



HEXAGON

ERDAS ECW JP2 SDK 5.5 Update 3

USER GUIDE

Version 5.5.0 Update 3

11 February 2021

Contents

Introduction	7
Overview.....	7
API Documentation.....	8
Code Listings	8
Intended Audience	8
Acknowledgements.....	8
What's New	9
Version 5.5 Update 3	9
Version 5.5 Update 2	9
Version 5.5 Update 1	9
Version 5.5	9
Version 5.4 Update 1	10
Version 5.4	10
Version 5.3	10
Version 5.2	11
Version 5.1	11
Version 5.0	12
Version 4.3	13
Version 4.2	13
Version 4.1	13
Version 3.3	13
Version 3.0	14
Version 2.0	14
About Image Compression.....	15
Introduction.....	15
Lossless or Lossy Compression	15
Wavelet Based Encoding.....	16
ECW Compression	17
Licensing	18
Overview.....	18
Understanding Gigapixel Limitations.....	19

Activating a License	19
System Requirements	21
Development Platforms	21
Windows.....	21
Linux	21
MacOS X.....	21
Android, iOS.....	21
Development Environments.....	22
Windows.....	22
Linux	22
MacOS X.....	22
Android.....	22
iOS	22
Runtime Platforms	23
Windows.....	23
Linux	23
MacOS X.....	23
Android.....	23
iOS	23
Installation	24
Windows	24
Linux.....	26
MacOS X	26
Directory Structure.....	26
File Formats.....	28
ECW Version 2	29
ECW Version 3	29
JPEG 2000	29
NITF	30
ECWP Streaming Protocol	31
ECWP Version 2.....	31
ECWP Version 3.....	31
Example ECWP resources	32

Features	33
ECWP Persistent Local Cache	33
Image Resampling	33
Data Scaling	34
Dynamic Range Calculation	34
Opacity Channels	35
NULL blocks	35
Comparison	36
J2I Index Files	39
Compression Methods	40
Scanline Encoder	40
Tiled Encoder	41
Block Size	41
Region Updates	41
Development	43
How Imagery is Accessed	43
How to Read a View	43
The SetFileView Concept	43
Viewing Areas Smaller than your Application Window Area	44
Requesting Odd-Aspect Views	45
Selecting Bands from an Image File to View	45
Blocking Reads or the Refresh Call-back Interface	45
Blocking Reads	46
Refresh Callbacks	46
Cancelling Reads	48
Multiple Image Views and Unlimited Image Size	48
Error Handling	48
Memory Management	49
Compression requirements	49
Memory Usage	49
Caching	49
Coordinate Information	50
Transparent Proxying	51

Delivering Your Application	51
Creating Compressed Images	51
Understanding Target Compression Ratios	52
Preserving Image Quality When Recompressing	52
Optimizing the Compression Ratio	52
Recompressing Imagery	53
Compressing Hyperspectral Imagery	54
Image Size Limitations	54
Compression Directory Limitations	54
Linking against the SDK	54
Geocoding Information	54
Datum	55
Projection	55
Units	55
Registration Point	56
Cell Size	56
RAW versus LOCAL images	56
Editing the Georeferencing Header Information	57
Geodetic Transform Database	57
GDT File Formats	57
How the SDK Stores Geocoding Information	58
Embedded Geography Markup Language (GML) Metadata	58
GML Examples	58
JPEG 2000 Registration Behaviour Change in version 5.2	60
Embedded GeoTIFF Metadata	61
Supported GeoTIFF Tags	61
Supported GeoTIFF GeoKeys	61
Support for World Files	61
Configuring the Use of Geocoding Data for JPEG 2000 Files	62
EPSG Codes	63
ECW Version 3 Metadata	64
Table 1: RPC	64
Table 2: Statistical Metadata	65
Table 3: General File Metadata	65

Examples	66
C/C++	66
Objective C	66
Java (JNI)	66
Android	66
FAQ	67
Licensing	67
Platform Support and Build Environment	69
Software Maintenance (SWM)	70
General.....	71
SDK Concepts	72
Advanced Concepts.....	74
Appendix A: JPEG 2000 Conformance.....	77
ISO/IEC 15444	77
Supported Extensions from Part 2	77
Default JPEG 2000 Encoding Format	78
GML in JP2.....	79
NITF 2.1.....	79
Appendix B: List of View Parameters.....	80
Appendix C: ECW Header Editor CLI Parameters	82
Appendix D: Screenshots.....	83
Support	85
About Hexagon.....	86

Introduction

Overview

The ERDAS ECW JPEG2000 Software Development Kit (SDK) may be used to add large image support to applications. It provides compression and use of very large images in the industry standard ECW (Enhanced Compressed Wavelet) image format and the ISO standard JPEG 2000 format.

The SDK enables software developers working with C or C++ to add image compression and decompression support for the ECW and JPEG 2000 file formats to their own GIS, CAD or imaging applications. The libraries are small and can be packaged as shared objects to install on a user system or in an application's executable code directory. The SDK libraries have a small, clean interface with only a few function calls. Subsampling and selective views of imagery are handled automatically. You can use the SDK library with a simple read-region call, or a progressive-update call. You can include ECW or JPEG 2000 compressed images of any size (including terabytes or larger) within your application and performance will remain the same, regardless whether the images reside locally or from a remote source delivered from ERDAS APOLLO and ECWP. The source is functionally hidden from your application, which needs only to open views into the image. The SDK manages the entire image access and decompression process for you.





API Documentation

Starting with SDK v5.1 the API documentation has been split from the User Guide and can now be found in HTML format in the “apidocs” folder in the SDK installation directory. The C, C++ APIs are documented as well as the common decompression and compression examples.

For the MacOS X platform, the Objective C API documentation is available directly from within XCode or via the “Help -> Documentation” menu. See Appendix A for an overview of the API documentation.

Code Listings

Code listings are displayed in a Courier font, as follows:

```
classid="clsid:AD90E32F-1FE2-11D3-88C8-006008A717FD"
```

Intended Audience

The User Guide and SDK is intended for programmers with a good understanding of C and C++ programming concepts and techniques. The document explains background concepts and techniques for implementing the compression and decompression features of the SDK into a third-party application.

- ❗ **Note:** The SDK is not useful for end-users as it provides no functional capability once installed. Effort must be made to integrate with third-party toolkits to leverage the power of the SDK. For users seeking a compression tool, refer to the **GeoCompressor** product.

Acknowledgements

We would like to acknowledge the following third-party applications and libraries:

- NSIS (Nullsoft Installation System)
- GeoTIFF Library
- TinyXML library
- J2000 library
- Zlib library
- Boost library
- Little CMS library

Appropriate acknowledgement notices from these libraries can be found in the “Third-Party” directory of the SDK installation.

What's New

Version 5.5 Update 3

Released: January 2021

- [EC-2537] PLT offset calculation was incorrect for some multi-tile JP2 files.
- [EC-2597] Fixed crash parsing JP2 with GMLv2 box.
- [EC-2565] Improve error messages when IO failure due to low disk space.
- [EC-2465] Changed log level of some JP2 logs to debug to be less verbose.

Version 5.5 Update 2

Released: September 2020

- Updated public GDAL driver to be compatible with 5.5 release and GDAL 3.1.
- Shipped missing NCSIOStreamOptions.h header on Windows, OS X and iOS.
- Updates to WASM decoder for RIA 2020.
- Default block size for ECW v3 now 128x128. Block sizes larger the 256x256 generate performance warnings.
- NCSFileInfoReported now reports blocksize.
- Fixed local ECWP disk cache limit being ignored on non-Windows platforms.
- Fixed issue with authentication and local ECWP disk cache.

Version 5.5 Update 1

Released: January 2020

- Android now requires Android API level 21 (5) (now compiled with c++_shared, no longer using unsupported gnustdl_static).
- Deprecated armeabi Android ABI
- Notable issues resolved:
 - [EC-2336] Corrected profile header in JP2K profiles
 - [EC-2335] ECWHeaderEditorCLI.exe crash on exit
 - [EC-2329] SDK can't decode 27-bit JPEG 2000
 - [EC-2313] getDefaultRGBABandlist() sometimes returns incorrect values
 - [EC-2301] Created script to integrate SDK into XCode framework
 - [EC-2295] Android platform update
 - [EC-2282] Decompression Example 7 fails in MacOS
 - [EC-2235] Issue reading JPEG 2000 raster format generated by IPP.

Version 5.5

Released: October 2019

- Support for Microsoft Visual Studio 2019 included (however there are no specific binaries available for msvc142, as the msvc141 versions are binary compatible. See [here](#)).
- Removed support for Microsoft Visual Studio 2013.
- Removed support for iOS armv7 (32-bit) (Apple have dropped support for 32 bit from iOS 11 onwards).
- Removed support for Max OS X x86 (Apple have dropped support for 32 bit on Mojave+).



- Increased GCC on Linux support to 8.x.
- Android now requires Android API level 9 (2.3).
- Added support for arm64-v8a platform on Android (64-bit).

Version 5.4 Update 1

Released: February 2019

- Addressed vulnerability FG-VD-19-015 reported by Fortinet.
- Modified JP2 output of RGB + extra bands to not write a colour definition (CDEF) box that allows Kakadu to correctly decode the image.
- Add support to ECW Header Editor to remove CDEF box for JP2 files specified above.
- Fixed reference counting to be more accurate for ECW cache.
- Fixed conflict on Linux with TinyXml symbols.
- Fixed issue where the low memory encoder, may call input tiles multiple times.
- Fixed CBlockingDecoder throwing exception in constructor.
- NCSOpenFileView failed on specific ECW file on Visual Studio 2017 x64 build, fixed with MSVC 141 update (optimization issue).
- Missing headers from Mac and Linux install (NCSStringUtils.h and NCSXmlUtils.h).
- Updated XCode projects to correct target build directory.
- Resolve output warning in CPacket::ParseHeader when decoding j2k conformance file.
- Resolve issue where SPOT JP2 files were not returning the correct EPSG code.
- Fixed crash in cexample7, fixed occasional read error in cexample6.

Version 5.4

Released: January 2018

- Many optimizations in JPEG 2000 codec, code-streams used in NITF 2.0 are faster to decode (NEPJ, NJPE).
- Several JP2 compliance issues resolved.
- Removed support for GCC libstdc++ standard library on Max OS X (no longer supported by Apple, requires 10.6 or higher).
- Added support for new C++11 ABI on GCC 5 or higher.
- Removed support for 32-bit binaries on Linux platform.
- Added support for Microsoft Visual Studio 2017.
- Removed support for Microsoft Visual Studio 2012.
- Added run-time platform support for Microsoft Windows Server 2016.
- Added support for AMD Ryzen CPU architecture.
- Bug fixes and speed optimizations.

Version 5.3

Released: May 2015

- Added support for Microsoft Visual Studio 2015.
- Removed support for Microsoft Visual Studio 2008 and 2010.
- Removed support for Windows CE.
- Added support for GCC 5 (Linux).
- Added support for Red Hat / Cent OS 7.x
- Removed support for Red Hat / Cent OS 5.x



- Added support for acceleration using AVX2 and FMA3 on supported CPUs.
- Added heap management options.
- Improved examples (multithreading, JNI).
- Bug fixes and speed optimizations.

Version 5.2

Released: January 2015

- New capability to update a region within an ECW version 3 file (and thus negating the need to recompress the entire scene).
- Platform support:
 - Addition of 64-bit ARM support (arm64-v8a) on iOS
 - Dropped development and deployment support for Windows XP
- Compiler support:
 - Addition of Microsoft Visual Studio 2013 (v12.0)
 - Deprecation of Microsoft Visual Studio 2008 (version 9.0)
- ECW 16 bit and general JP2 SSE optimizations (x86/x64).
- Assembly manifests have been added for Windows DLLs.
- Significant performance and memory improvements decoding certain format JPEG2000 images.
- Ability to open/browse images from local ECWP disk cache when device is offline.
- New Java JNI bindings on all supported platforms.
- New counters for read-time, write-time and reassembly-time have been added to highlight compression bottlenecks.
- Re-assembly time now considered when calculating encoding progress. For large compression jobs, it will no longer stay at 100% complete until assembling final output file.
- ECW v3 uint16 compression now uses per-band image statistics for more accurate compression results.
- Lower memory requirements in decoder on mobile.
- Many bug fixes and speed optimizations.

