JPEG Sequential Encoder on DM365

User's Guide

Literature Number: SPRUEV4B
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	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) JPEG Encoder 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 the reader is fluent in the C language, and have working knowledge of JPEG encoder. Good knowledge in eXpressDSP Algorithm Interface Standard (XDAIS), eXpressDSP Digital Media (XDM) standard, and IRES 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 - API Reference**, describes the data structures and interface functions used in the codec.
- **Appendix A - Feature Descriptions**, describes the additional features supported in JPEG encoder.
- **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.
- *XDAIS-DM (Digital Media) User Guide* (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*. 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
Exif	Exchangeable image file format
HDVICP	High Definition Video and Imaging Co-Processor sub-system
IRES	Resource manager Specifications and Protocols
JPEG	Joint Photographic Experts Group

Abbreviation	Description
JFIF	JPEG File Interchange Format
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 Encoder 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 and eXpressDSP, 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 Encoder 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. To facilitate these functions, the IALG interface defines the following APIs:

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

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

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

The IALG interface also defines three more optional APIs `algControl()`, `algNumAlloc()`, and `algMoved()`. For more details on these APIs, see *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 an image encoder system, you can use any of the available image encoders (such as JPEG, PNG, or JPEG2000) in your system. To enable easy integration with the client application, it is important that all codecs with

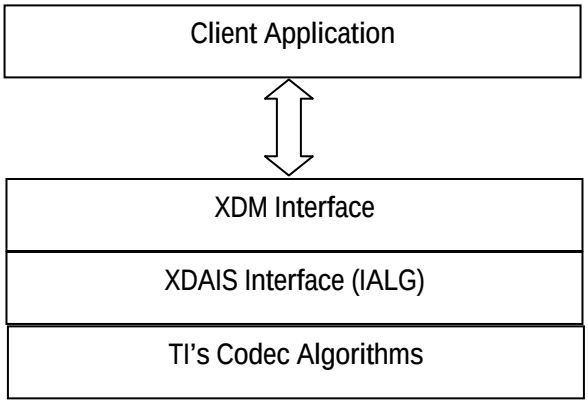
similar functionality use similar APIs. XDM was primarily defined as an extension to XDAIS to ensure uniformity across different classes of codecs (for example audio, video, image, and speech). The XDM standard defines the following two APIs:

- `control()`
- `process()`

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

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

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 encoder, then you can easily replace JPEG with another XDM-compliant image encoder with minimal changes to the client application.

For more details, see *XDAIS-DM (Digital Media) User Guide* (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 the 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 components include 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).

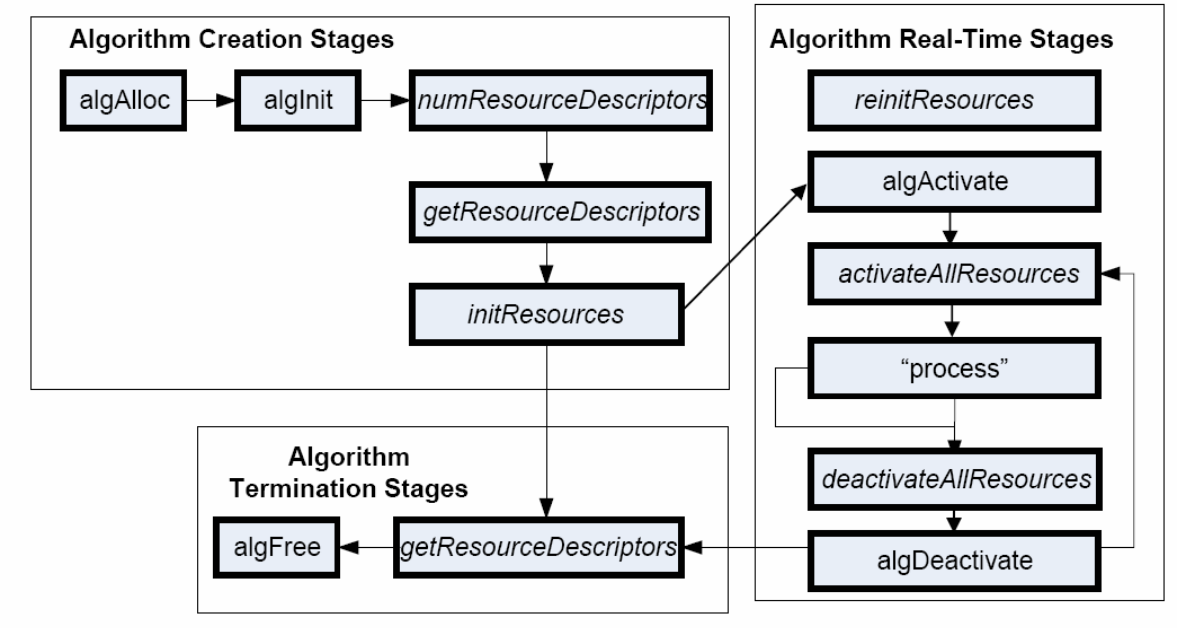


Figure 1-1. IRES Interface Definition and Function Calling Sequence.

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

1.2 Overview of JPEG Encoder

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

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

1.3 Supported Services and Features

This user guide accompanies TI's implementation of JPEG Encoder on the DM365 platform. This version of the codec has the following supported features of the standard:

- eXpressDSP Digital Media (XDM 1.0 IIMGENC1) interface and IRES interface compliant
- Supports JPEG baseline DCT encoding process with following limitations:
  - Non-interleaved scans are not supported.
  - Huffman tables and quantization tables for U and V components must be same.
  - No support for user defined Huffman tables. Default Huffman tables are used.
  - No support for number of components other than 3.
- Supports YUV 4:2:0/4:2:2 planar and YUV 4:2:2 interleaved data as input
- Supports YUV 4:2:0 semi-planar (NV12 format, that is,Y planar, Cb Cr interleaved) data as input
- Supports YUV4:2:2 and YUV4:2:0 encoded format
- Supports arbitrary image width and height (minimum width and height requirement of 97 and 16 pixels, respectively)
- Supports insertion of Application Maker and Comment Marker by test application
- Images with resolutions up to (Horizontal MCU size * 1024)*(Vertical MCU size * 1024) pixels can be encoded. This is the theoretical maximum; however, only images up to 10 Mpixels have been tested. If the codec memory and I/O buffer requirements exceed the DDR memory availability for frame based encoding, use ring buffer and slice mode encoding to encode higher resolution images.
- Supports restart interval
- Quantization tables are fixed with a quality factor (2 – 97) adjusting the quantization level

- Supports ring buffer configuration of bit-stream buffer for reducing buffer size requirement
- Supports rotation by 90, 180, and 270 degree
- Supports frame based encoding
- Supports slice mode encoding
- Supports frame level reentrancy
- Supports multi-instance of JPEG Encoder, and single/multi instance of JPEG Encoder with other DM365 codecs

1.4 Limitations

The following are the limitations:

- Does not support extended DCT based encoding process.
- Does not support loss-less encoding process
- Does not support hierarchical encoding process
- Does not support progressive scan
- Minimum image width and height requirement is 97 and 16 pixels, respectively
- Huffman tables are fixed by algorithm
- No support for number of components other than 3
- Ring buffer size should be multiples of 4096 bytes

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.

