Sequential JPEG Encoder Codec on DM355

User's Guide

Literature Number: SPRUFE5C
November 2007 – Revised April 2008

Contents

List of Figures

List of Tables

Read This First

This document describes how to install and work with Texas Instruments (TI) JPEG encoder implementation on the DM355 platform. It also provides a detailed application programming interface (API) reference and information on the sample application that accompanies this component.

About This Manual

TI codec implementations are based on the eXpressDSP digital media (xDM) standard. xDM is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS).

Intended Audience

This document is intended for system engineers who want to integrate TI codecs with other software to build a multimedia system based on the DM350 platform.

This document assumes that the reader is fluent in the C language, and have working knowledge of JPEG encoder. Good knowledge in eXpressDSP Algorithm Interface Standard (XDAIS), eXpressDSP digital media (xDM) standard, and IDMA3 will be helpful.

How to Use This Manual

This document includes the following chapters:

- Chapter 1 - Introduction, introduces the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- Chapter 2 - Installation Overview, describes how to install, build, and run the codec.
- Chapter 3 - Sample Usage, describes the sample usage of the codec.
- Chapter 4 – Features Supported, describes the additional features supported in jpeg encoder.
- Chapter 5 - API Reference, describes the data structures and interface functions used in the codec.

Related Documentation From Texas Instruments

The following documents describe TI DSP algorithm standards such as, XDAIS and XDM. To obtain a copy of any of these TI documents, visit the Texas Instruments website at www.ti.com.

- *TMS320 DSP Algorithm Standard API Reference* (SPRU360) describes all the APIs that are defined by the TMS320 DSP Algorithm Interface Standard (also known as XDAIS) specification.
- *A Technical Overview of eXpressDSP - Compliant Algorithms for DSP Software Producers* (SPRA579) describes how to make algorithms compliant with the TMS320 DSP Algorithm Standard which is part of TI's eXpressDSP technology initiative.
- *xDAIS-DM (Digital Media) User Guide* (SPRUEC8)
- *Using DMA with Framework Components for C64x+* (SPRAAG1)

Related Documentation

You can use the following documents to supplement this user's guide:

CCITT Recommendation T.81, specifying the JPEG standard. Available at http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

Abbreviations

The following abbreviations are used in this document.

Table 1. List of Abbreviations

Abbreviation	Description
CIF	Common Intermediate Format
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DMAN3	DMA Resource Manager
EVM	Evaluation Module
IDMA3	DMA Resource specification and negotiation protocol
JPEG	Joint Photographic Experts Group
MCU	Minimum Coded Unit
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media
YUV	Raw Image format
	Y: Luminance Component
	U,V : Chrominance components
Exif	Exchangeable image file format
JFIF	JPEG File Interchange Format

Text Conventions

The following conventions are used in this document:

- Text inside back-quotes (‘‘) represents pseudo-code.
- Program source code, function and macro names, parameters, and command line commands are shown in a mono-spaced font.

Product Support

When contacting TI for support on this codec, please quote the product name (JPEG Encoder on DM355) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

Code Composer Studio and eXpressDSP are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Software Copyright

Introduction

This chapter introduces XDAIS, XDM, and IDMA3. It also provides an overview of TI implementation of the JPEG Encoder on the DM355 platform and its supported features.

1.1 Overview of XDAIS, XDM, and IDMA3

TI's multimedia codec implementations are based on the eXpressDSP Digital Media (XDM) standard. XDM is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS). IDMA3 is the standard interface to algorithms for DMA resource specification and negotiation protocols. This interface allows the client application to query and provide the algorithm with its requested DMA resources.

1.1.1 XDAIS Overview

An eXpressDSP-compliant algorithm is a module that implements the abstract interface IALG. The IALG API takes the memory management function away from the algorithm and places it in the hosting framework. Thus, an interaction occurs between the algorithm and the framework. This interaction allows the client application to allocate memory for the algorithm and also to share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. To facilitate these functions, the IALG interface defines the following APIs:

- algAlloc()
- algInit()
- algActivate()
- algDeactivate()
- algFree()

The algAlloc() API allows the algorithm to communicate its memory requirements to the client application. The algInit() API allows the algorithm to initialize the memory allocated by the client application. The algFree() API allows the algorithm to communicate the memory to be freed when an instance is no longer required.

Once an algorithm instance object is created, it can be used to process data in real-time. The algActivate() API provides a notification to the algorithm instance that one or more algorithm processing methods is about to be run zero or more times in succession. After the processing methods have been run, the client application calls the algDeactivate() API prior to reusing any of the instance's scratch memory.

The IALG interface also defines three more optional APIs: algControl(), algNumAlloc(), and algMoved(). For more details on these APIs, see the *TMS320 DSP Algorithm Standard API Reference Guide* (SPRU360).

1.1.2 XDM Overview

In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building an image encoder system, you can use any of the available image encoders (such as JPEG, PNG, or JPEG2000) in your system. To enable easy integration with the client application, it is important that all codecs with similar functionality use similar APIs. XDM was primarily defined as an extension to XDAIS to ensure uniformity across different classes of codecs (for example audio, video, image, and speech). The XDM standard defines the following two APIs:

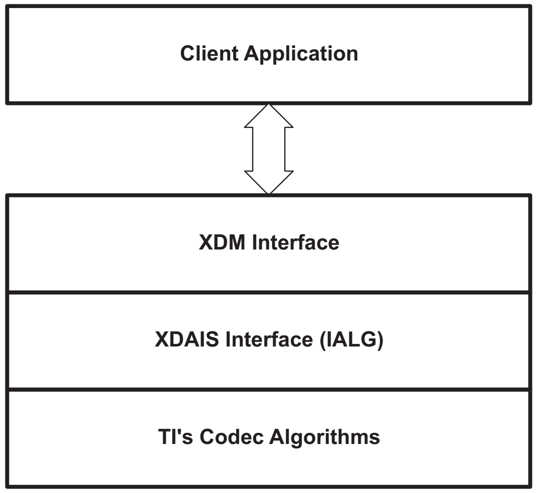
- control()
- process()

The control() API provides a standard way to control an algorithm instance and receive status information from the algorithm in real-time. The control() API replaces the algControl() API defined as part of the IALG interface. The process() API does the basic processing (encode/decode) of data.

Apart from defining standardized APIs for multimedia codecs, XDM also standardizes the generic parameters that the client application must pass to these APIs. The client application can define additional implementation specific parameters using extended data structures.

Figure 1-1 depicts the XDM interface to the client application.

Figure 1-1. XDM Interface to the Client Application



As depicted in Figure 1-1, XDM is an extension to XDAIS and forms an interface between the client application and the codec component. XDM insulates the client application from component-level changes. Since TI's multimedia algorithms are XDM compliant, it provides you with the flexibility to use any TI algorithm without changing the client application code. For example, if you have developed a client application using an XDM-compliant JPEG still image encoder, then you can easily replace JPEG with another XDM-compliant image encoder with minimal changes to the client application.

For more details, see the *xDAIS-DM (Digital Media) User Guide* (SPRUEC8).

1.1.3 IDMA3 Overview

Client applications use the algorithm's IDMA3 interface to query the algorithm's DMA resource requirements and grant the algorithm logical DMA resources via handles. Figure 1-2 shows a typical IDMA3 interface implemented by the algorithm module, which would be used by the client applications to query algorithms DMA needs. The algorithm specifies the number of separate EDMA/QDMA channels and PaRamSets it required, through memRecs. The IDMA3 standard defines the following APIs:

- dmaChangeChannels()
- dmaGetChannelCnt()
- dmaGetChannels()
- dmaInit()

The dmaChangeChannels( ) API is called by an application whenever logical channels are moved at run-time. This allows for the application to re-initialize the channel properties whenever allocated resources are not available. dmaGetChannelCnt() is called by an application to query an algorithm about its number of logical DMA channel requests. dmaGetChannels() is called by an application to query an algorithm about its DMA channel requests at initialization time, or to get the current channel holdings. Through this API, the algorithm specifies the number of TCCs and PaRamSets it would require and the properties of these resources when called during initialization time. dmaInit() is called by an application to grant DMA handle(s) to the algorithm at initialization.

For more details, see *Using DMA with Framework Components for C64x+ Application Report* (SPRAAG1).

1.2 Overview of JPEG Encoder

JPEG is the ISO/IEC recommended standard for image compression.

See the CCITT Recommendation T.81, specifying the JPEG standard document at http://www.w3.org/Graphics/JPEG/itu-t81.pdf for details on JPEG encoding/decoding process.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of JPEG Encoder on the DM355 platform.

This version of the codec has the following supported features of the standard:

- eXpressDSP™ Algorithm Interface Standard (XDAIS) software compliant
- eXpressDSP Digital Media (xDM) interface and IDMA3 compliant
- Implements IIMGENC1 interface of xDM
- Supports JPEG baseline DCT encoding process with following limitations:
  - Non-interleaved scans are not supported.
  - Huffman tables and quantization tables for U and V components must be same.
  - No support for user defined Huffman tables. Default Huffman tables are used.
  - No support for number of components other than 3.
- Supports YUV 4:2:0/4:2:2 planar and YUV 4:2:2 interleaved data as an input
- Supports YUV422 and YUV420 planar encoded format
- Supports arbitrary image width and height (minimum width/height requirement of 64 pixels)
- Images with resolutions up to 1000 Mpixels can be encoded. This is the theoretical maximum; however, only images up to 10 Mpixels have been tested. If the codec memory and I/O buffer requirements exceed the DDR memory availability for frame based encoding, use ring buffer and slice mode encoding to encode higher resolution images.
- Supports restart interval
- Quantization tables are fixed with a quality factor (1 – 97) adjusting the quantization level
- Supports ring buffer configuration of bitstream buffer for reducing buffer size requirement
- Supports Rotation by 0°, 90°, 180° and 270°
- Supports frame based encoding
- Supports slice mode encoding
- Supports frame level re-entrancy
- Supports multi instance of JPEG Encoder, and single/multi instance of JPEG Encoder with other DM355 codecs
- Validated on DM355 EVM

1.4 Limitations

The limitations will not be removed in future releases. These limitations are not defects but intentional or known deficiencies.

