JPEG Sequential Decoder on DM365

User's Guide

Literature Number: SPRUEV3B
August 2010

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
Wireless	www.ti.com/wireless-apps		

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

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) JPEG Sequential Decoder implementation on the DM365 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) 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 DM365 platform.

This document assumes that you are fluent in the C language, have a good 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 and IRES will be helpful.

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to 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 - API Reference**, describes the data structures and interface functions used in the codec.

- **Appendix A - Feature Descriptions**, describes the additional features supported in JPEG decoder.

- **Appendix B – Revision History,** highlights the changes made to SPRUEV4A codec specific user guide to make it SPRUEV4B.

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* (literature number 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* (literature number SPRA579) describes how to make algorithms compliant with the TMS320 DSP Algorithm Standard which is part of TI's eXpressDSP technology initiative.
- *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8)
- *Using DMA with Framework Components for C64x+* (literature number SPRAAG1).

Related Documentation

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

- *CCITT Recommendation T.81, specifying the JPEG standard*. It is 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
Exif	Exchangeable image file format
DMA	Direct Memory Access
DMAN3	DMA Resource Manager
EVM	Evaluation Module
IDMA3	DMA Resource specification and negotiation protocol
IRES	Resource manager Specifications and Protocols
JPEG	Joint Photographic Experts Group
JFIF	JPEG File Interchange Format

Abbreviation	Description
MCU	Minimum Coded Unit
MJCP	MPEG JPEG Co-Processor
VICP	Video and Imaging Co-Processor
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media
YUV	Raw Image format Y - Luminance Component U, V - Chrominance components

Note:

MJCP and VICP refer to the same hardware co-processor blocks.

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, quote the product name (JPEG Sequential Decoder on DM365) 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, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM644x, and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter introduces XDAIS, XDM, and IRES. It also provides an overview of TI's implementation of the JPEG Decoder on the DM365 platform and its supported features.

Topic **Page**

1.1 Overview of XDAIS, XDM, and IRES

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). IRES is the interface for management and utilization of special resource types such as hardware accelerators, certain types of memory and DMA. This interface allows the client application to query and provide the algorithm its requested 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* (literature number SPRU360).

1.1.2 XDM Overview

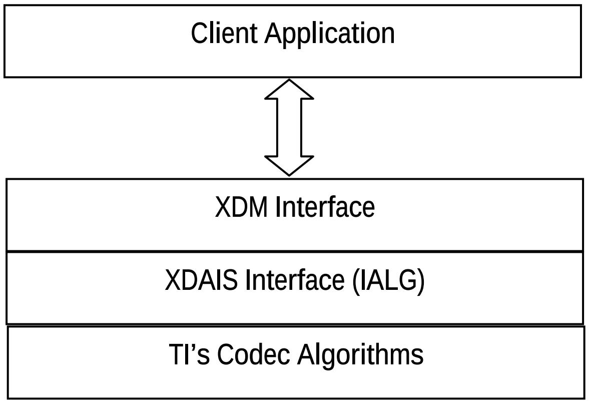
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a imaging decoder system, you can use any of the available image decoders (such as Sequential JPEG, Progressive JPEG Decoder) in your system. To enable easy 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.



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 *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8B).

1.1.3 IRES Overview

IRES is a generic, resource-agnostic, extendible resource query, initialization and activation interface. The application framework defines implements and supports concrete resource interfaces in the form of IRES extensions. Each algorithm implements generic IRES interface, to request one or more concrete IRES resources. IRES defines standard interface functions that the framework uses to query, initialize, activate/deactivate and reallocate concrete IRES resources. To create an algorithm instance within an application framework, the algorithm and the application framework agrees on the concrete IRES resource types that are requested. The framework calls the IRES interface functions, in addition to the IALG functions, to perform IRES resource initialization, activation and deactivation.

The IRES interface introduces support for a new standard protocol for cooperative preemption, in addition to the IALG-style non-cooperative sharing of scratch resources. Co-operative preemption allows activated algorithms to yield to higher priority tasks sharing common scratch resources. Framework component includes the following modules and interfaces to support algorithms requesting IRES-based resources:

- **IRES** - Standard interface allowing the client application to query and provide the algorithm with its requested IRES resources.

- **RMAN** - Generic IRES-based resource manager, which manages and grants concrete IRES resources to algorithms and applications. RMAN uses a new standard interface, the IRESMAN, to support run-time registration of concrete IRES resource managers.

Client applications call the algorithm’s IRES interface functions to query its concrete IRES resource requirements. If the requested IRES resource type matches a concrete IRES resource interface supported by the application framework, and if the resource is available, the client grants the algorithm logical IRES resource handles representing the allotted resources. Each handle provides the algorithm with access to the resource as defined by the concrete IRES resource interface.

IRES interface definition and function calling sequence is depicted in the following figure. For more details, see *Using IRES and RMAN Framework Components for C64x+* (literature number SPRAAI5).

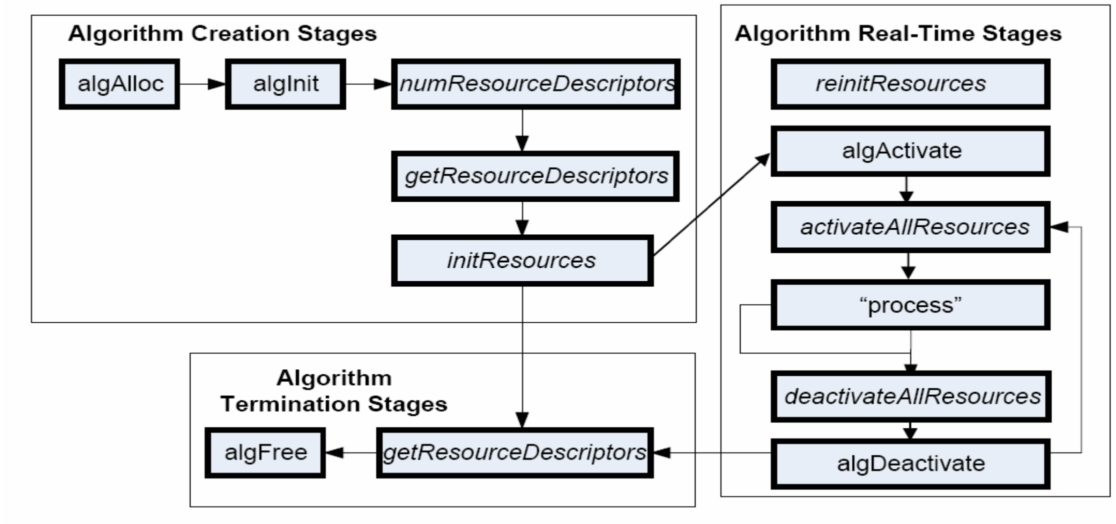


Figure 1-1. IRES Interface Definition and Function Calling Sequence.

