JPEG Progressive Support Decoder on C64x+

User's Guide

Literature Number: SPRUEA9C
June 2010

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) JPEG Progressive Support Decoder implementation on the C64x+ 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 C64x+ platform.

This document assumes that you are fluent in the C language, have a good working knowledge of Digital Signal Processing (DSP), digital signal processors, and DSP applications. Good knowledge of eXpressDSP Algorithm Interface Standard (XDAIS) and eXpressDSP Digital Media (XDM) standard will be helpful.

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Chapter 5 – Frequently Asked Questions,** answers frequently asked questions related to using JPEG Progressive Support Decoder on C64x+.

- **Appendix A - Sectional Decoding**, describes sectional decoding details for JPEG Progressive Support Decoder on C64x+.
- **Appendix B – Revision History,** highlights the changes made to the SPRUEA9B codec specific user guide to make it SPRUEA9C.

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 Rules and Guidelines* (literature number SPRU352) defines a set of requirements for DSP algorithms that, if followed, allow system integrators to quickly assemble production-quality systems from one or more such algorithms.
- *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.
- *Using the TMS320 DSP Algorithm Standard in a Static DSP System* (literature number SPRA577) describes how an eXpressDSP-compliant algorithm may be used effectively in a static system with limited memory.
- *DMA Guide for eXpressDSP-Compliant Algorithm Producers and Consumers* (literature number SPRA445) describes the DMA architecture specified by the TMS320 DSP Algorithm Standard (XDAIS). It also describes two sets of APIs used for accessing DMA resources: the IDMA2 abstract interface and the ACPY2 library.
- *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8)

The following documents describe TMS320 devices and related support tools:

- *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X* (literature number SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320c64x+ Megamodule* (literature number SPRAA68) describes the enhancements made to the internal memory and describes the new features which have been added to support the internal memory architecture's performance and protection.
- *TMS320C64x+ DSP Megamodule Reference Guide* (literature number SPRU871) describes the C64x+ megamodule peripherals.

- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature number SPRAA84) describes migration from the Texas Instruments TMS320C64x™ digital signal processor (DSP) to the TMS320C64x+™ DSP.
- *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (literature number SPRU187N) explains how to use compiler tools such as compiler, assembly optimizer, standalone simulator, library-build utility, and C++ name demangler.
- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.

Related Documentation

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

- *ISO/IEC IS 10918-1 Information Technology - Digital Compression and Coding of Continuous-Tone Still Images -- Part 1: Requirements and Guidelines* | *CCITT Recommendation T.8*1

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
COFF	Common Object File Format
CSL	Chip Support Library
DCT	Discrete Cosine Transform
DHT	Define Huffman Table
DQT	Define Quantization Table
DRI	Define Restart Interval
DSP	Digital Signal Processing
IJG	Independent JPEG Group
JFIF	JPEG File Interchange Format
JPEG	Joint Photographic Experts Group
MCU	Minimum Coded Unit
QDMA	Quick Direct Memory Access
SOF	Start Of Frame

Abbreviation	Description
SOI	Start Of Image
SOS	Start Of Scan
VLD	Variable Length Decoding
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media

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 Progressive Support Decoder on C64x+) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

Code Composer Studio, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM644x, and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the JPEG Progressive Support Decoder on the C64x+ platform and its supported features.

Topic	Page

1.1 Overview of XDAIS and XDM

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).

1.1.1 XDAIS Overview

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

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

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

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

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

1.1.2 XDM Overview

In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) 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

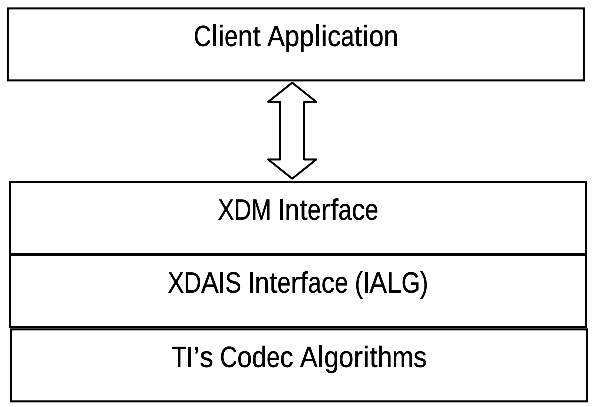
(For example audio, video, image, and speech). The XDM standard defines the following two APIs:

- `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 MPEG4 video decoder, then you can easily replace MPEG4 with another XDM-compliant video decoder, say H.263, with minimal changes to the client application.

For more details, see *eXpressDSP Digital Media (XDM) Standard API Reference* (literature number SPRUEC8).

1.2 Overview of JPEG Progressive Support Decoder

JPEG is an international standard for color image compression. This standard is defined in the ISO 10918-1 JPEG Draft International Standard | CCITT Recommendation T.81.

From this point onwards, all references to JPEG Decoder means JPEG Progressive Support Decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of JPEG Decoder on the C64x+ platform.

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

- Supports baseline sequential mode with both interleaved and non-interleaved input format
- Supports progressive mode
- Supports YUV 444, YUV 422, YUV 420, YUV 411, and Gray scale color sub-sampling formats
- Supports YUV 422 with sampling format ((1,2), (1, 1), (1,1)) for baseline sequential mode with interleaved input format
- Supports RGB16, BGR24, and BGR32 output formats
- Supports a maximum of three components
- Supports a maximum of three quantization tables
- Supports a maximum of four huffman tables each for AC and DC DCT coefficients
- Support arbitrary image size for both sequential and progressive JPEG images
- Supports 8-bit and 16-bit quantization tables
- Does not support source images of 12-bits per sample
- JPEG File Interchange Format (JFIF) header is skipped
- Restart management for bit stream with Define Restart Interval Marker (DRI) and Restart Marker (RST) are enabled

The other explicit features that TI's JPEG Decoder provides are:

- Supports sectional decoding
- Supports sub-region decoding
- Supports up scaling the output image by a factor of 2, 4, and 8
- Supports resizing the output image by a factor of 1/2, 1/4, and 1/8

- Supports on-the-fly resizing with respect to set `maxHeight` and `maxWidth`
- Supports YUV planar or YUV 422 interleaved output format
- Supports frame level decoding of images for sequential mode and scan level decoding for progressive mode
- All the data buffers and tables are placed in the external memory
- eXpressDSP Digital Media (XDM 1.0 IIMGDEC1) compliant

This page is intentionally left blank

Chapter 2

Installation Overview

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

2.1 System Requirements

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

2.1.1 Hardware

This codec has been built and tested on DM6446 EVM with XDS560 JTAG emulator.

This codec works on any of TI's C64x+ based platforms such as DM644x, DM648, DM643x, OMAP35xx and their derivatives.

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 3.3.49.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.14.

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a directory called 100_I_JPEG_D_02_00, under which another directory named C64XPLUS_PS_001 is created. Figure 2-1 shows the sub-directories created in C64XPLUS_PS_001.