Topic **Page**

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_E_01_00, under which another directory named jpeg_encoder is created.

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

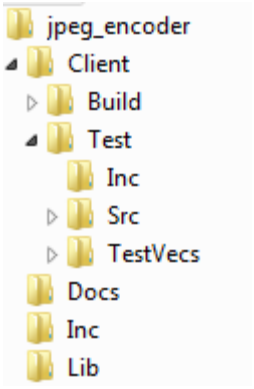


Figure 2-1. Component Directory Structure

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

Table 2-1. Component Directories

Sub-Directory	Description
jpeg_encoder/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_encoder /Docs	Contains user guide, datasheet, and release notes
jpeg encoder/Client/Test/Src	Contains application C files, makefile, and configuration file. Executable will be built in this folder.
jpeg_encoder/Client/Test/Inc	Contains header files needed for the application code
jpeg_encoder/Client/Test/TestVecs	Contains test vectors and configuration files
jpeg_encoder/Lib	Contains JPEG Encoder and other support libraries
jpeg_encoder/Inc	Contains JPEG Encoder 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 jpge365.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 jpegencTestApp.mak**" at the prompt and this generates an executable file, jpegencApp.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 encoder 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 will take input YUV files and dumps output JPEG files as specified in the in the command line arguments. To build and run the sample test application in Linux, follow these steps:

1) Verify that you have installed Framework Component (FC), XDC and LSP. For information about the versions, see Release Note.

2) Verify that libjpgenc.a library is built and present in jpeg_encoder/Lib directory.

3) Verify that support libraries (libimx.a) are present in jpeg_encoder/Lib directory.

4) Change directory to jpeg_encoder/Client/Test/Src and type `make clean` command followed by a `make` command. This will use the Makefile in that directory to build the test executable jpgenc-r in the jpeg_encoder/Client/Test/Src directory.

> **Note**:
>
> ARM tool chain that is arm_v5t_le-gcc (ARM gcc) compiler path needs to be set in user's environment path before building the JPEG Encoder executable.

5) To run the jpgenc-r executable on DM365 EVM board:

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

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

   c) 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.

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

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

   f) Change the directory to Client/Test/Src folder and execute the following command to run the JPEG encoder executable:
   $./jpgenc-r

This will run the JPEG encoder with base parameters. To run the JPEG Encoder with extended parameters execute the following command:
$./jpgenc-r –ext

2.7 Configuration Files

This codec is shipped along with:

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

2.7.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

```
X
Config
Input
Output
```

where:

- `X` must be set to 0 - for output dumping.
- `Config` is the Encoder configuration file. For details, see Sections 2.7.2 and 2.7.3.
- `Input` is the input file name (use complete path).
- `Output` is the output file name (use complete path).

A sample Testvecs.cfg file is as shown.

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

2.7.2 Encoder Configuration File For Base Parameters

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

A sample Testparams.cfg file is as shown.

```
# <ParameterName> = <ParameterValue> # Comment
#
################################################################
##############################
# Parameters
################################################################
##############################
maxHeight = 480
maxWidth  = 720
maxScans  = 15
dataEndianness = 1
forceChromaFormat = 2
inputChromaFormat = 4 #4:422ILE,9:420SEMI_PLANAR,
inputWidth = 720
inputHeight = 480
captureWidth = 720
numAU = 0
genHeader = 0
qValue = 97
```

2.7.3 Encoder Configuration File For Extended Parameters

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

A sample Testparams.cfg file is as shown.

```
# <ParameterName> = <ParameterValue> # Comment
#
################################################################
# Parameters
################################################################
maxHeight = 480
maxWidth  = 720
maxScans  = 15
dataEndianness = 1
forceChromaFormat = 2
inputChromaFormat = 4 #4:422ILE,9:420SEMI_PLANAR,
inputWidth = 720
inputHeight = 480
captureWidth = 720
numAU = 0
genHeader = 0
qValue = 97
rstInterval = 84
rotation = 0
disableEOI = 0
```

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 Encoder Client Interfacing Constraints

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

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

Channel Number	Associated PaRamSet Numbers
17 Channels(63 Compulsory)	All Param sets associated with TCCs and 23 additonal param sets .

Apart from these 16 TCCs requirements, it also needs 23 more PaRamSets that are allocated through the IRES Interface.

2) Channel mapping into Queue and EDMA Shadow regions setting is done by codec .

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. The `algActivate()` and `algDeactivate()` calls made by client application that are implemented by the codecs perform context save/restore to allow multiple instances of same codec and/or different codec combinations.

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

5) If multiple codecs are running with frame level re-entrancy, the client application has to perform time multiplexing of process calls of different codecs to meet desired timing requirements between video/image frames.

6) ARM and DDR clock must be set to the 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 to completion before another process call can be invoked.

