Sequential JPEG Decoder Codec on DM355

User's Guide

Literature Number: SPRUFE6B
February 2008

TEXAS INSTRUMENTS

I

Revision History

31 July 2006	Created	v. 0.1
04 Sep 2006	Updated	v 0.2
03 Oct 2006	Updated with scaling	v 0.3
15 Feb 2007	Minimum image width supported is 64 pixels for yuv422/420	v 0.4
17 July 2007	Updated with XDMv1.0 specific API changes and LINUX specific changes	v 0.5
11 Sep 2007	Added documentation on area decode and rotation	v 0.6
03 October	Updated with review comments from TI	v 1.0
18 Dec 2007	Updated API support and codec's extended error details	v1.1
08 Jan 2008	Updated parameter structure in API section	v1.2
06 Feb 2008	Added the sample code for algCreate and control call	v1.3

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) JPEG Decoder 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.

TI's codec implementations are based on the eXpressDSP Digital Media (XDM) v1.0 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's codecs with other software to build a multimedia system based on the DM355 platform.

This document assumes that you are fluent in the C language, having working knowledge of Digital Signal Processing (DSP), digital signal processors, and DSP applications. Good knowledge of eXpressDSP Algorithm Interface Standard (XDAIS) and eXpressDSP Digital Media (XDM) standard 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 decoder.
- **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's 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.

- *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 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 1-1. 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 Decoder 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

Contents

Figures

Tables

Chapter 1

Introduction

This chapter introduces XDAIS, XDM, and IDMA3. It also provides an overview of TI's implementation of the JPEG Decoder 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) 1.0 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 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 share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. In order to facilitate these functionalities, 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 *TMS320 DSP Algorithm Standard API Reference* (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 imaging decoder system, you can use any of the available image decoders (such as Sequential JPEG, Progressive JPEG Decoder) 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.

The following figure depicts the XDM interface to the client application.

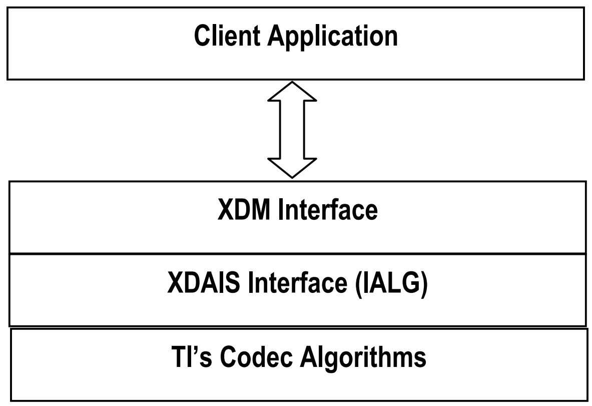


Figure 1-1. XDM interface to the client application

As depicted in the figure, 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 decoder, then you can easily replace JPEG with another XDM-compliant image decoder with minimal changes to the client application.

For more details, see *xDAIS-DM (Digital Media) User Guide* (SPRUEC8b [XDM v1.0 is employed]).

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 1-1 shows a typical IDMA3 interface implemented by the algorithm module, which is used by the client applications to query the algorithm's DMA needs. The algorithm specifies the number of separate EDMA/QDMA channels and PaRamSets it requires, through memRecs. The IDMA3 standard defines following APIs:

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

dmaChangeChannels() 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 requires 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+* (SPRAAG1).

1.2 Overview of JPEG Decoder

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 the JPEG encoding/decoding process.

1.3 Supported Services and Features

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

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

- eXpressDSP™ Algorithm Interface Standard (XDAIS) compliant
- eXpressDSP Digital Media (xDM) v1.0 interface and IDMA3 compliant
- Support baseline sequential process with the following limitations:
  - Cannot support non-interleaved scans
  - Only supports 1 and 3 components
  - Huffman tables and quantization tables for U and V components must be the same
- Supports a maximum of four (two tables each) for AC and DC DCT coefficients
- Supports YUV 422 interleaved output format only [Planar output is not supported]
- Supports yuv420, yuv422, yuv444, gray level with 8x8 pixels MCU
- Supports 8-bit quantization tables
- Supports frame level decoding of images
- Images with resolutions up to 700 Mpixels can be decoded. This is the theoretical maximum; however, only images up to 64 Mpixels have been tested. If the codec memory and I/O buffer requirements exceed the DDR memory availability for frame based decoding, use ring buffer and slice mode decoding to decode higher resolution images.
- JPEG File Interchange Format (JFIF) header is skipped

- Supports frame level re-entrancy for multiple instance support
- Supports resizing by various factors from 1/8 to 7/8
- Supports frame pitch greater than picture width, specified as display width parameter
- Supports Rotation and Decode area individually, but does not support both together
- Supports limited IDMA3 interface with user-configurable additional PaRamSet requirements
- Supports ring buffer configuration of bitstream buffer for reducing buffer size requirement
- Supports Rotation of 90, 180 and 270 degree
- Validated on DM355 EVM (MV 4.0)

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 process
- Does not support Lossless process
- Does not support Hierarchical process
- Does not support progressive scan
- Supports YUV 422 interleaved output format only. Planar output is not supported.
- Does not support yuv411, gray level with 16x16 pixels MCU
- Does not support image width less than 64 pixels for yuv420/422 and 32 pixels for yuv444
- Does not support source images of 12-bits per sample
- Ring buffer size should be multiple of 4096 Bytes
- Only limited support of IDMA3 interface. See Sec 3.1 for details.

Chapter 2

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.

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 board.

2.1.2 Software

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

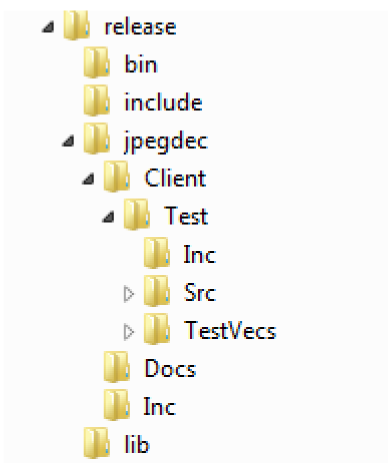
- **Linux:** MV Linux Pro 4.0 (kernel 2.6.10)
- **Code Generation Tools:** This project is compiled, assembled, and linked using the arm_v5t_le-gcc compiler.

2.2 Installing the Component

The codec component is released as tar-zipped file. To install the codec, follow the instructions in the Release notes. The code location is as follows:

JPEG Decoder algorithm code is in a directory ***jpegdec*** placed in DM355Codecs/release.

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



.

Figure 2-1. Component Directory Structure

Table 2-1 provides a description of the sub-directories created in the jpegdec directory.

Table 2-1. Component Directories

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

The DM355 JPEG Decoder library is put into the DM355Codecs/release/lib directory and the xdm headers are put in DM355Codecs/release/include directory.