For more details, see *Using IRES and RMAN Framework Components for C64x+* (literature number SPRAAI5).

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 DM365 platform.

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

- eXpressDSP Digital Media (XDM 1.0 IIMGDEC1) interface and IRES interface 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 4:2:2 interleaved data as an output
- Supports YUV 4:2:0 semi-planar (NV12 format, that is, Y planar, Cb Cr interleaved) data as an output

- Supports yuv420, yuv422, yuv444, gray level (with 8x8 pixels MCU) input format (yuv420 Vertical planar input format is supported only for 422ILE output format)
- Supports 8-bit quantization tables
- Supports frame level decoding of images
- Images with resolutions up to (Horizontal MCU size * 1024)*(Vertical MCU size * 1024) pixels 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-entrance
- 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 ring buffer configuration of bit-stream buffer for reducing buffer size requirement. Ring buffer size should be multiple of 4096 Bytes
- Supports rotation of 90, 180 and 270 degree
- Supports multi-instance of JPEG Decoder, and single/multi instance of JPEG Decoder with other DM365 codecs

1.4 Limitations

This codec does not support the following:

- Extended DCT-based process
- Lossless process
- Hierarchical process
- Progressive scan
- Input format of yuv411, gray level with 16x16 pixels MCU
- Decoded image width less than 64 pixels for yuv420/422 and 32 pixels for yuv444
- yuv422 Vertical Planar input for 420 Semi-planar output format
- Decoded height less than 32 pixels
- Source images of 12-bits per sample

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 for NO-OS Standalone

This section describes the hardware and software requirements for the normal functioning of the codec component in Code Composer Studio. For details about the version of the tools and software, see Release Note.

2.1.1 Hardware

- ❑ DM365 EVM (Set the bits 2 and 3 of switch SW4 to high(1) position; Set the bits 4 and 5 of SW5 to high(1) position)
- ❑ XDS560R JTAG

2.1.2 Software

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

- ❑ **Development Environment:** This project is developed using Code Composer Studio version 3.3.81.6 (Service Release-11)
- ❑ **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the TI ARM code generation tools
- ❑ DM365 functional simulator

2.2 System Requirements for Linux

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

2.2.1 Hardware

This codec has been tested as an executable on DM365 EVM board.

2.2.2 Software

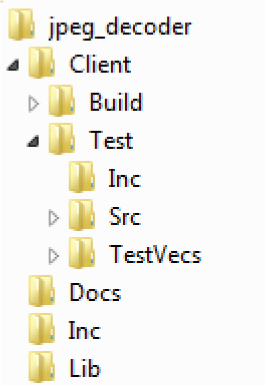
The following are the software requirements for the normal functioning of the codec:

- ❑ **Linux:** Monta Vista Linux 5.0
- ❑ **Code Generation Tools:** This project is compiled, assembled, and linked using the arm_v5t_le-gcc compiler.

2.3 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file to your local hard disk. The zip file extraction creates a top-level directory called 210_I_JPEG_D_01_00, under which another directory named jpeg_decoder is created.

Figure 2-1 shows the sub-directories created in the jpeg_decoder directory.



```
jpeg_decoder
  Client
    Build
    Test
      Inc
      Src
      TestVecs
  Docs
  Inc
  Lib
```

Figure 2-1. Component Directory Structure

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

Table 2-1. Component Directories

Sub-Directory	**Description**
jpeg_decoder/Client/Build	This folder is available only in NO-OS Standalone release package. Not required for linux release package. Contains makefile, cmd file and configuration file to build the NO-OS standalone test application.
jpeg_decoder/Client/Test/Src	Contains application C files, makefile, and configuration file. Executable will be built in this folder.
jpeg_decoder/Client/Test/Inc	Contains header files needed for the application code
jpeg_decoder/Client/Test/TestVecs	Contains test vectors and configuration files
jpeg_decoder/Docs	Contains user guide, datasheet, and release notes
jpeg_decoder/Lib	Contains JPEG Decoder and other support libraries
jpeg_decoder/Inc	Contains JPEG Decoder Interface file

2.4 Building the Sample Test Application for EVM Standalone (NO-OS)

The sample test application that accompanies this codec component will run in TI's Code Composer Studio development environment. To build the sample test application, follow these steps:

1) Verify that you have an installation of TI's Code Composer Studio version 3.3.81.6 (Service Release-11) and code generation tools as provided in the Release Note.

2) Verify if the codec object library jpgd365.lib and iMX library imx365.lib exists in the \Lib sub-directory.

3) Ensure that you have installed the XDC and Framework components releases with version numbers that are mentioned in the release notes.

4) For installing framework component, unzip the content at some location and set the path of the base folder in FC_INSTALL_DIR environment variable

5) Ensure that the installed XDC directory is in the general search PATH.

6) Open the MS-DOS command prompt at the directory \Client\Build\ sub-directory of the release folder.

7) Type the command “**gmake –f jpegdecTestApp.mak**” at the prompt and this generates an executable file, jpegdecApp.out in the \Client\Build\Out sub-directory.

2.5 Running the Sample Test Application on EVM Standalone (NO-OS)

The sample test application that accompanies this codec component will run in TI’s Code Composer Studio development environment. To run the sample test application in Code Composer Studio simulator, follow these steps:

1) Verify that you have an installation of TI’s Code Composer Studio version 3.3.81.6 with Service Release 11 and code generation tools as provided in the Release Note.

2) Verify the SDXDS560R JTAG driver installation version 30329A.

3) Check SW4 and SW5 switch positions of the DM365 EVM. Bit 2 and 3 of SW4 should be set to 1 and remaining should be set to 0. All bits should be set to 0 for SW5.