- Does not support extended DCT based encoding process
- Does not support loss-less encoding process
- Does not support hierarchical encoding process
- Does not support progressive scan
- Minimum image width/height requirement of 64 pixels
- Huffman tables are fixed by algorithm
- No support for number of components other than 3.
- Ring buffer size should be multiples of 4096 bytes
- Includes a standard JPEG header. Does not include a JFIF or EXIF style header. The application is expected to insert the APP0 (JFIF) or APP1 (EXIF) markers to create a JFIF or EXIF style header.
- Only limited support of IDMA3 interface. See Section 3.1 for details.

Chapter 2

SPRUFE5C–November 2007–Revised April 2008

Installation Overview

This chapter provides a brief description on the system requirements and instructions for installing the codec component. It also provides information on building and running the sample test application.

Topic **Page**

2.1 System Requirements

This section describes the hardware and software requirements for the normal functioning of the codec component.

2.1.1 Hardware

This codec has been tested as an executable on DM355 EVM board.

2.1.2 Software

The following are the software requirements for the normal functioning of the codec:

- Linux: Monta Vista Linux 4.0.1
- Code Generation Tools: This project is compiled, assembled, and linked using the arm_v5t_le-gcc compiler.

2.2 Installing the Component

To install the codec, follow the instructions in the Release notes. The code location is as follows:

JPEG Encoder algorithm code is in a directory jpegenc placed in DM355Codecs/release.

Figure 2-1 shows the sub-directories structure of jpegenc directory.

Figure 2-1. Component Directory Structure

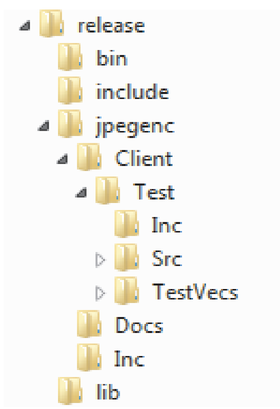


Table 2-1 provides a description of the sub-directories created in the release/jpegenc directory.

Table 2-1. Component Directories

Sub-Directory	Description
jpegenc /Docs	Contains user guide, datasheet, and release notes
jpegenc /Client/Test/Src	Contains application C files
jpegenc /Client/Test/Inc	Contains header files needed for the application code
jpegenc /Client/Test/TestVecs	Contains test vectors and configuration files
/Include	Contains the include files needed by application and codec
/lib	Contains JPEG Encoder and other support libraries
/bin	Contains JPEG Encoder executable “jpgenc”

The DM355 JPEG encoder library is put into the /release/lib directory and the xDM headers are put in /release/include directory.

2.3 Building the Sample Test Application on Linux

The sample test application that accompanies this codec component will take input YUV files and dumps output JPEG files as specified in the in the command line arguments. To build and run the sample test application in Linux, follow these steps:

Step 1. Verify that libjpgenc.a library is built and present in release/lib directory.

Step 2. Verify that support libraries (libimx.a, libimcop.a, libcosl.a, libdm355.a, libcmem.a) are present in DM355Codecs/release/lib directory.

Step 3. Change directory to /jpegenc/Client/Test/Src and type ‘make clean’ command followed by a ‘make’ command. This will use the makefile in that directory to build the test executable “jpgenc” into the release/bin/

Note: You must set the ARM tool chain i.e arm_v5t_le-gcc (ARM gcc) compiler path in your user's environment path before building the MPEG4 decoder executable.

To run the jpgencoder executable on DM355 EVM board, see the following instructions.

Step 4. Set up the DM355 EVM Board.

For information about setting up the DM355 environment, see the *DM355 Getting Started Guide* released in the “doc” directory in the DVSDK release package.

Step 5. Run the JPEG encoder executable

- For running the JPEG encoder executable, copy the executable "jpgenc" along with the entire "TestVecs" directory provided with the release package at project/jpegenc/Client/Test to the target directory.
- Copy the kernel modules `dm350mmap.ko` and `cmemk.ko` to the target directory. These modules are provided with the release package in project/bin directory.
- Copy `loadmodules.sh` provided with release package at project/bin to the target directory.
- Execute the following commands in sequence to run the JPEG encoder executable:

```
$./loadmodules.sh
$./jpgenc
```

This will run the JPEG encoder with base parameters.

- To run the JPEG encoder with extended parameters, change the config file in testvecs.cfg to testparams.cfg (TestVecs/Config/) and execute:

```
$./jpgenc -ext
```

2.4 Configuration Files

This codec is shipped along with:

- A generic configuration file (Testvecs.cfg) – specifies input yuv file, output file and parameter file for each test case.
- An Encoder parameter file (Testparams.cfg) – specifies the configuration parameters used by the test application to configure the encoder for a particular test case.
- The JPEG encoder has two modes: extended parameters mode and base parameters mode, which can be specified in a command line argument, as mentioned above.

2.4.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, Testvecs.cfg, for determining the input and output files for running the codec. The Testvecs.cfg file is available in the /Client/Test/TestVecs/Config sub-directory.

The format of the Testvecs.cfg file is:

```
X
Config
```

```
Input
Output
```

where:

- X must be set to 0 - for output dumping.
- Config is the Encoder configuration file. For details, see Section 2.4.2.
- Input is the input file name (use complete path).
- Output is the output file name (use complete path).

A sample Testvecs.cfg file is as shown:

```
0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\Input.yuv
..\..\Test\TestVecs\Output\Output.jpg
```

2.4.2 Encoder Configuration File for Base Parameters

The encoder configuration file, Testparams.cfg, contains the configuration parameters required for the encoder. The Testparams.cfg file is available in the /Client/Test/TestVecs/Config sub-directory.

A sample Testparams.cfg file is as shown:

```
# <ParameterName> = <ParameterValue> # Comment
#
##########################################################################################
# Parameters
##########################################################################################
maxHeight = 480
maxWidth  = 720
maxScans  = 15
dataEndianness = 1
forceChromaFormat = 2
inputChromaFormat = 4
inputWidth = 720
inputHeight = 480
captureWidth = 720
numAU = 0
genHeader = 0
qValue = 97
```

2.4.3 Encoder Configuration File for Extended Parameters

The encoder configuration file, Testparams.cfg contains the configuration parameters required for the encoder. The Testparams.cfg file is available in the /Client/Test/TestVecs/Config sub-directory

A sample Testparams.cfg file is as shown:

```
# <ParameterName> = <ParameterValue> # Comment
#
##########################################################################################
# Parameters
##########################################################################################
maxHeight = 480
maxWidth  = 720
maxScans  = 15
dataEndianness = 1
forceChromaFormat = 2
inputChromaFormat = 4
inputWidth = 720
inputHeight = 480
captureWidth = 720
numAU = 0
genHeader = 0
qValue = 97
rstInterval = 84
rotation = 0
disableEOI = 0
```

Sample Usage

This chapter provides a detailed description of the sample test application that accompanies this codec component.

Topic **Page**

3.1 JPEG Encoder Client interfacing constraints

The following constraints should be taken into account when implementing the client for the JPEG encoder library in this release:

- DMA requirements of JPEG encoder: Current implementation of the JPEG encoder uses the following TCCs for its DMA resource requirements, along with its associated PaRamSets. Apart from these 16 TCCs requirements, it also needs 23 more PaRamSets that are allocated through the IDMA3 interface.

Channel Number	Associated PaRamSet Numbers
34 to 49	34 to 49 (PaRamSet number = channel number)

- The client application should map all the DMA channels used by the JPEG encoder to the same queue. This is required for the codec to function normally. The codec does not map channels to queue.
- If there are multiple instances of a codec and/or different codec combinations, the application can use the same group of channels and PaRAM entries across multiple codecs. The AlgActivate and AlgDeactivate calls made by client application that are implemented by the codecs perform context save/restore to allow multiple instances of same codec and/or different codec combinations.
- Since all codecs use the same hardware resources, only one process call per codec should be invoked at a time (frame level reentrancy). The process call needs to be wrapped within activate and deactivate calls for context switch. Refer to XDM specification on activate/deactivate.
- If multiple codecs are running with frame level reentrancy, the client application has to perform time multiplexing of process calls of different codecs to meet desired timing requirements between video/image frames.
- The ARM and DDR clock must be set to the required frequency for running single or multiple codecs.
- The codec combinations feasibility is limited by processing time (computational hardware cycles) and DDR bandwidth.
- Codec atomicity is supported at frame level processing only. The process call has to run to completion before another process call can be invoked.

3.2 Overview of the Test Application – Usage in Single Instance Scenario

The test application exercises the IIMGENC1_Params extended class of the JPEG Encoder library. The main test application files are jpgeTest355.c and testFramework.h. These files are available in the /Client/Test/Src and /Client/Test/Inc sub-directories, respectively.

Figure 3-1 depicts the sequence of APIs exercised in the sample test application.

