ENC_MEM_DMA_TGT, The block will be used as a DMA target (i.e. data will be placed in the block) • ENC_MEM_DMA_SRC, The block will be used as a DMA source (i.e. data will be taken from this block).

String

Used to pass a string of characters via the API.

```
typedef struct
{
  IS_ARGTYPE;
  char *value;
  u32 length;
} ArgCharstar;
```

This type consists of a simple character array and array length pair. The length indicates the allocated length of the character array. If the length of the string placed in the array is less than the value of length, then the string should be null-terminated.

Fundamental

Used to pass a small number of fundamental values via the API.

```
typedef union
{
  bool boolval;
  u8 charval;
  u16 shortval;
  u32 longval;
  void* ptr;
} AnyFundamental;

typedef struct
{
  IS_ARGTYPE;
  u32 num_entries;
  AnyFundamental *array;
} ArgFundamental;
```

This type is intended for use in passing flexible quantities of trivial data items to API functions that require *EncArgument* parameters, and is implemented around a union type to avoid proliferation of data type definitions in the API. On creating an instance of this type, the programmer should create an array of `AnyFundamental` appropriate to the needs on hand, and attach that to the `ArgFundamental` instance, as follows:

```
ArgFundamental my_fund;
AnyFundamental* array = malloc(sizeof(AnyFundamental) * NUMBER_ELEMENTS);
my_fund.array = array;
my_fund.num_entries = NUMBER_ELEMENTS;
array[0].boolval = false // etc…
```

Any API call accepting this type specifies the number and type of elements to be used.

Core Data Types and their Type Codes

Each custom data type that can be passed as an argument over the API is allocated a unique type code. The codes for the types described in this section are shown in Core API Data-Type Codes.

Table 9 Core API Data-Type Codes

Type	Code
`ArgFundamental`	`ENC_ARGTYPE_FUNDAMENTAL`
`ArgCharstar`	`ENC_ARGTYPE_CHARSTAR`
`MemBlkDesc`	`ENC_ARGTYPE_MEMBLKDESC`
`StreamDescriptor`	`ENC_ARGTYPE_STREAMDESC`
`FrameDescriptor`	`ENC_ARGTYPE_FRAMEDESC`
`RawFrame`	`ENC_ARGTYPE_RAWFRAME`
`EncPacket`	`ENC_ARGTYPE_ENCPACKET`

Exported Functions

Overview

The exported function set is the primary means of interaction between an encoder and the enclosing application. It consists of a set of actions that are initiated by the application and are serviced by the API and encoder.

- init()
- uninit()
- encode()
- control()

init()

Initializes an encoder session. Call only once before encoding starts.

Synopsis

```
s32 init(u32              encoder_handle,
         u32              coding_type,
         StreamDescriptor *stream_desc,
         EncArgument      *extra_data);
```

NOTES The API must be initialized before this function is called. Refer to the description of init_arc_encoder_api.

This function must be called before any encoding or control functions are invoked on the encoder. The only call that may be made to an un-initialized encoder is init.

It is also important that the session handle used to initialize the encoder is used on subsequent function calls to the encoder. An unrecognized session handle is treated as if it were a call to an un-initialized encoder.

Parameters

encoder_handle	Encoder session ID
coding_type	Encoding type and profile being requested. The exact value of this will depend on the encoder type in use.
stream_desc	Details of the stream being encoded.
extra_data	Additional data specific to the encoder type. The exact type and contents of this argument depend on the encoder type in use.

Return Value

Encoder-specific implementations of this function might define additional return values; see the documentation for your encoder for details.

ENC_ERR_NONE	Success.
ENC_ERR_NOMOREINSTANCES	The maximum supported number of encoder sessions has been initialized.
ENC_ERR_CODINGTYPE	The encoder does not support the requested coding type.