Version 5.1

Released: March 2014

- First release of new Mobile licensing tier.
- Expanded platform support to include
 - MacOS X
 - Android (ARM & x86)
 - iOS
 - Windows CE
- JPEG2000 specific
 - Image structure metadata can now be retrieved using the GetParameter/SetParameter calls
 - Opacity band can now be set at a bit depth greater than 1
 - Fixed decoder crash
 - reading invalid files with a CUUIIDBOX m_nLength less than zero
 - p0_03.j2k compliance dataset
 - reading files with metadata box tag "meta"
- Windows specific
 - x64 redistributables now 25% smaller than v5.0

- MT / MD libraries are now included
 - Resolved CTL String conflict on VC100
 - Missing header files such as NCSRenderer.h
- Documentation
 - New HTML API Documentation now included
 - User Guide has been re-written and vastly improved
- Many bug fixes and optimizations including,
 - For UINT16 output, the Opacity band could introduce “salt-and-pepper” artefacts
 - Sub-sampling datasets using bilinear resampling with an Opacity channel incorrectly blended the background colour along the dataset edge
 - Improvements and bug fixes to examples, cexample03, cexample10, dexample4
- Performance
 - Single-threaded decoding across platforms increased +7-9% over 5.0

Version 5.0

Released: May 2013

- New ECW version 3 file format
 - 16-bit support for ECW v3.
 - Native opacity channels.
 - Custom metadata storage with native file info, statistics, and RPC support.
 - NULL block support. The format can efficiently store a NULL block where there is no data rather than encoding a black (0) block. This saves on storage and processing time.
- ECWP version 3 streaming for all ECW and JPEG 2000 files. Implementation is more efficient and is a stateless streaming protocol.
- New scaling API to automatically scale imagery between different bit depths (e.g. 16-bit to 8-bit for display purposes).
- Auto dynamic range calculation for improved visual imagery when viewing with non 8-bit data.
- Higher accuracy YUV to RGB conversion for improved decoding image quality.
 - Expanded SSE usage for improved performance throughout.
- GeoTIFF key/tag geo-referencing support for ECW v3 (similar in concept to GeoJP2).
- Removed external dependency on Intel Thread Building Blocks libraries.
- Removed external dependency on log4cpp.
- Added support for the Linux platform (32- and 64-bit).
- The ECW library is now a single unified library (no more NCSUtil or NCSCnet libraries).
- Removed read-only and read-write SDKs, one unified library now enables compression with OEM key license code.
- Added static libraries on all supported platforms.
- Simplified "C" API with consistent naming conventions, removed deprecated functions, and general API clean-up.
- Binaries for Microsoft Visual Studio 2008, 2010, 2012 and GCC 4.4 or higher on Linux.
- Optimized nearest neighbour and bilinear resampling routines to fix issues in 4.x where super-sampling would produce inaccurate results. Bilinear resampling is now also faster.
- Upgraded GDAL drivers for version 5.0 compatibility. Full decompression and compression support (with valid license), with support for ECWv3 metadata and ECWP version 3.
- Many JPEG 2000 performance and memory optimizations for different profiles.
- Removed J2i index files due to the above JP2 enhancements.
- New examples to demonstrate new features.
- New multi-threaded ECW JP2 check program included to check/validate files.
- Resolved issue with over-compression when writing lossy uint16 JPEG2000 files.



- Libraries are now fully Unicode on all platforms.

Version 4.3

Released: September 2012

- Improved the detection and handling of corrupt ECW files.
- Improved the stability and decoding performance of JPEG 2000 files.
- Improved the usage and creation of J2I files. J2I files are no longer created for multi-tile JPEG 2000 images, and are re-created when their associated JPEG 2000 file are modified.
- Stability enhancements to ECWP when decoding multiple streams simultaneously.
- Better estimation and handling of the memory required for compression.
- Fixed an issue decoding some NITF file streams.
- Many bug fixes and optimizations.

Version 4.2

Released: January 2011

- Maintenance release with many bug fixes and performance optimizations.

Version 4.1

Released: December 2010

- ISO standard "GML in JP2" geo-referencing support.
- Opacity channels for ECW and JPEG 2000 format (local and ECWP).
- Configurable persistent local cache of compressed blocks for streaming ECWP files (ECW and JPEG 2000).
- Configurable J2I index files (lower memory overhead and speed up decoding of many JPEG 2000 profiles).
- Configurable decoding of JPEG 2000 files with quality layers.
- Configurable buffered IO for JPEG 2000 reader.
- ECW decoder up to 4 times faster than version 3.x.
- Faster decoding and serving of JPEG 2000 format (including NPJE and EPJE profiles).
- Configurable "resilient" or "fussy" decoder modes for more robust ECW decoding.
- Higher scalability when using many file views.
- New C++ APIs.
- New caching algorithms.
- New configuration preferences.
- API compatible with "C" interface of version 3.x release.
- Full native 32 and 64-bit platform support (Windows).
- Microsoft Visual Studio 2008/2010/2012 support (Windows).
- Many bug fixes and optimizations.

Version 3.3

Released: September 2006

- Support for building on UNIX platforms using GNU autotools.
- Fix for threading issues on UNIX platforms.



- Fix to a decoding problem on big-endian architectures.
- Sample code with build files added to the distribution.
- Fix for a very minor bug in lossless compression.

Version 3.0

Released: September 2004

- Added ISO standard JPEG 2000 support.

Version 2.0

Released: September 2000

- Added many new features.
- Enhanced the ECW format (version 2).
- Added ECWP streaming.
- Added compression to the SDK.

About Image Compression

Introduction

Digital imagery is becoming more and more ubiquitous as time goes by. With the proliferation of means whereby image data can be obtained (digital cameras, satellite imaging and image scanning) there is now a vast amount of image data in use, all of which consumes valuable storage and bandwidth resources. The need to use data-store and bandwidth resources more efficiently is what drives the field of image compression. Image compression refers to a whole raft of techniques to encode image data for the purpose of reducing its size for easier transmission or persistence. A compressed image has undergone such encoding. The goal of an image compression scheme is to achieve the maximum possible degree of image file exchange and storage efficiency whilst preserving a minimum level of fidelity in the image that results after reconstruction from the compressed format.

Currently the most effective compression techniques that have been found for imagery employ frequency transforms, and of these, the most effective are wavelet based, employing the discrete wavelet transform to process a digital image into sub-bands prior to quantization and coding. Wavelet based compression results in very high compression ratios, whilst maintaining a correspondingly high degree of fidelity and quality in a reconstructed image. With advances in the processing power of ordinary computers, a compressed image may be used almost anywhere an uncompressed image can; the image, or required section of the image, is simply decompressed on the fly before being displayed, printed or processed.

Typically, a colour image such as an air photo can be compressed to less than 5% of its original size (20:1 compression ratio or better). At 20:1 compression, a 10GB colour image compresses down to 500MB in size. Images with less information can achieve even greater compression ratios. For example, ratios of 100:1 or greater are not uncommon for compressed topographic maps. Because the compressed imagery is composed of multi-resolution wavelet levels and there is less data to read you can experience fast roaming and zooming on the imagery on all types of storage media. This chapter discusses image compression issues, and describes the ECW (Enhanced Compression Wavelet) method.

Lossless or Lossy Compression

Lossless compression provides a compressed image that can decompress to an identical copy of the original image. Sometimes, this is also referred to as “numerically lossless” compression. This perfect reconstruction is the main advantage of lossless compression and still achieves 2:1 compression ratios. Numerically lossless is usually required where mathematical analyses, such as remote sensing or photogrammetric applications, require precise pixel digital numbers (DN’s) to be preserved.

- ❗ Note: Images are only considered lossless when compared at dataset resolution. Comparing super-sampled views may result in differences caused by resampling techniques used to sample the image at a different resolution.

Lossy compression provides a compressed image that can decompress to an approximate copy of the original image. Lossy compression sacrifices some data fidelity in order to achieve much higher compression rates. These higher compression ratios and faster decompression performance and lower storage requirements are the main advantages of lossy compression.



Visually lossless compression is the point at which lossy compression is perceptually indistinguishable from numerically lossless compression. Ultimately, this is the ideal point of any type of lossy image compression algorithm, because it gives you the perfect balance between quality, speed, and file storage savings. Both ECW and JPEG2000 formats can achieve visually lossless rates, but the rate is subjective and depends on the type of input image and the person using the compressed imagery. For example, the general public is unlikely to distinguish the quality difference between 2:1 lossless, 10:1, and a 30:1 lossy compressed image, whereas remote sensing scientists certainly will.

Please contact Hexagon Geospatial for more information about suggested target rates for your situation.

- Note: For more in-depth analysis of lossy compression and the concept of visually lossless compression, watch [How to Save Time and Storage Space with Industry-Leading Imagery Compression](#) webinar.

Wavelet Based Encoding

The most effective form of compression today is wavelet based image encoding. This technique is very effective at retaining data accuracy within highly compressed files. Unlike JPEG, which uses a block-based discrete cosine transformation (DCT) on blocks across the image, modern wavelet compression techniques enable compressions of 20:1 or greater, without visible degradation of the original image. Wavelet compression can also be used to generate lossless compressed imagery, at ratios of around 2:1.

Wavelet compression involves a way of analysing an uncompressed image in a recursive manner. This analysis results in a series of sequentially higher resolution images, each augmenting the information in the lower resolution images.

The primary steps in wavelet compression are:

- Performing a discrete wavelet transformation (DWT)
- Quantization of the wavelet-space image sub-bands; and then
- Encoding these sub-bands

Wavelet images are not compressed images as such. Rather, it is the quantization and encoding stages that provide the image compression. Image decompression, or reconstruction, is achieved by completing the above steps in reverse order. Thus, to restore an original image, the compressed image is decoded, de-quantized, and then an inverse discrete wavelet transformation (IDWT) is performed.

Wavelet mathematics embraces an entire range of methods, each offering different properties and advantages. Wavelet compression has not been widely used because the DWT operation consumes heavy processing power, and because most implementations perform DWT operations in memory, or by storing intermediate results on a hard disk. This limits the speed or the size of image compression. The ECW wavelet compression uses a breakthrough new technique for performing the DWT and inverse-DWT operations and along with the ECWP transmission protocol is recognized under US patents [6633688](#), [6442298](#), and [6201897](#). For more information about the technical implementation of the format, refer to the patents. ECW makes wavelet-based compression a practical reality.

Because wavelet compression inherently results in a set of multi-resolution images, it is suitable for working with large imagery to be viewed at different resolutions. This is because only the levels containing those details required for viewing are decompressed.

ECW Compression

ECW is an acronym for Enhanced Compressed Wavelet, a popular standard for compressing and using very large images. The primary advantage of the ECW technique is its superior speed. ECW is faster for several reasons:

- The ECW technique does not require intermediate tiles to be stored to disk and then recalled during the DWT transformation.
- The ECW technique takes advantage of CPU, L1 and L2 cache for its linear and unidirectional data flow through the DWT process.

The ECW speed advantage is exploited for more efficient compression in several ways:

- ECW employs multiple encoding techniques. Once an image has gone through DWT and quantization, it must be encoded. The ECW technique applies multiple, different encoding techniques, and automatically chooses the best encoding method over each area of an image. Where multiple techniques are equally good, ECW chooses the method that is fastest to decode.
- ECW uses asymmetrical wavelet filters. Because of its speed, the ECW compression engine can use a larger, and therefore slower, DWT filter bank for DWT encoding. This enables smaller, faster inverse DWT filters to be used during decoding. Therefore, the decoding of ECW imagery is much faster. ECW uses a 15 tap floating point filter bank for DWT compression, and a 3 tap integer-based filter bank for the inverse DWT decompression.

Even with the additional processing carried out as described above, the ECW compression is still at least 50% faster at compressing images than other compression techniques, when measured on the same file, on the same computer.

Licensing

Overview























The SDK is distributed as a single unified library containing both the compression and decompression functionality. You must choose the appropriate license and agree to the EULA when installing and ensure you have a valid OEM key to enable the compression functionality.

To better suit customer needs, Hexagon Geospatial now determines licensing according to three characteristics:

- Application type (desktop, server, or mobile)
- Capability (read-only or read-write)
- Redistribution rights (end-user or redistributable)

For read-write licenses (encoding), there is a further licensing tier based on the image size in gigapixels (1, 10, 100, or 1000 gigapixels). For mobile platforms, decoding is free up to 1 gigapixel input size for local images on the device, but free for unlimited ECWP images.

ERDAS ECW JPEG2000 SDK v5.x Licensing

PLATFORM		Read-only	Read-Write
SERVER	    	 <u>End-user</u>  <u>Redistributable</u>	 <u>Redistributable</u>  1 gigapixel 10 gigapixels 100 gigapixels 1000 gigapixels
	     	 <u>Redistributable</u> \$0 / freely available	 <u>Redistributable</u>  1 gigapixel 10 gigapixels 100 gigapixels 1000 gigapixels
	  	 <u>Redistributable</u>	



In summary:

- No license fee required:
 - Desktop Read-Only Redistributable
 - Mobile Read-Only Redistributable (local decoding restrictions apply)
- OEM licenses:
 - Desktop Read-Write Redistributable
 - Server Read-Only End-user
 - Server Read-Only Redistributable
 - Server Read-Write Redistributable

The license price will vary according to these characteristics. Please contact your local Hexagon's Geospatial division sales representative to discuss your requirements or to request a quote.

Prospective customers for the redistributable licenses must specify the product they want to license and provide any relevant platform requirements.

Understanding Gigapixel Limitations

The size of a file in gigapixels can be calculated by multiplying the number of rows (height) by the number of columns (width). For example an image with 20,000 rows and 20,000 columns would equal 400,000,000 pixels or 0.4 gigapixels. Note the number of bands are not included in the gigapixel calculation.

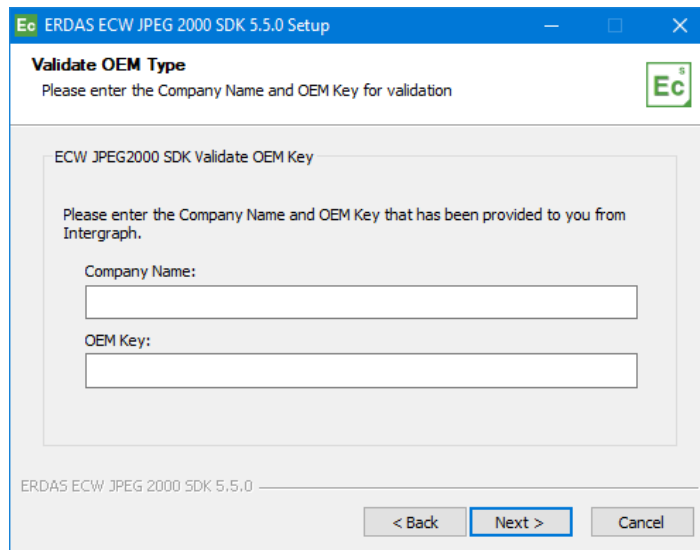
The SDK Read-Write licenses are available in

- 1 gigapixel
- 10 gigapixels
- 100 gigapixels
- 1000 gigapixels

Decompression in the SDK remains unlimited on Desktop, Server licenses and limited to 1 gigapixel on Mobile. An OEM Key is required to unlock unlimited decoding on mobile platforms.

Activating a License

All OEM license types as of the v5.1 installer now require a confirmation of a license key before installation will proceed. When one of these license types is selected, you must enter a valid company name and OEM Key.