Figure 3-1. Test Application Sample Implementation

Integration Layer	XDM-XDIAS-IDMA3 Interface	Codec Library
Param Setup		
Algorithm Instance Creation and Initialization	algNumAlloc () → algAlloc () → algInit () →	
DMA Channels Request and Granting	dma ChannelCnt () → dmaGetChannels () → dmaInit () →	
Process Call	algActivaate () → process () → algDeactivate () →	
Algorithm Instance Deletion	algNumAlloc () → algFree () →	

The test application is divided into four logical blocks:

- Parameter setup
- Algorithm instance creation and initialization
- Process call
- Algorithm instance deletion

3.2.1 Parameter Setup

Each codec component requires various codec configuration parameters to be set at initialization. The test application obtains the required parameters from the command line.

In this logical block, the test application does the following:

1. Reads the configuration parameters from the command line
2. Sets the IIMGENC1_Params structure based on the values it read
3. Reads the input YUV image into the application input buffer

After successful completion of the above steps, the test application does the algorithm instance creation and initialization.

3.2.2 Algorithm Instance Creation and Initialization

In this logical block, ALG_create() is called by the test application and accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs implemented by the codec are called in sequence by ALG_create():

1. algNumAlloc() - To query the algorithm about the number of memory records it requires.
2. algAlloc() - To query the algorithm about the memory requirement to be filled in the memory records.
3. algInit() - To initialize the algorithm with the memory structures provided by the application.

A sample implementation of the create function that calls algNumAlloc(), algAlloc(), and algInit() in sequence is provided in the ALG_create() function implemented in the alg_create.c file.

In addition, ALG_create() uses some APIs that deal with memory allocation, such as: _ALG_allocMemory(), _ALG_freeMemory(). They are provided in file alg_malloc.c .

Apart from algorithm memory allocation, the application needs to call the IDMA3_Create() function. This function uses the algorithm instance created in the previous call of ALG_create and provides the algorithm with the requisite DMA resources. The following APIs implemented by the algorithm are called in the following sequence:

1. dmaGetChannelCnt() - To query the algorithm about the number of memory records it requires. In the present implementation, it always defaults to 1.
2. dmaGetChannels() - To query the algorithm about the number of additional PaRamSets it requires in the channel records. In the current implementation, the algorithm uses hard-coded channels and its associated TCCs and PaRamSets internally. The client application that is using the algorithm's IDMA3 interface allocates additional PaRamSets requirements.
3. dmaInit() - To initialize the algorithm with continuous PaRamSet addresses allocated to the algorithm during this instance. A sample implementation of this function is included in the idma3_create.c file.

3.2.3 Process Call in Single Instance Scenario

After algorithm instance creation and initialization, the test application does the following:

1. Calls algActivate(), which initializes the encoder state and some hardware memories and registers.
2. Sets the input and output buffer descriptors required for the process() function call.
3. Calls the process() function to encode a single frame of data. The inputs to the process function are input and output buffer descriptors, and a pointer to the IIMGENC1_InArgs and IIMGENC1_OutArgs structures. The process() function should be called multiple times to encode multiple images.
4. Call algDeactivate(), which performs releasing of hardware resources and saving of encoder instance values.
5. process() is made a blocking call, but an internal OS specific layer enables the process to be pending on a semaphore while hardware performs a complete JPEG encode.
6. Other specific details of the process() function remains the same as described in section 3.1.3 and the constraints described in section 3.1.1 are applicable.

Note: algActivate () must be called at least once after codec instance creation and before the first call to process(), as it does hardware initialization.

3.2.4 Algorithm Instance Deletion

Once encoding is complete, the test application must delete the current algorithm instance. The following APIs are called in sequence:

1. algNumAlloc() - To query the algorithm about the number of memory records it used
2. algFree() - To query the algorithm to get the memory record information and then free them up for the application

A sample implementation of the delete function that calls algNumAlloc() and algFree() in sequence is provided in the ALG_delete() function implemented in the alg_create.c file.

3.3 Usage in Multiple Instance Scenario

If the client application supports multiple instances of JPEG encoder, the initialization and process calls are altered. One of the main issues in converting a single instance encoder to a multiple instance encoder is resource arbitration and data integrity of shared resources between various codec instances. Resources that are shared between instances and need to be protected include:

- DMA channels and PaRamSets
- MPEG-4-JPEG co-processor and their memory areas

To protect one instance of the JPEG encoder from overwriting into these shared resources when the other instance is actually using them, the application needs to implement mutexes in the test applications. The application developer can implement custom resource sharing mutex and call algorithm APIs after acquiring the corresponding mutex. Since all codecs (JPEG encoder/decoder and MPEG-4 encoder/decoder) use the same hardware resources, only one codec instance can run at a time.

Here are some of the API combinations that need to be protected with single mutex.

- `dmaInit()` of one instance initializes DMA resources when the other instance is actually active in its process() function.
- `control()` call of one instance sets post-processing function properties by setting the command length etc., when the other instance is active or has already set its post processing properties.
- `process()` call of one instance tries to use the same hardware resources [co-processor and DMA] when the other instance is active in its process() call.

If multiple instances of the JPEG encoder are used in parallel, the hardware must be reset between every process call and algorithm memory to be restored. This is achieved by calling algActivate() and algDeactivate() before and after process() calls.

Thus, the Process call section as explained in the previous section must change to include both algActivate() and algDeactivate()as mandatory calls for the algorithm.

3.3.1 Process Call with algActivate and algDeactivate

After algorithm instance creation and initialization, the test application does the following:

1. Sets the input and output buffer descriptors required for the process() function call.
2. Calls algActivate(), which initializes the encoder state and some hardware memories and registers.
3. Calls the process() function to encode a single frame of data. The inputs to the process function are input and output buffer descriptors, pointer to the IIMGENC1_InArgs and IIMGENC1_OutArgs structures.
4. Call algDeactivate(), which performs releasing of hardware resources and saving of encoder instance values.
5. Other specific details of the process() function remains same as described in section 3.1.3 and constraints describe in section 3.1.1 are applicable.

Note: In the multiple instance scenario, algActivate() and algDeactivate() are mandatory function calls before and after process()respectively.

Feature descriptions

This chapter describes special features not commonly found in a standard JPEG encoder such as:

- Ring-buffer configuration of the output bit stream buffer
- Slice-mode processing
- More than one input color format
- Rotation

4.1 Bit-stream Ring buffer in DDR

To minimize the output buffer memory requirement, the JPEG encoder stores the JPEG bitstream into a circular or ring buffer residing in DDR, which acts as an intermediary storage area between the final storage media (SD card, HD, memory stick, etc.) and the encoder. Thus, the size of the ring buffer can be much smaller than the final bitstream's size, effectively reducing the amount of physical DDR memory allocated for storing the bitstream. The complete bitstream is eventually stored on the media because, as JPEG fills one half of the ring buffer, the application empties the other half onto the media. The JPEG encoder and the application operate in parallel and on a different half, thus sustaining the maximum JPEG processing throughput. Figure 4-1 through Figure 4-4 depict the state of the ring buffer at different states of JPEG encode processing:

Figure 4-1. Ring buffer before JPEG encoder starts

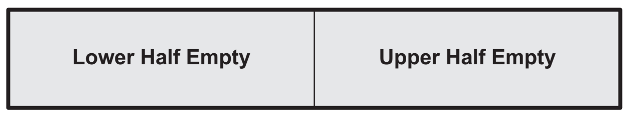


Figure 4-2. Ring buffer shortly after JPEG encoder starts

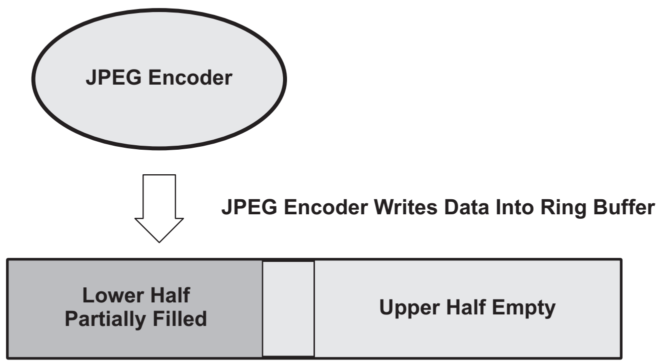


Figure 4-3. Ring buffer once JPEG encoder fills lower half

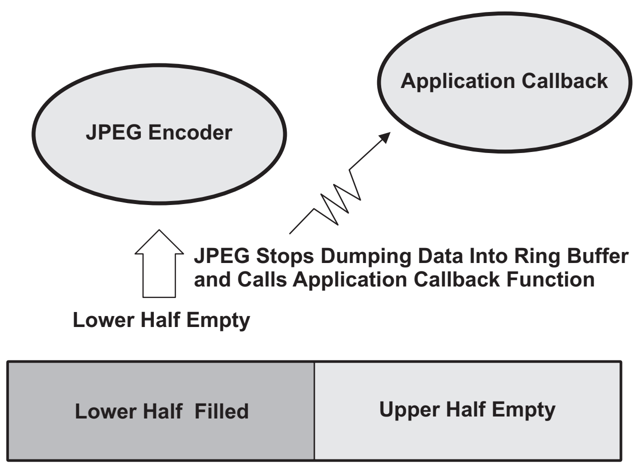
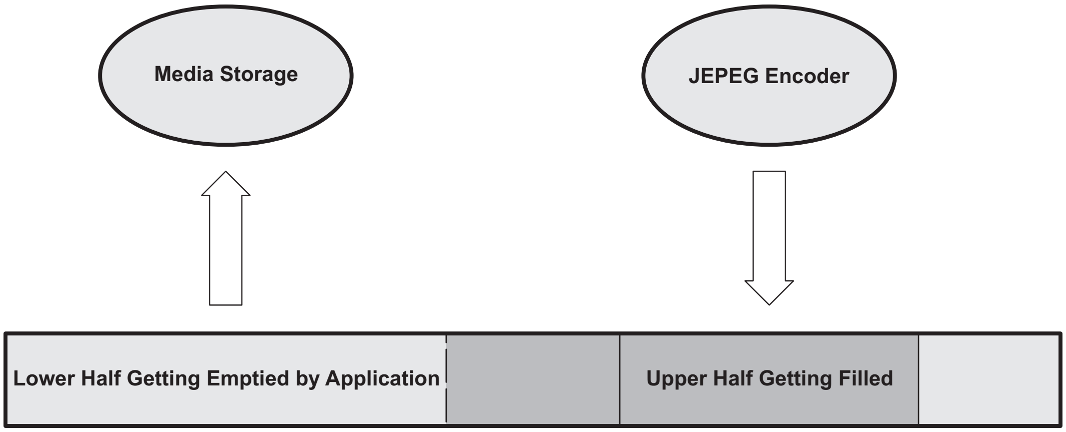


Figure 4-4. Ring Buffer Once Application Starts Filling First Half and JPEG Encode Starts Processing Second Half.