Example

Use of this function is straightforward, as shown in the pseudo-code fragment below.

```
# Always do this once and once only before any encoder interactions
init_arc_encoder_api();
# Get session parameters
setup = get_encoding_setup_from_whereever();
stream_desc = create_stream_descriptor(setup);
coding_type = get_encoding_type(setup);
handle = create_session_handle();
# Format setup data as encoder argument type
initialise_encargument(setup);
# Initialize session
if (init(handle, coding_type, stream_desc, &(ENC_ARG(setup)) != ENC_ERR_NONE)
{
  abort_session();
}
```

uninit()

Tears down an encoder session and frees up any session resources held in the API layer.

Synopsis

```
s32 uninit(u32 encoder_handle);
```

Parameters

Table 10 Parameters for the uninit Function

encoder_handle	Encoder session ID

Description

May only be called once, and then only on a previously initialized encoding session. No further encoding or control functions may be performed by the encoder session after this function is called.

Return Value

The common values that all implementations of this function returns are shown below.

ENC_ERR_NONE	Success
ENC_ERR_NOTINIT	Encoder session has not been initialized, or session has already been torn down.
ENC_ERR_BADHANDLE	Unrecognized encoder handle.

Example

```
if ((result = uninit(handle)) != ENC_ERR_NONE)
  probably_an_application_programming_error(result);
```

encode()

Synopsis

```
s32 encode(u32         encoder_handle,
           RawFrame    **in_frames,
           u32         *num_frames,
           EncPacket   **out_packets,
           u32         *num_packets,
           u32         *status,
           EncArgument *extra_data);
```

Enqueue input data for encoding. A call to this function might result in zero or more frames of data being consumed, and zero or more packets of output data being produced. Any output produced is not constrained to correspond to the input data passed in.

NOTE Implementations of this function should try to consume a single frame of data per call and return encoded data related to the input frame. This cannot, however, always be guaranteed.

Refer to the relevant encoder reference for precise details as to the behavior of the implementation of this function.

Parameters

encoder_handle	Encoder session ID
in_frames	An array of frames of YUV data for encoding. May contain zero elements if no further input is available, but output is still to be produced. The array should be arranged in presentation order.
num_frames	Points to a variable which, on call, contains the number of input frames submitted in in_frames, and on return, contains the number of frames consumed.
out_packets	A pointer to a data packet array, which will be populated in the correct output ordering by the encoder.
num_packets	Points to a variable which, on call, contains the number of packets that can be fit into the data packet array out_packets, and on return, contains the number of packets actually stored.
status	Pointer to a variable into which the encoder may write a word of status data.
extra_data	Additional encoder-specific data.

Return Value

The common values that all implementations of this function return are shown below.

`ENC_ERR_NONE`	Success
`ENC_ERR_NOTINIT`	Encoder session has not been initialized, or has already been torn down.
`ENC_ERR_BADHANDLE`	Unrecognized encoder handle.
`ENC_ERR_BADARG`	Incorrect type of `extra_data`; `extra_data` type descriptor not initialized correctly.
`ENC_ERR_OUTOFOPS`	Output packet array full.

Example

The use of this function is illustrated in the pseudo code below.

```
raw_frames = create_input_frame_array(max_num_input_frames);
```

```
out_packets = create_empty_array_for_output(max_num_output_packets);

while (input_available())
{
  num_output_packets = max_num_output_packets;
  num_read = get_frame_of _raw_data(&raw_frame[0]);
  enc_params = get_encoding_parameters_for_this_session(handle);
  result = encode(handle,
                  &raw_frames,
                  &num_read,
                  &out_packets,
                  &num_output_packets,
                  &status,
                  &(ENC_ARG(enc_params));
  if (result != ENC_ERR_NONE)
  {
    for i in 0 … num_output_packets
    {
      dispose_of_encoded_data(out_packets[i]);
      out_packets[i].unused = 1;
    }
  }
  else
    handle_error_and_decide_whether_to_end_session(result);
}
```

control()

Synopsis

```
s32 control(u32         encoder_handle,
            u32         function,
            EncArgument *extra_data);
```

Implements a generic interface onto control, maintenance and monitoring functions within both the API and the encoder itself

Parameters

encoder_handle	Encoder session ID
function	Function request code
extra_data	Additional data specific to the function requested

Common Function Codes

Table 11 Core API control Function Codes

Function	Definition	extra_data argument type
ENC_FN_SET_IMPIF	Set up the encoder's imported interface	**ArgFundamental** containing a pointer to the interface in the first element
ENC_FN_GET_IMPIF	Retrieve the encoder's imported interface	**ArgFundamental** receiving a pointer to the interface in the first element

Return Value

The common values that all implementations of this function return are shown below.