2.3 Building the Sample Test Application on Linux

The sample test application that accompanies this codec component will take jpeg input files and dumps output YUV files as specified in the configuration file. To build and run the sample test application, follow these steps:

1) Verify that libjpegdec.a library is present in DM355Codecs/release/lib directory.

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

3) Change directory to DM355Codecs/release/jpegdec/Client/Test/Src and type “make clean” followed by a “make” command. This will use the makefile in that directory to build the test executable jpgdec into the DM355Codecs/release/bin directory.

4) To run the jpgdec executable on your DM355 EVM board, see the following instructions.

   - Set up the DM355 environment.
     For information about setting up the DM355 environment, see the DVEVM Hardware Setup and the DVEVM Software Setup chapters in the *DVEVM Getting Started Guide*.

   - Copy the binary jpgdec and the entire TestVecs directory into target directory.

   - Run following commands from prompt
     $./jpgdec

2.4 Configuration Files

This codec is shipped along with:

- A generic configuration file (Testvecs.cfg) – specifies input .jpg file, output yuv file and parameter file for each test case.
- A Decoder parameter file (Testparams.cfg) – specifies the configuration parameters used by the test application to configure the Decoder for a particular test case.

2.4.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, Testvecs.cfg, for determining the parameter file for each test case. The Testvecs.cfg file is available in the DM355Codecs/release/jpegdec/Client/Test/TestVecs/Config sub-directory.

The format of the Testvecs.cfg file is:

```
X
Config
Input
Output/Reference
```

where:

- X:

  0 - for random pattern comparison, no input file read, no output file is created. Compliance checking is done by comparing checksum.

  1 - for compliance checking with reference output file. Input YUV file read, no output file is created

  2 - for writing the output to the output file

  Please note that in the current test app file provided only X=2 is supported and other values of X is ignored.

- `Config` is the Decoder parameter file.
- `Input` is the input JPEG file name (use complete path).
- `Output/Reference` is the output YUV file name.

A sample Testvecs.cfg file is as shown:

```
2
./TestVecs/Config/Testparams1.cfg
./TestVecs/Input/420/RST_01.jpg
./TestVecs/Output/420/RST_01.yuv
2
./TestVecs/Config/Testparams1.cfg
./Test/TestVecs/Input/420/RST_02.jpg
./Test/TestVecs/Output/420/RST_02.yuv
```

2.4.2 Decoder Parameter File

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

A sample Testparams.cfg file is as shown:

```
# New Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
#
################################################################################
##########
# Parameters
################################################################################
##########
Resize     = 0       # 0: No resizing, 1: resize by 1/2, resize by 1/4, resize
by 1/8
DisplayWidth    = 0       # 0: display width = image output width
rotation  = 0             # 0: No Rotation, 90, 180, 270
maxWidth = 720
maxHeight = 480
forceChromaFormat = 4     # 0: XDM_DEFAULT, 4: 422_ILE
dataEndianness = 1
subRegionUpLeftX = 0
subRegionUpLeftY = 0
subRegionDownRightX = 0
subRegionDownRightY = 0
```

Any field in the `IIMGDEC1_Params` structure (see Section 5.2.1.5) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, you must appropriately modify the array sTokenMap in the test application to handle these fields.

Chapter 3

Sample Usage

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

3.1 JPEG Decoder Client interfacing constraints

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

1) DMA requirements of JPEG Decoder: Current implementation of the JPEG decoder uses the following TCCs for its DMA resource requirements along with its associated PaRamSets:

Channel Number	Associated PaRamSet Numbers
33 to 47, 52 to 55	33 to 47, 52 to 55 (PaRamSet number = channel number)

Apart from these 19 TCCs requirements, it also needs 8 more PaRamSets that are allocated through the IDMA3 interface.

2) The client application shall map all the DMA channels used by JPEG decoder to the same queue. This is required for the codec to function normally. Codec shall not map channels to queue.

3) 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. AlgActivate and AlgDeactivate calls, implemented by the codec and made by the client application perform context save/restore to allow multiple instances of the same codec and/or different codec combinations.

4) As 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.

5) 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.

6) The ARM and DDR clock to be set to required frequency for running single or multiple codecs.

7) The codec combinations feasibility is limited by processing time (computational hardware cycles) and DDR bandwidth.

8) Codec atomicity is supported at frame level processing only. The process call has to run until 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 IIMGDEC1_Params extended class of the JPEG Decoder library. The main test application files are jpgdTest355.c and testFramework.h. These files are available in the /Client/Test/Src and /Client/Test/Inc sub-directories respectively.

The following figure illustrates the sequence of APIs exercised in the sample test application.

Integration Layer	XDM-XDIAS-IDMA3 Interface	Codec Library
Param Setup		
Algorithm Instance creation and initialization	**algNumAlloc ()** **algAlloc ()** **algInit ()**	
DMA channels request and granting	**dmaChannelCnt ()** **dmaGetChannels ()** **dmaInit ()**	
Process call	**algActivate ()** **process ()** **algDeactivate ()**	
Algorithm instance deletion	**algNumAlloc ()** **algFree ()**	

Figure 3-1. Test Application Sample Implementation

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 Decoder configuration files.

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

1) Opens the generic configuration file, Testvecs.cfg and reads the compliance checking parameter Decoder configuration file name (Testparams.cfg), and, if applicable, the input file name, and output/reference file name.

2) Opens the Decoder configuration file, (Testparams.cfg) and reads the various configuration parameters required for the algorithm. For more details on the configuration files, see Section □

3) Sets the IIMGDEC1_Params structure based on the values it reads from the Testparams.cfg file.

4) Reads the input bit stream 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()` use 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 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 with algActivate and algDeactivate

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

1) Calls `algActivate()`, which initializes the decoder 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, pointer to the `IIMGDEC1_InArgs` and `IIMGDEC1_OutArgs` structures. `process()` function should be called multiple times to decode multiple images.
4) Call `algDeactivate()`, which performs releasing of hardware resources and saving of decoder 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 complete JPEG decode.
6) Other specific details of the process() function remains same as the described in section 3.1.3 and constraints describe in sec 3.1.1 are applicable.

NOTE: `algActivate ()` is a mandatory call before `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 the JPEG decoder, initialization and process calls are altered. One of the main issues in converting a single instance decoder to a multiple instance decoder 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:

1) DMA channels and PaRamSets
2) JPEG Hardware Co-Processors and their memory areas

To protect one instance of the JPEG decoder 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 the 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 decoder 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 above section would change to include both algActivate() and algDeactivate() as mandatory calls of 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 decoder 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) Calls `algDeactivate(),` which performs releasing of hardware resources and saving of decoder instance values.
5) Other specific details of the process() function remains same as the described in section 3.1.3 and constraints describe in sec 3.1.1 are applicable.