Each major and minor release of the SDK requires a new key to be generated. V4 Keys will not work with v5.0 and v5.0 keys will not work with v5.1. Contact Hexagon Geospatial Support for more information or a new license key for customers on Software Maintenance.

When encoding (or decoding over 1 gigapixel on mobile platforms) you are required to enter your license within the application to unlock these features. The license key must be embedded in such a way that end-users cannot extract or reuse the key outside of your licensed application. OEM Key's cannot be redistributed publicly in any circumstances.

To validate a license when operating in the C API please refer to the [NCSCCompressSetOEMKey](#) method.

To validate a license when operating in the C++ API please refer to the [CNCSFile::SetOEMKey](#) method.

System Requirements

The SDK supports a variety of developer and target platforms each with varying levels of development environments. The system requirements for development and runtime platforms are described below for each license level.

Supported Platforms by License Level:

	Platform	Windows						Linux		MacOS	iOS		Android	
	Architecture	x86			x64			x86	x64	x64	ARM	ARM64	X86	ARM
	Development environment	VC12.0	VC14.0	VC14.1	VC12.0	VC14.0	VC14.1	GCC 4.6+	GCC 4.6+	XCode 5+	XCode 5+	XCode 5+	NDK 4.7+	NDK 4.7+
ECWSDK License Level	Desktop	✓	✓	✓	✓	✓	✓	✓	✓	✓				
	Server	✓	✓	✓	✓	✓	✓	✓	✓	✓				
	Mobile										✓	✓	✓	✓

Development Platforms

Windows

- Windows Vista (32 & 64 bit)
- Windows 7 (32 & 64 bit)
- Windows 8/8.1 (32 & 64 bit)
- Windows 10 (32 & 64 bit)

Linux

- Red Hat Enterprise 6.x and 7.x (64 bit).
- Cent OS 6.x and 7.x (64 bit).
- Most modern Linux distributions should work but are untested.

MacOS X

- MacOS X 10.7 or higher.

Android, iOS

- The host platform as specified in the relevant platform SDK.



Development Environments

The following compilers and IDEs are supported when targeting the following platforms:

Windows

- Microsoft Visual Studio 2013 (v12), 2015 (v14) & 2017 (v14.1) (32 and 64-bit)
- Dynamic (DLL) libraries are supplied as well as static libraries compiled for both a dynamically linked runtime (/MD and /MDd) and a statically linked runtime (/MT and /MTd). All targets supply both debug and release configurations.
- Apache Ant is required to build the Java JNI examples (optional).

Linux

- GCC v4.4.6 or higher (32 & 64-bit), shared libraries only
- GCC 5.0 or higher (32 & 64 bit), static and shared libraries with pre-compiled binaries for both the new C++ 11 ABI (_GLIBCXX_USE_CXX11_ABI=1) and the old (GCC 4) ABI (_GLIBCXX_USE_CXX11_ABI=0). For more information on the dual ABI support in GCC 5, see https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html.
- Debug and release variants are supplied for all targets.
- Any development environment supporting GCC is considered viable with the SDK.
- Apache Ant is required to build the Java JNI examples (optional).

MacOS X

- MacOS X v10.7 or higher
- XCode v5 or higher for 64 bit only
- Static Objective C library bindings are supplied, as well as a dynamic shared library (C/C++) and static framework (Objective C)
- Only the newer CLANG/LLVM libc++ standard library is supported (not libstdc++ as this has been deprecated by Apple)
- Apache Ant is required to build the Java JNI examples (optional)

Android

- Android development is independent of the development platform architecture and can be developed on Windows, Linux or MacOS X.
- Android Studio 3.5 or later is the preferred method for the development of Android applications with the SDK.
- Static SDK libraries (libNCSEcw.a) are supplied as are Java JNI bindings (libNCSEcwJNI.so)
- Binaries for armeabi-v7a, arm64-v8a, x86 and x86_64 are supplied
- The SDK was built with the NDK (side-by-side) v20.0.5594570
 - minSdkVersion 21
 - targetSdkVersion 26
 - ANDROID_STL c++_static is used (default)

iOS

- MacOS 10.7 or higher
- XCode 5.0 or higher
- A static framework is supplied (Objective C)
- 64 bit binaries (armv7, arm7s, arm64) and i386 binaries for emulator
- Only LLVM libc++ standard C++ library is currently supported



Runtime Platforms

The support runtime platforms for end user applications are listed below. Other platforms may also work, however are untested.

Windows

- Windows Vista, 7, 8 or 10 (32 and 64 bit).
- Windows Server 2016 or 2019 (64-bit).

Linux

- Red Hat Enterprise 6.x and 7.x (64-bit)
- Cent OS 6.x and 7.x (64-bit)
- Other Linux distributions are considered viable (64 bit)

MacOS X

- MacOS X 10.7 or higher (64 bit)

Android

- Android 2.3 or higher (ARM, ARMv7, ARMv8a, x86)

iOS

- iOS 7 (armv7, armv7s, arm64)
- iPhone Simulator 7 (i386)

Installation

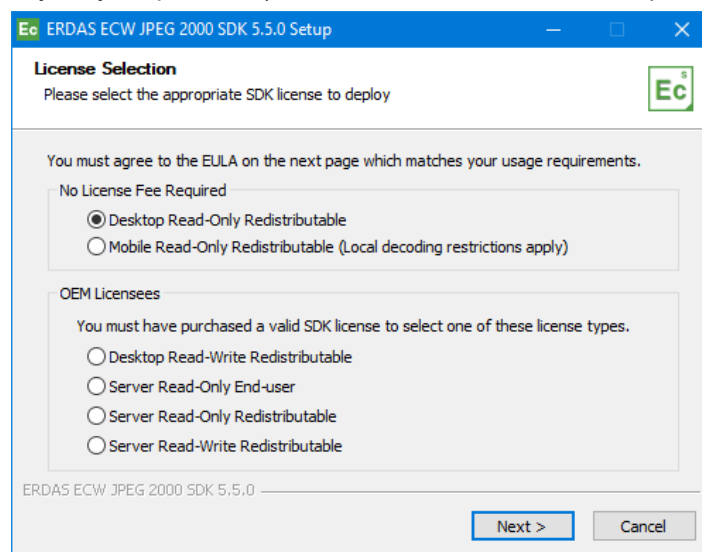
The SDK is available as three standalone installation packages, one for each runtime platform. Each run time platform has different included packages as shown below. Android is the only platform supported by all three installation packages

		Windows	Linux	MacOSX	iOS	Android
<i>Installer</i>	ECWJP2SDKSetup.exe	✓				✓
	ECWJP2SDKSetup.bin		✓			✓
	ECWJP2SDKSetup.dmg			✓	✓	✓

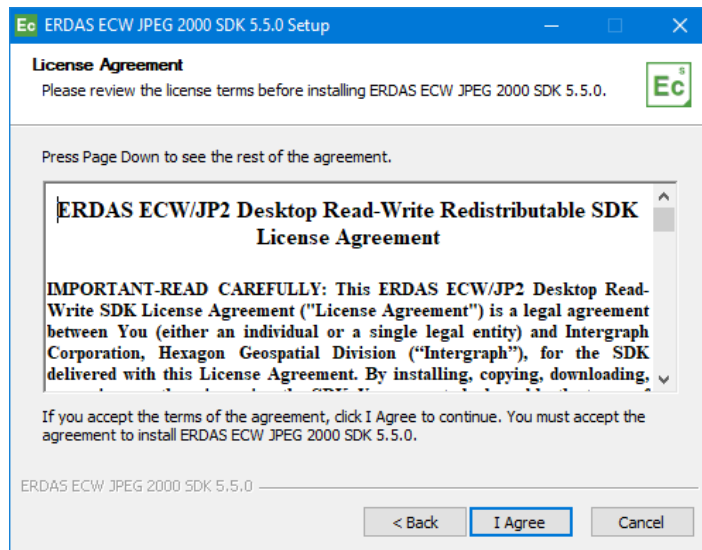
Windows

The installation steps shown below are for the Windows installer however both MacOSX and Linux installers have the same general workflow. The Linux installer is command line only.

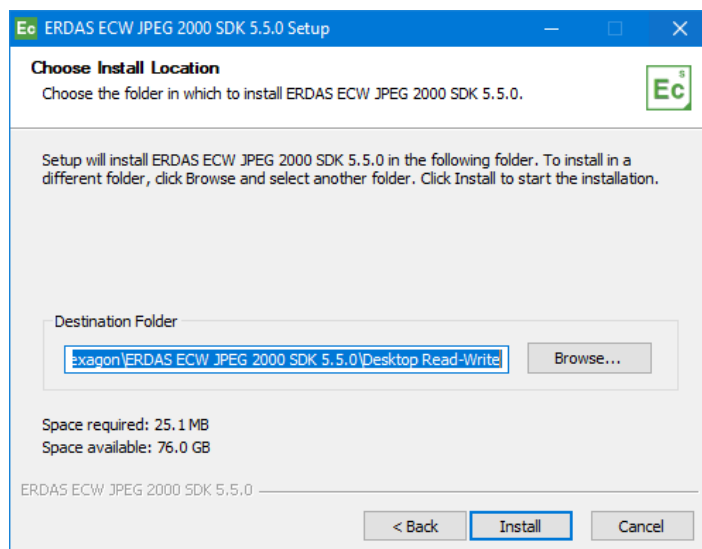
1. Run the setup binary for your platform (on Windows “ECWJP2SDKSetup_5.3.0.exe”):



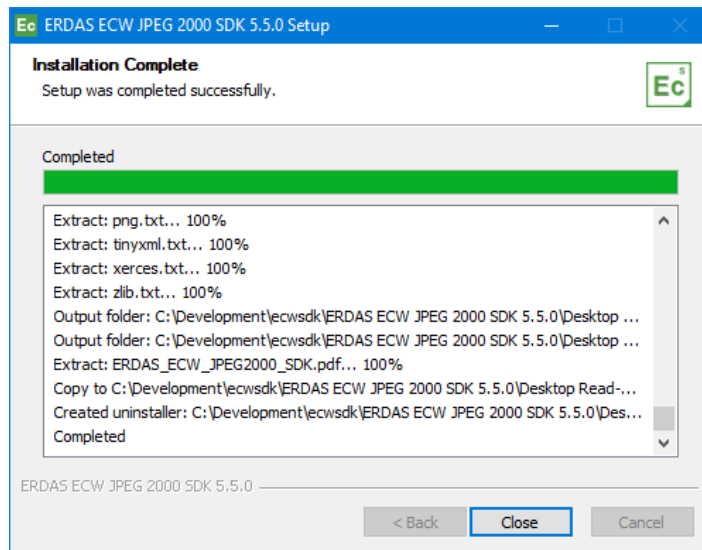
2. Select an appropriate license type and click **Next** to continue. Refer to the section on [Licensing](#) for more information about each license. To install multiple license levels, you must restart the installer and select an alternate complimentary license type.



3. You must accept the Hexagon End User License Agreement to install the product. Click **I Agree** to continue or **Cancel** to exit the installation.



4. Select a location to copy the SDK files to. Click **Install** to begin the installation process.



5. Once the installation is complete, click **Close**.

Linux

On Linux, the SDK is delivered as a script that needs to be run.

MacOS X

On MacOS X, the SDK can only be installed as the current user. Once the installation is complete, the user can run the script `IntegrateSDKWithXCodeFramework.sh` with elevated (root) privileges to create a system framework and integrate it with XCode.

Directory Structure

Depending on the license type chosen, the directory structure will contain different binaries and redistributables. Within the base SDK install directory one of the following sub-folders will be deployed,

- Desktop Read-Only
- Desktop Read-Write
- Mobile Read-Only
- Server Read-Only
- Server Read-Write
- Server Read-Only End-User

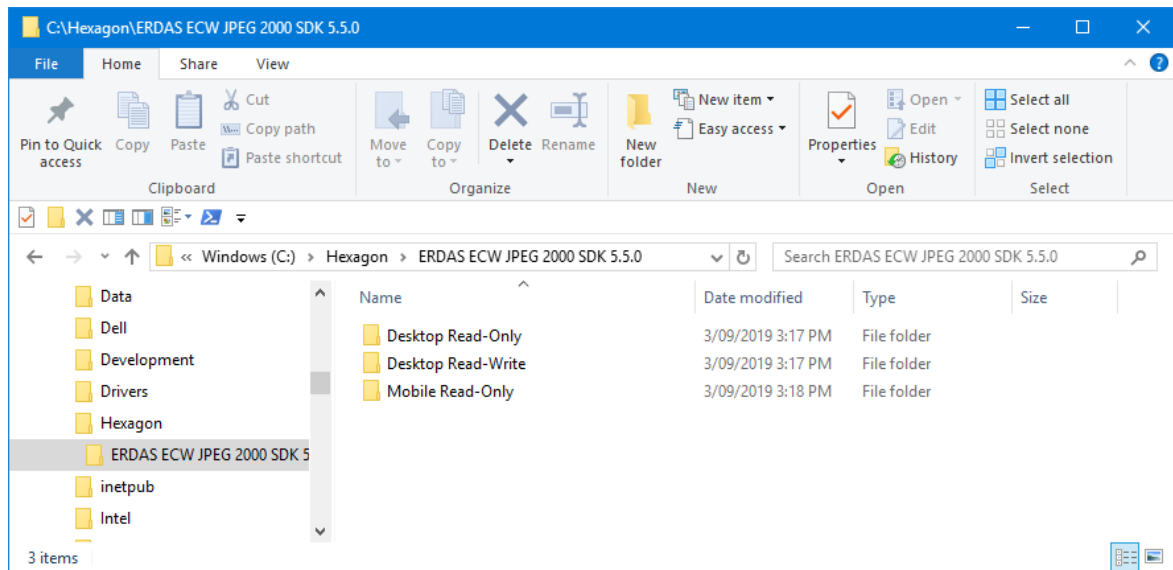


Figure 1 - Example Windows deployment with two licenses installed

The contents of these directories follow the same convention across platforms:

- **apidoc** – contains the html API documentation for the C, C++, Java (Android), Objective C (Max, iOS) and simplified C++ interfaces.
- **bin** – contains the platform binaries for your chosen compiler and operating system. On windows this contains the dynamic link libraries (.dll). On Unix platforms this directory may be empty.
- **etc** – contains data for mapping GDT datum and projection pairs to EPSG codes.
- **examples** – contains basic compression and decompression examples using the C, C++ and Java APIs.
- **include** – the headers files to include in your application when compiling.
- **lib** – the link libraries (static on all platforms and shared objects on Unix) to link your application against.
- **redistributable** – this folder contains the objects you may redistribute with your application for you platform.
- **testdata** – contains sample ECW and JPEG 2000 images used by the examples. These files can also be used to test the different capabilities of the SDK in your own applications.
- **Third-Party** – third party software license acknowledgements.
- **ERDAS_ECW_JPEG2000_SDK.pdf** – documentation.
- **EULA.rtf** – the End User License Agreement for the platform and license level you have chosen.