4.1.1 Mode of operation

The address and size of the ring buffer are passed to the JPEG encoder as input runtime arguments of the process function. The JPEG encoder manages this output ring buffer as follows.

As MCUs are encoded, the encoder fills the ring buffer with the generated bitstream. Each time half of the buffer is filled, the encoder will call a user-defined callback function. That callback function of type XDAS_Void (*halfBufCB)(Uint32 curBufPtr, XDAS_Void*arg)is passed to the encoder as a creation parameter during the ALG_create() function call.

The input argument curBufPtr is passed by the encoder and its value is the pointer to the first free byte in the ring buffer. All the bytes located before curBufPtr are valid bytes output by the encoder and that need to be saved into the storage media. The callback function must save curBufPtr so next time it is called, it knows where to save the data from. Note that the first time it is called is a special case, as the starting point of the valid data is the starting address of the ring buffer.

Note that successive values of curBufPtr are not necessarily in increasing order due to the circular nature of the ring buffer. The application must implement the case where curBufPtr rolls back to the beginning of the ring buffer

The second argument XDAS_Void*arg is a generic pointer that can be typecast to a pointer to a user-defined data structure and can be used by the application to pass extra information needed during the execution of the callback function. The example in Section 4.1.3 uses that feature to pass a structure that tracks the transfers between the ring buffer and the media storage.

4.1.2 Constraint

The ring buffer size must be a multiple of 4096 bytes.

4.1.3 Guidelines for Using Ring Buffer With JPEG Encoder

This section introduces few guidelines and tips to help you implement the ring buffer into an application using the JPEG encoder. It does not provide all the steps required to initialize/run the JPEG encoder, but only those related to ring buffer handling.

The following structure Ring2Media tracks the state of the transfers between the ring buffer and the storage media.

```
typedef struct Media2Ring{
  Int8* mediaPtr; // Pointer to first free location in the media buffer
  Int8* ringCurPtr; // Pointer to the first free location in the ring buffer
  Int8* ringStartPtr; // Pointer to the start of the ring buffer
  Int8* ringEndPtr; // Pointer to the end of the ring buffer
} Ring2Media;
```

The members mediaPtr and ringCurPtr are updated by the half-buffer callback function each time it is called.

Assuming there is a ring buffer array and media array defined as global:

```
Uint8 ringbuf[RINGBUFSIZE];
Uint8 media[MAX_IMG_WIDTH*MAX_IMG_HEIGHT*2];
```

The application creates and initializes an instance of Media2Ring as follows:

```
Ring2Meida ring2media={media, ringbuf, ringbuf, ringbuf + RINGBUFSIZE};
```

Note that the callback function that handles half-buffer can accept a second argument in addition to curBufPtr . Use this feature by passing the pointer to ring2meida to the callback function each time the encoder calls it.

The pointer to callback function and its second argument are passed to the encoder during creation time in the specific extended JPEG creation parameters structure extn_params of type IJPEGENC_Params.

```
extn_params.halfBufCB = (XDAS_Int32 (*)())JPEGENC_TI_DM355_HalfBufCB;
extn_params.halfBufCBarg= (void*) *)
```

Before calling the process() function, the starting address of ring buffer and its size are communicated to the encoder as run-time input parameters to the process function.

```
inArgs.ringBufStart= (XDAS_UInt8*)ringbuf;
inArgs.ringBufSize= RINGBUFSIZE;
```

The members ringCurPtr and mediaPtr of ring2media must be reinitialized to their initial values before each call to process() as the callback function updates them

```
ring2media.mediaPtr= media;
ring2media.ringCurPtr= ringbuf;
```

The process() function is normally called. During JPEG execution the half-buffer callback function is called by the codec each time half-buffer boundary is crossed. The responsibility of the callback function is to refresh the portion of data in the ring buffer delimited by ring2media.ringCurPtr and curBufPtr, the latter parameter being the first input argument of the callback function.

The callback function is also called at the end of JPEG processing by the codec to flush out the bitstream from the ring buffer into the storage media even though half buffer boundary is not reached.

The following is an example of half-buffer callback implementation using memcpy function for transfers. A more efficient implementation might use EDMA for memory transfers. The callback function should not wait for the EDMA transfers to complete before returning to JPEG to allow parallel processing with JPEG.

```
XDAS_Void JPEGENC_TI_DM355_HalfBufCB(XDAS_Int32 bufPtr, void *arg)
  {
    Uint32 i, x, y, numToXfer;
    Ring2Media *ring2media= arg;

/*
Detect if a pointer rollback occurred due the circular nature of the ring buffer,If it didn’t
occur then transfer is normal.
*/

    if ((XDAS_Int8*)bufPtr > ring2media ->ringCurPtr){
      numToXfer= (XDAS_Int8*)bufPtr- ring2media ->ringCurPtr;
      memcpy(ring2media ->ringCurPtr, ring2media ->mediaPtr, numToXfer);
     ring2media ->mediaPtr+= numToXfer;
     ring2media ->ringCurPtr+= numToXfer;
    }

     /*
If pointer rollback occurred then copy first end of the ring buffer into the storage media and
then copy the portion at the beginning of the ring buffer.
*/
    else {
      numToXfer= (XDAS_Int8*) ring2media ->ringEndPtr-
                          ring2media - >ringCurPtr;
      memcpy(ring2media ->ringCurPtr, ring2media ->mediaPtr, numToXfer);
      ring2media ->mediaPtr+= numToXfer;
      ring2media ->ringCurPtr= ring2media ->ringStartPtr;
      numToXfer= (XDAS_Int8*)bufPtr- ring2media ->ringStartPtr;
      memcpy(ring2media ->ringCurPtr, ring2media ->mediaPtr, numToXfer);
```

```
        ring2media ->mediaPtr+= numToXfer;
      ring2media ->ringCurPtr+= numToXfer;
    }

    return;
  }
```

Note how the members mediaPtr and ringCurPtr of the structure Ring2Media are updated. At the exit of the callback function, ring2meida->ringCurPtr should be the same value as bufPtr.

4.2 Slice-mode processing

Instead of processing an entire frame in one shot, JPEG can be configured so a call to process only encodes a slice of the frame.

To encode an entire frame, several calls to process function are needed. Between calls, it is possible to change the input pointer to YUV data and the output pointer.

This feature is useful for a system that doesn't have enough memory to store the YUV input data of the entire frame dumped by the sensor. The slice-based encode feature allows a smaller memory footprint to be used if the sensor can be controlled to dump any amount of YUV data at a chosen time. An entire frame is encoded by having the sensor dump a slice of data to a fixed location before the JPEG encodes it.

4.2.1 Slice Mode Processing Constraints

A slice size is expressed in number of MCUs and must be a multiple of the number of MCUs along the image's width, x 2. For instance, if the image width is W pixels and its color format is yuv422, then a slice size must be multiple of (W/16) x 2.

When slice mode processing is enabled, JPEG automatically inserts a restart marker at the end of each slice. Therefore, the slice size must remain constant in the processing of a frame; it is not possible to mix different slice sizes within the processing of the same frame. Only the last slice can be of different size because it ends with an EOI marker.

4.2.2 Slice Mode Processing Overhead

Because there is control overhead each time JPEG is started/stopped, you should try to process as few slices as possible per frame. For instance, a 1.2 Mpix frame partitioned in 20 slices would incur 15% overhead versus 11% overhead for a frame partitioned in 10 slices.

Also, the larger the frame is, the less affect the overhead has on the overall processing time. For instance, given a 4.4 Mpix frame, the overhead would only be 4% for a 20 slices frame and 2% for a 10 slices frame.

4.2.3 How to Operate Slice-Mode Processing Using JPEG APIs

Slice-mode processing is controlled by the run-time parameter numAU of the structure IIMGENC1_DynamicParams. Run time parameters are set when calling the control API. If numAU is set to XDM_DEFAULT, then entire frame will be encoded when the process API is called. Otherwise, it must be set to the number of MCUs contained in a slice.

The parameter numAU should be set such that it is multiple of (W/16) x 2, where W is the width of the image.

If that constraint is not respected, the encoder automatically rounds up numAU to the next valid value and returns it in the structure IIMGENC1_Status. It is then the responsibility of the application to use this corrected numAU as the effective slice's size.

The process API is then called as many times as there are slices in the image. After the first slice is encoded, header insertion must be disabled by a call to the control API. Also the parameter sliceNum in structure IJPEGENC_InArgs of the process API must be incremented each time process is called, otherwise, restart markers (0xFFD0, 0xFFD1, .., 0xFFD7) are not ordered correctly inside the bitstream. Note that the process API returns the current position of the input and output pointers in the member curInPtr and curOutPtr of the IJPEGENC_OutArgs structure. These values can be used to correctly initialize the input and output buffer pointers the next time the process API is called. The output buffer pointer will be equal to the currOutPtr value returned by the previous call to process API, which ensures bitstream continuity.