4) Open Setup Code Composer Studio version 3.3.

5) Select **File > Import**, browse for the .ccs file, and add it.

6) Save the configuration and exit from setup Code Composer Studio. **PDM** opens and displays both ARM926 and ARM968 processors.

7) Right click on ARM926 and connect.

8) Double-click ARM926 to launch Code Composer Studio IDE for the host processor.

9) Add the GEL file and initialize it properly

10) Select **File > Load Program** in Host Code Composer Studio, browse to the \Client\Build\Out\ sub-directory, select the codec executable created in Section 2.4 and load it into Code Composer Studio in preparation for execution.

11) Select **Debug > Run** in Host Code Composer Studio to execute decoder on host side.

    The sample test application takes the input files stored in the \Client\Test\Testvecs\Input sub-directory, runs the codec, and stores the output in \Client\Test\Testvecs\Output sub-directory.

12) Halt the coprocessor from Code Composer Studio IDE.

2.6 Building the Sample Test Application on Linux

The sample test application that accompanies this codec component takes the 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 you have installed Framework Component (FC), XDC, and LSP. For information on the versions, see the release notes.

2) Verify that libjpegdec.a library is present in jpeg_decoder/Lib directory.

3) Verify that support library (libimx.a) are present in jpeg_decoder/Lib directory.

4) Change directory to jpeg_decoder/Client/Test/Src and type `make clean` followed by a `make` command. This uses the Make file in that directory to build the test executable jpgdec-r into the jpeg_decoder/Client/Test/Src directory.

Note:

ARM tool chain, arm_v5t_le-gcc (ARM gcc) compiler path needs to be set in users environment path before building the JPEG decoder executable.

2.7 Running Sample Test Application

To run the jpgdec-r executable on your DM365 EVM board, follow these instructions.

1) Set up the DM365 EVM Board. (For information about setting up the DM365 environment, see the *DM365 Getting Started Guide,* released in the doc directory of DVSDK release package).

2) Ensure that complete Client folder is in the target file system.

3) Copy the kernel modules cmemk.ko, edmak.ko and irqk.ko to the target directory. These modules are provided with the release package in kernel_modules directory

4) Copy loadmodules.sh provided with release package at kernel_modules to the target directory.

5) Load the kernel modules by executing following command:
$./loadmodules.sh

6) Change the directory to Client/Test/Src folder and execute the following command to run the JPEG decoder executable.

   $./jpgdec-r

This will run the JPEG decoder with base parameters. To run the JPEG Decoder with extended parameters execute, $./jpgdec-r –ext

2.8 Configuration Files

This codec is shipped along with:

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

2.8.1 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 jpeg_decoder/Client/Test/TestVecs/Config sub-directory.

The format of the Testvecs.cfg file is:

```
X
Config
Input
Output
```

where:

- `X` may be set as:
  - 2 - for writing the output to the output file. In the current test app file only `X`=2 is supported.
- `Config` is the Decoder parameter file.
- `Input` is the input file name (use complete path).
- `Output` is the output YUV file name.

A sample Testvecs.cfg file is as shown:

```
2
../TestVecs/Config/Testparams.cfg
../TestVecs/Input/colorful_toys_cif.jpg
../TestVecs/Output/colorful_toys_cif_422ile.yuv
```