The directory structure for the different platform installers may vary due to platform dependencies. Windows has a much larger installation footprint than MacOS X and Linux installers due to the variety of the target compilers and options supported. Be sure to link against the correct versions of the libraries for your target compiler.

File Formats

The SDK supports ECW version 2, ECW version 3 and JPEG 2000 file formats. A high level functionality matrix can be seen below based on the features available in the ECWJP2 SDK.

Capability	ECW v2	ECW v3	JPEG2000
Region of interest	✓	✓	✓
Progressive display	✓	✓	✓
ECWP support	✓	✓	✓
Performance	highest	highest	Medium-high
Line compression	✓	✓	✓
Tile compression	✓	✓	✗
8-bit unsigned	✓	✓	✓
16-bit unsigned	✗	✓	✓
16-bit signed	✗	✗	✓
28-bit unsigned	✗	✗	✓
Visually lossless	✓	✓	✓
Numerically lossless	✗	✗	✓
Null block support	✗	✓	✗
Opacity band support	✓	✓	✓
Data statistics, histogram	✗	✓	✗
RPC storage	✗	✓	✗
Custom metadata	✗	✓	✓ ¹
Georeferencing	GDT	GeoTIFF Tags	GML in JP2, GeoJP2
Colour space support	Greyscale, RGB, Multiband	Greyscale, RGB, Multiband	Greyscale, RGB, Multiband
Largest known image ²	14 terapixels	14 terapixels	756 gigapixels

¹ Partial. Custom metadata can be written to JP2 UUID boxes however clients will have to explicitly understand the presence and structure of the contents.

² Compressed using ERDAS Image Compressor, as at January 2014.

ECW Version 2

- 8 bits (per channel) lossy RGB, Greyscale and Multiband images.
- Opacity channels (v4.2 SDK and above).
- Unlimited number of bands.
- Very high compression ratios while maintaining high visual quality.
- Single threaded scan-line based compression.
- Highly re-entrant decoder for efficient decoding of image data simultaneously in multiple threads.

ECW Version 3

- All of the features of ECW version 2 as per above.
- 16 bits (per channel) unsigned integer for RGB, Greyscale and Multiband images.
- Custom meta-data boxes (e.g. XML or binary objects) similar to the support in JPEG 2000. The user can get and set their own custom data and the SDK provides users a mechanism to implement extended file meta-data, statistics and RPC information (as defined below).
- Statistics (mode, median, min, max and histograms) stored for each band of data.
- RPC (Rapid Positioning Capability) information. RPC information involved various parameters (such as error, offset, scale and the coefficients) that are required to reconstruct an approximate transformation matrix. The SDK does not use these values to transform the image; the client should use this information to apply the transformation.
- Georeferencing using GeoTIFF tags. The georeferencing information is now stored in a custom meta-data box using the GeoTIFF key/value pair mechanism, in the same way as normal TIFF files. This makes implementation easier if previous GeoTIFF support is present. This data can also be edited using the header editor classes.
- NULL blocks add an additional capability for highly efficient storage. When compressing, if an area contains no data, the SDK can optimize this by not storing any block on disk and reconstructing a blank block on the fly when decoding. In version 2 format, the SDK would compress a "zero" block (typically black) that would compress to a very small amount of data, but would still be stored on disk. In version 3, no block is stored, and the decoder will reconstruct a NULL block dynamically based on the background colour of the image. This is much faster for decompression (as there is no disk access) and also is more efficient at compression so these blocks are never stored (increasing compression ratio for same quality image). This leads to much more efficient storage for situations like corridor mapping, where image data may only take up a small percentage of the full data extents.

JPEG 2000

JPEG 2000 is an international standard developed by the Joint Photographic Experts Group (JPEG). The JPEG image standard has found broad acceptance in digital imaging applications such as digital cameras and scanners, and the Internet.

JPEG 2000 is a substantial revision of the original JPEG standard. JPEG 2000 provides current and future application features and support, in addition to superior image compression. The feature set in JPEG 2000 includes:

- **Lossy and lossless compression:** JPEG 2000 provides lossy or lossless compression from a single algorithm. The lossless compression is within a few percent of the best (and most expensive) lossless compression available. Both lossy and lossless compression are available in a single code stream.
- **Progressive transmission:** JPEG 2000 supports progressive image code-stream organization. Such code-streams are particularly useful over a slow or narrow communication link. As more data is received, the transmitted image quality improves by some measure, such as resolution, size, spatial location, or image component. Within the compressed code-stream, JPEG 2000 can transmit image data in mixed dimensions of progressive measure.
- **Random access:** Spatial data is usually accessed randomly. The viewer examines the image in an ad hoc or random sequence, according to their interest at that time. JPEG 2000 provides several mechanisms for spatial or "region of interest" access, through varying resolution granularities.

- **Sequential encoding:** Low memory applications can scan and encode an image sequentially from top to bottom, without buffering the entire image in memory, using the JPEG 2000 standard. This build-up is achieved through a progression or tiling by spatial location through the image.
- **Domain processing:** JPEG 2000 processes compressed domains with scaling, translation, rotation, flipping and cropping capabilities.
- **Seamless and unified compression:** The unified compression architecture of JPEG 2000 enables seamless image component compression from 1 to 28 bits deep. This provides superior compression performance with continuous tone colour and grey scale images, as well as bi-level images.
- **Low bit rate performance:** JPEG 2000 delivers a substantial performance improvement over JPEG under low bit rate conditions, maintaining image fidelity.
- **Bit-error resilience:** JPEG 2000 provides integrity checks and block coding mechanisms to detect and rectify errors within coding blocks. This makes JPEG 2000 a strong choice for applications requiring robust error detection and correction.

JPEG 2000 can operate in four modes: hierarchical, lossless, progressive, or sequential. These modes are flexibly specified within the JPEG 2000 standard, which allows complex interactions between them, such as mixing hierarchical and progressive methods within a code-stream. Quality and resolution are both scalable, with different granularities corresponding to each level of access in an image. As a viewer randomly selects spatial regions, they can be transmitted and decoded at varying resolution and quality levels. Maximum resolution and size is chosen at compression time, but subsequent decompression or recompression can provide any level of image quality or resolution, up to the compression threshold. For example, an image compressed losslessly with JPEG 2000 can be subsequently decompressed at some lesser resolution to extract a lossy decompressed image. This extracted lossy image is identical to the image obtained when lossy compression is used on the original image. Therefore, you can decode and extract desired images without needing to decode the entire code-stream or source image file. The selected subset of image data will be identical to that obtained if only the selected data had been compressed in the first instance.

- 📘 Note: More information on the JPEG 2000 file format can be found at <https://jpeg.org/jpeg2000/>

NITF

NITF is an acronym for the National Imagery Transmission Format Standard (NITFS). The NITF file format works as container for a suite of standards for the storage and transmission of images and imagery related information. The SDK includes support for encoding and decoding the code-streams compliant with the NPJE and EPJE profiles specified within the NITF container (JPEG 2000 code-streams). It does **not** however include an implementation of the NITF container format itself. For an implementation that uses the SDK, see the Geospatial Data Abstraction Library (GDAL) at <https://gdal.org/> or ERDAS IMAGINE for NITF Decoding of JPEG2000 embedded code-streams.

ECWP Streaming Protocol

The SDK supports a custom image streaming protocol known as the **Enhanced Compressed Wavelet Protocol** (ECWP). ECWP provides numerous advantages over typical raster image delivery because the protocol delivers the compressed data directly to the client. This effectively creates a distributed decompression environment where the APOLLO ECWP Server performs no expensive operations to resample, reproject and compress to JPEG etc. By offloading the processing to the client, they not only get progressive display as data is being buffered from the Server, but it also means the Server can support thousands of concurrent users. ECWP is not the same as HTTP Byte-ranging and includes many optimizations specific only to ECWP.

ECWP uses a URL notation to provide the client SDK with a server host and a path to the ECW file that is to be streamed from. For example:

```
ecwp://www.server.com/folder/file.ecw
```

The URL represents a virtual path on the server where the ECW file resides. ECWP also supports encrypted connections using SSL (if configured on the server) by using the "ecwps://" notation. All versions of ECW files and JPEG 2000 files can be streamed over an ECWP connection. When opening an ECWP path, the URL is passed to the SDK functions exactly the same as if a local file name were specified; there is no difference in any of the SDK API functions between local files and URL streams.

ECWP streams are built on top of the HTTP protocol, so generally work wherever web browser access is available. Streaming is more efficient and faster than other traditional types of rendering, as only the compressed blocks required to reconstruct a view are sent to the client, and the client caches data to accelerate decompression and rendering. Version 1 of the SDK supported the ECWP version 1 protocol, version 2.x to 4.x supported ECWP version 1 & 2, and version 5 of the SDK supports version 1, 2, and 3.

The SDK has no capability to serve ECWP streams; it can only act as a client. This functionality is included only with the ERDAS APOLLO server family of products.

ECWP Version 2

All public versions of the SDK support the version 2 protocol. This protocol is built on top of HTTP and is synchronous in its requests. It uses older style HTTP keep-alive and persistent connections that can cause some issues through proxy servers.

The SDK does **not** support the creation of ECWP Streams for serving, only for consuming on the client. ECWP streams are supported in conjunction with the ERDAS APOLLO Server family of products (all versions) and older ERDAS Image Web Server products.

ECWP Version 3

Version 5.0 of the SDK features a new version of the ECWP streaming protocol that is faster, more lightweight, stateless, works transparently through proxies and firewalls and uses connection pooling to be more efficient on the client. There are no changes to the way the client opens an ECWP connection, the SDK automatically attempts to open an image in ECWP v3 mode, if that fails it will fall back to version 2.

ECWP version 3 streams are supported only in ERDAS APOLLO 2013 and later versions.



Example ECWP resources

The following ECWP URLs can be used for testing ECWP v3 or ECWP v2 clients. The server is located in Atlanta, GA and its contents can only be used for demonstration or testing purposes only.

- ❗ <ecwp://demo-apollo.hexagongeospatial.com/demo/rotterdam10cm2005.ecw>
- ❗ <ecwp://demo-apollo.hexagongeospatial.com/demo/Landsat742.ecw>
- ❗ <ecwp://demo-apollo.hexagongeospatial.com/demo/15.jp2>

Further example datasets can be found on the ERDAS APOLLO demonstration page located at https://demo-apollo.hexagongeospatial.com/erdas-apollo/index.html#. You can also use Hexagon's free ERDAS ER Viewer application to open or stream compressed ECW or JPEG 2000 files, or the free ECW and JPEG 2000 viewer that is shipped with the GeoCompressor application. These can be downloaded from <https://download.hexagongeospatial.com>.

Features

ECWP Persistent Local Cache

As of version 4.1 of the SDK, client side caching is available and can be deployed by your application. The ECWP persistent local cache enables compressed data blocks to be stored on the client's local disk. Before any future requests are made to the server the local cache will be searched. This lessens data transfer thus improving overall user performance. The application programmer can turn the cache on/off, set the maximum size of the cache, and specify the location using the `NCSGetConfig` and `NCSSetConfig` functions with the parameters: `NCSCFG_ECWP_CACHE_ENABLED`, `NCSCFG_ECWP_CACHE_SIZE_MB` and `NCSCFG_ECWP_CACHE_LOCATION` respectively. Consult the API documentation for full descriptions of these parameters.

If the cache reaches its allocated limit, the oldest data will be deleted from the cache to make room for the new data.

- ❗ Note: For security reasons, images secured via HTTPS or authenticated with a username and password are not cached locally by default. You can enable this using the `NCSCFG_ECWP_CACHE_ENABLE_SECURE` parameter.