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

Chapter 4

Feature Descriptions

This chapter provides some description on special features not commonly found in a standard JPEG decoder such as:

- Ring-buffer configuration of input bit stream buffer.
- Slice-mode processing.
- Resizing
- Area decode
- Rotation

4.1 Bitstream ring buffer in DDR

To minimize the memory requirement, the JPEG decoder reads the JPEG bitstream from a circular or ring buffer residing in DDR, which acts as an intermediary storage area between the originating storage media (SD card, HD, memory stick, etc.) and the decoder. Therefore, 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 processed eventually because as JPEG decodes one half of the ring buffer, the application fills the other half from the media. The JPEG decoder and the application operate in parallel and on a different half, thus sustaining the maximum JPEG processing throughput.

The figure below depicts the state of the ring buffer at different states of JPEG processing:

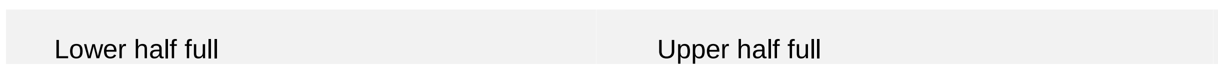


Figure 4-1. Ring buffer before JPEG decoder starts

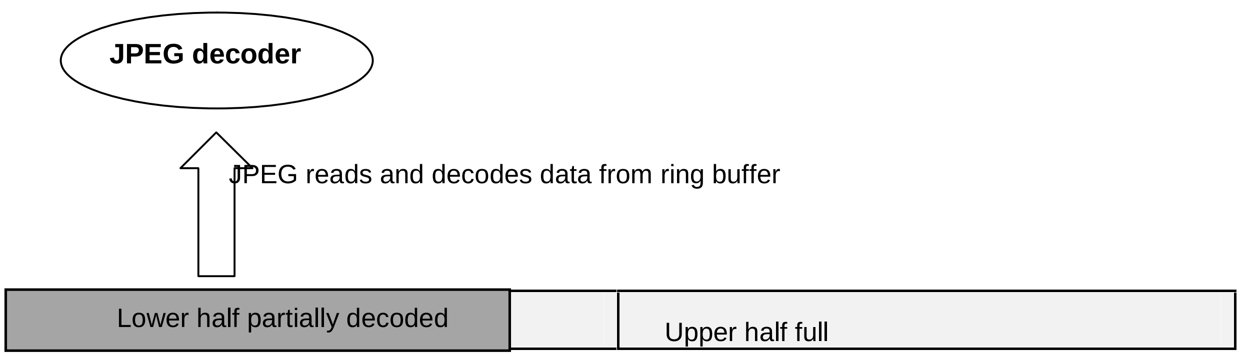


Figure 4-2. Ring buffer shortly after JPEG decoder starts

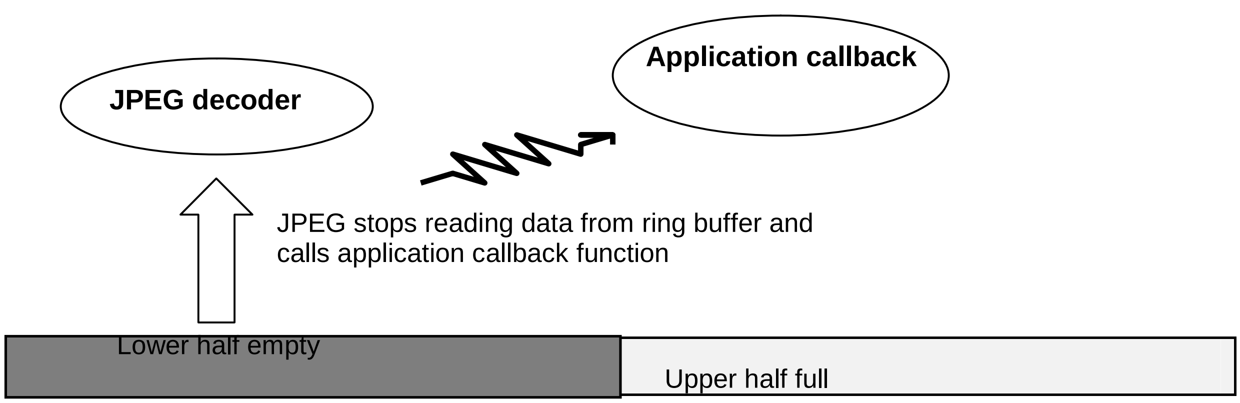


Figure 4-3. Ring buffer once JPEG decoder fills lower half

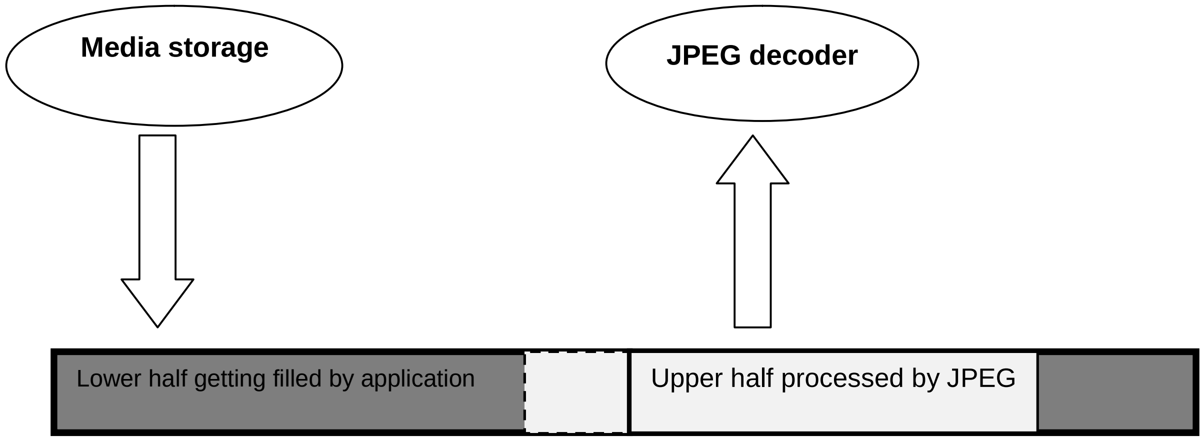


Figure 4-4. Ring buffer once application starts filling first half and JPEG decode starts processing second half.

4.1.1 Mode of operation

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

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

The input argument `curBufPtr` is passed by the decoder and its value is the pointer to the first free byte in the ring buffer. All the bytes located before `curBufPtr` are bytes already decoded by the decoder and can be overwritten by new bitstream data. The callback function must save `curBufPtr` so next time it is called, it knows where to overwrite the data from. However, the first time it is called is a particular 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 keeps track of the transfers between the ring buffer and the media storage.

4.1.2 Constraint

The ring buffer size must be multiple of 4096 bytes.

4.1.3 Guidelines for using ring buffer with JPEG decoder

This section introduces few guidelines and tips to help the programmer to implement ring buffer into an application using JPEG decoder. It doesn't provide all the steps required to initialize/run the JPEG decoder but only those related to ring buffer handling.