2.8.2 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, 2:resize by
1/4,
                     # 3: 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   # 4: 422_ILE,
                        # 9: 420 SemiPlanar.
dataEndianness = 1
subRegionUpLeftX = 0
subRegionUpLeftY = 0
subRegionDownRightX = 0
subRegionDownRightY = 0
```

Any field in the `IIMGDEC1_Params` structure (see Section 4.2.1.5) 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 modify the array `sTokenMap` in the test application to handle these fields.

This page is intentionally left blank

Chapter 3

Sample Usage

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

Topic **Page**

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 requires the following number of TCCs and Param sets.

Required Channels	Required Param Sets
22 channels	All param sets associated with TCCS and 11 additional param sets.

2) 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.

3) As all codecs use the same hardware resources, only one process call per codec is invoked at a time (frame level reentrancy). The process call needs to be wrapped within activate and deactivate calls for context switch. See XDM specification on activate/deactivate.

4) 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.

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

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

7) Codec atomicity is supported at frame level processing only. The process call runs until completion before another process call can be invoked.

3.2 Overview of the Test Application

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.

Following figure shows the sequence of APIs exercised in the sample test application.

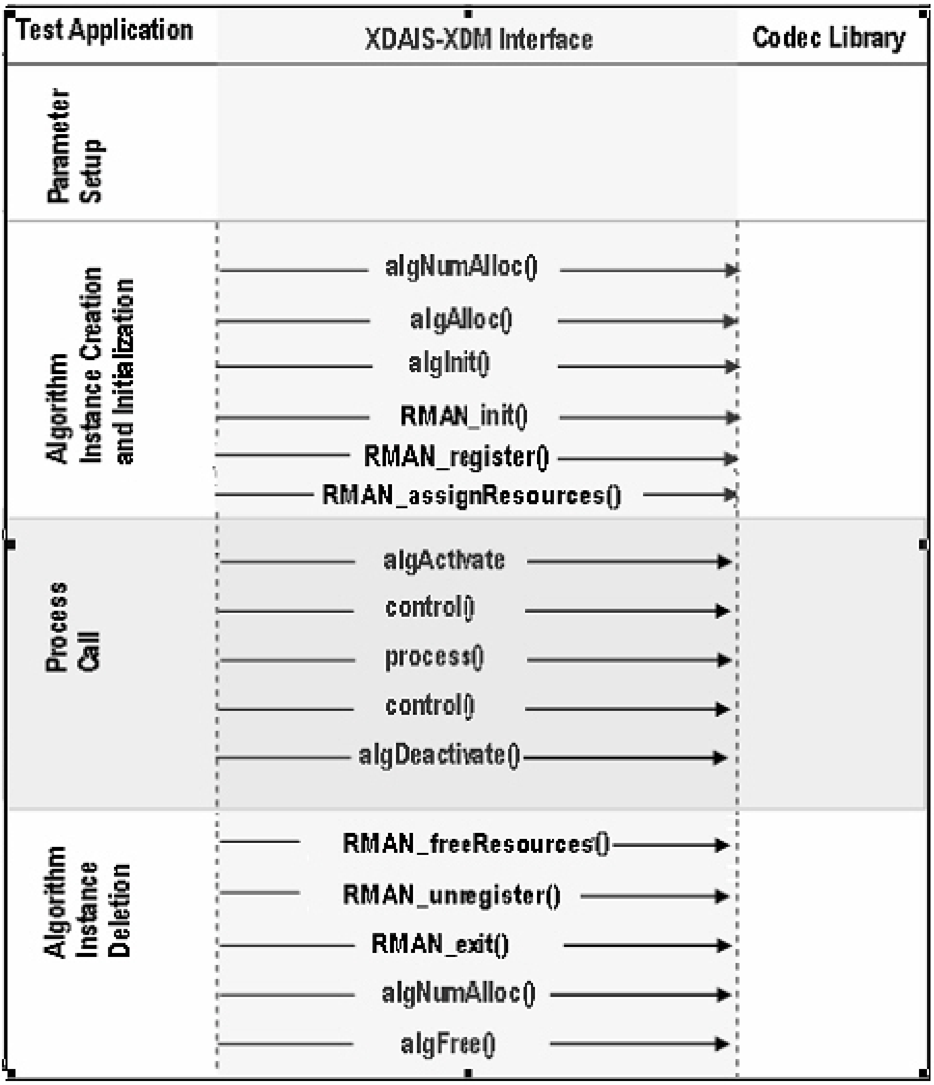


Figure 3-1. Test Application Sample Implementation.

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 2.8.

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 these steps, the test application does the algorithm instance creation and initialization.

3.2.2 Algorithm Instance Creation and Initialization

In this logical block, the test application accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs are called in sequence:

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.

After the algorithm instance is created successfully, the test application does DMA and VICP resource allocation for the algorithm. This requires initialization of RMAN and grant of VICP and DMA resources. This is implemented by calling RMAN interface functions in following sequence:

1) `RMAN_init` : To initialize the RMAN module.

2) `RMAN_register` : To register the VICP protocol / resource manager with generic resource manager.

3) `RMAN_assignresources` : To register resources to the algorithm as requested VICP protocol/resource manager. Process Call in single instance scenario.

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.

Note:

`algActivate()` is a mandatory call before first `process()`call, as it does hardware initialization.

3.2.3 Algorithm Instance Deletion

Once encoding/decoding is complete, the test application frees up the DMA and VICP resource allocated for algorithm. This is implemented by calling RMAN interface functions in following sequence.

1) `RMAN_freeResources()`: To free the resources allocated to the algorithm before process call.

2) `RMAN_unregister()`: To unregister VICP protocol/resource manager with the generic resource manager.

3) `RMAN_exit()`: To delete the generic IRES RMAN and release the memory.

After freeing the assigned resources, 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.

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.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:

- DMA `channels` and `PaRamSets`
- 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 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.

Some of the API combinations that need to be protected with single mutex are:

- `control()` call of one instance sets post-processing function properties by setting the command length, and so on. 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) will change to include both `algActivate()` and `algDeactivate()` as mandatory calls of the algorithm.

3.2.4 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.

Note:

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

This page is intentionally left blank

Chapter 4

API Reference

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

4.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. For each symbolic constant, the semantics or interpretation of the same is also provided.

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
XDM_DataFormat	XDM_BYTE	Big endian stream.
	XDM_LE_16	16-bit little endian stream.
	XDM_LE_32	32-bit little endian stream.
XDM_DecMode	XDM_DECODE_AU	Decode entire access unit.
	XDM_PARSE_HEADER	Performs JPEG header parsing.
XDM_ChromaFormat	XDM_CHROMA_NA	Not applicable
	XDM_YUV_420P	YUV 4:2:0 planar.
	XDM_YUV_422P	YUV 4:2:2 planar.
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian).
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian).
	XDM_YUV_444P	YUV 4:4:4 planar.
	XDM_YUV_411P	YUV 4:1:1 planar.
	XDM_GRAY	Gray format.
	XDM_CHROMAFORMAT_DEFAULT	Default chroma format value set to XDM_YUV_422ILE
	XDM_YUV_420SP	YUV 420 semi planar format
	XDM_RGB	RGB color format.
XDM_CmdId	XDM_GETSTATUS	Query JPEG Decoder instance to fill the IJPEGDEC_Status structure.
	XDM_SETPARAMS	Set run-time dynamic parameters.
	XDM_RESET	Reset the decoder.
	XDM_SETDEFAULT	Initialize creation time parameters to library defaults.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_FLUSH	Handle end of stream conditions. This command forces JPEG Decoder instance to output data without additional input. Not supported.
	XDM_GETBUFINFO	Query JPEG Decoder instance regarding properties of input and output buffers.
XDM_ErrorBit		The bit-fields in the 32-bit error code are interpreted as shown.
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore This bit is not used in this version of JPEG Decoder.
	XDM_INSUFFICIENTDATA	Bit 10 ❑ 1 - Insufficient input data ❑ 0 - Ignore
	XDM_CORRUPTEDDATA	Bit 11 ❑ 1 - Data problem/corruption ❑ 0 - Ignore
	XDM_CORRUPTEDHEADER	Bit 12 ❑ 1 - Corrupted frame header ❑ 0 - Ignore
	XDM_UNSUPPORTEDINPUT	Bit 13 ❑ 1 - Unsupported feature/parameter in input ❑ 0 - Ignore
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore
	XDM_FATALERROR	Bit 15 ❑ 1- Fatal error (stop decoding) ❑ 0 - Recoverable error

Note:

The remaining bits that are not mentioned in XDM_ErrorBit are interpreted as per the IJPEGDEC_ErrorStatus descriptions provide in Table 4-2.

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

Table 4-2. IJPEGDEC_ErrorStatus List.

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

4.2 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.

4.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- ❑ XDM1_BufDesc
- ❑ XDM1_SingleBufDesc
- ❑ XDM1_AlgBufInfo
- ❑ IIMGDEC1_Fxns
- ❑ IIMGDEC1_Params
- ❑ IIMGDEC1_DynamicParams
- ❑ IIMGDEC1_InArgs
- ❑ IIMGDEC1_Status
- ❑ IIMGDEC1_OutArgs

4.2.1.1 XDM1_BufDesc

|| Description

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

|| Fields

Field	Data Type	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.

4.2.1.2 XDM1_SingleBufDesc

|| Description

This structure contains elements required to hold one data buffer.

|| Fields

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

4.2.1.3 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	Data Type	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:

The JPEG Decoder has the following buffer information:

- If `GETBUFINFO` control command is called after `XDM_PARSE_HEADER`, the calculated input/output buffer size will be based on actual image width/height, else it will be based on `maxWidth`/`maxHeight` parameters value.
- Number of input buffer required is 1
- The input buffer size is the size of the bit-stream. Worst case input size is (height * width * 3) bytes for `YUV444`
- Number of output buffer required is 1 for `YUV 422ILE` and 2 for `YUV 420SP`
- The output buffer sizes (in bytes) = (height * width * 2) for `YUV 422ILE` case. See Datasheet for more details.

4.2.1.4 IIMGDEC1_Fxns

|| Description

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

|| Fields

Field	Data Type	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.

4.2.1.5 IIMGDEC1_Params

|| Description

This structure defines the creation parameters for an algorithm instance object. Set this data structure to `NULL`, if you are not sure of the values to be specified for these parameters.

|| Fields

Field	Data Type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the base or extended (if being used) data structure in bytes. Base: 24 Bytes (Default) Extended: 32 Bytes
`maxHeight`	`XDAS_Int32`	Input	Maximum image height. Set multiples of 16 to support all chroma format. Valid range: 32 to 16384 Default: 1600.
`maxWidth`	`XDAS_Int32`	Input	Maximum image width. Set multiples of 32 to support all chroma format. Valid range: 32 to 16384 Default: 2048.
`maxScans`	`XDAS_Int32`	Input	Maximum number of scans for progressive mode. Not supported.
`dataEndianness`	`XDAS_Int32`	Input	Endianness of input data. The JPEG decoder implementation supports only `XDM_BYTE` format.

Field	Data Type	Input/ Output	Description
forceChromaFormat	XDAS_Int32	Input	Force decoding in the given chroma format. See XDM_ChromaFormat enumeration for details. Supported values are: □ XDM_CHROMAFORMAT_DEFAULT- YUV 422 Interleaved. □ XDM_YUV_422ILE(4) - YUV 422 interleaved. (Default) □ XDM_YUV_420SP (9): YUV 420 Semi Planar.