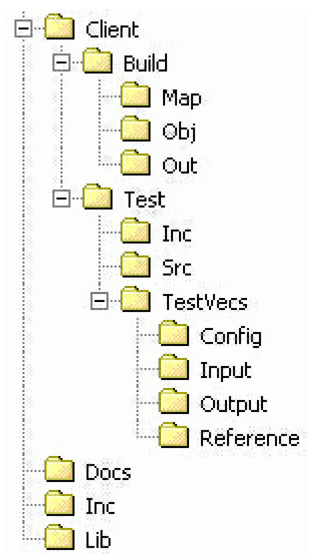


Figure 2-1. Component Directory Structure

Note:

If you are installing an evaluation version of this codec, the directory name will be 100E_I_JPEG_D_02_00.

Table 2-1 provides a description of the sub-directories created in the C64XPLUS_PS_001 directory.

Table 2-1. Component Directories

Sub-Directory	Description
\Inc	Contains XDM related header files which allow interface to the codec library
\Lib	Contains the codec library file
\Docs	Contains user guide and datasheet
\Client\Build	Contains the sample test application project (.pjt) file
\Client\Build\Map	Contains the memory map generated on compilation of the code
\Client\Build\Obj	Contains the intermediate .asm and/or .obj file generated on compilation of the code
\Client\Build\Out	Contains the final application executable (.out) file generated by the sample test application
\Client\Test\Src	Contains application C files
\Client\Test\Inc	Contains header files needed for the application code
\Client\Test\TestVecs\Input	Contains input test vectors
\Client\Test\TestVecs\Output	Contains output generated by the codec
\Client\Test\TestVecs\Reference	Contains read-only reference output to be used for cross-checking against codec output
\Client\Test\TestVecs\Config	Contains configuration parameter files

2.3 Before Building the Sample Test Application

This codec is accompanied by a sample test application. To run the sample test application, you need DSP/BIOS and TI Framework Components (FC).

This version of the codec has been validated with DSP/BIOS version 5.31.02 and Framework Component (FC) version 2.20.01.

2.3.1 Installing DSP/BIOS

You can download DSP/BIOS from the TI external website:

https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html

Install DSP/BIOS at the same location where you have installed Code Composer Studio. For example:

<install directory>\CCStudio_v3.3

The sample test application uses the following DSP/BIOS files:

- Header file, bcache.h available in the <install directory>\CCStudio_v3.3\<bios_directory>\packages\ti\bios\include directory.
- Library file, biosDM420.a64P available in the <install directory>\CCStudio_v3.3\<bios_directory>\packages\ti\bios\lib directory.

2.3.2 Installing Framework Component (FC)

You can download FC from the TI external website:

https://www-a.ti.com/downloads/sds_support/targetcontent/FC/index.html

Extract the FC zip file to the same location where you have installed Code Composer Studio. For example:

<install directory>\CCStudio_v3.3

The test application and the library use the following header files:

- _alg.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\packages\ti\sdo\fc\utils\api directory.
- alg.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\packages\ti\sdo\fc\utils\api directory
- ialg.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\fctools\packages\ti\xdais directory.

- iimgdec1.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\fctools\packages\ti\xdais\dm directory.
- xdas.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\fctools\packages\ti\xdais directory.
- xdm.h available in the <install directory>\CCStudio_v3.3\framework_components_2_20_01\fctools\packages\ti\xdais\dm directory.

2.4 Building and Running the Sample Test Application

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

1) Verify that you have installed TI's Code Composer Studio version 3.3.49 and code generation tools version 6.0.14.

2) Verify that the codec object library jpegdec_ti.l64P exists in the \Lib sub-directory.

3) Open the test application project file, TestAppDecoder.pjt in Code Composer Studio. This file is available in the \Client\Build sub-directory.

4) Select **Project > Build** to build the sample test application. This creates an executable file, TestAppDecoder.out in the \Client\Build\Out sub-directory.

5) Select **File > Load**, browse to the \Client\Build\Out sub-directory, select the codec executable created in step 4, and load it into Code Composer Studio in preparation for execution.

6) Select **Debug > Run** to execute the sample test application.

   The sample test application takes the input files stored in the \Client\Test\TestVecs\Input sub-directory, runs the codec, and uses the reference files stored in the \Client\Test\TestVecs\Reference sub-directory to verify that the codec is functioning as expected.

7) On successful completion, the application displays one of the following messages for each frame:

   - "Decoder compliance test passed/failed" (for compliance check mode)
   - "Decoder output dump completed" (for output dump mode)

2.5 Configuration Files

This codec is shipped along with:

- Generic configuration file (Testvecs.cfg) – specifies input and reference files for the sample test application.
- Decoder configuration file (Testparams.cfg) – specifies the configuration parameters used by the test application to configure the Decoder.

2.5.1 Generic Configuration File

The sample test application shipped along with the codec uses the configuration file, Testvecs.cfg for determining the input and reference files for running the codec and checking for compliance. 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/Reference
```

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file
- `Config` is the Decoder configuration file. For details, see Section 2.5.2.
- `Input` is the input file name (use complete path).
- `Output/Reference` is the output file name (if `X` is 0) or reference file name (if `X` is 1).

A sample Testvecs.cfg file is as shown.

```
1
..\..\Test\TestVecs\Config\Testparams_Inter.cfg
..\..\Test\TestVecs\Input\REMI0003.JPG
..\..\Test\TestVecs\Reference\REMI0003_Interleaved.yuv
0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\REMI0003.JPG
..\..\Test\TestVecs\Output\REMI0003_planar.yuv
```

2.5.2 Decoder Configuration File

The decoder configuration file, Testparams.cfg contains the configuration parameters required for the decoder. The Testparams.cfg file is available in the \Client\Test\TestVecs\Config sub-directory. In addition to the Testparams.cfg, the following config files are included for various input formats as specified:

- Interleaved format - Testparams_Inter.cfg
- Progressive mode - Testparams_Prog.cfg
- Interleaved format and progressive mode - Testparams_Prog_Inter.cfg

A sample Testparams.cfg file is as shown.

```
# New Input File Format is as follows
# <ParameterName> = <ParameterValue> # Comment
###########################################################
# Parameters
###########################################################
ImageWidth = 768       # Image width in Pels, must be
                          multiple of 16
ImageHeight = 512      # Image height in Pels, must be
                           multiple of 16
Scan = 15               # No. of Scan
ChromaFormat = 0        # 0 => JPEGDEC_DEFAULT_CHROMA_
                                FORMAT
                        # 1 => XDM_YUV_420P
                        # 2 => XDM_YUV_422P
                        # 3 => XDM_YUV_422IBE
                        # 4 => XDM_YUV_422ILE
                        # 5 => XDM_YUV_444P
                        # 6 => XDM_YUV_411P
                        # 7 => XDM_GRAY
                        # 8 => XDM_RGB
InputFormat = 0         # 1: Progressive
                        # 0: Sequential
ResizeOption = 0        # 0: No Resize
                        # 1: resize by 1/2
                        # 2: resize by 1/4
                        # 3: resize by 1/8
displayWidth = 0        # displayWidth
RGB_Format = 0          # Select RGB Format
                        # 0: BGR24
                        # 1: BGR32
                        # 2: RGB16
outImgRes = 0           # Select Output Resolution
                        # 0: Even Resolution
                        # 1: Actual Resolution
numAU = 0                # 0=>Default 1,2,3..N
                        # Where N is the maximum number of
                        # MCU's in a Frame
numMCU_row = 0          # 0=>Default 1,2,3..N Where N is
                        # the maximum number of rows in a
                        # Frame
x_org = 0               # 0=>Default start point of the
                        # X-axis in subregion
y_org = 0               # 0=>Default start point of the
                        # Y-axis in subregion
```