3.2 Overview of the Test Application

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

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

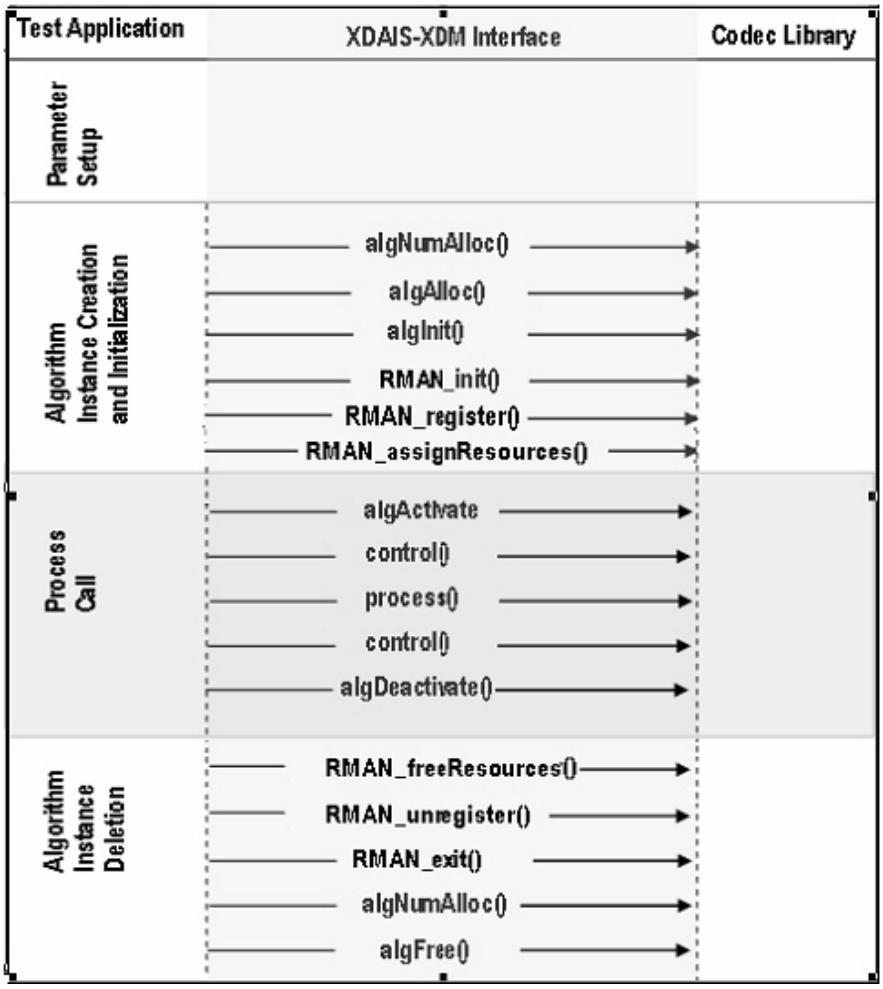


Figure 3-2. Test Application Sample Implementation.

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 command line.

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

1) Reads the configuration parameters from the command line

2) Sets the `IIMGENC1_Params` structure based on the values it read.

3) Reads the input YUV image into the application input buffer

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

3.2.2 Algorithm Instance Creation and Initialization

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

1) `algNumAlloc()` - To query the algorithm about the number of memory records it requires.

2) `algAlloc()` - To query the algorithm about the memory requirement to be filled in the memory records.

3) `algInit()` - To initialize the algorithm with the memory structures provided by the application.

After algorithm instance is successfully created, 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.

3.2.3 Process Call in Single Instance Scenario

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

1) Calls `algActivate()`, which initializes the encoder state and some hardware memories and registers.

2) Sets the input and output buffer descriptors required for the `process()` function call.

3) Calls the `process()` function to encode a single frame of data. The inputs to the process function are input and output buffer descriptors, and a pointer to the `IIMGENC1_InArgs` and `IIMGENC1_OutArgs` structures. The `process()` function should be called multiple times to encode multiple images.

4) Call `algDeactivate()`, which performs releasing of hardware resources and saving of encoder instance values.

5) `process`() is made a blocking call, but an internal OS specific layer enables the process to be pending on a semaphore while hardware performs a complete JPEG encode.

6) Other specific details of the `process()` function remains the same as described in this section and the constraints described in section 3.2.1 are applicable.

Note:

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

3.2.4 Algorithm Instance Deletion

After successful execution of algorithm 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.

Next, 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.

3.2.5 Usage in Multiple Instance Scenario

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

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

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

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

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

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

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

3.2.6 Process Call with algActivate and algDeactivate

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

1) Sets the input and output buffer descriptors required for the `process()` function call.
2) Calls `algActivate()`, which initializes the encoder state and some hardware memories and registers.
3) Calls the `process()` function to encode a single frame of data. The inputs to the process function are input and output buffer descriptors, pointer to the `IIMGENC1_InArgs` and `IIMGENC1_OutArgs` structures.

4) Call `algDeactivate()`, which performs releasing of hardware resources and saving of encoder instance values.

Other specific details of the `process()` function remains same as described in section 3.2.3 and constraints described in section 3.2.1 are applicable.

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.

Topic **Page**

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	Value	Description or Evaluation
XDM_DataFormat	XDM_BYTE	1	Big endian stream.
	XDM_LE_16	2	16-bit little endian stream. Not supported in this version of JPEG Encoder
	XDM_LE_32	3	32-bit little endian stream. This is not supported
XDM_ChromaFormat	XDM_CHROMA_NA	-1	Chroma format not applicable. This is not supported
	XDM_YUV_420P	1	YUV 4:2:0 planar.
	XDM_YUV_422P	2	YUV 4:2:2 planar.
	XDM_YUV_422IBE	3	YUV 4:2:2 interleaved (big endian). Not supported in this version of JPEG Encoder.
	XDM_YUV_422ILE	4	YUV 4:2:2 interleaved (little endian). Default choice for input color format.
	XDM_YUV_444P	5	YUV 4:4:4 planar. Not supported in this version of JPEG Encoder
	XDM_YUV_411P	6	YUV 4:1:1 planar. Not supported in this version of JPEG Encoder
	XDM_GRAY	7	Gray format. Not supported in this version of JPEG Encoder
	XDM_RGB	8	RGB color format. Not supported in this version of JPEG Encoder
	XDM_YUV_420SP	9	YUV 4:2:0 semi planar
	XDM_CHROMAFORMAT_DEFAULT	4	Default chroma format value set to XDM_YUV_422ILE

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
XDM_CmdId	XDM_GETSTATUS	0	Query algorithm instance to fill Status structure
	XDM_SETPARAMS	1	Set run-time dynamic parameters through the DynamicParams structure
	XDM_RESET	2	Reset the algorithm. Not supported in this version of JPEG Encoder
	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. Not supported in this version of JPEG Encoder…
	XDM_GETBUFINFO	5	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	6	Query the algorithm's version. The result will be returned in the data field of the respective _Status structure. This control command is currently not supported.
XDM_EncMode	XDM_ENCODE_AU	0	Encode entire access unit. Default value.
	XDM_GENERATE_HEADER	1	Encode only header.
	JPEGENC_TI_ENCODE_AU_NOHEADER	2	Encode raw data only (no header)
XDM_ErrorBit	XDM_APPLIEDCONCEALMENT	2^9	Bit 9 ❑ 1 - Applied concealment ❑ 0 – Ignore
	XDM_INSUFFICIENTDATA	2^10	Bit 10 ❑ 1 - Insufficient data ❑ 0 – Ignore
	XDM_CORRUPTEDDATA	2^11	Bit 11 ❑ 1 - Data problem/corruption ❑ 0 – Ignore

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	XDM_CORRUPTEDHEADER	2^12	Bit 12 ❑ 1 - Header problem/corruption ❑ 0 – Ignore
	XDM_UNSUPPORTEDINPUT	2^13	Bit 13 ❑ 1 - Unsupported feature/parameter in input ❑ 0 – Ignore
	XDM_UNSUPPORTEDPARAM	2^14	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 – Ignore
	XDM_FATALERROR	2^15	Bit 15 ❑ 1 - Fatal error (stop encoding) ❑ 0 - Recoverable error

Note:

- The remaining bits that are not mentioned in `XDM_ErrorBit` are used by codec for reporting extended errors. See the `DM365_JPEGENC_ERROR` structure in section 4.2.2.6 for more details.
- The algorithm can set multiple bits to 1, depending on the error condition.

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`
- `IIMGENC1_Fxns`
- `IIMGENC1_Params`
- `IIMGENC1_DynamicParams`
- `IIMGENC1_InArgs`
- `IIMGENC1_Status`
- `IIMGENC1_OutArgs`

4.2.1.1 XDM1_BufDesc

Description

This structure defines the buffer descriptor for input and output buffers in XDM 1.0.