4.2.1.6 IIMGDEC1_DynamicParams

‖ Description

This structure defines the run-time parameters for an algorithm instance object. Set this data structure to NULL, if you are not sure 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	Data Type	Input/ Output	Description
size	XDAS_Int32	Input	Size of the base or extended (if being used) data structure in bytes. Base: 16 Bytes (Default) Extended: 36 Bytes
numAU	XDAS_Int32	Input	Number of access units to decode. Setting this field to XDM_DEFAULT decodes the complete frame. Any value other than XDM_DEFAULT will decode that many MCUs. The minimum value should be number of MCUs per row.
decodeHeader	XDAS_Int32	Input	Flag indicating if only header parsing needs to be done. See XDM_DecMode enumeration for details. □ XDM_PARSE_HEADER - Return after header parsing is done. □ XDM_DECODE_AU - Return after complete frame decoding (default)
displayWidth	XDAS_Int32	Input	□ XDM_DEFAULT - Use the decoded image width as pitch (default) □ If any other value greater than the decoded image width is provided, then this value (in pixels) is used as pitch. Note: displayWidth should be => imageWidth else this parameter will be ignored and imageWidth will be used for pitch.

4.2.1.7 IIMGDEC1_InArgs

|| Description

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

|| Fields

Field	Data Type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the base or extended (if being used) data structure in bytes. Base: 8 Bytes (Default) Extended: 16 Bytes
`numBytes`	`XDAS_Int32`	Input	Number of valid input data in bytes in input buffer.

4.2.1.8 IIMGDEC1_Status

|| Description

This structure defines parameters that describe the status of the algorithm.

|| Fields

Field	Data Type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the base or extended (if being used) data structure in bytes. Base: 180 Bytes (Default) Extended: 228 Bytes
`extendedError`	`XDAS_Int32`	Output	Extended error enumeration. See `XDM_ErrorBit` enumeration for details.
`outputHeight`	`XDAS_Int32`	Output	Decoded image height.
`outputWidth`	`XDAS_Int32`	Output	Decoded image width.
`imageWidth`	`XDAS_Int32`	Output	Actual image width.
`outChromaformat`	`XDAS_Int32`	Output	Output chroma format. See `XDM_ChromaFormat` enumeration for details.
`totalAU`	`XDAS_Int32`	Output	Total number of MCUs.
`totalScan`	`XDAS_Int32`	Output	Total number of scans in the progressive image. Not supported.

Field	Data Type	Input/ Output	Description
`bufInfo`	`XDM_AlgBufInfo`	Output	Input and output buffer information. See `XDM_AlgBufInfo` data structure for details.

4.2.1.9 IIMGDEC1_OutArgs

‖ Description

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

‖ Fields

Field	Data Type	Input/ Output	Description
`size`	`XDAS_Int32`	Input	Size of the base or extended (if being used) data structure in bytes. Base: 20 Bytes (Default) Extended: 32 Bytes
`extendedError`	`XDAS_Int32`	Output	Extended error enumeration. See `XDM_ErrorBit` enumeration for details.
`bytesconsumed`	`XDAS_Int32`	Output	The numbers of input bytes consumed per decode call.
`currentAU`	`XDAS_Int32`	Output	Current MCU number.
`currentScan`	`XDAS_Int32`	Output	Current scan number (for progressive mode). Not supported.

4.2.2 JPEG Decoder Data Structures

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

- ❑ IJPEGDEC_Params
- ❑ IJPEGDEC_DynamicParams
- ❑ IJPEGDEC_InArgs
- ❑ IJPEGDEC_Status
- ❑ IJPEGDEC_OutArgs

4.2.2.1 IJPEGDEC_Params

|| Description

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

|| Fields

Field	Data Type	Input/ Output	Description
imgdecParams	IIMGDEC1_Params	Input	See IIMGDEC1_Params data structure for details.
halfBufCB	XDAS_Void (*) (Uint32, XDAS_Void *)	Input	Half buffer callback function pointer. This field should be set to NULL, if not used. Default: NULL
halfBufCBarg	XDAS_Void *	Input	Half buffer callback argument. This field should be set to NULL, if not used. Default: NULL

4.2.2.2 IJPEGDEC_DynamicParams

‖ Description

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

‖ Fields

Field	Data Type	Input/ Output	Description
`imgdecDynamicParams`	`IIMGDEC1_DynamicParams`	Input	See `IIMGDEC1_DynamicParams` 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 Note: Codec will internally round the width to next multiple of 16 for yuv 420/422 and 8 for yuv 444. It rounds the height to next multiple of 8 for yuv422/yuv444, and 16 for yvu420 before resize operation,
`subRegionUpLeftX`	`XDAS_Int16`	Input	X coordinate of upper left corner of area decode. This should be multiple of 16 for yuv 420/422 and 8 for yuv 444. Default: 0
`subRegionUpLeftY`	`XDAS_Int16`	Input	Y coordinate of upper left corner of area decode. This should be multiple of 8 for yuv422, yuv444, 16 for yvu420. Default: 0
`subRegionDownRightX`	`XDAS_Int16`	Input	X coordinate of lower right corner of area decode. This should be multiple of 16 for yuv 420/422 and 8 for yuv 444. Default: 0
`subRegionDownRightY`	`XDAS_Int16`	Input	Y coordinate of lower right corner of area decode. This should be multiple of 8 for yuv422, yuv444, 16 for yvu420. Default: 0
`rotation`	`XDAS_Int16`	Input	Set the rotation angle: 0: no rotation (default) 180, 90, 270.

Field	Data Type	Input/ Output	Description
decodingPreset	XDAS_Int16	Input	0 : High performance mode 1: Normal mode Note: ❑ If input image is of very high quality and contains lot of high frequency components, this parameter must be set to '1', else decoder may fail to decode. ❑ There is performance drop of 7%-20% based on input format and resolution in normal mode compared to high performance mode

Note:

To disable Area Decode feature, user must set all of the following fields to 0:

- ❑ subRegionUpLeftX
- ❑ subRegionUpLeftY
- ❑ subRegionDownRightX
- ❑ subRegionDownRightY

See section A.5 for details on Area decode.

4.2.2.3 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. The JPEG decoder returns the status parameters when the control function is called with XDM_GETSTATUS as command. Usually, application gets status parameters after header is parsed.

|| Fields

Field	Data Type	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 This parameter is not applicable, since this is only baseline sequential JPEG decoder.
imageHeight	XDAS_Int32	Output	Actual image height of the image.

Field	Data Type	Input/ Output	Description
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
nextFreeCmdPtr	XDAS_UInt16 *	Output	Reserved. Set it to NULL.
nextFreeImBufPtr	XDAS_UInt8 *	Output	Reserved. Set it to NULL.
nextFreeCoefBufPtr	XDAS_UInt8 *	Output	Reserved. Set it to NULL.
inputChromaFormat	XDAS_Int32	Output	Input image chroma format

4.2.2.4 IJPEGDEC_InArgs

‖ Description

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

‖ Fields

Field	Data Type	Input/ Output	Description
imgdecInArgs	IIMGDEC1_InArgs	Input	Base input run-time parameters. See IIMGDEC1_InArgs data structure for details
ringBufStart	XDAS_UInt8 *	Input	Pointer to starting point of bit-stream ring buffer. If ring buffer is not used, set it to NULL
ringBufSize	XDAS_Uint32	Input	Size of ring buffer in bytes. Set it to 0, if ring buffer is not used.

4.2.2.5 IJPEGDEC_OutArgs

‖ Description

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

‖ Fields

Field	Data Type	Input/ Output	Description
imgdecOutArgs	IIMGDEC1_OutArgs	Output	Base input run-time parameters. See IIMGDEC1_InArgs data structure for details
curInPtr	XDAS_Uint8*	Output	Current input pointer, pointing to bit-stream
curOutPtr	XDAS_Uint8*	Output	Current output pointer, pointing to full YUV display data for YUV 422ILE format and only luma (Y) data for YUV 420SP format.
curOutPtrChroma	XDAS_Uint8*	Output	Current output pointer, pointing to chroma data for YUV 420SP format. Not applicable for YUV 422ILE format.

4.3 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()`
- ❑ **Initialization** – `algInit()`
- ❑ **Control** – `control()`
- ❑ **Data processing** – `algActivate()`, `process()`, `algDeactivate()`
- ❑ **Termination** – `algFree()`

You must call these APIs in the following sequence:

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

`control()` can be called any time after calling the `algInit()` API.

`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* (literature number SPRU360).

4.3.1 Creation APIs

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

|| 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* (literature number 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* (literature number SPRU360).

‖ See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

Initialization API is used to initialize 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* (literature number 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()`

4.3.3 Control API

Control API is used for controlling the functioning of the algorithm instance during run-time. This is done by changing the status of the controllable parameters of the algorithm during run-time. These controllable parameters are defined in the `Status` data structure (see Data Structures section for details).

|| Name

control() – change run-time parameters and query the status

|| 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; /* algorithm specific control commands*/
```

```
IIMGDEC1_DynamicParams *params /* algorithm run-time
parameters */
```

```
IIMGDEC1_Status *status /* algorithm instance status
parameters */
```