ENC_ERR_NONE	Success
ENC_ERR_NOTINIT	Encoder has not been initialized, or encoder has already been torn down.
ENC_ERR_BADHANDLE	Unrecognized encoder handle.
ENC_ERR_BADARG	Incorrect type of extra_data; or, extra_data type descriptor not initialized correctly.
ENC_ERR_UNSUPPPORTED	Encoder does not support requested function
ENC_ERR_NOFUNC	Unknown function code
ENC_ERR_FAILED	Unspecified error

Example

The use of this function is illustrated in the pseudo code shown below.

```
function_code = get_code_for_this_request();
control_data = get_data_for_this_request();
if ((result = control(handle,
                      function_code,
                      &(ENC_ARG(control_data)))
{
  handle_control_error(function_code, result);
}
```

The Exported Interface Structure

The exported functions may also be accessed via the following data structure exported by each API instance:

```
typedef struct {
  fp_enc_init        init;
  fp_enc_uninit      uninit;
  fp_enc_encode      encode;
  fp_enc_control     control;
  u32                handle;
} encoder_interface;
```

This structure being made up of four function pointers, plus a variable intended to contain an encoder session ID. Each function accesses one of the functions described in this section, their type definitions being:

```
typedef s32 (*fp_enc_init)     (u32,
                                u32,
                                StreamDescriptor*,
                                EncArgument*);
typedef s32 (*fp_enc_uninit)   (u32);
typedef s32 (*fp_enc_encode)   (u32,
                                RawFrame*,
                                u32*,
                                EncPacket**,
                                u32*,
                                u32*,
                                EncArgument*);
typedef s32 (*fp_enc_control)  (u32,
                                u32,
                                EncArgument*);
```

Each instance of this structure must be fully populated with a complete set of valid function pointers to the relevant functions for the encoder type in question. The provision of the interface is mandatory - all API implementations support it. An exported interface instance can be either created by hand, or by using the `create_encoder` management function.

Imported Functions

Overview

The Imported Functions are implemented by the application and invoked by the encoder. They provide a means for the encoder to directly invoke functionality in the application without needing to return part-way through the execution of an exported function.

The imported functions are entirely optional - API implementations are not obliged to implement them, and should do so only as and when needed.

- frames_ready()
- data_needed()
- allocate()
- release()
- attention()

frames_ready()

Synopsis

```
s32 frames_ready(encoder_interface *this,
                 DataPacket        **packets,
                 u32               packet_count);
```

This function may be invoked by the encoder to notify the containing application of the availability of encoded data for immediate consumption. This function will only be required where the encoder does not permit the encoded data to be returned conventionally via the `encode` function.

Parameters

this	Pointer to the encoder's exported interface.
packets	Pointer to an array of packets of encoded data for disposal by the application.
packet_count	Number of packets in the array packets.

Return Value

The common values that all implementations of this function return are shown below.

`ENC_ERR_NONE`	Success
`ENC_ERR_FAILED`	Unspecified error

Example

A possible implementation of this function is illustrated by the pseudo-code below.

```
s32 my_frames_ready(encoder_interface* this,
                    DataPacket **packets,
                    u32 packet_count)
{
   for i in 0 … packet_count
   {
     dispose_of_packet_data(packets[i]);
```

```
      flag_packet_as_unused(packets[i]);
   }
   return ENC_ERR_NONE;
}
```

data_needed()

Synopsis

```
s32 data_needed(encoder_interface *this,
                RawFrames         **in_frames,
                u32               *num_frames);
```

This function may be invoked by the encoder when it requires more input data from the containing application. This function will only be required where the encoder does not support feeding of input data conventionally via the `encode` function.