Fields

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

4.2.1.2 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 a buffer address
`bufSize`	`XDAS_Int32`	Input	Size of `buf` in 8-bit bytes
`accessMask`	`XDAS_Int32`	Input	Mask filled by the algorithm, declaring how the buffer was accessed by the algorithm process

4.2.1.3 XDM_AlgBufInfo

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`minNumInBufs`	`XDAS_Int32`	Output	Number of input buffers: ❑ 3 for Planar Formats ❑ 2 for 420SemiPlanr ❑ 1 for 422ILE Format
`minNumOutBufs`	`XDAS_Int32`	Output	Number of output buffers 1
`minInBufSize[XDM_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output Input	Size in bytes required for each input buffer For 422P: ❑ (Width*Height) for Buffer0 ❑ (Width* Height) /2 for Buffer1 ❑ (Width* Height)/2 for Buffer2 For 422ILE: ❑ 2*(Width*Height) for Buffer 0 For 420P: ❑ (Width* Height) for Buffer0 ❑ (Width*Height)/4 for Buffer 1 ❑ (Width* Height)/4 for Buffer2 For 420SemiPlanar: ❑ (Width*Height) for Buffer0 ❑ (Width*Height)/2 for Buffer1
`minOutBufSize[XDM`	`XDAS_Int32`	Output	Size in bytes required for each output buffer

Field	Datatype	Input/ Output	Description
_MAX_IO_BUFFERS]			3*(Width*Height)/2 and 2*(width*Height) for YUV 420 planar and YUV 422 planar encoded format (forceChromaFormat) respectively

Note:

See the *JPEG Encoder Data Sheet* for I/O buffer details.

4.2.1.4 IIMGENC1_Fxns

Description

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

Fields

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

4.2.1.5 IIMGENC1_Params

Description

This structure defines the creation parameters 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. Default: 24
maxHeight	XDAS_Int32	Input	Maximum image height to be supported in pixels Default: 960 Supported value: 16 to 16384.

Field	Datatype	Input/ Output	Description
maxWidth	XDAS_Int32	Input	Maximum image width to be supported in pixels Default: 1280 Supported value: 97 to 16384.
maxScans	XDAS_Int32	Input	Maximum number of scans not supported for sequential encoder.
dataEndianness	XDAS_Int32	Input	Endianness of input data. Only XDM_BYTE supported.
forceChromaFormat	XDAS_Int32	Input	Force encoding in given Chroma format. Only XDM_YUV_420P (Default), and XDM_YUV_422P are supported.

4.2.1.6 IIMGENC1_DynamicParams

‖ Description

This structure defines the run-time parameters 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. Default: 32
numAU	XDAS_Int32	Input	Number of Access unit to encode. Set to XDM_DEFAULT to encode the entire frame Default:0
InputChromaFormat	XDAS_Int32	Input	Input chroma format. Only XDM_YUV_420P, XDM_YUV_422P, XDM_YUV_422ILE (default) and XDM_YUV_420SP formats supported
inputHeight	XDAS_Int32	Input	Height of input frame in pixels. Default: 480 Supported value: 16 to 16384.
inputWidth	XDAS_Int32	Input	Width of input frame in pixels. Default: 720 Supported value: 97 to 16384.
captureWidth	XDAS_Int32	Input	If the field is set to: ❑ 0 - Encoded image width is used as pitch. ❑ Any non-zero value, capture width is used as pitch (capture width should be >= image width). Default :0
generateHeader	XDAS_Int32	Input	Encode entire access unit or only header. See XDM_EncMode enumeration for details. Default :0

Field	Datatype	Input/ Output	Description
Qvalue	XDAS_Int32	Input	Q value Quality factor for encoder (2: Lowest quality, 97 Highest quality). If set to 0, default of 95 will be used.

4.2.1.7 IIMGENC1_InArgs

Description

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

Fields

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

4.2.1.8 IIMGENC1_OutArgs

Description

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

Fields

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

4.2.1.9 IIMGENC1_Status

Description

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

Fields

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

4.2.2 JPEG Encoder Data Structures

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

- `IJPEGENC_Params`
- `IJPEGENC_DynamicParams`
- `IJPEGENC_Status`
- `IJPEGENC_InArgs`
- `IJPEGENC_OutArgs`
- `DM365_JPEGENC_ERROR`

4.2.2.1 IJPEGENC_Params

‖ Description

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

‖ Fields

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

4.2.2.2 IJPEGENC _DynamicParams

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`imgencDynamicParams`	`IIMGENC1_DynamicParams`	Input	Base run-time parameters. See `IIMGENC1_DynamicParams` data structure for details
`rstInterval`	`Uint16`	Input	Restart interval in number of MCUs, must be > 3 Default value: 84.
`disableEOI`	`Uint16`	Input	❑ `XDM_DEFAULT`: EOI insertion enabled. ❑ 1: EOI insertion disabled ❑ Default :0
`rotation`	`Uint16`	Input	Specify degree of anticlockwise rotation. Can be 0 `(DEFAULT )`,90, 180, 270

`customQ`	`IJPEGENCQtab`	Input	Structure for user -defined quantization table.
			`{`
			`Uint8 luma[64];`
			`Uint8 chroma[64];`
			`}IJPEGENCQtab;`
			Set to NULL, if not used.
			Default :NULL

4.2.2.3 IJPEGENC_Status

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`imgencStatus`	`IIMGENC1_Status`	Output	Base status parameters. See `IIMGENC1_Status` data structure for details

Field	Datatype	Input/ Output	Description
bytesGenerated	XDAS_Int32	Output	Number of bytes generated by last call to JPEG process function.
numAU	XDAS_Int32	Output	Number of MCUs within a slice, recomputed by the JPEG encoder so it is in accordance with the constraint numAU % (2*IMGWIDTH/MCU_WIDTH)= 0
nextFreeCmdptr	XDAS_Uint16 *	Output	Reserved
nextFreeImBufPtr	XDAS_Uint8 *	Output	Reserved
nextFreeCoefBufPtr	XDAS_Uint8 *	Output	Reserved

4.2.2.4 IJPEGENC_InArgs

Description

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

Fields

Field	Datatype	Input/ Output	Description
imgencInArgs	IIMGENC1_InArgs	Input	Base input run-time parameters. See IIMGENC1_InArgs data structure for details
ringBufStart	XDAS_UInt8 *	Input	Pointer to starting point of bit-stream ring buffer
ringBufSize	XDAS_Uint32	Input	Size of ring buffer in bytes
sliceNum	XDAS_Int16	Input	Slice number. -1 if it is the last slice. Only effective when slice based encoding is enabled.
appDataType	unsigned short	Input	Bit 0 – 15 corresponding to APP0 – APP15 (if bit at position 'x' is 1, insert APPx) Default: 0
appDataLength[16]	unsigned short	Input	Element 'x' in array will specify the length of segment excluding Marker start code (2 bytes) and length (2 bytes) for APPx ❑ >0 (bytes): When bit 'x' of AppDataType = 1 ❑ =0: When bit 'x' of AppDataType = 0
insertCommentMarker	unsigned short	Input	❑ 0 = Do not insert comment marker (default) ❑ 1 = Insert comment marker