|| 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 the following values:

- ❑ XDM_GETSTATUS: fill structure IIMGDEC1_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 bit-stream 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 IJPEGDEC_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 IJPEGDEC_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 is an example for initializing the base dynamic parameters for 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 is an example for initializing the extended dynamic parameters for 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()`, `algActivate()`, `process()`

4.3.4 Data Processing API

Data processing API is used for processing the input data.

|| Name

`process()` – basic encoding/decoding call

|| Synopsis

```
XDAS_Int32 (*process)(IIMGDEC1_Handle handle, XDM1_BufDesc
*inBufs, XDM1_BufDesc *outBufs, IIMGDEC1_InArgs *inargs,
IIMGDEC1_OutArgs *outargs);
```

|| Arguments

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

XDM1_BufDesc *inBufs; /* algorithm input buffer descriptor
*/

XDM1_BufDesc *outBufs; /* algorithm output buffer
descriptor */

IIMGDEC1_InArgs *inargs /* algorithm run-time input arguments
*/

IIMGDEC1_OutArgs *outargs /* algorithm run-time 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().
- 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()

4.3.5 Termination API

Termination API is used to terminate the algorithm instance and free 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* (literature number SPRU360).

|| See Also

algAlloc()

Appendix A

Feature Descriptions

This chapter provides descriptions 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, rotation, and area decode.