The following structure *Media2Ring* can be used to keep track of 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
} Media2Ring;
```

The members *mediaPtr* and *ringCurPtr* will be updated by the half-buffer callback function each time they are called.

Assuming that 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:

```
Media2Ring media2ring={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 *media2ring* to the callback function each time the decoder calls it.

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

```
extn_params.halfBufCB = (XDAS_Int32 (*)())JPEGDEC_TI_DM355_HalfBufCB;
extn_params.halfBufCBarg= (void*)&media2ring;
```

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

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

The members *ringCurPtr* and *mediaPtr* of *media2ring* must be reinitialized to their initial values before each call to process() since the callback function updates them.

```
ing2media.mediaPtr= media;
media2ring.ringCurPtr= ringbuf;
```

Also, the ring buffer must be filled by the application prior to the first call of the JPEG decoder's process function:

```
memcpy(media2ring.ringCurPtr, media2ring.mediaPtr, RINGBUF_SIZE);
media2ring.mediaPtr+= RINGBUF_SIZE;
```

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 *media2ring.ringCurPtr* and *curBufPtr*, the latter parameter being the first input argument of the callback function.

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 JPEGDEC_TI_DM355_HalfBufCB(XDAS_Int32 bufPtr, void *arg)
{
      Uint32 i, x, y, numToXfer;
      Media2Ring *media2ring= 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 > media2ring->ringCurPtr){
        numToXfer= (XDAS_Int8*)bufPtr-media2ring->ringCurPtr;
        memcpy(media2ring->ringCurPtr, media2ring->mediaPtr, numToXfer);
       media2ring->mediaPtr+= numToXfer;
       media2ring->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*)media2ring->ringEndPtr-
                   media2ring->ringCurPtr;
        memcpy(media2ring->ringCurPtr, media2ring->mediaPtr, numToXfer);
        media2ring->mediaPtr+= numToXfer;
        media2ring->ringCurPtr= media2ring->ringStartPtr;
        numToXfer= (XDAS_Int8*)bufPtr-media2ring->ringStartPtr;
        memcpy(media2ring->ringCurPtr, media2ring->mediaPtr, numToXfer);
        media2ring->mediaPtr+= numToXfer;
        media2ring->ringCurPtr+= numToXfer;
      }

      return;
}
```

Note how the members *mediaPtr* and *ringCurPtr* of the structure *Media2Ring* are updated. At the exit of the callback function, *media2ring->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 decodes a slice of the frame.

To decode an entire frame, several calls to `process` function are needed. Between calls, it is possible to change the output pointer to YUV data. However, contrary to the JPEG decoder, the output pointer cannot be changed.

This feature is useful for a system that doesn't have enough memory to store the YUV output data of the entire frame dumped by the decoder. The slice based decode feature allows a smaller memory footprint to be used.

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.

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, as it ends with 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 impact the overhead has on the overall processing time. For instance, given a 4.4 Mpix frame, the overhead would be only 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 `IIMGDEC1_DynamicParams`. Run time parameters are set when calling the `control` API. If `numAU` is set to `XDM_DEFAULT` then entire frame will be decoded 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/w) x 2, where W is the width of the image and w is the width of a MCU.

If that constraint is not respected, the decoder automatically rounds up `numAU` to the next valid value and returns it in the structure `IIMGDEC1_`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. Note that the `process` API returns the current position of the input and output pointers in the member `curInPtr` and `curOutPtr` of `the IJPEGDEC_OutArgs` structure. The `curOutPtr` value can be used to initialize correctly the output buffer pointers next time the `process` API is called. If the output buffer pointer is equal to the `currOutPtr` value returned by the previous call to `process` API, then slices are stitched together as non-slice processing of a whole frame would do.

Note that JPEG decoder slice based decoding is simpler to operate than JPEG decoder's because there is no need to update a `sliceNum` parameter each time process function is called and last slice does not require special parameter settings.

Slice-mode decoding seamlessly operates with the input bitstream's ring-buffer configuration so both are automatically enabled.

4.2.4 Example of application code that operates slice-mode decoding

The following example implements the different steps described in the previous section. Note that some initialization sections are skipped, see the file jpgdTest355.c for the full example.

```
inArgs.ringBufStart= ringbuf;

inArgs.ringBufSize= RINGBUF_SIZE;

/* Basic Algorithm process() call, to parse header  */

retVal = IIMGDECFxns->process(

          (IIMGDEC1_Handle)handle,

          (XDM1_BufDesc *)&inputBufDesc,

          (XDM1_BufDesc *)&outputBufDesc,

          (IIMGDEC1_InArgs *)&inArgs,

          (IIMGDEC1_OutArgs *)&outArgs);

bytesConsumed  += outArgs.imgdecOutArgs.bytesconsumed;


/* Call get status to get number of total MCUs */

IIMGDECFxns->control((IIMGDEC1_Handle)handle, XDM_GETSTATUS,

          (IIMGDEC1_DynamicParams *)&extn_dynamicParams,

          (IIMGDEC1_Status *)&status);

totalAU= status.imgdecStatus.totalAU;

/* Set run-time parameters such as: no header decoding and size of
slice */

extn_dynamicParams.imgdecDynamicParams.decodeHeader = XDM_DECODE_AU;

extn_dynamicParams.imgdecDynamicParams.numAU= totalAU/20;


/* Set Run time parameters in the Algorithm via control()    */

IIMGDECFxns->control((IIMGDEC1_Handle)handle, XDM_SETPARAMS,

            (IIMGDEC1_DynamicParams *)&extn_dynamicParams,

            (IIMGDEC1_Status *)&status);

numAU= status.numAU;

inputBufDesc.descs[0].buf = outArgs.curInPtr;