Field	Datatype	Input/ Output	Description
commentSegLen	unsigned short	Input	Length of comment segment excluding Marker start code (2 bytes) and length (2bytes) ❑ >0 (bytes): When InsertAppData = 1 ❑ =0: When InsertAppData = 0 (default)

4.2.2.5 IJPEGENC_OutArgs

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
imgencOutArgs	IIMGENC1_OutArgs	Output	Base input run-time parameters. See IIMGENC1_InArgs data structure for details
curInPtrY	XDAS_Uint8*	Output	Current input pointer, pointing to YUV interleaved data for YUV422 interleaved input or Y data for planar input
curInPtrU	XDAS_Uint8*	Output	Current input pointer, pointing to U data for planar input
curInPtrV	XDAS_Uint8*	Output	Current input pointer, pointing to V data for planar input
curOutPtr	XDAS_Uint8*	Output	Current output pointer, pointing to bit-stream
offsetAppData	unsigned int	Output	>=0 (bytes), Valid offset value when InsertAppData = 1
offsetCommentData	unsigned int	Output	>=0 (bytes), Valid offset value when InsertCommentMarker = 1

4.2.2.6 DM365_JPEGENC_ERROR

JPEG encoder supports this enumeration structure to report errors in creation time and run-time parameters.

Error	Value (bit 0 to 7 of error code)	Description
DM365_JPEGENC_INVALID_MAXWIDTH	1	Error in max width parameter
DM365_JPEGENC_INVALID_MAXHEIGHT	2	Error in max height parameter
DM365_JPEGENC_INVALID_MAXSCANS	3	Error in max scans parameter
DM365_JPEGENC_INVALID_DATAENDIANNESS	4	Error in data endianness parameter
DM365_JPEGENC_INVALID_FORCECHROMAFORMA	5	Error in chroma format parameter
DM365_JPEGENC_INVALID_NUMAU	6	Error in numAU parameter
DM365_JPEGENC_INVALID_INPUTCHROMAFORMAT	7	Error in input chroma parameter
DM365_JPEGENC_INVALID_INPUTHEIGHT	8	Error in input height parameter
DM365_JPEGENC_INVALID_INPUTWIDTH	9	Error in input width parameter
DM365_JPEGENC_INVALID_CAPTUREWIDTH	10	Error in capture width parameter
DM365_JPEGENC_INVALID_GENERATEHEADER	11	Error in generate header parameter
DM365_JPEGENC_INVALID_QVALUE	12	Error in q value parameter
DM365_JPEGENC_INVALID_RSTINTERVAL	13	Error in reset interval parameter
DM365_JPEGENC_INVALID_ROTATION	14	Error in rotation parameter
DM365_JPEGENC_INVALID_DISABLEEOI	15	Error in disable EOI parameter

4.3 Interface Functions

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

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

You must call these APIs in the following sequence:

1) `algNumAlloc()`
2) `algAlloc()`
3) `algInit()`
4) `algActivate()`
5) `process()`
6) `algDeactivate()`
7) `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* (literature number SPRU360).

4.3.1 Creation APIs

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

> **Note**:
>
> See the *JPEG Encoder 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* (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 Params structure and creating an instance with base parameters.

```
{
      ……….
      ……….
    IIMGENC1 Params      params;

    // Set the create time base parameters
    params.size = sizeof(IIMGENC1 Params);
```

```
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM DEFAULT;
    params.dataEndianness = XDM BYTE;
    params.forceChromaFormat= XDM YUV 420P;
   handle = (IALG Handle) ALG create((IALG Fxns *)&
JPEGENC TI IJPEGENC,

(IALG Handle) NULL,
                                                    (IALG Params
*) &params)
……….
……….
}
```

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

```
{
      ……..
      ……..

    IIMGENC1 Params                  params;
    IJPEGENC Params                  extParams;

    // Set the create time base parameters
    params.size = sizeof(IIMGENC1 Params);
    params.maxHeight = 480;
    params.maxWidth = 720;
    params.maxScans= XDM DEFAULT;
    params.dataEndianness = XDM BYTE;
    params.forceChromaFormat= XDM YUV 420P;

    // Set the create time extended parameters

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

handle = (IALG Handle) ALG create((IALG Fxns *)&
JPEGENC TI IJPEGENC,

(IALG Handle) NULL,

(IALG Params *) &extParams)

      ……..
      ……..

}
```

‖ See Also

`algAlloc(), algMoved()`

4.3.3 Control processing API

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

‖ Name

`control()` – control call

‖ Synopsis

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

‖ Arguments

```
IIMGENC1_Handle handle; /* algorithm instance handle */

IIMGENC1_Cmd id; /* id of command */

IIMGENC1_DynamicParams *params; /* pointer to dynamic
parameters */

IIMGENC1_Status *status /* pointer to status structure */
```

‖ Return Value

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

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

‖ Description

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

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

- `XDM_GETSTATUS`: fill structure `IIMGENC1_Status` whose pointer is passed as 4th argument.
- `XDM_SETPARAMS`: set dynamic params contained in the structure whose pointer is passed as 3rd argument.
- `XDM_RESET`: reset the encoder so next time `process()` is called, a new bit-stream is encoded.
- `XDM_SETDEFAULT`: set the dynamic params to the following default values:
  - `XDM_GETBUFINFO`: get required number of I/O buffers and their sizes. Results are returned in the `bufInfo` member of the structure `IIMGENC1_Status` whose pointer is passed as 4th argument.

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

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

‖ Post conditions

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

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

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

```
                    {
      ……
      ……

    IIMGENC1 DynamicParams      dynParams;
    IIMGENC1 Status             status;
    IJPEGENC DynamicParams      extDynParams;
      ……
      ……
    // Set the dynamic base parameters
    dynParams.size = sizeof(IJPEGENC DynamicParams);
    dynParams.numAU= XDM DEFAULT;
    dynParams.inputChromaFormat = XDM YUV 422ILE;
    dynParams.inputHeight = 480;
    dynParams.inputWidth = 720;
    dynParams.captureWidth = 720;
    dynParams.generateHeader = XDM DEFAULT;
    dynParams.qValue = XDM DEFAULT;

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

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


/* Set Dynamic Params */
retVal=iimgEncfxns->control(  (IIMGENC1 Handle)handle,
XDM SETPARAMS,
                              (IIMGENC1 DynamicParams *)&
extDynParams,
                              (IIMGENC1 Status *)&status);
      ……
      ……

                      }
```

‖ See Also

`algInit()`, `algActivate()`, `process()`

4.3.4 Data Processing API

Data processing API is used for processing the input data.