```
x_length = 0            # 0=>Default X-length of the
                        # subregion
y_length = 0            # 0=>Default Y-length of the
                        # subregion
alpha_rgb = 0           # 0=> Default value to fill
                        # rgb32
```

Note:

The values remain same as in the Testparams file except for the following parameters:

- `ChromaFormat` = 4 (in Testparams_Inter.cfg file for interleaved format)
- `InputFormat` = 1 and `ChromaFormat` = 0 (in Testparams_Prog.cfg file for planar format)
- `InputFormat` = 1 and `ChromaFormat` = 4 (in Testparams_Prog_Inter.cfg file for interleaved format)

Any field in the `IIMGDEC1_Params` structure (see Section 4.3.1.7) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, ensure to modify the test application appropriately to handle these fields.

2.6 Standards Conformance and User-Defined Inputs

To check the conformance of the codec for the default input file shipped along with the codec, follow the steps as described in Section 2.4.

To check the conformance of the codec for other input files of your choice, follow these steps:

1) Copy the input files to the \Client\Test\TestVecs\Inputs sub-directory.

2) Copy the reference files to the \Client\Test\TestVecs\Reference sub-directory.

3) Edit the configuration file, Testvecs.cfg available in the \Client\Test\TestVecs\Config sub-directory. For details on the format of the Testvecs.cfg file, see Section 2.5.1.

4) Execute the sample test application. On successful completion, the application displays one of the following messages for each frame:

   o "Decoder compliance test passed/failed" (if `X` is 1)

   o "Decoder output dump completed" (if `X` is 0)

If you have chosen the option to write to an output file (`X` is 0), you can use any standard file comparison utility to compare the codec output with the reference output and check for conformance.

2.7 Uninstalling the Component

To uninstall the component, delete the codec directory from your hard disk.

2.8 Evaluation Version

If you are using an evaluation version of this codec, a Texas Instruments logo will be visible in the output.

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.

3.1 Overview of the Test Application

The test application exercises the `IIMGDEC1` base class of the JPEG Decoder library. The main test application files are TestAppDecoder.c and TestAppDecoder.h. These files are available in the \Client\Test\Src and \Client\Test\Inc sub-directories respectively.

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

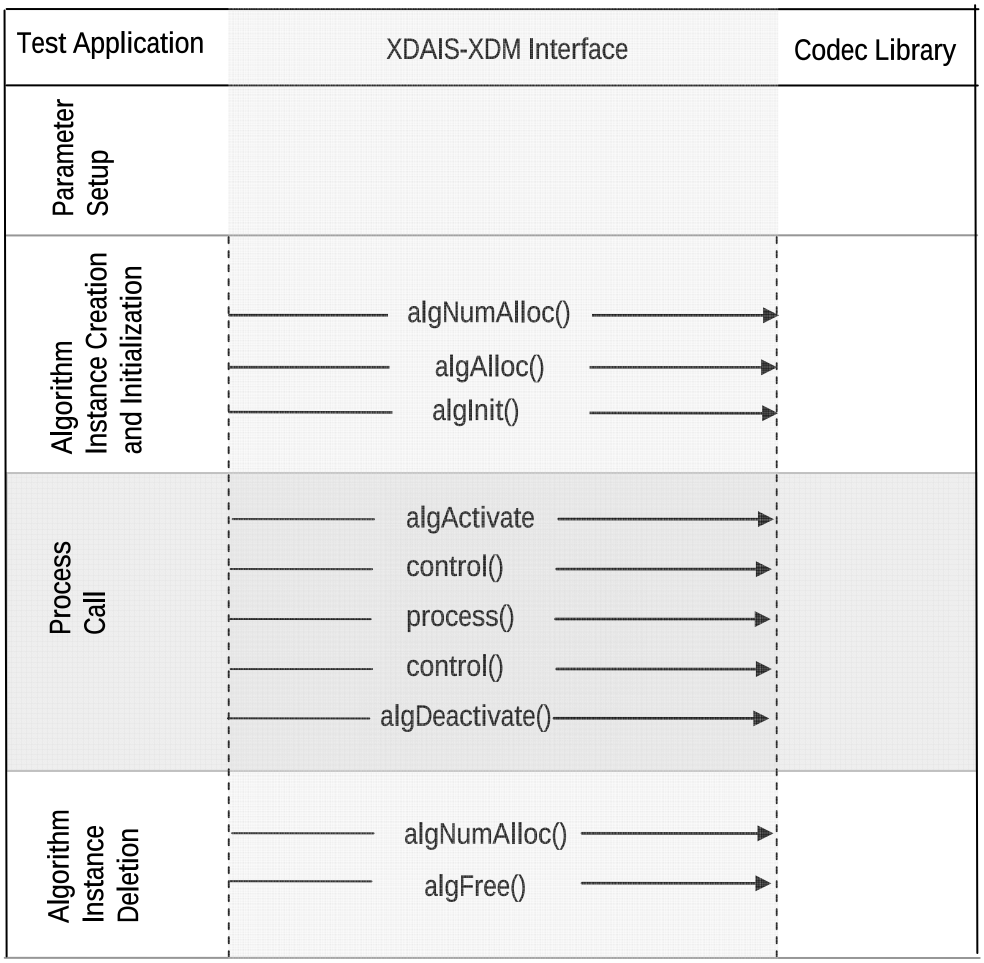


Figure 3-1. Test Application Sample Implementation

The test application is divided into four logical blocks:

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

3.1.1 Parameter Setup

Each codec component requires various codec configuration parameters to be set at initialization. For example, a video codec requires parameters such as video height, video width,and so on. The test application obtains the required parameters from the Decoder configuration files.

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

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

2) Opens the Decoder configuration file, (Testparams.cfg) and reads the various configuration parameters required for the algorithm.

   For more details on the configuration files, see Section 2.5.

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

4) Reads the input bit-stream into the application input buffer.

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

3.1.2 Algorithm Instance Creation and Initialization

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

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

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

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

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

3.1.3 Process Call

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

1) Sets the dynamic parameters (if they change during run-time) by calling the `control()` function with the `XDM_SETPARAMS` command.

2) Sets the input and output buffer descriptors required for the `process()` function call. The input and output buffer descriptors are obtained by calling the `control()` function with the `XDM_GETBUFINFO` command.

3) Call s the `process()` function to encode/decode a single frame of data. The behavior of the algorithm can be controlled using various dynamic parameters (see Section 4.3.1.8). The inputs to the process function are input and output buffer descriptors, pointer to the `IIMGDEC1_InArgs` and `IIMGDEC1_OutArgs` structures.

The `control()` and `process()` functions should be called only within the scope of the `algActivate()` and `algDeactivate()` XDAIS functions which activate and deactivate the algorithm instance respectively. Once an algorithm is activated, there could be any ordering of `control()` and `process()` functions. The following APIs are called in sequence:

1) `algActivate()` - To activate the algorithm instance.

2) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the six available control commands.

3) `process()` - To call the Decoder with appropriate input/output buffer and arguments information.

4) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the six available control commands.

5) `algDeactivate()` - To deactivate the algorithm instance.

The do-while loop encapsulates frame level `process()` call and updates the input buffer pointer every time before the next call. The do-while loop breaks off either when an error condition occurs or when the input buffer exhausts. It also protects the `process()` call from file operations by placing appropriate calls for cache operations as well. The test application does a cache invalidate for the valid input buffers before `process()` and a cache write back invalidate for output buffers after `process()`.

In the sample test application, after calling `algDeactivate()`, the output data is either dumped to a file or compared with a reference file.

Note:

In the test application, the `process()` should be called twice. First when the decode header is set to `XDM_PARSE_HEADER` it decodes only the header and next with decode header set to `XDM_DECODE_AU` decodes the entire access unit.

3.1.4 Algorithm Instance Deletion

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

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

2) `algFree()` - To query the algorithm to get the memory record information.

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.

This page is intentionally left blank

Chapter 4

API Reference

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

4.1 Symbolic Constants and Enumerated Data Types

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

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