Parameters

this	Pointer to the encoder's exported interface.
in_frames	Pointer to an array in which to place new input frames
num_frames	Pointer to a variable into which the number of frames placed into `in_frames` should be placed

Return Value

The common values that all implementations of this function return are shown below.

`ENC_ERR_NONE`	Success
`ENC_ERR_FAILED`	Unspecified error

Example

A possible implementation of this function is illustrated by the pseudo-code below.

```
s32 my_data_needed(encoder_interface *this,
                RawFrames         **in_frames,
                u32               *num_frames)
{
  *num_frames = 0;
  for i in 0 … some_reasonable_limit_such_as_one
  {
    in_frames[i] = obtain_an_empty_input_frame();
    get_frame_of _raw_data(&in_frames[1]);
    *num_frames++;
  }
  return ENC_ERR_NONE;
}
```

allocate()

Synopsis

```
s32 allocate(encoder_interface *this,
             u32               size,
             u32               flags,
             u32               num_blocks,
             MemBlkDesc        **blocks);
```

This function may be invoked by the encoder when it requires one or more blocks of special purpose memory from the containing application.

NOTE This function is used only for memory set aside by the system for special purposes - for example, managed un-cached regions, frame buffer memory, etc. Stack and heap memory is allocated by the C run time, as normal. If you wish to modify these allocation strategies, modify your C run time.

Parameters

this	Pointer to the encoder's exported interface.
size	Size in bytes of the requested block or blocks.
flags	Set of flags describing the memory being requested. See the table Memory Descriptor Flags.
num_blocks	Number of blocks requested
blocks	Array to contain descriptors for the allocated blocks, one descriptor per block.

Return Value

The common values that all implementations of this function return are shown below.

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Operation failed - no memory allocated.

NOTE If this function fails, it is assumed that nothing was allocated. In the event of a failure part-way through a request for multiple blocks, all blocks allocated up to that point in the request must be freed.

Example

An implementation of this function is illustrated by the pseudo-code shown below.

```
s32 my_allocate(encoder_interface *this,
                u32               size,
                u32               flags,
                u32               num_blocks,
                MemBlkDesc        **blocks)
{
  request_type = get_purpose_of_request(flags);
  alignment = get_alignment(flags);
  for i in 0 … num_blocks
  {
    blocks[i] = new MemBlkDesc;
    blocks[i].virt = allocate_aligned(request_type, size, alignment);
    if (blocks[i].virt == 0)
      release_allocated_blocks(blocks, i);
    blocks[i].phys = virt_to_phys(request_type, blocks[i].virt);
```

```
    blocks[i].flags = flags;
    blocks[i].size = size;
    blocks[i].tag = get_tag_if_needed_for this_type(request_type, blocks[i].virt);
  }
  return ENC_ERR_NONE;
}
```

release()

Synopsis

```
s32 release(encoder_interface *this,
            MemBlkDesc        **blocks);
```

Release a block of memory allocated via the API.

Parameters

this	A pointer to the encoder's exported interface.
blocks	A null-terminated array of descriptors for the memory to be released - one descriptor per block.

Return Value

The common values that all implementations of this function return are shown below.

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Unspecified error

Example

```
s32 my_release(encoder_interface *this,
           MemBlkDesc        **blocks)
{
  while blocks[0] != NULL
  {
    free_according_to_type (blocks[0].virt,blocks[0].flags);
    blocks++;
  }
  return ENC_ERR_NONE;
}
```

attention()

Synopsis

```
s32 attention(u32 encoder_interface *this,
              u32 cause);
```

Receive notification of some encoder event that requires external intervention.

Parameters

this	Pointer to the encoder's exported interface.
cause	Event requiring attention.

Cause Codes

No cause codes are defined for this function in the base API specification. See your encoder reference for details of any events requiring application support.