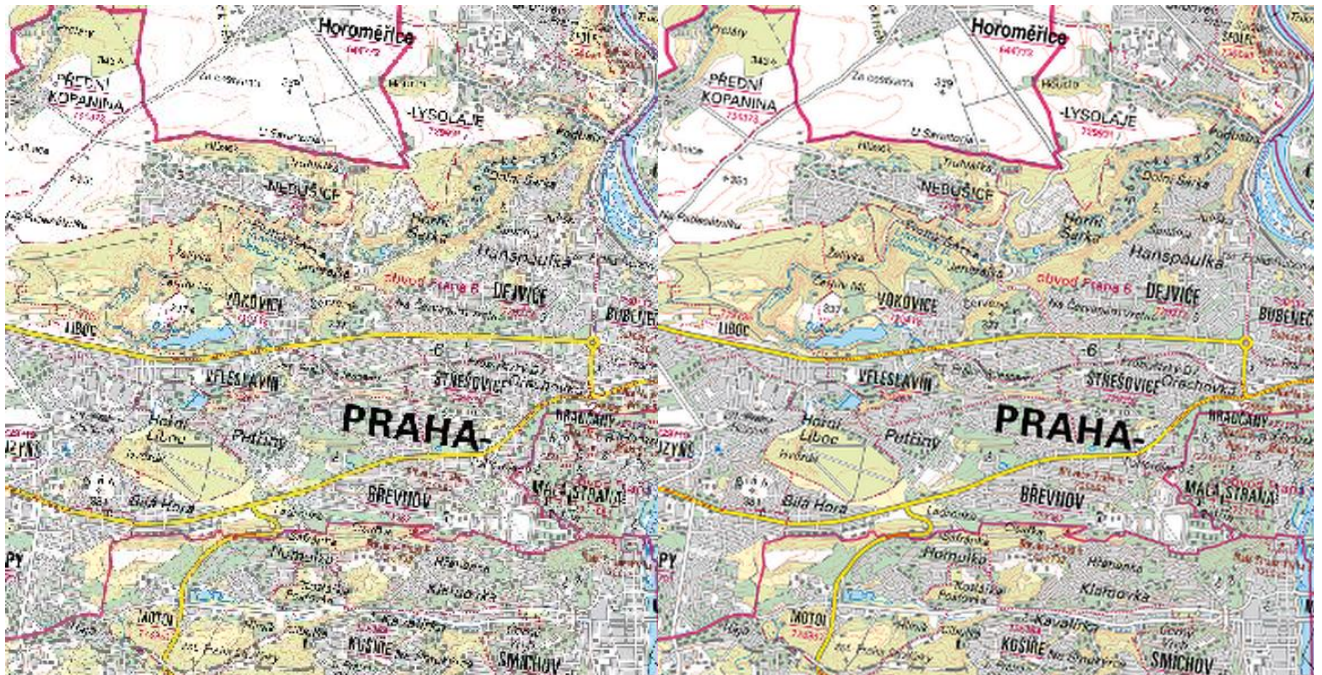
Image Resampling

When decoding, the SDK supports bilinear as well as nearest neighbour resampling. The sampling method can be set per view, the default remains nearest neighbour. Nearest neighbour resampling is faster but produces less accurate results, bilinear is slower but produces a higher quality image.

Additionally, the SDK now resamples automatically when requesting image views past 1:1 dataset resolution. Previously, the SDK would return an error on `SetView` and the application would have to resample the image accordingly.

To change the resample method in C++, set the class member `m_eResampleMethod` of the `CNCSFile` class to

`NCS_RESAMPLE_NEAREST_NEIGHBOUR_INTERPOLATION` or `NCS_RESAMPLE_BILINEAR_INTERPOLATION` as required. This is currently not supported in the "C" API.



Nearest neighbour resampling

Bilinear resampling

Data Scaling

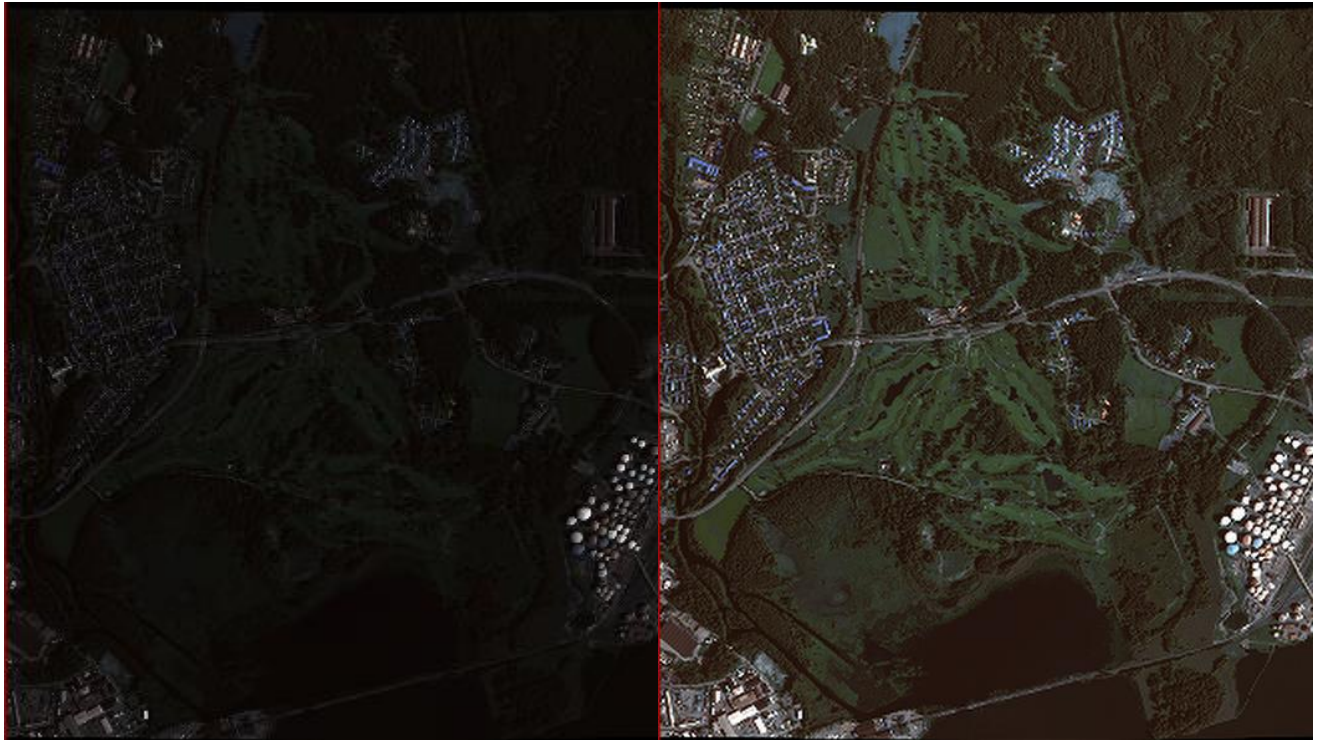
The SDK supports scaling data between different bit depths when decompressing. By specifying a different cell size in the output line or tile when decoding, the SDK will automatically scale the values to the numeric limits of the data type supplied. The scaling is just a linear scale between cell types and will be truncated if the source range exceeds the destination range.

To scale the data using a transform see the next section – Dynamic Range Calculation.

Dynamic Range Calculation

The SDK can apply a scaling algorithm based on statistics to deliver a more dynamic image when decoding. When applied, the SDK calculates fast/approximate statistics, then applies a transform utilizing the max and min value and optionally adds a clip percent. This is useful when displaying high bit depth images whose range sits within a small subsection of the available mathematical precision. For example, a 12-bit image stored in the lower 12 bits of a 16 unsigned integer image and be scaled to an 8 bit image using only the first 12 bits as input calculation for the transform. This produces a much more dynamic image, as can be seen in the below example.

To change the transform in C++, use the class method `SetTransform` of the `CNCSFile` class to set the minimum and maximum values, with an optional clip percentage. This is currently not supported in the "C" API.



No range adjustment

99% dynamic range adjustment applied

- Note: The dynamic range calculation is based on approximate image statistics and is designed for quick enhancements to give good results in most situations. For applications requiring more complex control, range adjustment or other enhancements are typically applied outside of the SDK.

Opacity Channels

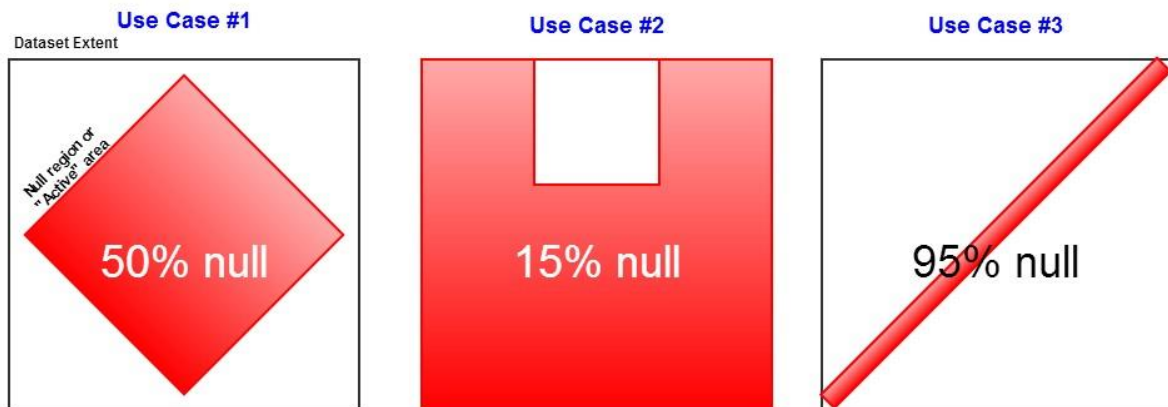
Opacity channels were introduced in version 4.1 of the SDK for ECW format, version 2 files. An opacity channel represents a “No Data” value for a particular pixel to indicate that the pixel should be either fully transparent or fully opaque (e.g. 0 or 255 for 8 bit data). Opacity channels are lossless and are usually stored as 1 bit (0 or 1). The work flow for compressing and decompressing ECW files with opacity is exactly the same to that of JPEG 2000 for previously releases of the SDK.

For an example on how to compress a file with an opacity channel, refer to compression example 1 for more information on implementing this feature.

Note: Older version of the SDK prior to 4.1 will ignore opacity channels in ECW files.

NULL blocks

A major improvement with ECW v3 is the introduction of null blocks that can offer further file storage savings and compression performance compared with ECW v2 or JPEG2000 without sacrificing image quality. The key criteria as to whether null blocks should be enabled is the relationship of the input data extent to the amount of null or no-data areas and the size of the input image. Generally speaking the higher amount of null area the greater the gains that enabling null blocks will provide.



An important observation in these examples are the first and third use-cases. Both have 4 vertices however clearly the percentage ratio to data is a lot higher in the third example at 95%. Therefore null blocks will provide the greatest benefit to the third image both in terms of additional file storage savings and compression speed. The first example will still benefit and is still a good candidate for null blocks however will not see as significant gains.

- Note: Null blocks in this way will always provide varying levels of optimizations depending on input as highlighted in the following examples.

Comparison

The following examples highlight real world gains enabling ECW v3 Null Blocks within the ERDAS Image Compressor that implements the Null block v5.1 functionality. This capability requires development effort to implement the region handling. See Compression Example 8.

Sydney Landsat scene

Input image

Dimensions: 15,221 x 14,661 px

(0.223 gigapixel)

Structure: 3 Band, RGB UINT8

Opacity band: false

Projection: EPSG:32656

Null region (yellow)

Null Vertices: 4

Ratio to data: 30.234%



Null blocks enabled		Null blocks disabled	
Hardware		Hardware	
Platform:	Windows 7 / Server 2008 R2	Platform:	Windows 7 / Server 2008 R2
CPU Model:	Intel(R) Core(TM) i7 CPU Q 740 @ 1.73GHz	CPU Model:	Intel(R) Core(TM) i7 CPU Q 740 @ 1.73GHz
CPU Cores:	8	CPU Cores:	8
Memory:	8,128.00 MB	Memory:	8,128.00 MB
Memory cache		Memory cache	
System:	512.00 MB	System:	512.00 MB
Read:	1,911.85 MB	Read:	1,911.85 MB
Write:	120.15 MB	Write:	120.15 MB
Threads:	8	Threads:	8
Precincts:	73376	Precincts:	73376
Total Blocks:	18344	Total Blocks:	18344
Data Blocks:	13251	Data Blocks:	18344
Null Blocks:	5093	Null Blocks:	0
Duration:	0 hours 0 mins 55 seconds	Duration:	0 hours 1 mins 20 seconds
Target Ratio:	15:1	Target Ratio:	15:1
Actual Ratio:	31.3:1	Actual Ratio:	30.7:1
Throughput:	11.5 MB / sec	Throughput:	8.0 MB / sec
Output Data		Output Data	
File Name:	f:\landsat-null.ecw	File Name:	f:\landsat-no-null.ecw
File Type:	ECW v3	File Type:	ECW v3
Data Writer:	ECW JPEG2000 SDK v5.1	Data Writer:	ECW JPEG2000 SDK v5.1
Dimensions:	15,221 x 14,661 px	Dimensions:	15,221 x 14,661 px
Structure:	4 Band, RGB UINT8	Structure:	4 Band, RGB UINT8
File Size:	20.39 MB	File Size:	20.79 MB

1. File savings: 400kb (~ 2% smaller)
2. Time savings: 25 seconds (~ 30% faster)

The small file size difference is expected in this example despite the 30% ratio to data because the input image is only small at 0.2 gigapixels. This means there are fewer resolution levels within the ECW file that in turn means there are fewer null blocks in the output. Irrespective of the small file size improvement, enabling null blocks increases compression speed by 30%, which can be significant depending on use-case, for example compressing thousands of images in batch.