XDM_AccessMode	XDM_ACCESSMODE_READ	JPEG Decoder instance reads from the buffer using the CPU.
	XDM_ACCESSMODE_WRITE	JPEG Decoder instance writes to the buffer using the CPU.
XDM_DataFormat	XDM_BYTE	Big endian stream.
	XDM_LE_16	16-bit little endian stream.
	XDM_LE_32	32-bit little endian stream.
XDM_DecMode	XDM_DECODE_AU	Decode entire access unit.
	XDM_PARSE_HEADER	Performs JPEG header parsing.
XDM_EncMode	XDM_ENCODE_AU	Encode entire access unit, including headers.
	XDM_GENERATE_HEADER	Encode only header
XDM_ChromaFormat	XDM_CHROMA_NA	Chroma format not applicable
	XDM_YUV_420P	YUV 4:2:0 planar.
	XDM_YUV_422P	YUV 4:2:2 planar.
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian).
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian).
	XDM_YUV_444P	YUV 4:4:4 planar.
	XDM_YUV_411P	YUV 4:1:1 planar.
	XDM_GRAY	Gray format.
	XDM_RGB	RGB color format.
	JPEGDEC_DEFAULT_CHROMA_FORMAT	Output follows input format
	XDM_CHROMAFORMAT_DEFAULT	YUV 4:2:2 interleaved (little endian).
XDM_CmdId	XDM_GETSTATUS	Query JPEG Decoder instance to fill the IJPEGDEC_Status structure.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	`XDM_SETPARAMS`	Set run-time dynamic parameters.
	`XDM_RESET`	Reset the decoder.All fields in the internal data structures are reset and all internal buffers are flushed.
	`XDM_SETDEFAULT`	Initialize all fields in the param structure to their default values. The application can change specific parameters using `XDM_SETPARAMS`.
	`XDM_FLUSH`	Handle end of stream conditions. This command forces JPEG Decoder to output data without additional input. The recommende sequence is to call the control() function (with `XDM_FLUSH`) followed by repeated calls to the process() function until it returns an error.
	`XDM_GETBUFINFO`	Query JPEG Decoder instance regarding properties of input and output buffers.
	`XDM_GETVERSION`	Query JPEG Decoder instance version. The result will be returned in the data field of the respective `status` structure.
`XDM_ErrorBit`	`XDM_APPLIEDCONCEALMENT`	The bit-fields in the 32-bit error code are interpreted as shown. Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore This bit is not used in this version of JPEG Decoder.
	`XDM_INSUFFICIENTDATA`	Bit 10 ❑ 1 - Insufficient input data ❑ 0 - Ignore
	`XDM_CORRUPTEDDATA`	Bit 11 ❑ 1 - Data problem/corruption ❑ 0 - Ignore
	`XDM_CORRUPTEDHEADER`	Bit 12 ❑ 1 - Corrupted frame header ❑ 0 - Ignore
	`XDM_UNSUPPORTEDINPUT`	Bit 13 ❑ 1 - Unsupported feature/parameter in input ❑ 0 - Ignore

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore
	XDM_FATALERROR	Bit 15 ❑ 1- Fatal error (stop decoding) ❑ 0 - Recoverable error

Note:

The remaining bits that are not mentioned in `XDM_ErrorBit` are interpreted as:

- Bit 16-32: Reserved
- Bit 8: Reserved
- Bit 0-7: Codec and implementation specific (see Tables under Section 4.2.1)

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

The JPEG Decoder specific error status messages are listed in Section 4.2.1. The value column indicates the decimal value of the last 8-bits reserved for codec specific error statuses.

4.2 Behavioral Specification of the Decoder

Behavioral specification defines how the decoder reacts to errors and recommended steps to be taken by the application to recover from such errors.

4.2.1 Classification of Errors

Errors that the decoder returns can be grouped into the following categories:

- **Fatal errors** - These are non-recoverable errors that cause the application to re-initialize, reset, or re-instantiate the decoder for resuming normal operation. The application cannot decode until you reset the decoder.
- **Non-Fatal bit- stream errors** - These are errors for which the application can proceed with the decoding without resetting the decoder. The current frame is erroneous and hence decoder can continue with the decoding skipping the erroneous frame. The `process()` function returns `FAILURE`.
- **Non-Fatal errors** – These are errors for which the decoder can continue decoding the current frame. The error does not affect the decoding process. The `process()` function returns `SUCCESS` and sets the extended error status accordingly.

Table 4-2. JPEG Decoder Fatal Error Statuses.

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
None	-	-	-

Table 4-3. JPEG Decoder Non-Fatal Bit Stream Error Statuses

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
`IJPEGDEC_ErrorStatus`	`JPEGDEC_ERROR_UNSUPPORTED_FORMAT`	01	SOI not found
	`JPEGDEC_ERROR_NOT_BASELINE_INTL`	03	Image is not sequential
	`JPEGDEC_ERROR_DISPLAY_WIDTH`	04	Invalid display width (`displaywidth < imagewidth`)
	`JPEGDEC_ERROR_VLD`	05	Error in Variable Length Decoding (VLD)
	`JPEGDEC_ERROR_SCAN_FREQ`	06	Invalid scan frequency

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	JPEGDEC_ERROR_RST_MARKER	07	Missing restart marker
	JPEGDEC_ERROR_MISSING_MARKER	08	Missing either SOS, SOF, DHT, or DQT marker
	JPEGDEC_ERROR_BAD_MARKER_LENGTH	09	Invalid marker length
	JPEGDEC_END_OF_IMAGE	10	Reached end of picture
	JPEGDEC_ERROR_BAD_DQT	11	Error in quantization table
	JPEGDEC_ERROR_DHT ERROR	12	Error in huffman table
	JPEGDEC_ERROR_BAD_DRI_LEN	13	Bad DRI length
	JPEGDEC_ERROR_SOS_NO_SOF	14	Invalid JPEG file structure: SOS before SOF
	JPEGDEC_ERROR_SOF_DUPLICATE	15	Invalid JPEG file structure: two SOF markers
	JPEGDEC_ERROR_BAD_SOF_DATA	16	Bad SOF marker length or component
	JPEGDEC_ERROR_SOS_COMP_ID	17	Invalid component ID in SOS marker
	JPEGDEC_ERROR_BAD_SOS_LEN	18	Invalid length of SOS or bad component numbers
	JPEGDEC_ERROR_SOS_INVALID	19	SOS marker is invalid
	JPEGDEC_ERROR_BAD_PROGRESSION	20	Invalid progressive parameters
	JPEGDEC_ERROR_BAD_PRECISION	21	Sample precision not equal to 8
	JPEGDEC_ERROR_COMPONENT_COUNT	22	Too many or few color components in the scan or frame
	JPEGDEC_ERROR_BAD_MCU_SIZE	23	Sampling factors too large for interleaved scan
	JPEGDEC_ERROR_EMPTY_IMAGE	24	Invalid image width or height or number of components

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	JPEGDEC_ERROR_IMAGE_SIZE	25	Input Image size is greater than Maximum set size and resizing factor required is more than 3. Note: Resizing factor will be calculated to fit the image in a given output buffer size and decoder outputs the resized image.
	JPEGDEC_ERROR_CORRUPTED_BITSTREAM	26	Corrupted bit-stream
	JPEGDEC_ERROR_OUTSIZE	27	Output buffer size is too Small for the image size
	JPEGDEC_ERROR_DCTBUFSIZE	28	Error in DCT buffer size
	JPEGDEC_ERROR_NULL_PTR	29	NULL pointer
	JPEGDEC_ERROR_INPUT_PARAMETER	30	Error occurred in the interface parameter
	JPEGDEC_ERROR_MAX_SCAN	31	Exceed maximum number of scans
	JPEGDEC_ERROR_INVALID_numAU	32	Invalid numAU to decode in sectional decoding
	JPEGDEC_ERROR_INVALID_mcu_multiplication	33	Invalid x_org or invalid y_org for sub-region decoding