Return Value

The common values that all implementations of this function return are shown below.

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Unspecified error

Example

```
s32 my_attention(u32 encoder_interface *this,
                 u32 cause)
{
   switch (cause)
   {
      case SIMPLY_DONE:
         fix_the_problem_here();
         break;

      case INTERVENE_IN_ENCODER:
         function_code = get_code_for_this_request(cause);
         control_data = get_data_for_this_request(cause);
         this->control(this->handle,
                       function_code,
                       control_data.typeinfo) != ENC_ERR_NONE)
         break;

      default:
         return ENC_ERR_FAILED;
   }
   return ENC_ERR_NONE;
}
```

Imported Interface Structure

As with the exported functions, an interface structure is provided to aggregate a set of imported functions:

```
typedef struct
{
  fp_enc_frames_ready   frames_ready;
  fp_enc_data_needed    data_needed;
  fp_enc_allocate       allocate;
  fp_enc_release        release;
  fp_enc_attention      attention;
} encoder_import_interface;
```

The type definitions used in this structure being:

```
typedef s32 (*fp_enc_frames_ready) (encoder_interface *,
                                    EncPacket **,
                                    u32);
typedef s32 (*fp_enc_data_needed)  (encoder_interface*,
                                    RawFrame **,
                                    u32 *);
typedef s32 (*fp_enc_allocate)     (encoder_interface *,
                                    u32,
                                    u32,
                                    u32
                                    MemBlkDesc **);
typedef s32 (*fp_enc_release)      (encoder_interface *,
                                    MemBlkDesc **);
typedef s32 (*fp_enc_attention)    (encoder_interface *,
                                    u32);
```

NOTE Unlike the exported interface, the use of which is optional, the imported interface must be used to pass service functions to the API. Unused entries must be set to 0.

The imported interface currently in use can be obtained by making an `ENC_FN_GET_IMPIF` request to an encoder wrapper's **control** function. An imported interface can be modified using the `ENC_FN_SET_IMPIF` control request.

NOTE A default imported interface is set up when the API is initialized. Use the following procedure to set up a non-standard imported interface:

1. Retrieve the current imported interface using `ENC_FN_GET_IMPIF`.
2. Modify the retrieved interface, as needed.
3. Activate the new interface using `ENC_FN_SET_IMPIF`.

Error Codes

The Base API defines the error codes summarized in the table Core API Error Codes. Encoder-specific implementations add error codes as appropriate . See the individual encoder documentation for details.

Table 12 Core API Error Codes

Value	Meaning
ENC_ERR_NONE	No error - operation completed successfully
ENC_ERR_FAILED	Operation could be completed
ENC_ERR_NOTINIT	Encoding session needs to be initialized first
ENC_ERR_NOFUNC	Unrecognized or unsupported function request code
ENC_ERR_BAD_IMPIF	Incorrectly formed import interface
ENC_ERR_CODINGTYPE	Coding type unsupported by encoder
ENC_ERR_OOM	Out of memory
ENC_ERR_BADARG	Error in encoder-specific data
ENC_ERR_NOMOREINSTANCES	All available instances of the encoder are in use
ENC_ERR_BADHANDLE	Unrecognized session ID
ENC_ERR_NOSTATS	Encoding statistics unavailable
ENC_ERR_EOF	End of input encountered
ENC_ERR_OUTOFOPS	Output packet array overflow

Management Functions

In order to simplify the use of the API, a small set of functions is provided to create and initialize encoder interfaces and API data structures:

- create_encoder() Create a populated exported interface for a specific encoder type
- release_encoder() Destroy an encoder interface created by **create_encoder()**
- create_argument() create an instance of a given data type with the type descriptor information fully populated
- destroy_argument() destroy an argument created using **create_argument()**.

The objects performing the creation use an integer code to identify the type of the object to be created; these type codes are given for each type throughout the API documentation.