Before calling the process API for the last slice, the control API must be called to set numAU to the number of remaining MCUs left to finish encoding the image. Also, the sliceNum parameter of the structure IJPEGENC_InArgs passed to the process API must be set to -1, to inform the JPEG encoder that it is the last slice to be encoded, otherwise, the EOI marker is not appended.

Slice-mode encoding seamlessly operates with the output bitstream's ring-buffer configuration and both are automatically enabled.

4.2.4 Example of Application Code That Operates Slice-Mode Encoding

```
// Call get status to get number of total AU
iimgEncfxns->control((IIMGENC1_Handle)handle,
    IJPEGENC_GETSTATUS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams, (IIMGENC1_Status*)&status);
totalAU= status.imgencStatus.totalAU;
// Set number of MCUs per slice.
extn_dynamicParams.imgencDynamicParams.numAU= totalAU/20;
// Call control function to setup dynamic params
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams, (IIMGENC1_Status*)&status);
numAU= status.numAU; // Get real numAU computed by codec
// Call to JPEG encode processing, encode 1st slice with header
ring2media.mediaPtr= media;
ring2media.ringCurPtr= ringbuf;
inArgs.ringBufStart= (XDAS_UInt8*)ringbuf;
inArgs.ringBufSize= RINGBUFSIZE;
inArgs.sliceNum= 0;
retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                    (XDM1_BufDesc  *)&inputBufDesc,
                    (XDM1_BufDesc  *)&outputBufDesc,
                    (IIMGENC1_InArgs *)&inArgs,
                   (IIMGENC1_OutArgs *)&outArgs);
// Disable header insertion
extn_dynamicParams.imgencDynamicParams.generateHeader= XDM_ENCODE_AU;
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams, (IIMGENC1_Status*)&status);
bytesGenerated= outArgs.imgencOutArgs.bytesGenerated;
// Repeat JPEG encoding as many times as necessary until last slice
for (i=numAU;i<totalAU-numAU;i+= numAU){
    inArgs.sliceNum++;
    inputBufDesc.descs[0].buf     = outArgs.curInPtr;
outputBufDesc.descs[0].bufs     = outArgs.curOutPtr;
    // The line below is actually ignored by codec
    outputBufDesc.bufSizes[0] -=  outArgs.imgencOutArgs.bytesGenerated;
    retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                  (XDM1_BufDesc  *)&inputBufDesc,
                  (XDM1_BufDesc  *)&outputBufDesc,
                  (IIMGENC1_InArgs *)&inArgs,
                  (IIMGENC1_OutArgs *)&outArgs);
    bytesGenerated+= outArgs.imgencOutArgs.bytesGenerated;
}
// For last slice, re-adjust numAU
extn_dynamicParams.imgencDynamicParams.numAU= totalAU-i;
// Call control function to setup dynamic params
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams *)&extn_dynamicParams,   (IIMGENC1_Status *)&status);
// Call JPEG for the last slice
inArgs.sliceNum= -1;   // -1 means last slice for JPEG encoder.
inputBufDesc.descs[0]buf     = outArgs.curInPtr;
outputBufDesc.descs[0].buf     = outArgs.curOutPtr;
```

```
retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                (XDM1_BufDesc   *)&inputBufDesc,
              (XDM1_BufDesc   *)&outputBufDesc,
                (IIMGENC1_InArgs *)&inArgs,
                (IIMGENC1_OutArgs *)&outArgs);
bytesGenerated+= outArgs.imgencOutArgs.bytesGenerated;
```

4.3 Color Formats Supported

Three input formats are supported: YUV422 interleaved, YUV420 planar, and YUV422 planar. Input format is set by initializing the parameter inputChromaFormat of the structure IIMGENC1_DynamicParams before calling the control API() with command XDM_SETPARAMS. The symbols XDM_YUV_422ILE, XDM_YUV_420P or XDM_YUV_422P must be used. Input color format can be changed before any process() API call. When planar format is chosen, the pointers to U and V components must be passed to the encoder through the XDM1_BufDesc structure when calling the process() API.

Two output formats are supported: yuv420 or yuv422. Output format is set at creation time when calling the algInit() API by setting the member forceChromatFormat of the structure IIMGENC1_Params. Use either the XDM_YUV_420P symbol or the XDM_YUV_422P symbol to initialize this member.

4.4 Rotation

Rotation is internally supported by the JPEG encoder, so the application does not need to spend any extra resources to perform this task. The following clockwise rotations are supported: 90, 180, and 270. The rotation feature is enabled by setting the rotation member of the IJPEGENC_DynamicParams structure passed when calling the control() API with command XDM_SETPARAMS.

Here is a brief description of how rotation is performed within the codec when 90 rotation is enabled for YUV422 interleaved input:

Full frame case when slice based encoding is disabled— Blocks of 8(H)x16(V) pixels are fetched from the bottom right corner of the input frame, internally rotated to blocks of 16x8 and encoded by the JPEG encoder. The encoding progression in reference to the original image is from bottom to top and left to right. In reference to the encoded image, it is left to right, but top to bottom.

Case when slice mode encoding is enabled— The application must feed a vertical band of MCUs to the JPEG encoder. Blocks of 8(H)x16(V) pixels are fetched from the bottom right corner of the band, internally rotated to blocks of 16x8, and encoded by the JPEG encoder. The encoding progression in reference to the vertical band is from bottom to top and left to right. In reference to the encoded image, it is left to right, but top to bottom.

Please refer to Figure 4-5 and Figure 4-6 for illustrations of this process.

Figure 4-5. Rotation processing flow, full frame case

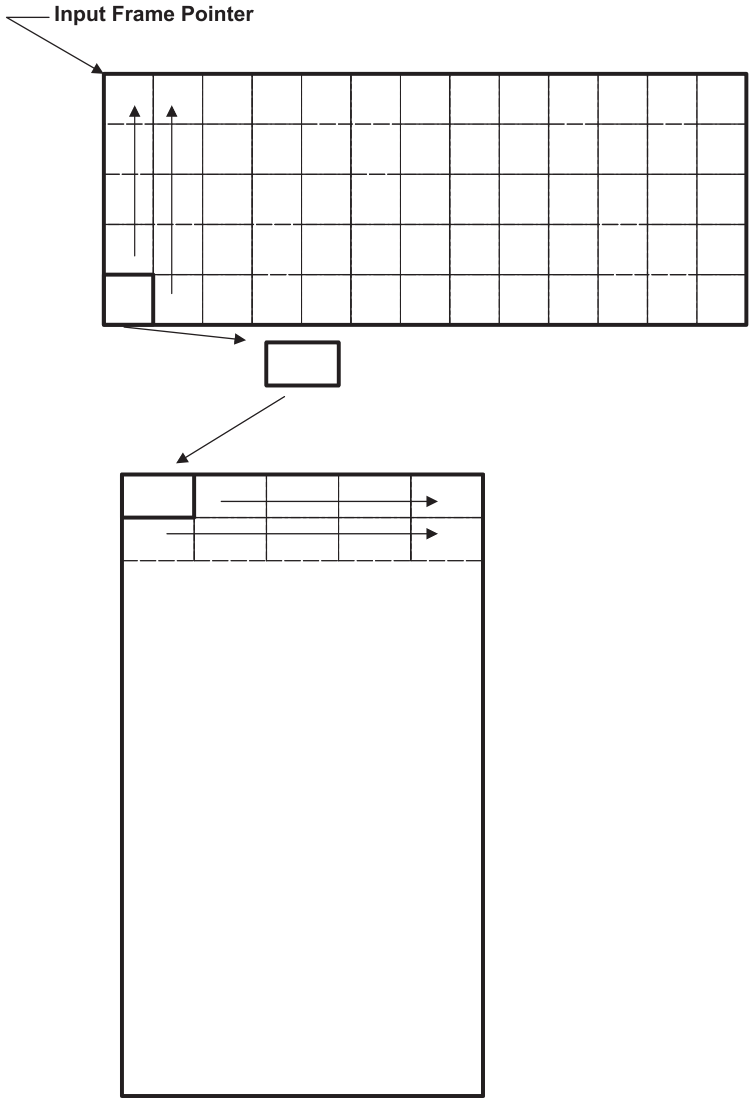
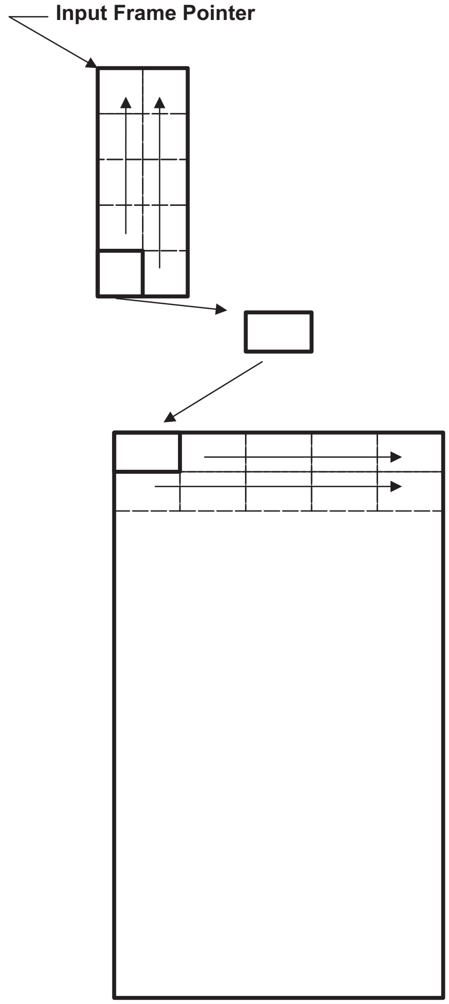


Figure 4-6. Rotation processing flow, slice mode case



Slice-mode processing is controlled by the run-time parameter numAU of the structure IIMGENC1_DynamicParams. Run time parameters are set when calling the control API. If numAU is set to XDM_DEFAULT, then entire frame will be encoded when the process API is called. Otherwise, it must be set to the number of MCUs contained in a slice.