	JPEGDEC_ERROR_INVALID_Xlength_Ylength	34	Invalid X length or invalid Y length for sub-region decoding

Table 4-4. JPEG Decoder Non-Fatal Error Statuses

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
IJPEGDEC_ErrorStatus	JPEGDEC_ERROR_PROG_MEM_ALOC	2	Input image is progressive

4.2.2 Recommended Steps to Recover from Error

The categories of JPEG Decoder specific errors and the recommended actions to be performed by the application to recover from these errors are listed in Table 4-5.

Table 4-5. Recommended Steps on the Application Side

Error Category	Recommended Steps
Fatal errors	❑ The application has to reinstantiate the decoder to continue decoding.
Non-fatal bit stream errors	❑ Application can skip the erroneous frame and continue decoding the other frames. ❑ If the application decides to abort the execution of the particular sequence, it should call `reset()` API before it starts decoding the next sequence.
Non-fatal errors	❑ `JPEGDEC_ERROR_NOT_BASELINE_INTL`, indicates that the library does not support progressive images. ❑ `JPEGDEC_ERROR_PROG_MEM_ALOC`, application should set the `progressiveDecFlag` in the `IJPEGDEC_Params` data structure to support decoding progressive images in case of extended structures. ❑ `JPEGDEC_ERROR_INPUT_PARAMETER`, application should check and set the dynamic input parameters correctly.
No extended error is set by the algorithm and `process()` API	Check if you have assigned `NULL` pointers for any of the inputs of the `process()` API like `inBufs`, `outBufs`, `inargs`, `outargs`.

4.3 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.

4.3.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- `XDM_BufDesc`
- `XDM_SingleBufDesc`
- `XDM1_SingleBufDesc`
- `XDM1_BufDesc`
- `XDM1_AlgBufInfo`
- `IIMGDEC1_Fxns`
- `IIMGDEC1_Params`
- `IIMGDEC1_DynamicParams`
- `IIMGDEC1_InArgs`
- `IIMGDEC1_Status`
- `IIMGDEC1_OutArgs`

4.3.1.1 XDM_BufDesc

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`**bufs`	`XDAS_Int8`	Input	Pointer to an array containing buffer addresses
`numBufs`	`XDAS_Int32`	Input	Number of buffers
`*bufSizes`	`XDAS_Int32`	Input	Size of each buffer in 8-bit bytes

4.3.1.2 XDM_SingleBufDesc

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`*buf`	`XDAS_Int8`	Input	Pointer to a buffer address
`bufSize`	`XDAS_Int32`	Input	Size of each buffer in 8-bit bytes

4.3.1.3 XDM1_SingleBufDesc

|| Description

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

|| Fields

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

4.3.1.4 XDM1_BufDesc

|| Description

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

|| 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

4.3.1.5 XDM1_AlgBufInfo

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`minNumInBufs`	`XDAS_Int32`	Output	Minimum number of input buffers
`minNumOutBufs`	`XDAS_Int32`	Output	Minimum number of output buffers
`minInBufSize[XDM_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	Minimum size, in 8-bit bytes, required for each input buffer
`minOutBufSize[XDM_MAX_IO_BUFFERS]`	`XDAS_Int32`	Output	Minimum size, in 8- bit bytes, required for each output buffer

Note:

The JPEG Decoder has the following buffer information:

- Number of input buffer required is 1
- Number of output buffers required are 1 for `XDM_YUV_422ILE` and 3 for `XDM_DEFAULT`
- The input buffer size is equal to `(maxHeight` * `maxWidth)`. See `IIMGDEC1_Params` data structure for details.

Algorithm populates output buffer sizes with respect to the maximum height and maximum width if `control()` API is called with

XDM_GETBUFINFO control command. Each output buffer size is equal to a value of 2 * (maxHeight * maxWidth) for XDM_YUV_422ILE and (maxHeight * maxWidth) for XDM_DEFAULT. Algorithm populates actual output buffer sizes with respect to the image format if control() API is called with XDM_GETSTATUS control command after the process() call is made with header parsing.

4.3.1.6 IIMGDEC1_Fxns

|| Description

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

|| Fields

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

4.3.1.7 IIMGDEC1_Params

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes. Default: sizeof (IIMGDEC1_Params)
maxHeight	XDAS_Int32	Input	Maximum image height. Set multiples of 16 to support all chromaformat. The default value is 1600.
maxWidth	XDAS_Int32	Input	Maximum image width. Set multiples of 32 to support all chromaformat. The default value is 2048.

Field	Datatype	Input/ Output	Description
maxScans	XDAS_Int32	Input	Maximum number of scans for progressive mode. The default value is 15.
dataEndianness	XDAS_Int32	Input	Endianness of input data. The JPEG decoder implementation supports only XDM_BYTE format.
forceChromaFormat	XDAS_Int32	Input	Force decoding in the given chroma format. See XDM_ChromaFormat enumeration for details. To set Planar / Interleaved / RGB output YUV format: ❑ JPEGDEC_DEFAULT_CHROMA_FORMAT - YUV planar (same as the data format in the encoded data). ❑ XDM_YUV_422ILE - YUV 422 interleaved. ❑ XDM_RGB - RGB output (set RGB_format variable of IJPEGDEC_DynamicParams structure to select RGB format as BGR24 or RGB16 or BGR32) Default: JPEGDEC_DEFAULT_CHROMA_FORMAT

4.3.1.8 IIMGDEC1_DynamicParams

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes. Default: sizeof (IIMGDEC1_DynamicParams)
numAU	XDAS_Int32	Input	Number of access units to decode. Setting this field to XDM_DEFAULT decodes the complete frame. Any value other than XDM_DEFAULT will decode that many number of MCUs. The minimum value should be number of MCUs per row.
decodeHeader	XDAS_Int32	Input	Flag indicating if only header parsing needs to be done. See XDM_DecMode enumeration for details. ❑ XDM_PARSE_HEADER - Return after header parsing is done. ❑ XDM_DECODE_AU - Return after complete frame decoding. Default: XDM_DECODE_AU

Field	Datatype	Input/ Output	Description
displayWidth	XDAS_Int32	Input	❑ XDM_DEFAULT - Use the decoded image width as pitch. ❑ If any other value greater than the decoded image width is provided, then this value (in pixels) is used as pitch. Note: displayWidth should be => imageWidth

4.3.1.9 IIMGDEC1_InArgs

|| Description

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

|| Fields

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

4.3.1.10 IIMGDEC1_Status

|| Description

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

|| 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 enumeration. See XDM_ErrorBit enumeration for details.
data	XDM1_SingleDesc	Output	Buffer descriptor for data passing
outputHeight	XDAS_Int32	Output	Decoded image height.
outputWidth	XDAS_Int32	Output	Decoded image width.
imageWidth	XDAS_Int32	Output	Actual image width.

Field	Datatype	Input/ Output	Description
outChromaformat	XDAS_Int32	Output	Outputchromaformat. See XDM_ChromaFormat enumeration for details.