Corridor mapping example

Input image

Dimensions: 372,535 x 477,806 px

(177.999 gigapixel)

Structure: 4 Band, RGB UINT8

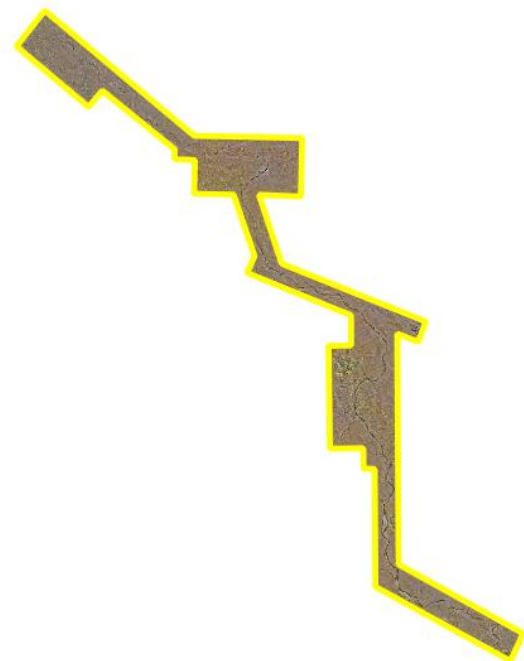
Opacity band: true

Projection: EPSG:28350

Null region (yellow)

Null Vertices: 33

Ratio to data: 89.731%



Null blocks enabled				Null blocks disabled			
Hardware				Hardware			
Platform:	Windows 7 / Server 2008 R2			Platform:	Windows 7 / Server 2008 R2		
CPU Model:	Intel(R) Xeon(R) CPU	E5410	@ 2.33GHz	CPU Model:	Intel(R) Xeon(R) CPU	E5410	@ 2.33GHz
CPU Cores:	8			CPU Cores:	8		
Memory:	16,380.00 MB			Memory:	16,380.00 MB		
Memory cache				Memory cache			
System:	512.00 MB			System:	512.00 MB		



Read: 2,319.47 MB

Write: 1,775.53 MB

Threads: 8

Precincts: 57967752

Total Blocks: 14491938

Data Blocks: 1506776

Null Blocks: 12985162

Duration: 0 hours 53 mins 45 seconds

Target Ratio: 15:1

Actual Ratio: 209.4:1

Throughput: 210.6 MB / sec

Output Data

File Name: g:\corridor-null.ecw

File Type: ECW v3

Data Writer: ECW JPEG2000 SDK v5.1

Dimensions: 372,535 x 477,806 px

Structure: 4 Band, RGB UINT8

File Size: 3,242.55 MB

Read: 2,319.47 MB

Write: 1,775.53 MB

Threads: 8

Precincts: 57967752

Total Blocks: 14491938

Data Blocks: 14491938

Null Blocks: 0

Duration: 3 hours 41 mins 19 seconds

Target Ratio: 15:1

Actual Ratio: 169.3:1

Throughput: 51.2 MB / sec

Output Data

File Name: g:\corridor-no-null.ecw

File Type: ECW v3

Data Writer: ECW JPEG2000 SDK v5.1

Dimensions: 372,535 x 477,806 px

Structure: 4 Band, RGB UINT8

File Size: 4,011.66 MB

1. File savings: 769mb (~ 19% smaller)
2. Time savings: 2 hours 48 minutes (~ 410% faster)

This example shows the strengths of enabling null blocks. It has a relatively simple input region and a high level of null data of 89%. Unlike the previous example, we can now observe significant gains to both compression speed and file savings with no degradation to image quality.

J2I Index Files

With older versions of the SDK prior to 5.0, J2I index files are automatically created when a JPEG 2000 file stream is opened. Index files use the same base name as the file, with a ".j2i" extension. Creating an on-disk index file lowers memory consumption for large files (as the indexes do not have to be held in memory) and

can speed up decoding in certain circumstances where the index is complex or expensive to calculate on the fly. When opening a file for read, if no index exists, the open function will block until the index is created, then continue to open the file as normal. If an index file already exists, the SDK will use it directly.

- Automatic J2I index file generation can be turned off with the global configuration option `NCSCFG_JP2_AUTOGEN_J2I` in the API Reference chapter.

In version 5 and above, J2I files are no longer used as the JPEG 2000 decoding algorithm has been optimized and they are no longer necessary. Preference settings for J2I are now silently ignored.

Compression Methods

The SDK offers two interfaces for image compression, the older (v4 and below) scanline (single threaded) encoder and the newer (v5 and higher) tiled (multi-threaded) encoder. Each one is described below.

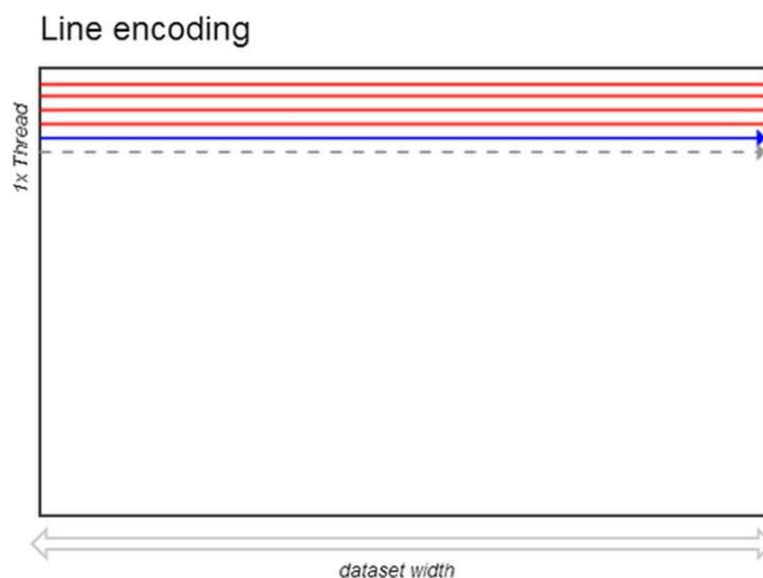
The two algorithms are suited for different situations. In order to determine the ideal method for your situation, benchmarks should be performed particularly where compression speed is an important measure.

Note: Output ECW files are binary identical irrespective of the compression method used.

Scanline Encoder

The scanline encoder represents a scanline based reader that reads each scanline from the top of the file, compresses and continues to the next scanline until the end (bottom) of the file. Scanline based encoding is single-threaded and cannot scale across multiple CPU cores, however it benefits from lower memory requirements to compress. This method is particularly suited for scanline structured input data such as strip based TIFF files, or file formats that do not perform well with multi-threaded readers.

When using the scanline encoder, the input is read from the top to the bottom of the image one scanline at a time and fed into the encoder.



Refer to compression examples 1 and 2 for more information on using the scanline encoder API.

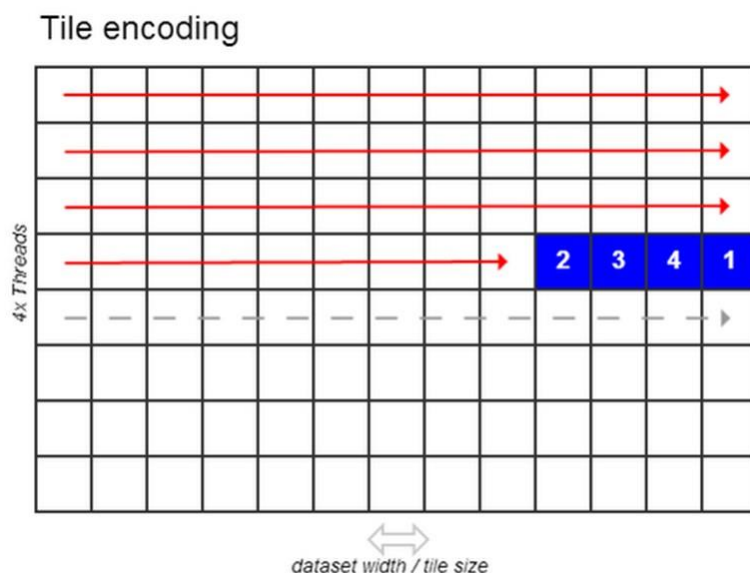
Tiled Encoder

The tile encoder is a new parallel algorithm introduced in the ERDAS ECWJP2 SDK v5 that reads input data in discrete tiles across multiple reader threads. Each thread processes independently in a thread-pool across the width of the dataset and is then repeated down the image.

The new algorithm results in efficient scaling across CPU Cores however it requires more memory to compress the same input as the scanline based encoder. There is a further trade-off with Disk I/O as the concurrent threads increase load and requires more data to be processed than line, increasing the likelihood of reaching a disk bottleneck. It's also possible that the input data readers have not been optimized for multi-threaded reading creating additional bottlenecks.

Fast I/O is a requirement in order to feed data to the multiple worker threads otherwise CPU utilization will be low and performance may be slower than the line algorithm. Where data I/O is sufficient, the tile algorithm can be more than 400% faster than line depending on hardware and input format used.

When using the tile encoder, small tiles (default 64x64) are read in parallel, moving across the image, then down the image until the entire input dataset has been read. Image tiles are read in parallel and run in a thread pool.



Refer to compression example 8 for more information on using the tile encoder API.

Block Size

Note that when compressing ECW files, the SDK expects a block size. This is the size of the tile that the underlying wavelet transform is processed and stored on. Smaller block sizes are more efficient for ECWP delivery over the internet. Larger block sizes may decode faster and produce higher compression ratios. The block size must be between 64 and 32768 inclusive, and must be a power of 2. If the block size does not meet this condition, the compression will fail. The default block size is 64.

Region Updates

New to version 5.2 is the ability to update a region of an ECW version 3 file with new imagery. This API is not available for ECW version 2 files or JPEG 2000 images. The API is based on the tile encoder API (described



above) and supports opacity channels and NULL block writing (e.g. it is easy to NULL out a region of a defined file).

Region update is useful when you want to update a small area within a very large mosaic without recompressing the entire file and can be substantially faster than recompressing the entire image. Care should be taken when updating mosaics so that the same type of data and same input range of the data is used in the new input imagery as was used in the original image. In this way, the update will be relatively seamless and maintain a consistent look.

Refer to compression example 13 for more information on updating regions of ECW files.

- ❗ Note: **GeoCompressor 2020** product supports region updates of ECW files. It can use multiple inputs (mosaic), optimize the update with NULL blocks, supports opacity channels and can also define the region using a shapefile. For more information go to the product page at <https://www.hexagongeospatial.com/products/power-portfolio/geocompressor>.

Development

This section describes the structure of the SDK and how to set up a development environment for your SDK implementations. Many of these descriptions are set out for the PC platform, however are still relevant for other operating systems. Refer to the section on [System Requirements](#) to determine the correct compilers and prerequisites for your system.

How Imagery is Accessed

Images consist of rows of data, and a number of columns of data, with one or more bands (values) of data at each pixel in the array of data. For example, a compressed image might consist of 200,000 rows x 300,000 columns x 3 bands for a Red-Green-Blue image. Your application simply requests a region to view, and the library does the rest. When working with imagery, your application opens one or more views to the image(s) desired. It then performs one or more `SetFileViews` for each view opened and reads imagery for each `SetFileView` area. Despite the huge size of the images that the SDK can process, the ongoing region-specific decompression of data is always transparent to your development process.

How to Read a View

The essential information and procedure for accessing a file view is as follows:

- What image(s) to view, process, display or print: The `NCSOpenFileView()` function opens a view into an ECW JPEG2000 image file. This image file can also be served from a local or remote ERDAS APOLLO, in which case the image would be defined by its URL.
- Obtain information about the image that has been opened. The `NCSGetViewFileInfo()` function obtains details about the size of the image, and its world coordinate system (size of pixels, map projection, and so forth). Set a desired view area into the image, and how large a display area is required for that view. The `NCSSetFileView()` function specifies the area you wish to view, and how large an area your application is using in its display.
- Read data from a view. You can use calls that return information in a `BIL` (Band Interleaved by Line), `RGB` or `BGR` format. For example, the `NCSReadViewLineBGR()` function returns data in an order that can be directly placed into a Windows bitmap.
- Close a view. You can close a view with `NCSCloseFileView()`. There are some additional functions, particularly when using the `Refresh` call-back interface into the library. However, the above outline gives the basic interface approach into the library.

The SetFileView Concept

The ECW JPEG2000 SDK viewing and decompression library is very powerful, in that it will present an image to you at a resolution that your application requires rather than the resolution that the original image presents.

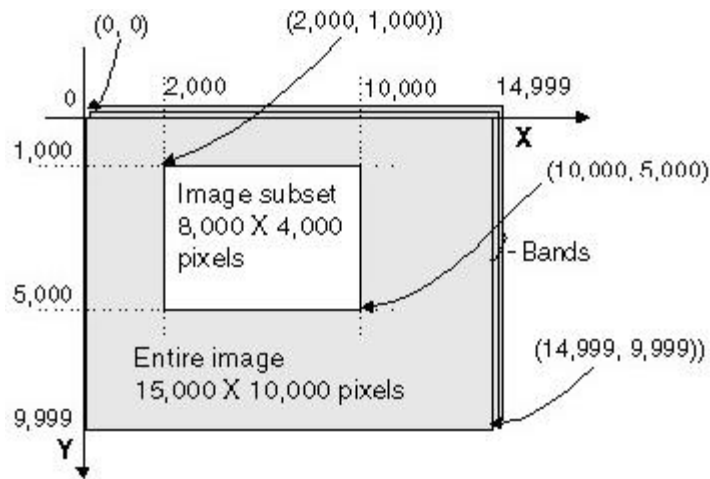
Consider the example of an application that currently has a window open that is 500x300 pixels in size, and you are opening an image that is 15,000 x 10,000 pixels in size. When displaying an overview of the image, (the entire image area), you probably would prefer not having to read all 15,000 x 10,000 pixels to display an overview of the image. The advantage is that when you call the `NCSSetFileView()` function, you can specify:

- The area of the image to view. This can be any area from the entire image size down to a smaller portion of the image. You specify this as the top left and bottom right coordinates of the required area. Since the SDK treats

each pixel in the compressed image as an area rather than a point, the bottom-most and right-most row and column respectively are included in the extracted view.

- The size in which your application requires the image to be displayed.
- The bands of information that you wish to view from the image.
- If the view is to be read using the blocking of refresh call-back interface. This is specified when you open a view, not when you set a view area for an opened view.