An additional set of functions are required to initialize and tear down the API:

- init_arc_encoder_api()
- uninit_arc_encoder_api()

Similarly each encoder type provides initialization and de-initialization, if its management functions are required:

- init_mgmt_if()
- uninit_mgmt_if()

create_encoder()

Synopsis

```
encoder_interface* create_encoder (u32 encoder_type,
                                   u32 handle)
```

Creates an export interface for the specified encoder type.

Parameters

encoder_type	Identity code for the required encoder type
handle	Encoding session ID to be placed into the resulting interface

Return Value

This function returns the address of a fully populated encoder export interface, or 0 if the request failed.

release_encoder()

Synopsis

```
s32 release_encoder(u32 encoder_type,
                    u32 handle);
```

Release an encoder interface structure previously created by create_encoder().

NOTE This function does not de-initialize the encoding session. Do so separately before calling this function.

Parameters

encoder_type	Identity code for the required encoder type
handle	Encoding session ID to be placed into the resulting interface

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

create_argument()

Synopsis

```
EncArgument *create_argument  (u32 typecode);
```

Create an instance of an API-defined custom data type.

Parameters

typecode Identity code of the requested data type

Return Value

This function returns a pointer to an instance of the requested data type if successful, 0 otherwise.

destroy_argument()

Synopsis

```
s32  destroy_argument (EncArgument *victim);
```

Release the memory held by an object created using create_argument

Arguments

victim A pointer to the object to be released.

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

init_arc_encoder_api()

Synopsis

```
s32 init_arc_encoder_api()
```

Initialize the encoder API prior to starting encoding.

NOTE This function initializes the API services, in particular the dedicated memory allocators. It is imperative that the API be initialized before attempting any activity on the encoders.

Arguments

None.

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

uninit_arc_encoder_api()

Synopsis

```
s32 uninit_arc_encoder_api()
```

Tear down the encoder API.

Arguments

None.

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

init_mgmt_if()

Synopsis

```
s32 init_mgmt_if()
```

Initialize the encoder-specific sections of the API.

NOTE This function only registers the encoder's custom data types and export interface with the API for use with the create_argument()and create_encoder() functions. Its use is only mandatory if these functions are needed.

Arguments

None

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

uninit_mgmt_if()

Synopsis

```
s32 uninit_mgmt_if()
```

Tear down the encoder-specific sections of the API.

Arguments

None.

Return Value

ENC_ERR_NONE	Success
ENC_ERR_FAILED	Request failed

Example Use of the Management Functions

An illustrative use of the management functions is shown in the example Example usage of the Management Functions. Again, the functions that would be provided by an API implementation are highlighted in *emphasis*.

Example 3 Example usage of the Management Functions

```
s32 result;
MyEncoderDataType* myenc_data;
encoder_interface* myenc_exportif;
u32 handle = get_session_id();

// Initialize the API
result = init_arc_encoder_api();
check_for_error_and_handle_appropriately(result);
result = init_myencoder_mgmt_if();
check_for_error_and_handle_appropriately(result);

// Get interface and a data object
myenc_exportif = create_encoder(handle);
myenc_data = create_argument(MYENC_DATA_TYPE);
if ((myenc_exportif == 0) || (myenc_data == 0))
  bit_of_a_problem();

// Now can use them to do something useful
result = myenc_exportif->control(handle, MYFUNCTIONCODE, ENC_PTRARG(myenc_data));

// If we ever get to the point where we want to stop using the encoders,
// do the following the order shown
uninit_myencoder_mgmt_if();
uninit_arc_encoder_api();
```

Platform Support

The ARC Encoder API as supplied is targeted for standalone use on an ARC processor, that is, with no operating system, relying on the MetaWare C libraries to provide a run-time environment. The API is intended to be easily portable to new runtime environments capable of supporting the ARC Video Subsystem Family hardware. This section is intended to act as a primer for developers faced with this porting task.