totalAU	XDAS_Int32	Output	Total number of MCUs.
totalScan	XDAS_Int32	Output	Total number of scans in the progressive image.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

4.3.1.11 IIMGDEC1_OutArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
extendedError	XDAS_Int32	Output	Extended error enumeration. See XDM_ErrorBit enumeration for details.
bytesconsumed	XDAS_Int32	Output	The numbers of input bytes consumed per decode call.
currentAU	XDAS_Int32	Output	Current MCU number.
currentScan	XDAS_Int32	Output	Current scan number (for progressive mode).

> **Note:**
>
> Total AU and Current AU are used in case of sectional decoding and are updated correctly. In other cases, the values returned may not be correct.

4.3.2 JPEG Decoder Data Structures

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

- IJPEGDEC_Params
- IJPEGDEC_fxns
- IJPEGDEC_DynamicParams
- IJPEGDEC_InArgs
- IJPEGDEC_Status
- IJPEGDEC_OutArgs

4.3.2.1 IJPEGDEC_Params

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
imgdecParams	IIMGDEC1_Params	Input	See IIMGDEC1_Params data structure for details.
progressiveDecFlag	XDAS_Int32	Input	Set flag value 1 if progressive decoding is required in addition to baseline sequential mode.
outImgRes	XDAS_Int32	Input	Set the output image resolution ❑ 0 - Always Even Image resolution ❑ 1 - Outputs Actual Image resolution

Note:

- For outImgRes set resolution as even, for non-standard images, output resolution will be as follows:
  - In case of 422ILE, 422P, 420P, 444P format, output resolution will be in multiple of 2 (example: 227*149 => 226*148)
  - In case of 411 format output resolution will be in multiple of 4 (example: 227*149 => 224*148)
- For outImgRes set resolution as actual, for non-standard images, output resolution will be as follows:
  - In case of 422ILE, 422P, 420P format, output resolution will be in multiple of 2 (example: 227*149 => 226*148)

- In case of 411 format, output resolution will be in multiple of 4 (example:.227*149 => 224*148)
- In case of 444P and all RGB output format, output resolution will be actual image resolution

4.3.2.2 IJPEGDEC_fxns

|| Description

This structure defines all of the operations on JPEG Decoder instance object.

|| Fields

Field	Datatype	Input/ Output	Description
iimgdec	IIMGDEC1_fxns	input	See IIMGDEC1_fxns data structures for details

4.3.2.3 IJPEGDEC_DynamicParams

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
imgdecDynamicParams	IIMGDEC1_DynamicParams	Input	See IIMGDEC1_DynamicParams data structure for details.
Frame_numbytes	XDAS_Int32	Input	Total number of bytes used for sectional decoding
progDisplay	XDAS_Int32	Input	Set the display option for progressive mode: ❑ 1 - Output buffer contains the partially (progressively) decoded image after each scan is decoded. ❑ 0 - Output buffer contains the decoded image only after all the scans are decoded.

Field	Datatype	Input/ Output	Description
resizeOption	XDAS_Int32	Input	Resize output image ❑ 0 - No resizing ❑ 1 - Resize output image by 1/2 ❑ 2 - Resize output image by 1/4 ❑ 3 - Resize output image by 1/8 ❑ 4 - Up-scale output image by 2 ❑ 5 - Up-scale output image by 4 ❑ 6 - Up-scale output image by 8
RGB_Format	XDAS_Int32	Input	Set the output RGB format ❑ 0 - BGR24 ❑ 1 - BGR32 ❑ 2 - RGB16
numMCU_row	XDAS_Int32	Input	Number of rows of access units to decode. Setting this field to XDM_DEFAULT decodes the complete frame. Any value other than XDM_DEFAULT will decode that many number of rows Set numMCU_row to any integer value other then zero for sectional decoding
x_org	XDAS_Int32	Input	start point of the X-axis of the subregion
y_org	XDAS_Int32	Input	start point of the Y-axis of the subregion
x_length	XDAS_Int32	Input	X-length of the subregion
y_length	XDAS_Int32	Input	Y-length of the subregion
alpha_rgb	XDAS_UInt8	Input	Alpha value to fill rgb32 Default value: 0

Note:

- If input image size is greater than maximum set size, resizing factor will be calculated to fit the image within a given output buffer size and decoder outputs the resized image. The resizeOption field will be set accordingly.
- To support RGB output format, set forceChromaFormat in IIMGDEC1_Params as XDM_RGB and set the value for RGB_format accordingly.
- numAU in IIMGDEC1_DynamicParams or numMCU_row should be set

for sectional decoding

- `x_org` and `y_org` should set as multiples of the number specified in the following table based on the `inputchromaformat`

Input Format	Multiplication Factor	
	x_org	y_org
XDM_YUV_420	16	16
XDM_YUV_422	16	8
XDM_YUV_444	8	8
XDM_YUV_411	32	8
XDM_GRAY	8	8

- `x_org` + `x_length` should be less then the original width
- `y_org` + `y_length` should be less then the original height

4.3.2.4 IJPEGDEC_InArgs

|| Description

This structure defines the run-time input arguments for a JPEG Decoder instance object.

|| Fields

Field	Datatype	Input/ Output	Description
imgdecInArgs	IIMGDEC1_InArgs	Input	See IIMGDEC1_InArgs data structure for details.

4.3.2.5 IJPEGDEC_Status

|| Description

This structure defines parameters that describe the status of the JPEG Decoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, `IIMGDEC1_Status`.

|| Fields

Field	Datatype	Input/ Output	Description
imgdecStatus	IIMGDEC1_Status	Output	See IIMGDEC1_Status data structure for details.
mode	XDAS_Int32	Output	Mode of operation: ❑ 0 - Baseline sequential interleave ❑ 1 - Baseline sequential non interleave ❑ 2 - Progressive

Field	Datatype	Input/ Output	Description
imageHeight	XDAS_Int32	Output	Actual height of the image.
decImageSize	XDAS_Int32	Output	Size of the decoded image in bytes.
lastMCU	XDAS_Int32	Output	Flag indicating that the last Minimum Coded Unit (MCU) has been processed: ❑ 1 – Decoded all MCU's ❑ 0 – Decoding not completed
hSampleFreq	XDAS_Int32	Output	Horizontal Sampling frequency for chroma component
vSampleFreq	XDAS_Int32	Output	Vertical Sampling frequency for chroma component
End_of_seq	XDAS_Int8	Output	End of sequence
End_of_scan	XDAS_Int8	Output	End of scan
Bytesgenerated[3]	XDAS_Int32	Output	Slice decoded image after each call

4.3.2.6 IJPEGDEC_OutArgs

‖ Description

This structure defines the run-time output arguments for the JPEG Decoder instance object.

‖ Fields

Field	Datatype	Input/ Output	Description
imgdecOutArgs	IIMGDEC1_OutArgs	Output	See IIMGDEC1_OutArgs data structure for details.

4.4 Interface Functions

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

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

You must call these APIs in the following sequence:

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

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

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

4.4.1 Creation APIs

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

|| Name

`algNumAlloc()` – determine the number of buffers that an algorithm requires

|| Synopsis