‖ Name

`process()` – basic encoding

‖ Synopsis

```
XDAS_Int32 (*process)(IIMGENC1_Handle handle, XDM1_BufDesc
*inBufs, XDM1_BufDesc *outBufs, IIMGENC1_InArgs *inargs,
IIMGENC1_OutArgs *outargs);
```

‖ Arguments

```
IIMGENC1_Handle handle; /* algorithm instance handle */

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

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

IIMGENC1_InArgs *inargs /* algorithm run-time input
arguments */

IIMGENC1_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 `IIMGENC1_InArgs` data structure that defines the run-time input arguments for an algorithm instance object.

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

> **Note**:
>
> If you are using extended data structures, the fourth and fifth arguments must be pointers to the extended `InArgs` and `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.

‖ Post conditions

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

- If the process operation is successful, the return value from this operation is equal to `IALG_EOK;` otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.
- After successful return from `process()` function, `algDeactivate()` can be called.

‖ Example

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

‖ See Also

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

4.3.5 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* (literature number SPRU360).

‖ See Also

`algAlloc()`

This page is intentionally left blank

Appendix A

Feature Descriptions

This appendix describes special features not commonly found in a standard JPEG encoder such as ring-buffer configuration of the output bit-stream buffer, slice-mode processing, more than one input color format, and rotation.

A.1 Bit-stream Ring Buffer in DDR

To minimize the output buffer memory requirement, the JPEG encoder stores the JPEG bit-stream into a circular or ring buffer residing in DDR, which acts as an intermediary storage area between the final storage media (SD card, HD, memory stick, etc.) and the encoder. Thus, the size of the ring buffer can be much smaller than the final bit-stream's size, effectively reducing the amount of physical DDR memory allocated for storing the bit-stream. The complete bit-stream is eventually stored on the media because, as JPEG fills one half of the ring buffer, the application empties the other half onto the media. The JPEG encoder and the application operate in parallel and on a different half, thus sustaining the maximum JPEG processing throughput. Figures A.1, A.2, A.3, and A.4 depict the state of the ring buffer at different states of JPEG encode processing:

Figure A-1. Ring Buffer Before JPEG Encoder Starts.

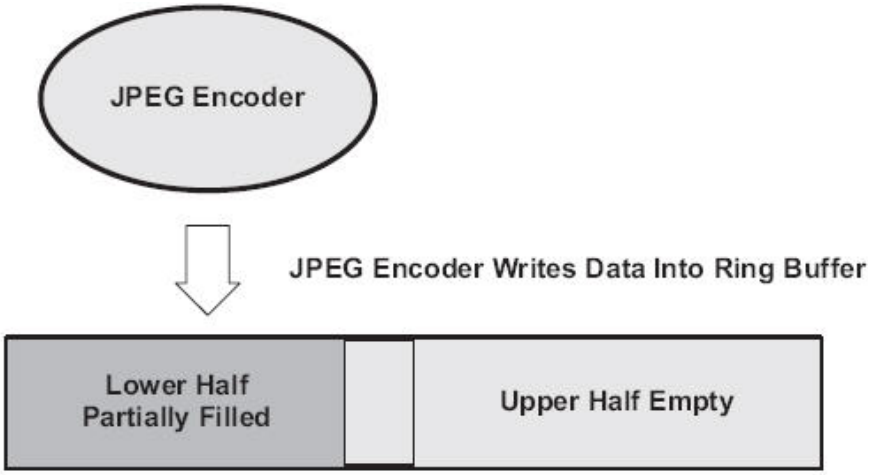


Figure A-2. Ring Buffer Shortly After JPEG Encoder Starts

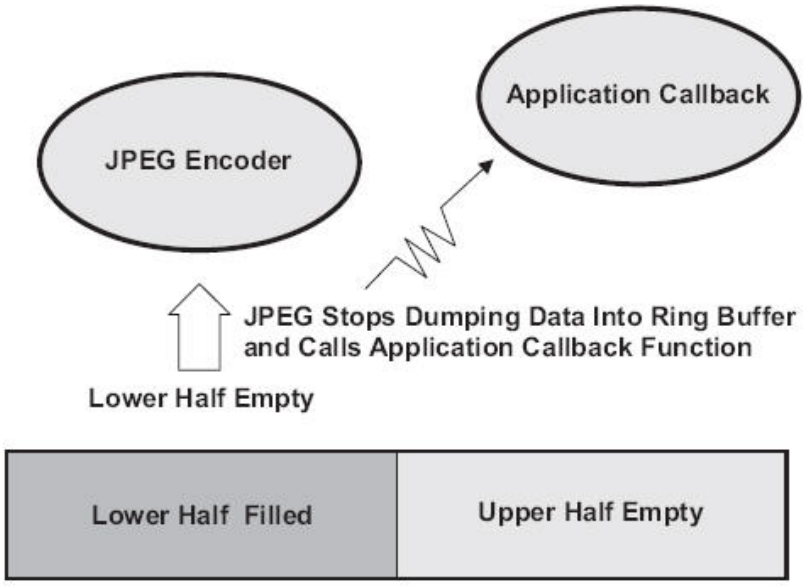


Figure A-3. Ring Buffer Once JPEG Encoder Fills Lower Half

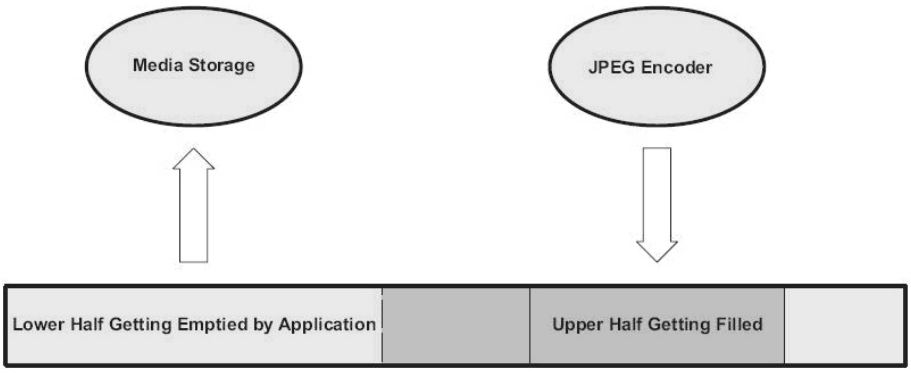


Figure A-4. Ring Buffer Once Application Fills First Half and JPEG Encoder Starts Processing Second Half.

A.1.1 Operation Mode

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

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

The input argument `curBufPtr` is passed by the encoder and its value is the pointer to the first free byte in the ring buffer. All the bytes located before `curBufPtr` are valid bytes output by the encoder and that need to be saved into the storage media. The callback function must save

curBufPtr. When it is called the next time, it knows where to save the data from. However, the first time it is called is a special case, as the starting point of the valid data is the starting address of the ring buffer.

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 a multiple of 4096 bytes.

A.1.3 Guidelines for Using Ring Buffer With JPEG Encoder

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

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

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

                    } Ring2Media;
```

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

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

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

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

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

The callback function that handles half-buffer can accept a second argument in addition to curBufPtr . Use this feature by passing the

pointer to `ring2meida` to the callback function each time the encoder calls it.