The following diagram illustrates how you would specify the coordinates of the area to be viewed:



For example, to view the entire image example above, you might do:

```
// a view of the entire image into a 500x300 view area
error = NCSSetFileView(pView,nBands, pBandList,
                      0, 0, 14999, 9999, 500, 300);
```

Whereas to view the smaller area of the image, you might do:

```
// a view of a portion of the image into a 500x300 view area
error = NCSSetFileView(pView,nBands,pBandList,
                      2000, 1000, 10000, 5000, 500, 300);
```

- ❗ *Note: For more accuracy you can call the `NCSSetFileViewEx()` function instead of `NCSSetFileView()`. This allows you to specify the world coordinates of a georeferenced image for the image view.*

Viewing Areas Smaller than your Application Window Area

You should not request an area from the image that is smaller than the window size. For example, if your window size is 500 x 300, this is the smallest view area you should request from the library. This is because, when zooming to sub-pixel levels (as in this example), it is faster to perform pixel zooming using higher level graphics operations. These can quickly zoom bitmaps using graphics hardware assist, rather than using a low level library such as the ECW JPEG2000 library to perform this operation for you.

Requesting Odd-Aspect Views

You can ask for odd-aspect views of imagery. For example, you could request the library to return the area from (1000, 2000) to (2000, 5000) into your window view area of 500x300. This might be desirable in cases where the original data is a non-square pixel size (seismic data is an example of this type of data). The library will automatically scale data in the X or Y direction to meet your requirements. To perform this automatically, your application should use the `NCSGetViewFileInfo()` to find out the world coordinate size for each pixel, and take this value into account when displaying imagery.

Selecting Bands from an Image File to View

A compressed image file may contain from one to any number of bands.

Typically a file will contain 1 (grayscale) or 3 (colour) bands, but not in every case (e.g. with perspectival imagery). When you perform a `SetFileView()`, you specify the number of bands to view, and the band numbers to view. For example, you might wish to read 3 bands from a 7 band compressed image, and you may wish to read band numbers 5, 4, and 2. You do this by indicating the number of bands (`nBands`) and allocating an array equal to this size, which is filled with the actual bands to read.

If your application is not performing any image processing functions, and is simply designed to display a good image regardless of the number of input bands, we recommend the following approach:

- For images with three or less bands, specify the number of bands in the image. For images with more than three bands, specify 3 bands as the number to view.
 - Select the first bands in the file. For example, use band 0 for a 1 band file, bands 0 and 1 for a 2 band file, 0, 1, and 2 for a 3 band file, and bands 0, 1, and 2 for a 20 band file.
 - Use one of the read line functions that will return data converted into a RGB view (the RGB or BGR calls). Use the BGR call if you are using Windows style bitmaps, as you do not have to perform any conversion on the image
 - In this case, the library will fill the RGB (or BGR) line array in a way to always ensure a good looking image. For example, it will automatically fill all of red, green and blue for an input view containing only 1 band, which ensures that a grayscale view will still appear correctly in your RGB or BGR based bitmap.
- Note:** When developing applications with the SDK, be aware that in keeping with programming convention, band numbering commences at zero. For example, in a three band RGB image, the first, second and third bands (red, green and blue respectively) must be specified as 0, 1, and 2 when writing an application built on the SDK.

Blocking Reads or the Refresh Call-back Interface

There are two ways to access images:

- **Blocking reads:** You perform a `SetFileView`, read the view, and perform another `SetFileView`, and so on. For each read of data, the library will block until all imagery is read from disk (or the network) before processing the data.
- **Refresh callback interface:** Multiple `SetFileViews` are performed whenever you choose, even if reading is not complete. Each set view will trigger an asynchronous call-back in a separate thread where you can then proceed to read the data. These happen multiple times *for each setview*, for example every time new data arrives from the network. Typically you would use this style of read when reading from an ERDAS APOLLO server over ECWP.

In the C API, when reading in progressive mode, you specify a function pointer as the call-back. To use blocking reads, you specify `NULL` as the call-back function when performing the `NCSOpenFileView()` call.



How the two methods operate, and when you might use the different techniques, depends upon your application. You can mix both methods (for different views) into the library at the same time. Typically, use the different approaches as follows:

When to Use Blocking Reads

Use blocking reads in the following situations:

- Your application is not threaded (there is no need to be able to perform updates of the image display within another thread).
- You are printing an image out to a printer (so your view area is large and static).
- You have a simple use for the SDK and do not want to unnecessarily debug multi-threaded logic.
- Your application cannot persist "state" information between each view request for an image.

When to Use Refresh Reads

Use refresh reads in the following situations:

- Your application can refresh the image view on demand.
- Your application is thread safe.
- Your application is highly interactive, for example roaming and zooming over imagery in real time, and needs to respond rapidly to user input.
- Your application is primarily designed to display imagery via the Internet using the ERDAS APOLLO technology, and may need to compensate for the varying latency of an Internet connection.

Blocking Reads

The `dexample1.c` program in the `examples\decompression\dexample1` directory demonstrates the use of blocking reads. This is the more conventional approach if the application wants to set a view area, read some imagery from that view area or set another view area.

This is called the blocking reads interface because when your application reads the imagery line by line for a view area, the library will block your application until the imagery is available to be read. In the case of a local image file the delay will be very short however in the case of viewing an image from ERDAS APOLLO via the Internet or your local intranet it may take some time to assemble a complete view of the requested area, particularly when accessing data over low bandwidth connections. After you set a view into an image and read imagery line by line, the reads will block your application when data is not yet available for a line. In such circumstances the use of the blocking reads interface may not be appropriate.

The library responds to a read call from the application by waiting a preset time and then returning whatever data is available. With slow connections to a remote server, this time could expire before all the data has been received from the server. Your application could then display an incomplete image. To overcome this problem you should preferably use refresh callbacks. Failing that, you could create a Refresh button that calls the same `SetFileView`, and then reads the image data that has been cached in the progressive view. The data will remain cached as long as the DLL is not unloaded.

Refresh Callbacks

The refresh callback interface allows multi-thread applications to perform `SetFileView()` calls in one thread, while having another thread (the refresh callback thread) perform the read. This has an important implication, in that the view area and extents in the refresh callback may be different from the most recent



`SetFileView()` performed. You must use the `NCSGetViewInfo()` function call while in the refresh callback, to determine the actual view area and size currently available.

There is no direct correlation between `SetFileView()` in the main thread and reading a view within the refresh callback thread.

You may receive multiple refresh callbacks for a single `SetFileView()` when an image is being served from ERDAS APOLLO, and new information for the view area is transmitted continuously. This is known as progressive updates.

Some `SetFileViews()` might not ever be issued as a refresh callback at all. This will be the case when your application is issuing many `SetFileViews()`, for example, when the user is roaming and zooming, in which case the library will automatically cancel some `SetFileViews` if there are too many currently pending.

Although the refresh callback interface requires threads in your application, it is often actually easier to implement than simple blocking reads. This is because your application no longer has to regard the user waiting while an image is redrawn. You simply issue `SetFileView()` whenever you want the view area to change, and the library will optimize the reading, and call your application to update the on-screen view when appropriate. Do multiple `SetFileViews` per `OpenFileView` to increase performance.

If you are using the refresh callback approach, you must keep a `FileView` open while the view is being updated. If you are using the blocking reads approach, you have two alternatives, as shown below:

The following approach is the preferred interface to the library:

```
// Open a file view
NCSOpenFileView()

// Set a file view, image will be read

// asynchronously in a separate thread
NCSSetFileView()

// Set another file view, continue setting file views
// and reading asynchronously until finished
NCSSetFileView()

// Close the file view
NCSCloseFileView()
```

This will provide better performance than doing the following for each view you want to read:

```
NCSOpenFileView()

NCSSetFileView()

//Read view
NCSCloseFileView()
```

This is because the SDK library can cache image information between `NCSSetFileViews`, allowing your application to perform better. However, the library will still cache file information even if your application can not keep state information between requests to view different areas of an image. This means that you can still obtain very high performance in such cases.

Cancelling Reads

You do not have to read all lines from a view that you have set. For example, if your application decides that it needs to have a new view into an image, and you are still not through reading from an existing view, you can quit performing the line-by-line reads, and go ahead and perform a new `setview`. This is done by returning the `NCSReadStatus` as `NCS_READ_CANCELLED` in the read callback.

Multiple Image Views and Unlimited Image Size

You can open views to as many images as you like at the same time. There are no internal limits. The library has been tested with as many as 10,000 compressed images open at the same time.

You can open as many simultaneous views into the same image as you like. There are no internal limits (other than memory).

The compressed image can be of any size (e.g. you can open views into terabyte sized compressed images).



35 terabyte image of Germany compressed to 1 terabyte.

Error Handling

Most of the decompression calls in the SDK return either an `NCSError` enumerated value, or a `CNCSError` object (which has an `NCSError` value as a data member). Functions are provided to obtain meaningful error messages from these return values: `NCSGetErrorText` can be used to retrieve error information from an `NCSError` value, and `CNCSError::GetErrorMessage` can be used to query a `CNCSError` object for error text.

Memory Management

Compression requirements

The following table highlights the fixed write memory cache requirements for different input characteristics.

Input image size	Bands	Gigapixels	Output Format	Compression Method	Write Memory Cache (MB)
50,000 x 50,000 px	3	2.5	ECW	Tile	234
	3	2.5	ECW	Line	143
100,000 x 100,000 px	3	10	ECW	Tile	416
	3	10	ECW	Line	234
500,000 x 100,000 px	3	50	ECW	Tile	1,872
	3	50	ECW	Line	962
	8	50	ECW	Tile	4,906
100,000 x 500,000 px	3	50	ECW	Tile	416
	3	50	ECW	Line	234
	3	50	JPEG2000	Line	202
	8	50	ECW	Tile	1,023
500,000 x 500,000 px	3	250	ECW	Tile	1,873
	3	250	ECW	Line	963
	3	250	JPEG2000	Line	801
	8	250	ECW	Tile	4,909
	8	250	JPEG2000	Line	2,048
1,000,000 x 1,000,000 px	3	1000	ECW	Tile	3,695
	3	1000	ECW	Line	1,873
	8	1000	ECW	Tile	9,102

Notes:

- The drop in memory when using line over tile compression method.
- The drop in memory when the width of the input changes, despite identical gigapixel value.
- The large increase in memory requirements moving from 3 to 8 bands.
- The small memory drop using our default JPEG2000 profile over ECW in line mode.
- Although not shown, write memory requirements do not depend on the target compression ratio. A target of 10:1 will require the same memory as 50:1 for equivalent input characteristics.

Memory Usage

The SDK requires very little memory while decompressing imagery for a view area. Imagery is decompressed on the fly on a line-by-line basis so that even if you open up (for example) a huge view (say 1,000,000 x 1,000,000) into a TB (1,000,000 x 1,000,000) image, the library will perform this for you.

Caching

SDK performs a range of caching operations to speed access to compressed image files, although by default it will never use more than one quarter (25%) of physical RAM for caching operations.

The size of the cache allocated by the SDK during its normal operation can be capped or controlled by calling the global configuration function:

`NCSSetConfig(NCSCFG_CACHE_MAXMEM, nCacheSize)` where

`nCacheSize` is a 32 bit unsigned integer specifying the desired cache size. There is also a 64 bit alternative parameter `NCSCFG_CACHE_MAXMEM_64`.

When accessing imagery over an Internet connection via ECWP, the SDK caches imagery data in the main memory on the client side to speed roaming and zooming over areas that have already been accessed.

When accessing imagery from a local ECW file, the SDK also caches the data connected with that particular file and shares it amongst all open views on that file.

When a file view is closed, by default, the contents of the cache are not freed. The reason for this behaviour is that if another view is immediately reopened it will be able to access the cached data, improving performance.

The default behaviour is to persist data from a particular ECW or JPEG 2000 file in the cache until one half hour has passed.

While data from a particular file is still cached, the SDK maintains a lock on that file even if there are no open file views connected with it. It is necessary to release the cache to remove this lock.

Via the C API, this is done using a call to `NCSCloseFileView(..., TRUE)` where the second argument is set to `TRUE` to specify that cached data should be released.

Via the C++ API, the same operation is performed using `CNCSFile::Close(TRUE)`.

A shortlist of SDK functionality of interest for controlling the cache is:

```
NCSCloseFileView(..., TRUE)
NCSSetConfig(NCSCFG_CACHE_MAXMEM, ...)
NCSSetConfig(NCSCFG_FORCE_FILE_REOPEN, ...)
NCSSetConfig(NCSCFG_CACHE_MAXOPEN, ...)
CNCSFile::Close(TRUE)
```

You should use these functions to implement the caching behaviour required by your application.

Coordinate Information

ECW and JPEG 2000 files contain embedded image coordinate information in addition to compressed image data. Using the SDK, geographical metadata can be obtained from an image file in either of the two formats. The primary use of this data is to specify the geographical location depicted in the image. You can extract and use this important data for georeferencing the image or in mosaics of multiple images.

- See the section entitled [Geocoding Information](#) for more information on how geographical metadata is included in ECW and JPEG 2000 files

To obtain coordinate information from an ECW file using the C language API of the SDK, call `NCSGetViewFileInfo()`, which will return a pointer to an `NCSFileInfo` data structure. Refer to the API documentation for the individual members of this structure.

Transparent Proxying

In SDK versions before 5.0, a further problem could occur where the connection to ERDAS APOLLO is via a proxy. ECWP packets could be blocked by the proxy if authentication is enabled. If this happens, you could, as a workaround, configure the local network to use transparent proxying, a feature supported by many different types of proxies. The following procedure describes how to use transparent proxying as a workaround:

- Disable the authentication on the proxy.
- Install the client side proxy on all the client PCs on the network.

This replaces the networking DLLs on the client machines and makes the proxy transparent to the applications running on the client. It does this by handling the authentication itself rather than having the applications do it. In version 5.0 of the SDK, the ECWP version 3 protocol should work transparently with most proxies, so this configuration should be unnecessary.

Delivering Your Application

If you are delivering on a Windows platform, your application will normally consist of an executable (.exe) file and a number of Dynamic Link Library (.dll) files. These application files must be installed to run your application.

The NCSEcw.dll file should be installed in the bin directory. Keeping these files in the same directory makes all these libraries available to the system.

On Linux, you should ship the libNCSEcw.so file in your library directory, and set the LD_LIBRARY_PATH environment variable to locate your shared libraries at runtime.