For 90 or 180 rotation, the parameter numAU should be set such that it is multiple of (H/16) x 2, where H is the height of the original image (before rotation).

If that constraint is not met, the encoder automatically rounds up numAU to the next valid value and returns it in the structure IIMGENC1_Status. It is then the responsibility of the application to use this corrected numAU as the effective slice's size.

API Reference

This chapter provides a detailed description of the data structures and interfaces functions used in the codec component.

Topic		Page

5.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. Described alongside the macro or enumeration is the semantics or interpretation of the same in terms of what value it stands for and what it means.

Table 5-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
XDM_DataFormat	XDM_BYTE	1	Big endian stream
	XDM_LE_16	2	16-bit little endian stream
	XDM_LE_32	3	32-bit little endian stream
XDM_ChromaFormat	XDM_CHROMA_NA	-1	Chroma format not applicable
	XDM_YUV_420P	1	YUV 4:2:0 planar
	XDM_YUV_422P	2	YUV 4:2:2 planar
	XDM_YUV_422IBE	3	YUV 4:2:2 interleaved (big endian)
	XDM_YUV_422ILE	4	YUV 4:2:2 interleaved (little endian). Default choice for input color format.
	XDM_YUV_444P	5	YUV 4:4:4 planar
	XDM_YUV_411P	6	YUV 4:1:1 planar
	XDM_GRAY	7	Gray format
	XDM_RGB	8	RGB color format
	XDM_CHROMAFORMAT_DEFAULT	4	Default chroma format value set to XDM_YUV_422ILE
XDM_CmdId	XDM_GETSTATUS	0	Query algorithm instance to fill Status structure
	XDM_SETPARAMS	1	Set run time dynamic parameters via the DynamicParams structure
	XDM_RESET	2	Reset the algorithm
	XDM_SETDEFAULT	3	Initialize all fields in Params structure to default values specified in the library
	XDM_FLUSH	4	Handle end of stream conditions. This command forces algorithm instance to output data without additional input. This command is not implemented.
	XDM_GETBUFINFO	5	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	6	Query the algorithm's version. The result will be returned in the data field of the respective _Status structure. This control command is presently not supported.
XDM_EncMode	XDM_ENCODE_AU	0	Encode entire access unit. Default value.
	XDM_GENERATE_HEADER	1	Encode only header.
	JPEGENC_TI_ENCODE_AU_NOHEADER	2	Encode raw data only (no header)

Table 5-1. List of Enumerated Data Types (continued)

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
XDM_ErrorBit	XDM_APPLIEDCONCEALMENT	9	Bit 9 1 - Applied concealment 0 - Ignore
	XDM_INSUFFICIENTDATA	10	Bit 10 1 - Insufficient data 0 - Ignore
	XDM_CORRUPTEDDATA	11	Bit 11 1 - Data problem/corruption 0 - Ignore
	XDM_CORRUPTEDHEADER	12	Bit 12 1- Header problem/corruption 0 - Ignore
	XDM_UNSUPPORTEDINPUT	13	Bit 13 1 - Unsupported feature/parameter in input 0 - Ignore
	XDM_UNSUPPORTEDPARAM	14	Bit 14 1 - Unsupported input parameter or configuration 0 – Ignore
	XDM_FATALERROR	15	Bit 15 1 - Fatal error (stop encoding) 0 - Recoverable error
XDM_EncodingPreset	XDM_DEFAULT	0	Default setting of the algorithm specific creation time parameters
	XDM_HIGH_QUALITY	1	Set algorithm specific creation time parameters for high quality (default setting). Not supported in this version of the JPEG Encoder.
	XDM_HIGH_SPEED	2	Set algorithm specific creation time parameters for high speed. Not supported in this version of the JPEG Encoder.
	XDM_USER_DEFINED	3	User defined configuration using advanced parameters. Not supported in this version of the JPEG Encoder.

Note: The remaining bits that are not mentioned in XDM_ErrorBit are used by codec for reporting extended errors. Please refer to the DM355_JPEGENC_ERROR structure in Section 5.2.2.1 for more details.

The algorithm can set multiple bits to 1, depending on the error condition.

5.2 Data Structures

This section describes the XDM defined data structures that are common across codec classes. These XDM data structures can be extended to define any implementation specific parameters for a codec component..

Table 5-2. Data Structures

Title Page

Table 5-2. Data Structures (continued)

5.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- XDM1_BufDesc
- XDM1_SingleBufDesc
- XDM1_AlgBufInfo
- IIMGENC1_Fxns
- IIMGENC1_Params
- IIMGENC1_DynamicParams
- IIMGENC1_InArgs
- IIMGENC1_Status
- IIMGENC1_OutArgs

XDM1_BufDesc

Description

This structure defines the buffer descriptor for input and output buffers in XDM1.0

Fields

Field	Datatype	Input/ Output	Description
numBufs	XDAS_Int32	Input	Number of buffers
descs[XDM_MAX_IO_BUFFERS]	XDM1_SingleBufDesc	Input	Array of buffer descriptors

XDM1_SingleBufDesc

Description

This structure defines the single buffer descriptor for input and output buffers in XDM1.0

Fields

Field	Datatype	Input/ Output	Description
*buf	XDAS_Int8	Input	Pointer to a buffer address
bufSize	XDAS_Int32	Input	Size of buf in 8-bit bytes
accessMask	XDAS_Int32	Input	Mask filled by the algorithm, declaring how the buffer was accessed by the algorithm process

XDM_AlgBufInfo

Description

This structure defines the buffer information descriptor for input and output buffers. This structure is filled when you invoke the control() function with the XDM_GETBUFINFO command.

Fields

Field	Datatype	Input/ Output	Description
minNumInBufs	XDAS_Int32	Output	Number of input buffers
minNumOutBufs	XDAS_Int32	Output	Number of output buffers
minInBufSize[XDM_MAX_IO_BUFFERS]	XDAS_Int32	Output	Size in bytes required for each input buffer
minOutBufSize[XDM_MAX_IO_BUFFERS]	XDAS_Int32	Output	Size in bytes required for each output buffer

Note: For more information regarding I/O buffers, see the *Sequential JPEG Encoder Codec on DM355 Datasheet* (SPRS490).

IIMGENC1_Fxns

Description

This structure contains pointers to all the XDAIS and XDM interface functions.

Fields

Field	Datatype	Input/ Output	Description
ialg	IALG_Fxns	Input	Structure containing pointers to all the XDAIS interface functions. For more details, see *TMS320 DSP Algorithm Standard API Reference* (SPRU360).
*process	XDAS_Int32	Input	Pointer to the process() function.
*control	XDAS_Int32	Input	Pointer to the control() function.

IIMGENC1_Params

Description

This structure defines the creation parameters for an algorithm instance object. Set this data structure to NULL, if you are unsure of the values to be specified for these parameters.

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
maxHeight	XDAS_Int32	Input	Maximum image height to be supported in pixels. Minimum supported height is 64 pixels.
maxWidth	XDAS_Int32	Input	Maximum image width to be supported in pixels. Minimum supported width is 64 pixels.
maxScans	XDAS_Int32	Input	Maximum number of scans. Not supported in this encoder.

Field	Datatype	Input/ Output	Description
dataEndianness	XDAS_Int32	Input	Endianness of input data.. Only XDM_BYTE (Default) is supported. See XDM_DataFormat enumeration for details.
forceChromaFormat	XDAS_Int32	Input	Force encoding in given Chroma format. Only XDM_DEFAULT, XDM_YUV_420P, and XDM_YUV_422P (Default) are supported.

IIMGENC1_DynamicParams

Description This structure defines the run time parameters for an algorithm instance object. Set this data structure to NULL, if you are unsure of the values to be specified for these parameters.

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
numAU	XDAS_Int32	Input	Number of Access unit to encode. Set to XDM_DEFAULT to encode the entire frame
inputHeight	XDAS_Int32	Input	Height of input frame in pixels. Minimum supported Height is 64 pixels.
inputWidth	XDAS_Int32	Input	Width of input frame in pixels. Minimum supported Width is 64 pixels.
inputChromaFormat	XDAS_Int32	Input	Input chroma format. Only XDM_DEFAULT, XDM_YUV_420P, XDM_YUV_422P, and XDM_YUV_422ILE (Default) are supported.
generateHeader	XDAS_Int32	Input	Encode entire access unit or only header. See XDM_EncMode enumeration for details.
captureWidth	XDAS_Int32	Input	If the field is set to: 0 - Encoded image width is used as pitch. Any non-zero value, capture width is used as pitch (capture width should be ≥ to image width).
qValue	XDAS_Int32	Input	Q value Quality factor for encoder (1: Lowest quality, 97 Highest quality)

IIMGENC1_InArgs

Description This structure defines the run time input arguments for an algorithm instance object.

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.

IIMGENC1_OutArgs

Description This structure defines the run time output arguments for an algorithm instance object.