/*Basic Algorithm process() call */
```

```
  // Repeat JPEG encoding as many times as necessary until last slice
  for (i=0;i<totalAU;i+= numAU){
      if (retVal = IIMGDECFxns->process((IIMGDEC1_Handle)handle,
                   (XDM1_BufDesc *)&inputBufDesc,
                    (XDM1_BufDesc *)&outputBufDesc,
                    (IIMGDEC1_InArgs *)&inArgs,
                    (IIMGDEC1_OutArgs *)&outArgs)){
            printf("!!!! Error during JPEG decode !!!!\n");
                              break; // break on error.
      /*Error code is in outArgs.imgdecOutArgs.extendedError*/
      }
  // we just stitch the slices one after the other.
      outputBufDesc.descs[0].buf = outArgs.curOutPtr; sequentially
      bytesConsumed += outArgs.imgdecOutArgs.bytesconsumed;
} /* End of For loop */
```

4.3 Resizing

The JPEG decoder possesses some simple resizing capabilities; it can downsize the output along each dimension by a factor of 1/8, ¼, 3/8, ½, 5/8, 3/4, or 7/8.

The application sets the resize ratio by setting `resizeOption` of `IJPEGDEC_DynamicParams`. The interpretation of resizeOption value is as follows:

0: No resize

1: 1/2 resize factor applied to horizontal and vertical dimension.

2: 1/4 resize factor applied to horizontal and vertical dimension.

3: 1/8 resize factor applied to horizontal and vertical dimension.

4: 3/8 resize factor applied to horizontal and vertical dimension.

5: 5/8 resize factor applied to horizontal and vertical dimension.

6: 6/8 resize factor applied to horizontal and vertical dimension.

7: 7/8 resize factor applied to horizontal and vertical dimension.

This feature can be used to save memory for the output buffer. For instance, if the display is VGA size (640x480) and the decoded bitstream is 3296x2480, then the application can set the resize option to ¼, so the output is reduced to an 824 x 620 image That image can be further resized using the preview engine to exactly fit the display size. The output buffer must be large enough to contain an 824 x 620 image.

Finally, if resizing is enabled (`resizeOption` not 0), and if post-processing is enabled, the post-processing input format is forced to block format.

If resizing is disabled, then the application can choose either yuv422 interleaved or block format for the post-processing input format.

4.4 Rotation

On-the fly rotation can be performed by the decoder during image decoding. Choices of rotation are 90, 270, and 180 degrees rotation. Use the parameter `rotation` in the structure `IJPEGDEC_DynamicParams` to set the appropriate rotation. If no rotation is desired, the parameter must be set to 0.

When the rotation is 90 and slice mode is enabled, then the outputBufDesc.descs[0].buf has to be updated.

The following example implements the update,

```
cformat= status.imgdecStatus.outputChromaFormat;

      if (extn_dynamicParams.rotation== 90) {

        Uint16 sliceWidth;


    sliceWidth=numAU*mcuWidth[cformat]/status.imgdecStatus.imageWidth)*m
    cuHeight[cformat];

        sliceWidth= (sliceWidth*resizeOption)/8;

        outputBufDesc.descs[0].buf+= 2*(status.imgdecStatus.outputWidth -
sliceWidth);