Topic **Page**

A.1 Bit-stream Ring Buffer in DDR

To minimize the memory requirement, the JPEG decoder reads the JPEG bit-stream 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, and so on) and the decoder. Therefore, the size of the ring buffer can be smaller than the final bit-streams size, effectively reducing the amount of physical DDR memory allocated for storing the bit-stream. The complete bit-stream is processed eventually 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 following figure shows the state of the ring buffer at different states of JPEG processing.

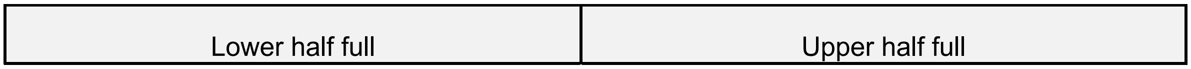


Figure A-1. Ring Buffer Before JPEG Decoder Starts

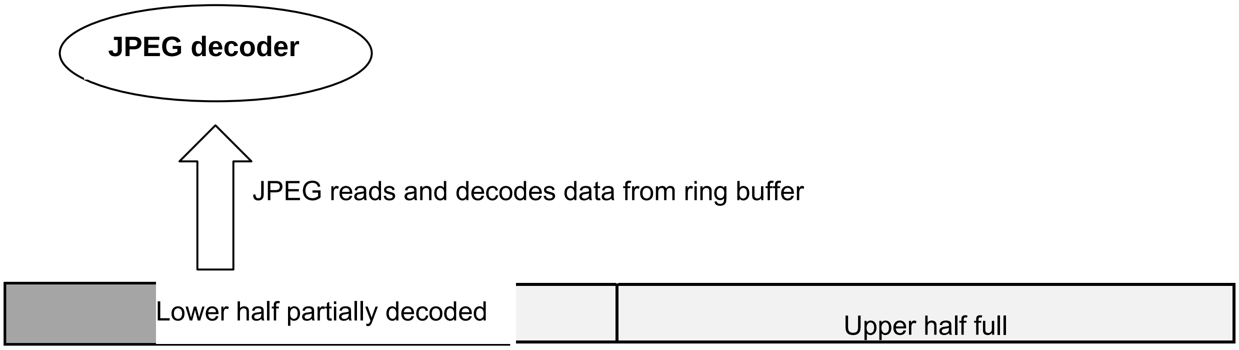

JPEG decoder
JPEG reads and decodes data from ring buffer
Lower half partially decoded
Upper half full

Figure A-2. Ring Buffer Shortly After JPEG Decoder Starts.

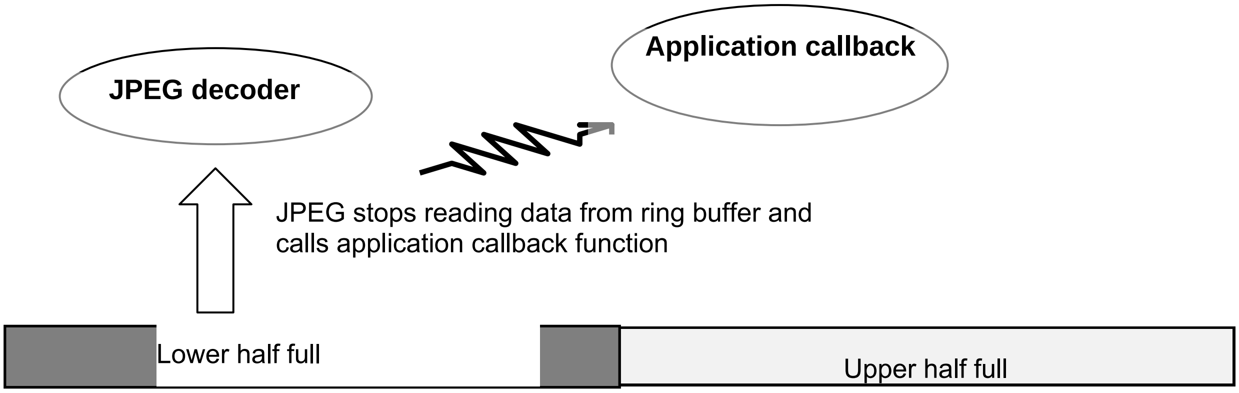

Application callback
JPEG decoder
JPEG stops reading data from ring buffer and calls application callback function
Lower half full
Upper half full

Figure A-3. Ring Buffer Once JPEG Decoder Fills Lower Half

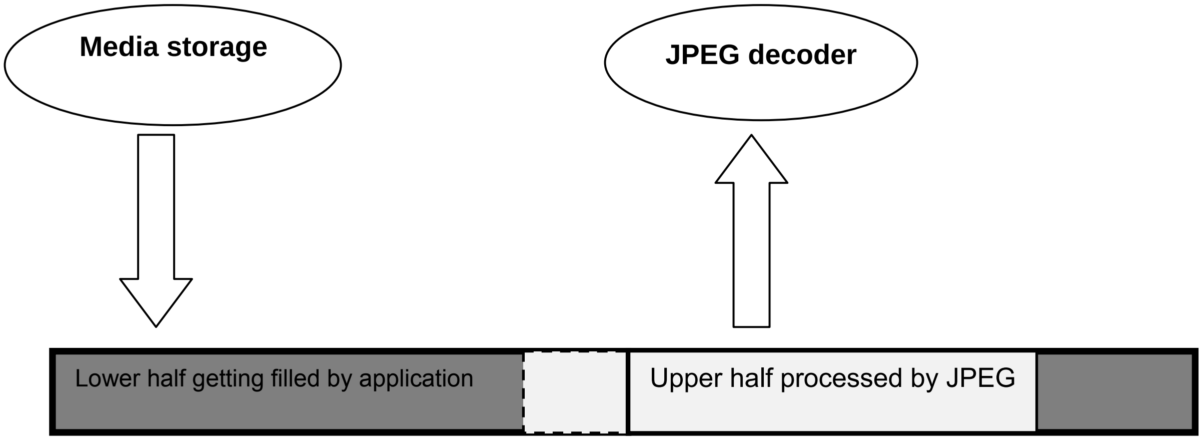

Media storage
JPEG decoder
Lower half getting filled by application
Upper half processed by JPEG

Figure A-4. Ring Buffer After Application Fills First Half and JPEG Decoder Processes Second Half

A.1.1 Operation Mode

The address and size of the ring buffer are passed to the JPEG decoder as run-time input arguments of the `process` function. This section explains how JPEG decoder manages this output ring buffer.