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

```
extn_params of type IJPEGENC_Params.
extn_params.halfBufCB = (XDAS_Int32
(*)())JPEGENC_TI_DM365_HalfBufCB;
extn_params.halfBufCBarg= (void*)&ring2media;
```

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

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

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

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

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

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

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

For Application Marker and Comment Marker support, you need to pass one extra parameter in `JPEGENC_TI_DM365_HalfBufCB` function.This flag set to 1 for application data and comment data transfer to media. If marker support is not required, it needs to be set to zero. After the exit of the last call back function, the value of `ring2meida->ringCurPtr` should be the same value as `bufPtr`.

```
XDAS_Void JPEGENC_TI_DM365_HalfBufCB(XDAS_Int32 bufPtr, void
*arg, unsigned int Mflag)
  {
       Uint32 i, x, y, numToXfer;
       Ring2Media *ring2media= arg;

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

       if ((XDAS_Int8*)bufPtr > ring2media ->ringCurPtr){
         numToXfer= (XDAS_Int8*)bufPtr- ring2media -
>ringCurPtr;
         memcpy(ring2media ->ringCurPtr, ring2media -
>mediaPtr, numToXfer);
        ring2media ->mediaPtr+= numToXfer;
if(Mflag == 1){
              ring2media->ringCurPtr = ring2media-
>ringStartPtr;
         }
         else{
              ring2media->ringCurPtr += numToXfer;
         }

       }

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

       return;
  }
```

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

A.2 Slice-mode Processing

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

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

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

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 the image width, multiplied 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.

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

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

A.2.3 Operating Slice-mode Processing Using JPEG APIs

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

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

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

The process API is then called as many times as there are slices in the image. After the first slice is encoded, header insertion must be disabled by a call to the control API. Also the parameter sliceNum in structure IJPEGENC_InArgs of the process API must be incremented each time process is called, otherwise, restart markers (0xFFD0, 0xFFD1, .., 0xFFD7) are not ordered correctly inside the bit-stream.

The process API returns the current position of the input and output pointers in the member curInPtr and curOutPtr of the IJPEGENC_OutArgs structure. These values can be used to correctly initialize the input and output buffer pointers the next time the process API is called. The output buffer pointer will be equal to the currOutPtr value returned by the previous call to process API, which ensures bit-stream continuity.

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

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

A.2.4 Example of Application Code that Operates Slice-mode Encoding

```
// Call get status to get number of total AU
iimgEncfxns->control((IIMGENC1_Handle)handle,
    IJPEGENC_GETSTATUS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams,
(IIMGENC1_Status*)&status);
totalAU= status.imgencStatus.totalAU;
// Set number of MCUs per slice.
extn_dynamicParams.imgencDynamicParams.numAU= totalAU/20;
// Call control function to setup dynamic params
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams,
(IIMGENC1_Status*)&status);
numAU= status.numAU; // Get real numAU computed by codec
// Call to JPEG encode processing, encode 1st slice with
header
ring2media.mediaPtr= media;
ring2media.ringCurPtr= ringbuf;
inArgs.ringBufStart= (XDAS_UInt8*)ringbuf;
inArgs.ringBufSize= RINGBUFSIZE;
inArgs.sliceNum= 0;
retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                          (XDM1_BufDesc  *)&inputBufDesc,
                          (XDM1_BufDesc  *)&outputBufDesc,
                          (IIMGENC1_InArgs *)&inArgs,
                          (IIMGENC1_OutArgs *)&outArgs);
// Disable header insertion
extn_dynamicParams.imgencDynamicParams.generateHeader=
XDM_ENCODE_AU;
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams*)&extn_dynamicParams,
(IIMGENC1_Status*)&status);
bytesGenerated= outArgs.imgencOutArgs.bytesGenerated;
```

```
// Repeat JPEG encoding as many times as necessary until last
slice
for (i=numAU;i<totalAU-numAU;i+= numAU){
      inArgs.sliceNum++;
      inputBufDesc.descs[0].buf     = outArgs.curInPtr;
outputBufDesc.descs[0].bufs     = outArgs.curOutPtr;
      // The line below is actually ignored by codec
      outputBufDesc.bufSizes[0] -=
outArgs.imgencOutArgs.bytesGenerated;
      retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                  (XDM1_BufDesc  *)&inputBufDesc,
                  (XDM1_BufDesc  *)&outputBufDesc,
                  (IIMGENC1_InArgs *)&inArgs,
                  (IIMGENC1_OutArgs *)&outArgs);
      bytesGenerated+= outArgs.imgencOutArgs.bytesGenerated;
}
// For last slice, re-adjust numAU
extn_dynamicParams.imgencDynamicParams.numAU= totalAU-i;
// Call control function to setup dynamic params
iimgEncfxns->control((IIMGENC1_Handle)handle, XDM_SETPARAMS,
(IIMGENC1_DynamicParams *)&extn_dynamicParams,
(IIMGENC1_Status *)&status);
// Call JPEG for the last slice
inArgs.sliceNum= -1;   // -1 means last slice for JPEG
encoder.
inputBufDesc.descs[0]buf     = outArgs.curInPtr;
outputBufDesc.descs[0].buf     = outArgs.curOutPtr;
retVal = iimgEncfxns->process((IIMGENC1_Handle)handle,
                  (XDM1_BufDesc  *)&inputBufDesc,
                  (XDM1_BufDesc  *)&outputBufDesc,
                  (IIMGENC1_InArgs *)&inArgs,
                  (IIMGENC1_OutArgs *)&outArgs);
bytesGenerated+= outArgs.imgencOutArgs.bytesGenerated;
```

A.3 Color Formats

This section provides information about the supported inputs and output color formats.

The following input color formats are supported

- YUV422 interleaved
- YUV420 planar
- 420Semi_Planar and YUV422 planar

Input format is set by initializing the parameter `inputChromaFormat` of the structure `IIMGENC1_DynamicParams` before calling the `control()` API with command `XDM_SETPARAMS`. The symbols `XDM_YUV_422ILE`, `XDM_YUV_420P` or `XDM_YUV_422P` must be used. The value of 420Semi_Planar is 9 . Input color format can be changed before any `process()` API call. When planar format is chosen, the pointers to U and V components must be passed to the encoder through the `XDM1_BufDesc` structure when calling the process() API.

The following output formats are supported:

- yuv420
- yuv422

Output format is set at creation time when calling the `algInit()` API by setting the member `forceChromatFormat` of the structure `IIMGENC1_Params`. Use either the `XDM_YUV_420P` symbol or the `XDM_YUV_422P` symbol to initialize this member.