      }
```

Rotation, post-processing, and resizing features can be enabled at the same time. Rotation and area decode features cannot be enabled at the same time.

4.5 Area Decode

With this feature, the application can choose to output a sub-area within the whole image. If the original image is much larger than the display, then the end result will be equivalent to zooming into a portion of the image.

The following figure illustrates the area decode feature:

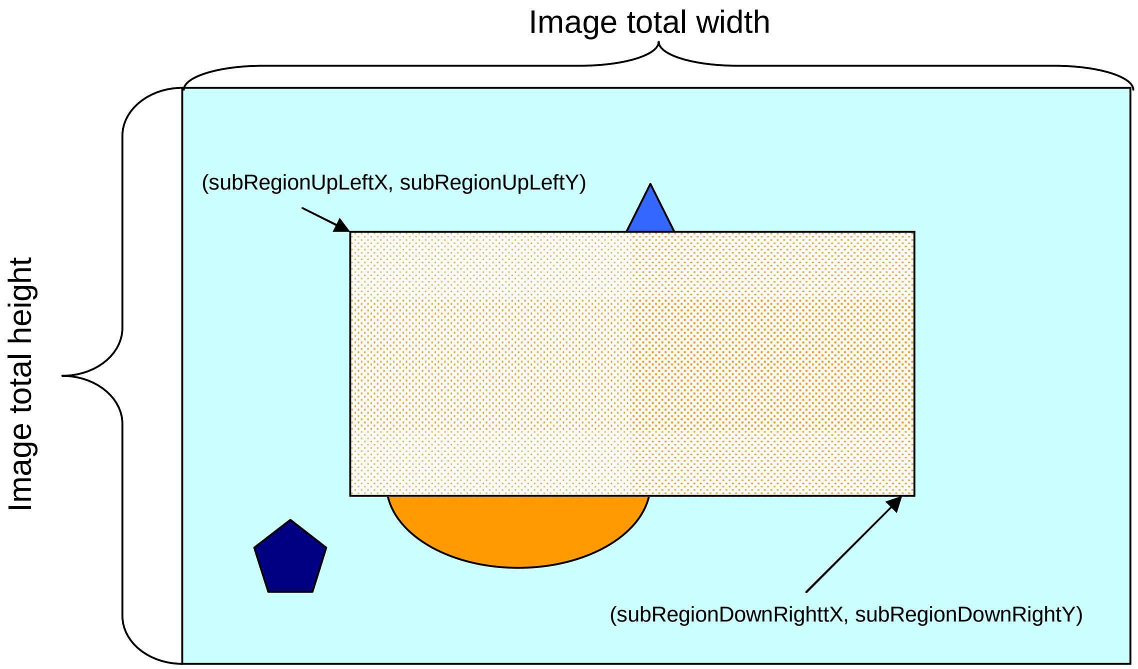


Figure 4-5. Area Decode Example

The slightly dotted area is the area that the decoder will output. The upper left corner of the dotted area will match the upper left corner of the display.

The application passes the coordinates of the upper left corner and lower right corner of the decode area to the JPEG decoder interface by setting the parameters subRegionUpLeftX, subRegionUpLeftY, subRegionDownRightX, subRegionDownRightY in the structure IJPEGDEC_DynamicParams. These coordinates must be multiples of 16 or 8 (depending on the color format) or the decoder will automatically internally round them down. If all coordinates are 0s, the decoder decodes the entire image.

Chapter 5

API Reference

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

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. Not used in this version of JPEG Decoder.
	XDM_LE_16	2	16-bit little endian stream. Not used in this version of JPEG Decoder.
	XDM_LE_32	3	32-bit little endian stream. . Not used in this version of JPEG Decoder.
XDM_ChromaFormat	XDM_CHROMA_NA	-1	Not applicable
	XDM_YUV_420P	1	YUV 4:2:0 planar. Used to specify output color format. Not supported in this version of JPEG Decoder.
	XDM_YUV_422P	2	YUV 4:2:2 planar. Used to specify output color format. Not supported in this version of JPEG Decoder.
	XDM_YUV_422IBE	3	YUV 4:2:2 interleaved (big endian). Used to specify output color format. Not supported in this version of JPEG Decoder.
	XDM_YUV_422ILE	4	YUV 4:2:2 interleaved (little endian). Default choice for output color format.
	XDM_YUV_444P	5	YUV 4:4:4 planar. Used to specify output color format. Not supported in this version of JPEG Decoder.

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	XDM_YUV_411P	6	YUV 4:1:1 planar. Used to specify output color format. Not supported in this version of JPEG Decoder.
	XDM_GRAY	7	Gray format. Used to specify output color format. Not supported in this version of JPEG Decoder.
	XDM_RGB	8	RGB color format. Used to specify output color format. Not supported in this version of JPEG Decoder.
	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.
	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 @c data field of the respective _Status structure. This control command is presently not supported.

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
XDM_DecMode	XDM_DECODE_AU	0	Decode entire access unit. Default value.
	XDM_PARSE_HEADER	1	Parse only header.
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

Note:
The remaining bits that are not mentioned in `XDM_ErrorBit` are interpreted as per the `IJPEGDEC_ErrorStatus` descriptions given below.

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

Table 5-2. IJPEGDEC_ErrorStatus List

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IJPEGDEC_ErrorStatus	JPEGDEC_ERROR_INSUFFICIENT_DATA	Bit 0: 1 - Input buffer underflow 0 - Ignore
	JPEGDEC_ERROR_DISPLAY_WIDTH	Bit 1: 1 - Invalid display width 0 - Ignore
	JPEGDEC_ERROR_INVALID_ROTATION_PARAM	Bit 2: 1 - Invalid rotation 0 - Ignore
	JPEGDEC_ERROR_INVALID_RESIZE	Bit 3: 1 - Invalid resize 0 - Ignore
	JPEGDEC_ERROR_INVALID_numAU	Bit 4: 1 - Invalid numAU 0 - Ignore
	JPEGDEC_ERROR_INVALID_DecodeHeader	Bit 5: 1 - When DecodeHeader is other than 0 or 1 0 - Ignore
	JPEGDEC_ERROR_UNSUPPORTED_ChromaFormat	Bit 6: 1 - Invalid force chroma 0 - Ignore
	JPEGDEC_ERROR_UNSUPPORTED_dataEndianness	Bit 7: 1 - Invalid dataEndianness 0 - Ignore
	JPEGDEC_ERROR_INVALID_SUBWINDOW	Bit 8: 1 - Invalid decode area 0 - Ignore

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.

5.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- `XDM1_BufDesc`
- `XDM1_SingleBufDesc`
- `XDM_AlgBufInfo`
- `IIMGDEC1_Fxns`
- `IIMGDEC1_Params`
- `IIMGDEC1_DynamicParams`
- `IIMGDEC1_InArgs`
- `IIMGDEC1_Status`
- `IIMGDEC1_OutArgs`
- `IDMA3_Handle`
- `IDMA3_ChannelRec`

5.2.1.1 XDM1_BufDesc

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
numBufs	XDAS_Int32	Input	Number of buffers contained
descs	XDM1_SingleBufDesc (*)[XDM_MAX_IO_BUFFERS]	Input	An array of single buffer descriptor objects. XDM_MAX_IO_BUFFERS is defined to be 16.

5.2.1.2 XDM1_SingleBufDesc

|| Description

This structure contains elements required to hold one data buffer..

|| Fields

Field	Datatype	Input/ Output	Description
*buf	XDAS_Int8	Input	Pointer to the vector containing buffer address
bufSize	XDAS_Int32	Input	Size of buffer in bytes

5.2.1.3 XDM1_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[XD	XDAS_Int32	Output	Size in bytes required for each

M_MAX_IO_BUFFERS]			input buffer
minOutBufSize[XDM_MAX_IO_BUFFERS]	XDAS_Int32	Output	Size in bytes required for each output buffer

> Note:
>
> For JPEG Decoder, the buffer details are:
>
> - Number of input buffer required is 1 for the bitstream.
> - The input buffer size is the size of the bitstream. Worst case input size is (height * width * 3) bytes for YUV444
> - Number of output buffer required is 1 for `YUV 422ILE`
> - The output buffer sizes (in bytes) = (height * width * 2)

5.2.1.4 IIMGDEC1_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 (*process)(IIMGDEC1_Handle handle, XDM1_BufDesc *inBufs, XDM1_BufDesc *outBufs, IIMGDEC1_InArgs *inargs, IIMGDEC1_OutArgs *outargs)	Input	Pointer to the `process()` function.

*control	XDAS_Int32 (*control)(IIMGDEC1_Handle handle, IIMGDEC1_Cmd id, IIMGDEC1_DynamicParams *params, IIMGDEC1_Status *status)	Input	Pointer to the control() function.

5.2.1.5 IIMGDEC1_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. Default is 1600.
maxWidth	XDAS_Int32	Input	Maximum image width to be supported in pixels. Default is 2048.
maxScans	XDAS_Int32	Input	Not supported in this version of the JPEG decoder.
dataEndianness	XDAS_Int32	Input	Endianness of output data. This version of the JPEG decoder supports only XDM_BYTE (Default).
forceChromaFormat	XDAS_Int32	Input	Force decoding in given Chroma format. This version of the JPEG decoder supports only XDM_YUV_422ILE (Default).

5.2.1.6 IIMGDEC1_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. Run time parameters change the behavior of the JPEG processing and can be set before each call to the process() function.

|| 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 decode, must be set to XDM_DEFAULT in case of decoding entire frame.
`decodeHeader`	`XDAS_Int32`	Input	Decode entire access unit or only header. See `XDM_DecMode` enumeration for details.
`displayWidth`	`XDAS_Int32`	Input	If the field is set to: ❑ 0 - Use image width as pitch. ❑ Any non-zero value, display width is used as pitch (if capture width is greater than image width).

5.2.1.7 IIMGDEC1_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.
`numBytes`	`XDAS_Int32`	Input	Number of valid input data in bytes in input buffer

5.2.1.8 IIMGDEC1_Status

|| Description

This structure defines parameters that describe the status of 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	Extended error code. See XDM_ErrorBit enumeration for details.
outputHeight	XDAS_Int32	Output	Output height
outputWidth	XDAS_Int32	Output	Output width (image width rounded up to a multiple of the MCU width)
imageWidth	XDAS_Int32	Output	image width
outChromatformat	XDAS_Int32	Output	Output chroma format: XDM_ChromaFormat
totalAU	XDAS_Int32	Output	Total number of Access Units (say MCU) in the image.
totalScan	XDAS_Int32	Output	Total number of scans
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

5.2.1.9 IIMGDEC1_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	Extended error code. See XDM_ErrorBit enumeration for details.
currentAU	XDAS_Int32	Output	Current Access Unit (MCU) Number
currentScan	XDA_Int32	Output	Current scan number
bytesConsumed	XDAS_Int32	Output	The number of bytes consumed.

5.2.1.10 IDMA3_Handle

‖ Description

IDMA3_Handle is a pointer of type IDMA3_Obj holds the private state associated with each logical DMA channel.

‖ Fields