As MCUs are decoded, the application fills the ring buffer with the bit-stream. Each time half the buffer is decoded, the decoder will call a user-defined call back function. The call back 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 over written by the new bit-stream data. The callback function should save curBufPtr. When it is called the next time, 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.

The 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 A.1.3 uses that feature to pass a structure that tracks the transfers between the ring buffer and the media storage.

A.1.2 Constraint

The ring buffer size must be multiple of 4096 bytes.

A.1.3 Guidelines for Using Ring Buffer with JPEG Decoder

This section provides few guidelines to help the you implement the ring buffer into an application using JPEG decoder. It does not provide all the steps required to initialize or run the JPEG decoder but only those related to handling the ring buffer.

The structure, Media2Ring, can be used to track 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:

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;
```

In addition, the ring buffer must be filled by the application prior to the first call of the JPEG decoders process function:

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

```
media2ring.mediaPtr+= RINGBUF_SIZE;
```

The `process()` function is 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:

The members, `mediaPtr` and `ringCurPtr`, of the structure `Media2Ring` are updated such that at the exit of the callback function, `media2ring->ringCurPtr` should be the same value as `bufPtr`.

A.2 Slice Mode Processing

JPEG decoder can be configured to call a `process` only to decode a slice of the frame, instead of processing an entire frame at one time.

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 does not 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.

A.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 with the image width, multiplied by 2. For example, 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.

A.2.2 Slice Mode Processing Overhead

As there is control overhead each time JPEG is started/stopped, you should try to process as few slices as possible per frame. For example, a 1.2 Mpix frame partitioned in 20 slices will 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 example, given a 4.4 Mpix frame, the overhead will only be 4% for a 20 slices frame and 2% for a 10 slices frame.

A.2.3 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 considered, the decoder automatically rounds up `numAU` to the next valid value and returns it in the structure `IIMGDEC1_Status`. Then, it is 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.

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 the output buffer pointers correctly 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 is done.

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 the last slice does not require special parameter settings.

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

A.2.4 Example of Application Code That Operates Slice-mode Decoding

The following example implements the different steps described in the previous section.

Note:

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 */
```

A.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 example, if the display is VGA size (640x480) and the decoded bit-stream is 3296x2480, then the application can set the resize option to ¼. Therefore, the output is reduced to an 824 x 620 image. This 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.

Note:

Codec will internally round the width to next multiple of 16 for yuv 420/422 and 8 for yuv 444. It rounds the height to next multiple of 8 for yuv422/yuv444 and 16 for yuv420 before resize operation,

A.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 `rotation` parameter in the structure `IJPEGDEC_DynamicParams` to set the appropriate rotation. If rotation is not required, the parameter must be set to zero.

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)*mcuHeight[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.

A.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.

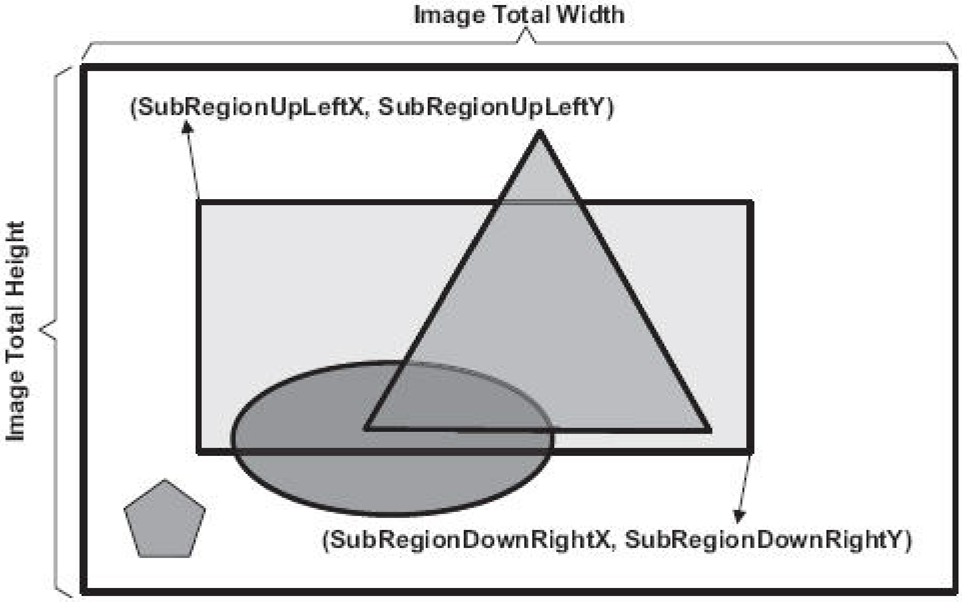


Image Total Width
Image Total Height
(SubRegionUpLeftX, SubRegionUpLeftY)
(SubRegionDownRightX, SubRegionDownRightY)

Figure A-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`. X coordinates must be multiples of 16 for YUV 420/422 format and multiple of 8 for YUV 444 format and Y coordinates should me multiple of 16 for YUV 420 and multiple of 8 for YUV 422/444 format (if not, the decoder will automatically round them internally to multiple of 16 or 8). If all coordinates are zeros, the decoder decodes the entire image.

Decoded image width/height is calculated using:.

- ❑ Width = (`subRegionDownRightX - subRegionUpLeftX)`
- ❑ Height = `(subRegionDownRightY- subRegionUpLeftY)`

Appendix B

Revision History

This revision history highlights the changes made to SPRUEV3A codec specific user guide to make it SPRUEV3B.

Table 4-1. JPEG Sequential Encoder on DM365

Section	**Changes**
Global	❑ There are no changes in the user guide for this release of JPEG Sequential Decoder on DM365