Platform Dependencies

The API has been designed to isolate the encoder from the runtime platform so far as efficiently possible. This is largely done via the imported interface functions:

- allocate
- release
- attention

allocate

This is the means by which an encoder will request a system resource - currently this is limited to blocks of dedicated system memory for input and output. A request for memory will indicate the purpose the memory is to be used for, the number of identically-sized blocks required, the size of each block, and the address alignment on which the blocks are to placed.

release

This function has the opposite effect to `allocate` - its releases the resource back to the outer environment for reuse.

attention

This function performs a catchall role: any service request from the encoder that cannot be accommodated by any other means will be delivered using this function.

Porting the Encoders to a New Platform

Implementing allocate and release

The allocate and release functions are needed for the API to work properly. They should be implemented and accessed by the default import interface created by the API.

The **allocate** function is presented with a set of arguments, three of which indicate what it should allocate:

- *num_blocks* indicates the number of blocks it should allocate;
- *size* indicates the size in bytes of each block;
- *flags* provides a set of flags providing details of how the memory should be placed:
  - The purpose of the block can be used to select the device or the region in the system memory map from which to allocate;
  - The alignment flags indicate the byte alignment on which each block should be placed;
  - The information flags provide additional hints; importantly, the ENC_MEM_DC_PAD flag indicates that each block should have sole usage of the data cache lines that it occupies. This

requires that the base address of each block be a multiple of the data cache line length, and that the size of each block be a multiple of the data-cache-line length.

Cache Management

The ARC Video Subsystem Family makes extensive use of DMA engines to transfer data to and from its internal memories. Since the data being transferred consists of the encoding application's input and output data, particular attention must be paid to managing the data cache in order to keep it in synchronization with the actual contents of memory. In particular, the following points should be noted:

- Data that is to be read into the encoder via DMA needs to be flushed out to main memory before the DMA starts;
- The cache lines corresponding to data that has been written out from the encoder via DMA must be invalidated prior to access by the main processor;
- The ARC processor permits access to memory in a cache-bypass mode: see the *ARCompact Programmer's Reference Manual* for details of the **.di** forms of the **ld** and **st** instructions, and the *MetaWare C/C++ Language Reference* for details on the **_Uncached** keyword;
- The data cache may be disabled completely by setting its control register appropriately. This course of action is discouraged owing to the resulting impact on system performance.

It is also important to bear in mind the correspondence between memory and the cache when performing cache management - in particular, the invalidation of a cache line for DMA purposes can be undermined if a variable on that cache line is modified by program code before the DMA has completed. It is recommended that memory being used for DMA purposes is sized and aligned so as to have sole occupancy of data cache lines.

Cache Control Using the MMU

Where the ARC 700 MMU is available, cache control can be achieved by creating a set of page mappings onto the DMA buffers, and setting the protection bits in the associated page table entries accordingly. Please refer to the ARC700 MMU Reference Guide for further details.

Threading

The Video Encoder API is inherently single-threaded. It is strongly recommended that any deployments of the encoder in a multi-threaded environment maintain both the API and any application code interacting with the API in a single thread.

Also note that the API and application should occupy the memory space, since objects are passed over the API by reference. Thus, for example, the encoding application and API must be in the same virtual address space on a protected mode operating system.

API Initialization and Teardown

The API provides a pair of management functions to set up and tear down the API.

- init_arc_encoder_api()
- uninit_arc_encoder_api()

These functions are intended to be used for platform specific initialization such as claiming resources for the lifetime of the API, and subsequent teardown, if needed. It is recommended that selection of the platform initialization functions be performed by the use of C preprocessor definitions, so that the type of platform supported is a build-time option.

What the API Does Not Cover

Heap and Stack Management

The API does not interfere with the C run-time **malloc()** function and **new** operator. If a custom heap management strategy is needed, appropriate versions of these functions should be made available in the C runtime libraries when building the encoder, API and application. Placement and sizing of the heap and stack can also be achieved by the use of a linker command file at build time. Refer to the MetaWare documentation for details about how to do this with the MetaWare tools.

Devices and File systems

Device access and I/O are the responsibility of code outside of the encoder and API. There is no code in either the encoder or API that requires any form of I/O device or file system - this should be addressed at the application level.