Field	Datatype	Input/ Output	Description
numTccs	unsigned short	Output	The number of TCCs allocated to this channel. In the present implementation since TCCs are fixed this value is set to zero.
numPaRams	unsigned short	Output	The number of PaRam entries allocated to this channel.
*tccTable	unsigned char	Output	TCCs assigned to channel - set to NULL.
paRamAddr	Uns *	Output	PaRAMs assigned to channel
qdmaChan	unsigned short	Output	Physical QDMA Channel assigned to handle - set to zero

			since no QDMA channels are used in current implementation.
transferPending	Bool	Output	Set to true when a new transfer is started on this channel. Set to false when a wait/sync operation is performed on this channel..
`env`	`void *`	Output	IDMA3_ProtocolHandle ('protocol') dependent private channel memory The memory for the 'env' is allocated and reclaimed by the framework when this IDMA3 channel has been requested with a non-NULL 'protocol'. The size, type and alignment of the allocated 'env' memory is obtained by calling the channel's 'protocol'->getEnvMemRec() function. During channel creation, the 'env' pointer must always be created as a private and persistent memory assigned to the IDMA3 channel object. However, the framework/resource manager is also allowed to allocate requested internal 'env' memory as 'scratch' memory which can only be used when the channel is in active state. In the 'scratch' allocation case, the framework/resource manager must still allocate the 'env' as 'persistent', possibly in external memory, and must pass the address of the 'scratch' 'internal' 'env' memory in the first word of the 'env' memory. If the channel 'env' memory is created as 'persistent' with no 'scratch' shadow, then the first word of the env memory must be set to NULL.
protocol	IDMA3_ProtocolHandle	Output	The channel protocol functions used by the DMA manager to determine memory requirements for the 'env'.
persistent	Bool	Output	Indicates if the channel has been allocated with persistent property.

5.2.1.11 IDMA3_ChannelRec

‖ Description

DMA Channel Descriptor to logical DMA channels.

‖ Fields

Field	Datatype	Input/ Output	Description
handle	IDMA3_Handle	Input	Handle to logical DMA channel
numTransfers	Int	Output	Number of DMA transfers that are submitted using this logical channel handle. Single (==1) or Linked (>= 2). In the current implementation this is set to number of PaRamSets required by the application.
numWaits	Int	Output	Number of individual transfers that can be waited in a linked start. (Always set to 1 - for single transfers or for waiting all)
priority	IDMA3_Priority	Output	Relative priority recommendation: High, Medium, Low. - set to IDMA3_PRIORITY_LOW always
protocol	IDMA3_ProtocolHandle	Output	When non-NULL, the protocol object provides interface for querying and initializing logical DMA channel for use by the given protocol. The protocol can be IDMA3_PROTOCOL_NULL in this case no 'env' is allocated In current implementation its set to NULL always.
persistent	Bool	Output	When persistent is set to TRUE, the PaRAMs and TCCs will be allocated exclusively for this channel. They cannot be shared with any other IDMA3 channel. In the current implementation, this is always set to TRUE.

5.2.2 JPEG Decoder Data Structures

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

- IJPEGDEC_Params
- IJPEGDEC_DynamicParams
- IJPEGDEC_Status
- IJPEGDEC_InArgs
- IJPEGDEC_OutArgs

5.2.2.1 IJPEGDEC_Params

|| Description

This structure defines the base creation parameters and any other implementation specific parameters for the JPEG Decoder instance object. The base creation parameters are defined in the XDM data structure, IIMGDEC1_Params.

|| Fields

Field	Datatype	Input/ Output	Description
imgdecParams	IIMGDEC1_Params	Input	Base creation parameters. See IIMGDEC1_Params data structure for details
halfBufCB	XDAS_Void (*) (Uint32 curBufPtr, XDAS_Void*arg)	Input	Half buffer callback function pointer
halfBufCBarg	XDAS_Void *	Input	Half buffer callback argument

5.2.2.2 IJPEGDEC_DynamicParams

‖ Description

This structure defines the base runtime creation parameters and any other implementation specific runtime parameters for the JPEG Decoder instance object. The base runtime parameters are defined in the XDM data structure, `IIMGDEC1_DynamicParams`.

‖ Fields

Field	Datatype	Input/ Output	Description
imgdecDynamicParams	`IIMGDEC1_DynamicParams`	Input	Base creation parameters. See `IIMGDEC1_Params` data structure for details
disableEOI	XDAS_Int16	Input	0: EOI decoding enabled (Default). 1: EOI decoding disabled
resizeOption	XDAS_Int32	Input	Set the resize option: 0: no resizing (Default) 1: resize 1/2 2: resize 1/4 3: resize 1/8 4: resize 3/8 5: resize 5/8 6: resize 6/8 7: resize 7/8
postProc	IJPEGDECPostP	Input	Pointer to post-processing object. This version of the JPEG decoder does not support this field. Please set this as NULL.
subRegionUpLeftX	XDAS_Int16	Input	X coordinate of upper left corner of area decode. Must be multiple of 16.
subRegionUpLeftY	XDAS_Int16	Input	Y coordinate of upper left corner of area decode. Must be multiple of 8 for yuv422, yuv444, 16 for yvu420.
subRegionDownRightX	XDAS_Int16	Input	X coordinate of lower right corner of area decode. Must be multiple of 16.

Field	Datatype	Input/ Output	Description
subRegionDownRightY	XDAS_Int16	Input	Y coordinate of lower right corner of area decode. Must be multiple of 8 for yuv422, yuv444, 16 for yvu420.
rotation	XDAS_Int16	Input	Set the rotation angle: 0: no rotation (default) 180, 90, 270.

5.2.2.3 IJPEGDEC_Status

|| Description

This structure defines the base status parameters and any other implementation specific status parameters for the JPEG Decoder instance object. The base status parameters are defined in the XDM data structure, `IIMGDEC1_Status`. Status parameters are returned by the JPEG decoder upon calling the control function with `XDM_GETSTATUS` as command. Usually application gets status parameters after header is parsed.

|| Fields

Field	Datatype	Input/ Output	Description
imgdecStatus	`IIMGDEC1_Status`	Output	Base status parameters. See `IIMGDEC1_Status` data structure for details
mode	XDAS_Int32	Output	0: baseline sequential 1: progressive
imageHeight	XDAS_Int32	Output	Actual image height of the image.
stride[3]	XDAS_Int32	Output	Stride values for Y,U and V components. This version does not support this.
decImageSize	XDAS_Int32	Output	Size of the decoded image in bytes
lastMCU	XDAS_Int32	Output	Last MCU in the frame 0: Not last
numAU	XDAS_Int32	Output	Number of MCUs in a slice computed by the decoder

Field	Datatype	Input/ Output	Description
nextFreeCmdPtr	XDAS_Uint16*	Output	Pointer to next free word in co-processor command memory – not used in current implementation.
nextFreeImBufPtr	XDAS_Uint8*	Output	Pointer to next free byte in image buffer – not used in current implementation.
nextFreeCoefBufPtr	XDAS_Uint8*	Output	Pointer to next free byte in co-processor coeff memory – not used in current implementation.

5.2.2.4 IJPEGDEC_InArgs

|| Description

This structure defines the base runtime input parameters and any other implementation specific runtime input parameters for the JPEG Decoder instance object. The base runtime parameters are defined in the XDM data structure, `IIMGDEC1_InArgs`.

|| Fields

Field	Datatype	Input/ Output	Description
imgdecInArgs	`IIMGDEC1_InArgs`	Input	Base input runtime parameters. See `IIMGDEC1_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

5.2.2.5 IJPEGDEC_OutArgs

|| Description

This structure defines the base runtime output parameters and any other implementation specific runtime output parameters for the JPEG Decoder instance object. The base runtime parameters are defined in the XDM data structure, `IIMGDEC1_OutArgs`.

‖ **Fields**

Field	Datatype	Input/ Output	Description
imgdecOutArgs	IIMGDEC1_OutArgs	Output	Base input runtime parameters. See IIMGDEC1_InArgs data structure for details
curInPtr	XDAS_Uint8*	Output	Current input pointer, pointing to bitstream
curOutPtr	XDAS_Uint8*	Output	Current output pointer, pointing to YUV display data

5.3 Interface Functions

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

- **Creation** – `algNumAlloc()`, `algAlloc()`, `dmaGetChannelCnt()`, `dmaGetChannels()`
- **Initialization** – `algInit()`, `dmaInit()`
- **Termination** – `algFree()`

You must call these APIs in the following sequence:

```
1) algNumAlloc()
2) algAlloc()
3) algInit()
4) control()
5) algActivate() – optional for single instance case
6) process()
7) algDeactivate() – optional for single instance case
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* (SPRU360).

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 Decoder Data Sheet for External Data Memory requirements

Name

`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* (SPRU360).

See Also

`algAlloc()`

Name

`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 *TMS320 DSP Algorithm Standard API Reference* (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 Data Structures section for details).

Name

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

```
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 *TMS320 DSP Algorithm Standard API Reference* (SPRU360).

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

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

    IIMGDEC1_Params      params;

    // Set the create time base parameters
    params.size = sizeof(IIMGDEC1_Params);
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM_DEFAULT;
    params.dataEndianness = XDM_BYTE;
    params.forceChromaFormat= XDM_YUV_422ILE;

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

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

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

    IIMGDEC1_Params      params;
    IJPEGDEC_Params      extParams;

    // Set the create time base parameters
    params.size = sizeof(IJPEGDEC_Params);
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM_DEFAULT;
    params.dataEndianness = XDM_BYTE;
    params.forceChromaFormat= XDM_YUV_422ILE;

    // Set the create time extended parameters
    extParams.imgdecParams = params;
    extParams.halfBufCB = NULL;
    extParams.halfBufCBarg = NULL;
```

```
handle = (IALG_Handle) ALG_create((IALG_Fxns *)& JPEGDEC_TI_IJPEGDEC,
                                          (IALG_Handle) NULL,
                                          (IALG_Params *) &extParams)
........................
........................
}
```

See Also

algAlloc(), algMoved()

5.3.3 Control Processing API

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

Name

control() – control call

Synopsis

```
XDAS_Int32 (*control)( IIMGDEC1_Handle handle, IIMGDEC1_Cmd id,
IIMGDEC1_DynamicParams *params, IIMGDEC1_Status *status);
```

Arguments

```
IIMGDEC1_Handle handle; /* algorithm instance handle */

IIMGDEC1_Cmd id; /* id of command */

IIMGDEC1_DynamicParams *params; /* pointer to dynamic parameters */

IIMGDEC1_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/decoding. 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 IIMGDEC_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 decoder so next time process() is called, a new bitstream is decoded.

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

XDM_FLUSH: not supported in this version of JPEG decoder

XDM_GETBUFINFO: get required number of I/O buffers and their sizes. Results are returned in the bufInfo member of the structure IIMGDEC1_Status whose pointer is passed as 4th argument.

The third argument is a pointer to a dynamic params structure of type IIMGDEC1_DynamicParams or IJPEGDEC1_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 IIMGDEC1_Status or IJPEGDEC1_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 parameters of dynamic parameters structure must be set before making control call to XDM_SETPARAMS.

Postconditions

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

If the control call 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 base dynamic parameters for a 720x480 input.

```
{
    IIMGDEC1_DynamicParams      dynParams;
    IIMGDEC1_Status             status;

........................
........................

    // Set the dynamic base parameters
    dynParams.size = sizeof(IIMGDEC1_DynamicParams);
    dynParams.numAU= XDM_DEFAULT;
    dynParams.decodeHeader = XDM_DEFAULT;
    dynParams.displayWidth = 720;

/* Set Dynamic Params */
retVal = IIMGDECFxns->control((IIMGDEC1_Handle)handle, XDM_SETPARAMS,
                             (IIMGDEC1_DynamicParams *)& dynParams,
                             (IIMGDEC1_Status *)&status);
........................
........................

}
```

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

```
{
```

```
........................
........................
    IIMGDEC1_DynamicParams      dynParams;
    IIMGDEC1_Status             status;
    IJPEGDEC_DynamicParams      extDynParams;
........................
........................
    // Set the dynamic base parameters
    dynParams.size = sizeof(IIMGDEC1_DynamicParams);
    dynParams.numAU= XDM_DEFAULT;
    dynParams.decodeHeader = XDM_DEFAULT;
    dynParams.displayWidth = 720;

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

    extDynParams.disableEOI = 0;
    extDynParams.resizeOption = 0;
    extDynParams.subRegionUpLeftX = XDM_DEFAULT;
    extDynParams.subRegionUpLeftY = XDM_DEFAULT;
    extDynParams.subRegionDownRightX= XDM_DEFAULT;
    extDynParams.subRegionDownRightY= XDM_DEFAULT;
    extDynParams.rotation= 0;

/* Control call to Set Dynamic Params */
retVal = IIMGDECFxns->control((IIMGDEC1_Handle)handle, XDM_SETPARAMS,
                              (IIMGDEC1_DynamicParams *)& extDynParams,
                              (IIMGDEC1_Status *)&status);
........................
........................
}
```

See Also

`algInit()`, `algDeactivate()`, `process()`

5.3.4 Data Processing API

The Data processing API processes the input data.

Name

`process()` – basic encoding/decoding call

Synopsis

```
XDAS_Int32 (*process)(IIMGDEC1_Handle handle, XDM_BufDesc *inBufs,
XDM_BufDesc *outBufs, IIMGDEC1_InArgs *inargs, IIMGDEC1_OutArgs *outargs);
```

Arguments

`IIMGDEC1_Handle handle; /* algorithm instance handle */`

`XDM_BufDesc *inBufs; /* algorithm input buffer descriptor */`

```
XDM_BufDesc *outBufs; /* algorithm output buffer descriptor */

IIMGDEC1_InArgs *inargs /* algorithm runtime input arguments */

IIMGDEC1_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 `IIMGDEC1_InArgs` data structure that defines the run time input arguments for an algorithm instance object.

The last argument is a pointer to the `IIMGDEC1_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 `OutArgs` data 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.

Postconditions

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, jpgeTest355_fileIO.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.

Name

`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 *TMS320 DSP Algorithm Standard API Reference* (SPRU360).

See Also

`algAlloc()`

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303,
Dallas, Texas 75265