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
extendedError	XDAS_Int32	Output	extendedErrorField to report the extended errors returned by codec
bytesGenerated	XDAS_Int32	Output	Number of bytes generated during the process() cal
currentAU	XDAS_Int32	Output	Current access unit number

IIMGENC1_Status

Description This structure defines the run time output arguments for an algorithm instance object.

Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
extendedError	XDAS_Int32	Output	@extendedErrorField
data	XDM1_SingleBufDesc	Input	Buffer descriptor for data passing.
			This buffer can be used as either input or output, depending on the command. The buffer will be provided by the application, and returned to the application upon return of the control() call. The algorithm must not retain a pointer to this data.
totalAU	XDAS_Int32	Output	Total number of Access Units.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details

5.2.2 *JPEG Encoder Data Structures*

This section includes the following JPEG Encoder specific extended data structures:

- IJPEGENC1_Params
- IJPEGENC1_DynamicParams
- IJPEGENC1_Status
- IJPEGENC1_InArgs
- IJPEGENC1_OutArgs
- DM355_JPEGENC_ERROR

IJPEGENC1_Params

Description

This structure defines the base creation parameters and any other implementation specific parameters (extended) for the JPEG Encoder instance object. The base creation parameters are defined in the XDM data structure, IIMGENC1_Params.

Fields

Field	Datatype	Input/ Output	Description
imgencParams	IIMGENC1_Params	Input	Base creation parameters. See IIMGENC1_Params data structure for details
halfBufCB	XDAS_Void (*) (Uint32 curBufPtr, XDAS_Void*arg)	Input	Half buffer callback function pointer. Must be set to NULL if not used.
halfBufCBarg	XDAS_Void *	Input	Half buffer callback argument. Must be set to NULL if not used.

IJPEGENC1_DynamicParams

Description

This structure defines the base runtime creation parameters and any other implementation specific runtime parameters (extended) for the JPEG Encoder instance object. The base runtime parameters are defined in the XDM data structure, IIMGENC1_DynamicParams.

Fields

Field	Datatype	Input/ Output	Description
imgencDynamicParams	IIMGENC1_DynamicParams	Input	Base runtime parameters. See IIMGENC1_Params data structure for details
rstInterval	XDAS_Uint16	Input	Restart interval in number of MCUs, must be > 3.
disableEOI	XDAS_Uint16	Input	XDM_DEFAULT: EOI insertion enabled. 1: EOI insertion disabled
rotation	XDAS_Uint16	Input	Specify degree of clock-wise rotation. Can be 0 (XDM_DEFAULT),90, 180, 270
customQ	IJPEGENCQtab	Input	Structure for user -defined quantization table. { Uint8 luma[64]; Uint8 chroma[64]; }IJPEGENCQtab; Should be set to NULL if not used.

IJPEGENC1_Status

Description

This structure defines the base status parameters and any other implementation specific status parameters for the JPEG Encoder instance object. The base status parameters are defined in the XDM data structure, IIMGENC1_Status.

Fields

Field	Datatype	Input/ Output	Description
imgencStatus	IIMGENC1_Status	Output	Base status parameters. See IIMGENC1_Status data structure for details
bytesGenerated	XDAS_Int32	Output	Number of bytes generated by last call to JPEG process function.
numAU	XDAS_Int32	Output	Number of MCUs within a slice, recomputed by the JPEG encoder so it respects the constraint numAU % (2*IMGWIDTH/MCU_WIDTH)= 0

IJPEGENC1_InArgs

Description This structure defines the base runtime input parameters and any other implementation specific runtime input parameters for the JPEG Encoder instance object. The base runtime parameters are defined in the XDM data structure, IIMGENC1_InArgs.

Fields

Field	Datatype	Input/ Output	Description
imgencInArgs	IIMGENC1_InArgs	Input	Base input runtime parameters. See IIMGENC1_InArgs data structure for details
ringBufStart	XDAS_UInt8 *	Input	Pointer to starting point of bitstream ring buffer
ringBufSize	XDAS_Uint32	Input	Size of ring buffer in bytes

Field	Datatype	Input/ Output	Description
sliceNum	XDAS_Int16	Input	Slice number. -1 if last slice. Only effective when slice based encoding enabled.

IJPEGENC1_OutArgs

Description This structure defines the base runtime output parameters and any other implementation specific runtime output parameters for the JPEG Encoder instance object. The base runtime parameters are defined in the XDM data structure, IIMGENC1_OutArgs.

Fields

Field	Datatype	Input/ Output	Description
imgencOutArgs	IIMGENC1_Out Args	Output	Base input runtime parameters. See IIMGENC1_InArgs data structure for details
curInPtrY	XDAS_Uint8*	Output	Current input pointer, pointing to YUV interleaved data for YUV422 interleaved input or Y data for planar input
curInPtrU	XDAS_Uint8*	Output	Current input pointer, pointing to U data for planar input
curInPtrV	XDAS_Uint8*	Output	Current input pointer, pointing to V data for planar input
curOutPtr	XDAS_Uint8*	Output	Current output pointer, pointing to bitstream

5.2.2.1 DM355_JPEGENC_ERROR

JPEG encoder supports the enum structure in Table 5-3 to report errors in creation time parameters and run time parameters.

Table 5-3. Enumeration Structure

ERROR	Bit Position	Description
DM355_JPEGENC_INVALID_MAXWIDTH	1	1 -Error in max width parameter 0- No error
DM355_JPEGENC_INVALID_MAXHEIGHT	2	1- Error in max height parameter 0 - No error
DM355_JPEGENC_INVALID_MAXSCANS	3	1- Error in max scans parameter 0 - No error
DM355_JPEGENC_INVALID_DATAENADIANNESS	4	1- Error in data endianness parameter 0 - No error
DM355_JPEGENC_INVALID_FORCECHROMAFORMA	5	1- Error in chroma format parameter 0 - No error
DM355_JPEGENC_INVALID_NUMAU	6	1- Error in numAU parameter 0 - No error
DM355_JPEGENC_INVALID_INPUTCHROMAFORMA	7	1- Error in input chroma parameter 0 - No error
DM355_JPEGENC_INVALID_INPUTHEIGHT	8	1- Error in input height parameter 0 - No error
DM355_JPEGENC_INVALID_INPUTWIDTH	16	1- Error in input width parameter 0 - No error
DM355_JPEGENC_INVALID_CAPTUREWIDTH	17	1- Error in capture width parameter 0 - No error
DM355_JPEGENC_INVALID_GENERATEHEADER	18	1- Error in generate header parameter 0 - No error
DM355_JPEGENC_INVALID_QVALUE	19	1- Error in q value parameter 0 - No error
DM355_JPEGENC_INVALID_RSTINTERVAL	20	1- Error in reset interval parameter 0 - No error

Table 5-3. Enumeration Structure (continued)

ERROR	Bit Position	Description
DM355_JPEGENC_INVALID_ROTATION	21	1- Error in rotation parameter 0 - No error
DM355_JPEGENC_INVALID_DISABLEEOI	22	1- Error in disable EOI parameter 0 - No error

5.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the JPEG encoder. The APIs are logically grouped into the following categories:

- Creation – algNumAlloc(), algAlloc(), dmaGetChannelCnt(), dmaGetChannels()
- Initialization – algInit(), dmaInit()
- Control Processing – control(), algActivate(), process(), algDeactivate()
- Termination – algFree()

You must call these APIs in the following sequence:

1. algNumAlloc()
2. algAlloc()
3. algInit()
4. control()
5. algActivate()
6. process()
7. algDeactivate()
8. algFree()

algNumAlloc(), algAlloc(), algInit(), algActivate(), algDeactivate(), and algFree() are standard XDAIS APIs. This document includes only a brief description for the standard XDAIS APIs. For more details, see *TMS320 DSP Algorithm Standard API Reference Guide* (SPRU360).

Table 5-4. API List

5.3.1 Creation APIs

Creation APIs create an instance of the component. The term creation could mean allocating system resources, typically memory.

Note: Please see the JPEG Encoder Data Sheet for External Data Memory requirements

algNumAlloc() *Determine the number of buffers that an algorithm requires*

Synopsis XDAS_Int32 algNumAlloc(Void);

Arguments Void

Return Value XDAS_Int32; /* number of buffers required */

Description algNumAlloc() returns the number of buffers that the algAlloc() method requires. This operation allows you to allocate sufficient space to call the algAlloc() method.

algNumAlloc() may be called at any time and can be called repeatedly without any side effects. It always returns the same result. The algNumAlloc() API is optional.

For more details, see *TMS320 DSP Algorithm Standard API Reference Guide* (SPRU360).

See Also `algAlloc()`

algAlloc() — *Determine the attributes of all buffers that an algorithm requires*

Synopsis

```
XDAS_Int32 algAlloc(const IALG_Params *params, IALG_Fxns **parentFxns,
IALG_MemRec memTab[]);
```

Arguments

```
IALG_Params *params; /* algorithm specific attributes */
IALG_Fxns **parentFxns;/* output parent algorithm functions */
IALG_MemRec memTab[]; /* output array of memory records */
```

Return Value `XDAS_Int32 /* number of buffers required */`

Description algAlloc() returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm. If successful, this function returns a positive non-zero value indicating the number of records initialized.

The first argument to algAlloc() is a pointer to a structure that defines the creation parameters. This pointer may be NULL; however, in this case, algAlloc() must assume default creation parameters and must not fail.

The second argument to algAlloc() is an output parameter. algAlloc() may return a pointer to its parent's IALG functions. If an algorithm does not require a parent object to be created, this pointer must be set to NULL.

The third argument is a pointer to a memory space of size nbufs * sizeof(IALG_MemRec) where, nbufs is the number of buffers returned by algNumAlloc() and IALG_MemRec is the buffer-descriptor structure defined in ialg.h.