A.4 Rotation

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

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

- Full frame case when slice based encoding is disabled

  Blocks of 8(Width)x16(Height) pixels are fetched from the top left corner of the input frame, internally rotated to blocks of 16x8 and encoded by the JPEG encoder. The encoding progression in reference to the original image is from top to bottom and right to left. In reference to the encoded image, it is top to bottom, but left to right . The following figure shows this process.

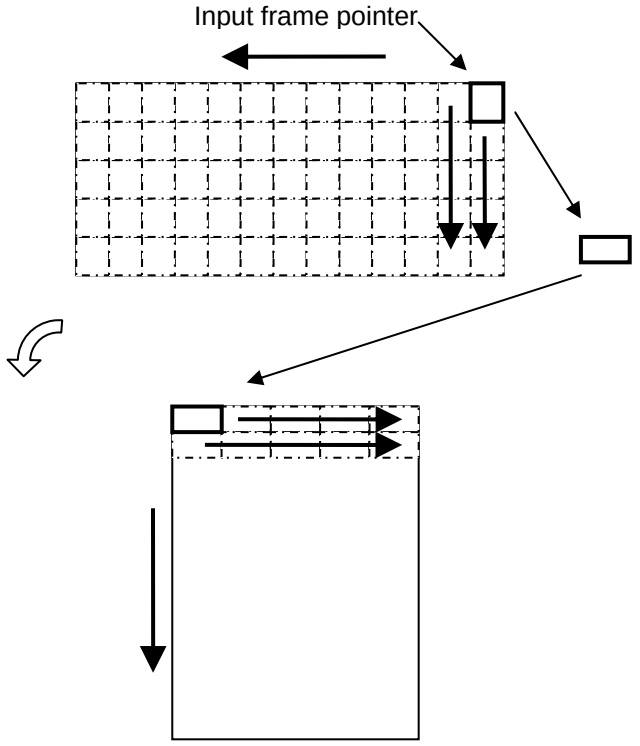


Figure A-5. Rotation Processing Flow for Full Frame

- Case when slice mode encoding is enabled

  The application must feed a vertical band of MCUs to the JPEG encoder. Blocks of 8(Width) x 16(Height) pixels are fetched from the top left corner of the band, internally rotated to blocks of 16x8, and encoded by the JPEG encoder. The encoding progression in reference to the vertical band is from top to bottom and right to left. In reference to the encoded image, it is top to bottom, but left to right.

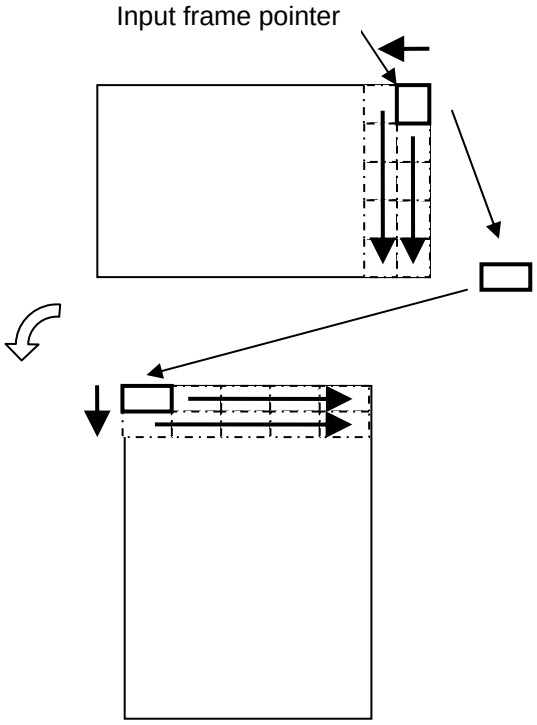


Figure A-6. Rotation Processing Flow for Slice Mode

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

For 90° or 270° rotation, the parameter `numAU` should be set such that it is multiple `of` (H/16) x 2, where H is the height of the original image (before rotation).

If that constraint is not met, the encoder automatically rounds up `numAU` to the next valid value and returns it in the structure `IIMGENC1_Status`. It is then the responsibility of the application to use this corrected `numAU` as the effective slice's size. Hence, it is advisable to read the value of `numAU` from the status structure after the `XDM_SETPARAMS` call and utilize this value of `numAU` for the process call and all further processing.

When rotation is set to 90° or 180°, the input pointer is set to the rightmost block before encoding the first slice.

When rotation is set to 90°, the application should set the input pointer to original Y, U and V buffers before encoding the last slice.

A.5 Application Marker and Comment Marker Insertion

In addition to JPEG Encoder Header, insertion of some extra data such as Application Marker and Comment Marker data is also supported. The marker header insertion is done by codecs, but specific user data will be inserted by the application.

Application needs to set the following parameters of `IJPEGENC_InArgs` structure (in extended mode) before calling the `process()` function.

Parameter Name	Type	Value (Range)
`appDataType`	`unsigned short`	Bit 0 – 15 corresponding to APP0 – APP15 (if bit at position 'x' is 1, insert APPx) By default, APP0 is supported in this codec. Hence, testApp should check these cases ❑ APP0 marker and data will be inserted by default inside codec (exception, client specifying APP0 insertion) ❑ If APP0 is specified by client, then codec will insert only marker and length (data is not inserted) as in other cases (APP1 – APP15, data is not inserted inside the codec).
`appDataLength[16]`	`unsigned short`	Element 'x' in array will specify the length of segment excluding Marker start code (2 bytes) and length (2 bytes) for APPx ❑ >0 (bytes). when bit 'x' of `AppDataType` = 1 ❑ =0, when bit 'x' of `AppDataType` = 0 **Error cases** ❑ >0 but bit 'x' of `AppDataType` = 0, codec will assume that no user data needs to be inserted ❑ =0 but bit 'x' of `AppDataType`, codec will assume that no user data needs to be inserted
`insertCommentMarker`	`unsigned short`	❑ 0 = Do not insert comment marker ❑ 1 = Insert comment marker
`commentSegLen`	`unsigned short`	Length of comment segment excluding Marker start code (2 bytes) and length (2 bytes) ❑ >0 (bytes): when `InsertAppData` = 1 ❑ =0: when `InsertAppData` = 0 **Error cases** ❑ >0 but `InsertCommentMarker` = 0, codec will assume that no user data needs to be inserted ❑ =0 but `InsertCommentMarker` = 1, codec will assume that no user data

Parameter Name	Type	Value (Range)
		needs to be inserted

After the encoding process, codec will return following parameters in `IJPEGENC_OutArgs` structure (in extended mode), which can be used by testApp to insert the data.

Parameter Name	Type	Value (Range)
`offsetAppData`	`unsigned int`	>=0 (bytes), Valid offset value from start of encoded stream, when `InsertAppData` = 1
`offsetCommentData`	`unsigned int`	>=0 (bytes), Valid offset value when `InsertCommentMarker` = 1

Appendix B

Revision History

This revision history highlights the changes made to SPRUEV4A codec specific user guide to make it SPRUEV4B.

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 Encoder on DM365