```
XDAS_Int32 algNumAlloc(Void);
```

|| Arguments

```
Void
```

|| Return Value

```
XDAS_Int32; /* number of buffers required */
```

|| Description

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

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

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

|| See Also

`algAlloc()`

‖ Name

`algAlloc()` – determine the attributes of all buffers that an algorithm requires

‖ Synopsis

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

‖ Arguments

```
IALG_Params *params; /* algorithm specific attributes */

IALG_Fxns **parentFxns;/* output parent algorithm
functions */

IALG_MemRec memTab[]; /* output array of memory records */
```

‖ Return Value

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

‖ Description

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

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

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

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

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

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

‖ See Also

`algNumAlloc(), algFree()`

4.4.2 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).

‖ See Also

`algAlloc()`, `algMoved()`

4.4.3 Control API

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

|| Name

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

|| Synopsis

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

|| Arguments

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

IIMGDEC1_Cmd id; /* algorithm specific control commands*/

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

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

|| Return Value

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

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

|| Description

This function changes the run-time parameters of an algorithm instance and queries the algorithm's status. `control()` must only be called after a successful call to `algInit()` and must never be called after a call to `algFree()`.

The first argument to `control()` is a handle to an algorithm instance.

The second argument is an algorithm specific control command. See `XDM_CmdId` enumeration for details.

The third and fourth arguments are pointers to the `IIMGDEC1_DynamicParams` and `IIMGDEC1_Status` data structures respectively.

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

- `control()` can only be called after a successful return from `algInit()` and `algActivate()`.
- If algorithm uses DMA resources, `control()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.

‖ Postconditions

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.
- If the control command is not recognized, the return value from this operation is not equal to `IALG_EOK`.

‖ Example

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

‖ See Also

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

4.4.4 Data Processing API

Data processing API is used for processing the input data.

|| Name

`algActivate()` – initialize scratch memory buffers prior to processing.

|| Synopsis

```
Void algActivate(IALG_Handle handle);
```

|| Arguments

```
IALG_Handle handle; /* algorithm instance handle */
```

|| Return Value

`Void`

|| Description

`algActivate()` initializes any of the instance's scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to `algActivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be initialized prior to calling any of the algorithm's processing methods.

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

|| See Also

`algDeactivate()`

|| Name

process() – basic encoding/decoding call

|| Synopsis

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

|| Arguments

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

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

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

IIMGDEC1_InArgs *inargs /* algorithm runtime input arguments
*/

IIMGDEC1_OutArgs *outargs /* algorithm runtime output
arguments */
```

|| Return Value

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

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

|| Description

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

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see XDM1_BufDesc data structure for details).

The fourth argument is a pointer to the IIMGDEC1_InArgs data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the IIMGDEC1_OutArgs data structure that defines the run-time output arguments for an algorithm instance object.

Note:

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

|| Preconditions

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

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

- If algorithm uses DMA resources, `process()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.
- Buffer descriptor for input and output buffers must be valid.
- Input buffers must have valid input data.

|| Postconditions

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

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

|| Example

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

|| See Also

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

|| Name

`algDeactivate()` – save all persistent data to non-scratch memory

|| Synopsis

```
Void algDeactivate(IALG_Handle handle);
```

|| Arguments

```
IALG_Handle handle; /* algorithm instance handle */
```

|| Return Value

`Void`

|| Description

`algDeactivate()` saves any persistent information to non-scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to `algDeactivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be saved prior to next cycle of `algActivate()` and processing.

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

|| See Also

`algActivate()`

4.4.5 Termination API

Termination API is used to terminate the algorithm instance and free up the memory space that it uses.

|| Name

`algFree()` – determine the addresses of all memory buffers used by the algorithm

|| Synopsis

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

|| Arguments

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

|| Return Value

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

|| Description

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

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

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

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

|| See Also

`algAlloc()`

Chapter 5

Frequently Asked Questions

This section answers frequently asked questions related to using JPEG Progressive Support Decoder on C64x+.