After calling this function, memTab[] is filled up with the memory requirements of an algorithm.

For more details, see T*MS320 DSP Algorithm Standard API Reference Guide* (SPRU360).

See Also `algNumAlloc(), algFree()`

5.3.2 Initialization API

The Initialization API initializes an instance of the algorithm. The initialization parameters are defined in the Params structure (see the Data Structures section for details).

algInit() — *Initialize an algorithm instance*

Synopsis

```
XDAS_Int32 algInit(IALG_Handle handle, IALG_MemRec memTab[], IALG_Handle parent,
IALG_Params *params);
```

Arguments

```
IALG_Handle handle; /* algorithm instance handle*/
IALG_memRec memTab[]; /* array of allocated buffers */
IALG_Handle parent; /* handle to the parent instance */
IALG_Params *params; /* algorithm initialization parameters */
```

```
Return Value
```

Return Value

```
IALG_EOK; /* status indicating success */
IALG_EFAIL; /* status indicating failure */
```

Description

algInit() performs all initialization necessary to complete the run time creation of an algorithm instance object. After a successful return from algInit(), the instance object is ready to be used to process data.

The first argument to algInit() is a handle to an algorithm instance. This value is initialized to the base field of memTab[0].

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers allocated for an algorithm instance. The number of initialized records is identical to the number returned by a prior call to algAlloc().

The third argument is a handle to the parent instance object. If there is no parent object, this parameter must be set to NULL.

The last argument is a pointer to a structure that defines the algorithm initialization parameters.

For more details, see T*MS320 DSP Algorithm Standard API Reference Guide* (SPRU360).

The following sample code is an example of initializing Params structure and creating an instance with base parameters.

```
{
    .........................................
    IIMGENC1_Params    params;

    // Set the create time base parameters
    params.size = sizeof(IIMGENC1_Params);
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM_DEFAULT;
    params.dataEndianness = XDM_BYTE;
    params.forceChromaFormat= XDM_YUV_420P;

   handle = (IALG_Handle) ALG_create((IALG_Fxns *)& JPEGENC_TI_IJPEGENC,
                                            (IALG_Handle) NULL,
                                            (IALG_Params *) &params)
...............................................
}
```

The following sample code is an example of initializing Params structure and creating an instance with extended parameters

```
{
    .........................................

    IIMGENC1_Params            params;
    IJPEGENC_Params            extParams;

    // Set the create time base parameters
    params.size = sizeof(IIMGENC1_Params);
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM_DEFAULT;
    params.dataEndianness = XDM_BYTE;
    params.forceChromaFormat= XDM_YUV_420P;

    // Set the create time extended parameters

    extParams.imgencParams = params;
    extParams.halfBufCB = NULL;
    extParams.halfBufCBarg = NULL;

handle = (IALG_Handle) ALG_create((IALG_Fxns *)& JPEGENC_TI_IJPEGENC,
                                            (IALG_Handle) NULL,
                                            (IALG_Params *) &extParams)
```

```
...............................................
}
```

See Also `algAlloc(), algMoved()`

5.3.3 Control Processing API

The Control API is used before a call to process() to enquire about the number and size of I/O buffers, or to set the dynamic params, or get status of encoding.

control() — *Control call*

Synopsis

```
XDAS_Int32 (*control)( IIMGENC1_Handle handle, IIMGENC1_Cmd id,
IIMGENC1_DynamicParams *params, IIMGENC1_Status *status);
```

Arguments

```
IIMGENC1_Handle handle; /* algorithm instance handle */
IIMGENC1_Cmd id; /* id of command */
IIMGENC1_DynamicParams *params; /* pointer to dynamic parameters */
IIMGENC1_Status *status /* pointer to status structure */
```

Return Value

```
IALG_EOK; /* status indicating success */
IALG_EFAIL; /* status indicating failure */
```

Description

This function does the basic encoding. The first argument to control() is a handle to an algorithm instance

The second argument is the command id, which can be of these following values:

XDM_GETSTATUS: fill structure IIMGENC1_Status whose pointer is passed as 4th argument.

XDM_SETPARAMS: set dynamic params contained in the structure whose pointer is passed as 3rd argument.

XDM_RESET: reset the encoder so next time process() is called, a new bitstream is encoded.

XDM_SETDEFAULT: set the dynamic params to the following default values:

XDM_GETBUFINFO: get required number of I/O buffers and their sizes. Results are returned in the bufInfo member of the structure IIMGENC1_Status whose pointer is passed as 4th argument.

The third argument is a pointer to a dynamic params structure of type IIMGENC1_DynamicParams or IJPEGENC_DynamicParams (typecast to the previous one). This argument is used whenever command ID is XDM_SETPARAMS.

The fourth argument is a pointer to a structure of type IIMGENC1_Status or IJPEGENC_Status (typecast to the previous one). This argument is used whenever command ID is XDM_GETSTATUS or XDM_GETBUFINFO.

Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

control() can only be called after a successful return from algInit() and algActivate().

handle must be a valid handle for the algorithm’s instance object.

All the parameters of dynamic parameters structure must be set before making call to XDM_SETPARAMS control function

Post conditions

The following conditions are true immediately after returning from this function.

If the control operation is successful, the return value from this operation is equal to IALG_EOK; otherwise it is equal to either IALG_EFAIL or an algorithm specific return value.

The following code gives an example for initializing the extended dynamic parameters for a 720x480 input.

```
{
...............................................

    IIMGENC1_DynamicParams    dynParams;
    IIMGENC1_Status        status;
    IJPEGENC_DynamicParams    extDynParams;
...............................................
    // Set the dynamic base parameters
    dynParams.size = sizeof(IJPEGENC_DynamicParams);
    dynParams.numAU= XDM_DEFAULT;
    dynParams.inputChromaFormat = XDM_YUV_422ILE;
    dynParams.inputHeight = 480;
    dynParams.inputWidth = 720;
    dynParams.captureWidth = 720;
    dynParams.generateHeader = XDM_DEFAULT;
    dynParams.qValue = XDM_DEFAULT;

    // Set the extended dynamic parameters
    extDynParams.imgencDynamicParams = dynParams;

    extDynParams.rstInterval = 84;
    extDynParams.disableEOI = XDM_DEFAULT;
    extDynParams.rotation = XDM_DEFAULT;
    extDynParams.customQ = NULL;
    extDynParams.preProc = NULL;
    extDynParams.overlay = NULL;

/* Set Dynamic Params */
retVal=iimgEncfxns->control(  (IIMGENC1_Handle)handle, XDM_SETPARAMS,
                             (IIMGENC1_DynamicParams *)& extDynParams,
                             (IIMGENC1_Status *)&status);
...............................................
  }
```

See Also `algInit(), algDeactivate(), process()`

5.3.4 Data Processing API

The Data processing API processes the input data.

process() *Basic encoding call*

Synopsis

```
XDAS_Int32 (*process)(IIMGENC1_Handle handle, XDM_BufDesc *inBufs, XDM_BufDesc
*outBufs, IIMGENC1_InArgs *inargs, IIMGENC1_OutArgs *outargs);
```

Arguments

```
IIMGENC1_Handle handle; /* algorithm instance handle */
XDM_BufDesc *inBufs; /* algorithm input buffer descriptor */
XDM_BufDesc *outBufs; /* algorithm output buffer descriptor */
IIMGENC1_InArgs *inargs /* algorithm runtime input arguments */
IIMGENC1_OutArgs *outargs /* algorithm runtime output arguments */
```

Return Value

```
IALG_EOK; /* status indicating success */
IALG_EFAIL; /* status indicating failure */
```

Description

This function does the basic encoding/decoding. The first argument to process() is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see XDM_BufDesc data structure for details).

The fourth argument is a pointer to the IIMGENC1_InArgs data structure that defines the run time input arguments for an algorithm instance object.

The last argument is a pointer to the IIMGENC1_OutArgs data structure that defines the run time output arguments for an algorithm instance object.

Note: If you are using extended data structures, the fourth and fifth arguments must be pointers to the extended InArgs and OutArgsdata structures respectively. Also, ensure that the size field is set to the size of the extended data structure. Depending on the value set for the size field, the algorithm uses either basic or extended parameters.

Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

process() can only be called after a successful return from algInit() and algActivate().

If algorithm uses DMA resources, process() can only be called after a successful return from DMAN3_init().

handle must be a valid handle for the algorithm's instance object.

Buffer descriptor for input and output buffers must be valid.

Input buffers must have valid input data.

Post conditions

The following conditions are true immediately after returning from this function.

If the process operation is successful, the return value from this operation is equal to IALG_EOK; otherwise it is equal to either IALG_EFAIL or an algorithm specific return value.

After successful return from process() function, algDeactivate() can be called.

Example

See test application file, jpgeTest.c available in the /Client/Test/Src sub-directory.

See Also

`algInit(), algDeactivate(), control()`

5.3.5 Termination API

The Termination API terminates the algorithm instance and frees up the memory space that it uses

algFree()

Determine the addresses of all memory buffers used by the algorithm

Synopsis

```
XDAS_Int32 algFree(IALG_Handle handle, IALG_MemRec memTab[]);
```

Arguments

```
IALG_Handle handle; /* handle to the algorithm instance */
IALG_MemRec memTab[]; /* output array of memory records */
```

Return Value

```
XDAS_Int32; /* Number of buffers used by the algorithm */
```

Description

algFree() determines the addresses of all memory buffers used by the algorithm. The primary aim of doing so is to free up these memory regions after closing an instance of the algorithm.

The first argument to algFree() is a handle to the algorithm instance.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm instance.

For more details, see T*MS320 DSP Algorithm Standard API Reference* (SPRU360).

See Also

`algAlloc()`