Table 5-1. FAQs for JPEG Progressive Support Decoder on C64x+

Questions	Answers
Does JPEG Decoder support sub-sectional decoding of the image starting at an arbitrary location?	JPEG Decoder does support decoding of a sub-sectional decoding from any location by setting `x_org`, `y_org` of `IJPEGDEC_DynamicParams` as the origin and `x_length` and `y_length` as the dimensions of the sub-section. However, there are some constraints on values, which is explained in section 4.3.2.3.
What is the difference between `XDM_AlgBufInfo.minOutBufSize` and `IJPEGDEC_Status.decImageSize`?	A minimum buffer of size `XDM_AlgBufInfo.minOutBufSize` should be provided by the application to store the decoded output. There is no restriction on the value taken by `XDM_AlgBufInfo.minOutBufSize`. `IJPEGDEC_Status.decImageSize` provides the actual size of the decoded image. `IJPEGDEC_Status.decImageSize` should not be used for output buffer allocation.
Does the JPEG Decoder support non-multiple of 16 image size?	The decoder supports decoding of bit streams with non-multiple of 16 image size. The output buffer in this case can be of the actual image size. The height and width values used to create decoder instance should be multiple of 16.
Does JPEG Decoder support progressive decoding?	JPEG Decoder supports decoding of Baseline Sequential and Progressive images.
What are the output formats supported by the JPEG Decoder?	JPEG Decoder supports default encoding format (output is in the same format as the input), YUV 422 interleaved, RGB16, BGR24 and BGR32 output formats (internal conversion from encoding format to the mentioned output format).
Does JPEG Decoder support rescaling of the output?	Yes, this version of JPEG Decoder supports downscaling of the output by a factor of 2, 4 or 8 and also up-scaling by factor of 2, 4 or 8 using `IJPEGDEC_DynamicParams.resizeOption`. See section 4.3.2.3 and Appendix A for details.
Can the resolution of JPEG image be obtained without decoding the entire image?	Yes, This can be done by setting `IIMGDEC_DynamicParams.decodeHeader = XDM_PARSE_HEADER` instead of `XDM_DECODE_AU`.
What will happen if an odd resolution encoded image is provided as an input to the decoder?	JPEG Decoder supports decoding of odd resolution images. However, the resolution of output image will be changed by the decoder algorithm depending upon the encoding format of the image. See section 4.3.2.1 for details.

This page is intentionally left blank

Appendix A

Sectional Decoding

This appendix contains Sectional Decoding details for JPEG Progressive Support Decoder on C64x+.

A.1 Overview

Sectional decoding is same as slice based JPEG decoding. The decoder decodes N number of rows of MCUs, where N is less than or equal to the total number of rows of MCUs in the image. Decoding starts from top left image corner and progresses towards the bottom until the entire image is decoded.

The following parameters are used for sectional decoding:

- `IIMGDEC1_DynamicParams. numAU`
- `IJPEGDEC_DynamicParams. numMCU_row`

A.1.1 IIMGDEC1_DynamicParams. numAU

Number of MCUs to be decoded for each slice, `numAU` should be in the multiples of MCUs per line. If any other value is set, then the algorithm recalculates the next MCUs per line and then will round it to the next MCUs per line. (See section 4.3.1.8).

A.1.2 IJPEGDEC_DynamicParams. numMCU_row

Number of Line or row to be decoded for each slice. For example, if set to 1, then for each process call, it decodes 1 line or row (See section 4.3.2.3).

> **Note:**
>
> If both the values are set, then algorithm will consider the highest value and calculate accordingly.

A.1.3 Input Buffer Requirement

Sufficient input buffer is required for sectional decoding. If the data is less than or equal to input data, then the algorithm will return error.

To calculate input buffer size:

`Size = IJPEGDEC_DynamicParams. numMCU_row * WIDTH * X`

where, `WIDTH` is the image width.

The application takes care of the value of `X`, so that `Size` would be sufficient to decode `IJPEGDEC_DynamicParams. numMCU_row`.

The pseudo code to calculate input buffer size is as shown.

```
inputsize = 5000;
   if(status_one.imgdecStatus.imageWidth > 1000)
   {
      inputsize =  status_one.imgdecStatus.imageWidth * 20;
   }

   if(dynamiparams.numMCU_row !=0) // for sliced
       {
             if(byteremain <=(inputsize*
dynamiparams.numMCU_row))
       inargs.imgdecInArgs.numBytes = byteremain;
             else

             inargs.imgdecInArgs.numBytes  = inputsize*
dynamiparams.numMCU_row;
             if(inargs.imgdecInArgs.numBytes >
inargs.frame_numbytes)
                    inargs.imgdecInArgs.numBytes =
inargs.frame_numbytes;

       }

 ret =JPEGDEC_decode(handle_one,( XDM_BufDesc *)&inBufs,
                    (XDM_BufDesc *)&outBufs_one,
(IIMGDEC_InArgs *)&inargs,
                    (IIMGDEC_OutArgs *)&outargs_one);
 byteremain -= outargs_one.imgdecOutArgs.bytesconsumed;

 if(dynamiparams.numMCU_row )
 {
        inBufs.bufs[0] +=
outargs_one.imgdecOutArgs.bytesconsumed;
               inargs.imgdecInArgs.numBytes -=
outargs_one.imgdecOutArgs.bytesconsumed;

 }
```

Note:

The pseudo code should repeat until end of sequence.

A.2 Sub-region Decoding

The decoder decodes only a particular region of the image. The application passes the (x,y) coordinates and the corresponding x-length and y-length fields from the upper layers (exact details to be specified by the algorithm owner). The algorithm decodes all rows of MCUs contained within that region (sub-optimal from a memory/performance standpoint, but easier to implement), or decode the region itself (optimal solution, but more complex). If the implementation decodes only a region, there are restrictions imposed on the upper layers to ensure that the input data passed is aligned on proper MCU boundaries (example, 16x16 in case of 4:2:0 input data).

The following parameters are used for sub-region decoding:

- `IJPEGDEC_DynamicParams. x_org`
- `IJPEGDEC_DynamicParams. y_org`
- `IJPEGDEC_DynamicParams. x_length`
- `IJPEGDEC_DynamicParams. y_length`

Any part of the image can be selected for decoding.

Example 1: Consider the VGA (640x480) resolution

```
x_org  =  0

y_org = 0

x_length = 128

y_length = 128
```

The output of the decoder is 128 x 128 starting origin for the decoder is (0, 0)

Example 2: Consider the VGA (640x480) resolution

```
x_org  =  16

y_org = 8

x_length = 176

y_length = 144
```

The output of the decoder would be 176 x 144 starting origin for the decoder is (16, 8).

The values of `x_org` and `y_org` depends on the format of `input chromaformat(`See the note in section 4.3.2.3).

A.3 Sub-region Scaling

The following parameters are used for sub-region scaling:

- `IJPEGDEC_DynamicParams. resizeOption`
- `IJPEGDEC_DynamicParams. x_org`
- `IJPEGDEC_DynamicParams. y_org`
- `IJPEGDEC_DynamicParams. x_length`
- `IJPEGDEC_DynamicParams. y_length`

Example 1: Consider the VGA (640x480) resolution

```
resizeOption = 1

x_org  =  0
```

```
y_org = 0

x_length = 128

y_length = 128
```

The output of the decoder would be 64 x 64 (resized by 2) starting origin for the decoder is (0,0).

Example 2: Consider the VGA (640x480) resolution

```
resizeOption = 4

x_org  =  0

y_org = 0

x_length = 128

y_length = 128
```

The output of the decoder is 256 x 256 (up scaled by 2) starting origin for the decoder is (0,0).

Appendix B

Revision History

This user guide revision history highlights the changes made to the SPRUEA9B codec specific user guide to make it SPRUEA9C.

Table B-1.Revision History for JPEG Progressive Support Decoder on C64x+

Section	Addition/Deletion/Modification
Global	Note: There are no changes in user guide for this release