On MacOS X, you should ship the libNCSEcw.dylib in your library directory, and set the DYLD_LIBRARY_PATH environment variable to locate your shared libraries at runtime.

Alternatively on all platforms, you can link against the static libraries in which case you do not need to ship the extra files.

Creating Compressed Images

The applications you develop using the SDK must be able to supply the following information to the compression engine:

- You must have a valid company key and OEM license code to create ECW or JPEG 2000 images. See the compression examples of how to set your OEM code in the application to enable compression.
- The name of the output compressed file to create or the name of the source image to compress. In the latter case, the software will generate a default output file name based on the input file name.
- The image information such as height, width and number of bands, etc.
- How you want the image compressed, e.g. as a grayscale file, as a colour (RGB) file, or as a multi-band file.
- The desired compression ratio to use; typically between 20:1 to 50:1 for colour compression, and 10:1 to 20:1 for grayscale compression.
- Optionally, you can supply geocoding information such as datum, projection and units, etc. to be embedded in the compressed image file.



Understanding Target Compression Ratios

Some SDKs compress to a target output size (using “rate control”) when compressing an image. This means that if you compress several images as part of a mosaic, each one will differ in quality as the compressor tries to fit it to the target output size. The overall mosaic will appear inconsistent with varying degrees of quality for each image. This is useful for compressing to a known size such as a DVD, but can be problematic for other uses.

The ECW SDK takes a different approach. The target compression ratio specifies a **quality** ratio. When encoding an image, once the desired quality ratio has been achieved, it will no longer continue to throw away data since the quality level has been reached. It will do this irrespective of the implied output size calculated from the input compression ratio. The advantage of this approach, is that multiple images compressed over the same area for a mosaic with the same target ratio, will have **exactly the same quality** as each other and will appear consistent. You cannot do this if you specify an output file size as each image is different and will compress differently.

Preserving Image Quality When Recompressing

Your application will need to provide a target compression ratio value to the compression engine. This value specifies the desired compression ratio that the user would like to achieve from the compression process. After compressing the image, the compression engine will indicate the actual compression ratio achieved. For example, after compression you may note that the actual compression ratio achieved was in fact 40:1, resulting in an output file size of only 25MB. This would be the difference between the Target Compression Ratio (what you set) and the Actual Compression Ratio (what you achieved). Except when compressing very small files (less than 2MB in size), the Actual Compression Ratio will generally be equal to or greater than the Target compression, sometimes significantly greater.

What causes this difference in output file size is due to the compression engine using this value as a measure of how much information content to preserve in the image. If your image has areas that are conducive to compression (e.g. desert or bodies of water), a greater rate of compression may be achieved while still keeping the desired information content and quality. The compression engine uses multiple wavelet encoding techniques simultaneously, and adapts the best techniques depending upon the area being compressed. It is important to understand that encoding techniques are applied after image quantization and do not affect the quality, even though the compression ratio is higher than what might have been requested.

Additionally, the SDK adds an enhancement filter when decoding to give sharper images at high compression ratios. This should be turned off when recompressing from ECW sources using the `NCSSetConfig(NCSCFG_TEXTURE_DITHER, (BOOLEAN) FALSE)` setting.

Optimizing the Compression Ratio

When compressing imagery, the target compression ratio is specified.

The following table indicates typical target Compression ratios:

Imagery	Application	Target Compression Ratio
Color airphoto mosaic	High quality printed maps.	25:1
Color airphoto mosaic	Internet or email distribution.	40:1
Grayscale airphoto mosaic	High quality printed maps.	10:1 to 15:1
Grayscale airphoto mosaic	Internet or email distribution.	15:1 to 30:1
Lossless (JPEG 2000 Only)	Imagery with perfect reconstruction.	1:1

- Note: Compressing to ECW and specifying a target of less than 5:1 is not recommended as the output quality at that level of compression will not change significantly. To gain further insight into the role of target rates, compression speed and output quality, watch [How to Save Time and Storage Space with Industry-Leading Imagery Compression](#) webinar.
- Sample data compressed at multiple target ratios are also included.

Depending on the imagery, your final compression ratio may be higher than the target compression rate. Imagery with large areas that are similar (for example desert, forests, golf courses or water) often achieves a much higher actual compression rate.

Scanned topographical maps also often achieve a higher compression rate, as do images with smooth changes, such as colour-draped DEMs.

When compressing to the JPEG 2000 file format, which supports lossless compressed images, lossless compression is specified by selecting a target compression ratio of 1:1. This does not correspond to the actual compression rate, which will generally be higher (between 2:1 and 2.5:1).

When you compress individual images that will later be decompressed/ recompressed, we recommend that you use a lower compression rate that is evenly divisible by the ultimate planned compression rate for the output mosaic. This will ensure optimum quality of your compressed mosaic. For example, if you plan to compress the final mosaic at a target rate of 20:1, use a target rate of 10:1 or perhaps 5:1 for the individual images that you are compressing. This way you still reduce disk space significantly, but ensure that you lose very little quality in the multi-compression process.

Recompressing Imagery

The actual compression ratio is calculated using the original, uncompressed size of images that have been previously saved in a compressed (e.g. 8-bit LZW) format. Therefore, it is possible that the compressed ECW or JPEG 2000 image file might be larger than the input file. For example, if we have a 2300x2300 RGB image, its uncompressed size would be 2300x2300x3=15MB. Using 8-bit LZW compression, the file size could be reduced to 800KB; i.e. 30 times smaller. If this file was saved as a compressed ECW or JPEG 2000 image with an actual compression ratio of 25:1, the output would be larger than the input 800KB file.

- The compressed ECW or JPEG 2000 image will still be faster to pan and zoom locally and over the internet than an LZW compressed TIFF image file that is the same size or even smaller, due to the special characteristics of progressive image retrieval from an image compressed using wavelet technology. This performance differential will increase dramatically with large image sizes (1 gigapixel+).



Compressing Hyperspectral Imagery

The SDK is unique in that it will allow you to compress multi-band hyperspectral imagery to ECW or JPEG 2000 formats. To do this you must specify the `MULTIBAND` compression format option when performing the compression process. The SDK supports storage of up to 65,535 bands.

Image Size Limitations

ECW compression is more efficient when it is used to compress large image files. In the case of extremely small images less than 128 x 128 pixels in size, the SDK will return an error message if the application developer attempts to compress the data to the ECW format. No such minimum is in place for compression to JPEG 2000 output and files as small as 1 x 1 pixel can be created using this format. There is technically no upper limit on the size of images that can be compressed using the SDK however OEM License Keys enforce gigapixel limitations. See the section on [Licensing](#) for more information.

Compression Directory Limitations

The ECW JPEG2000 SDK compression creates temporary (.tmp) files in the output directory. These files contain packet information, sometimes in very large numbers. If the output directory is accessed in parallel with compression, then this can degrade the performance of the compression. Tiled images (JP2) are particularly susceptible because the number of temporary files generated is proportional to the number of tiles in the image. Specifying a separate physical disk for input, output and temp files can improve the performance (throughput) of the compression engine significantly.

When compressing, the default SDK parameters should be used whenever possible, unless your application has specific requirements to deviate from the default parameters. Choosing inappropriate compression parameters can impact compression performance detrimentally. In such cases, the process of creating and deleting an excessive number of temporary files could hinder the compression substantially.

Linking against the SDK

When linking against the static libraries on Windows, you must define `NCSECW_EXPORTS` as a pre-processor definition. Both the dynamic and static libraries on Windows are linked against the dynamic Microsoft Visual C (MSVC) run-time library (/MD and /MDd).

When linking on Windows platforms, for dynamic libraries link against `NCSEcw.lib` and `NCSEcwd.lib` for release and debug modes respectively. For static libraries, link against `NCSEcwS.lib` and `NCSEcwSd.lib` for release and debug modes respectively.

When linking on Linux platforms, for dynamic libraries link against `libNCSEcw.so` and `libNCSEcwd.so` for release and debug modes respectively. For static libraries, link against `libNCSEcwS.a` and `libNCSEcwSd.a` for release and debug modes respectively.

Consult the examples for more information on which libraries to link against.

Geocoding Information

An ECW or JPEG 2000 compressed image file can contain embedded geocoding information. This information can be retrieved when the image is decompressed. Geocoding provides a georeference, indicating where the

image is geographically located. Geocoding enables compressed ECW or JPEG 2000 files to form mosaics of very large images. The geocoding information consists of the components described in the following sections.

- Datum
- Projection
- Units
- Registration point
- Cell size

Datum

The datum represents a mathematical approximation of the shape of earth's surface at a specified location. Common datums are:

- North American Datum (NAD27 and NAD83)
- Geocentric Datum of Australia (GDA94)
- World Geodetic System (WGS72 and WGS84)

Projection

A map projection is the mathematical function used to plot a point on an ellipsoid on to a plane sheet of paper. There are probably 20 or 30 different types of map projections commonly used. These try to preserve different characteristics of the geometry of the earth's surface. The following is a list of common projection types:

- Albers Equal Area
- Azimuthal Equidistant
- Conic Equidistant
- Lambert Conformal Conic
- Modified Polyconic
- Mollweide
- Mercator
- Regular Polyconic
- Sinusoidal
- Stereographic
- Transverse Mercator
- Van der Grinten

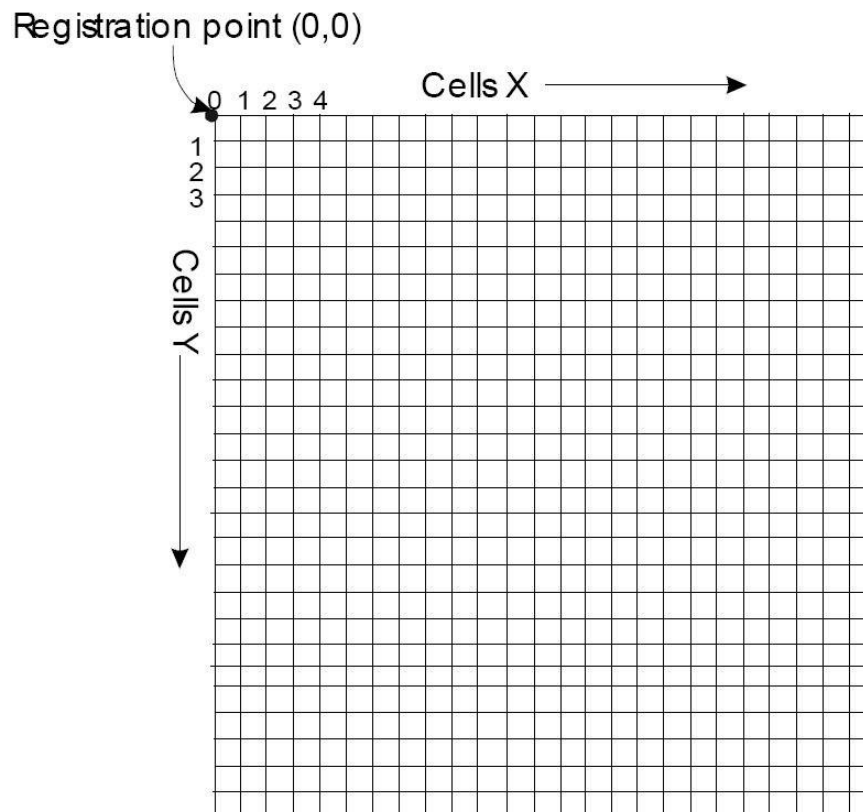
Units

The measurement units are usually set for the specific projection. They can be:

- Meters
 - Feet (US survey feet where 1 meter = 39.37 inches, or 1 foot = 0.30480061 meters)
 - Degrees Latitude/Longitude
- ❗ The default setting for RAW images, i.e. those that do not contain geocoding information, is meters.

Registration Point

The projection, datum and units information tell us the shape of and the area covered by the image, but they do not show where it is located. To convey this information we require a single registration point with world coordinates on the image. For all ECW compressed images this registration point is the origin or top left corner of the top left cell (0,0). The following diagram shows the (0,0) position of the registration point in an image.



Not all images have their registration point at the top left cell (0, 0), as required by the ECW format. Given the cell sizes and the actual reference point it is possible to calculate the world coordinates at point (0, 0).

For example, consider an image that has a registration point at cell (5, 6) with world coordinates (480E, 360N). If the X and Y cell size is 1 meter, the world coordinates at cell (0, 0) will be (480-5)E, (360+6)N, i.e. (475E, 366N).

Cell Size

The cell size represents the physical size of each image pixel in relation to ground units. Cell size can be different in the X and Y dimension, although are often the same and are measured in the units specified above.

RAW versus LOCAL images

Datasets compressed with no georeferencing information are considered "RAW" and have a projection of "RAW" and a datum of "RAW". These images have the origin as "0, 0" and is the bottom left pixel of the image. The world coordinates reflect the pixels coordinates of the image. The units are usually meters, however this has no meaning as the coordinates are in pixel space and the projection is unknown. RAW images cannot be



reprojected or mosaicked as their location and tie point are unknown. These images are only useful for display purposes.

Datasets may sometimes have some georeferencing information (such as a tie point, a cell size and an origin) but have no valid projection information. These images are known as “LOCAL” images and usually have a projection of “LOCAL” and a datum of “WGS84”. Although the absolute location of the image is unknown, we can still perform some operations on these images, such as measure distances and orient similar images together (mosaic). These images however cannot be reprojected to another coordinate system, or overlaid with images in a known map projection.

Editing the Georeferencing Header Information

You do not have to re-compress an entire ECW or JPEG 2000 file to edit the georeferencing information in the header. There are C and C++ convenience functions/classes to do this automatically for you. Refer to the `NCSEditReadInfo`, `NCSEditWriteInfo` functions and the `NCS::CHeaderEditor` class in the HTML API documentation for more information.

The header editor functionality is demonstrated in decompression examples 6, 9, 10, 11, 12 13, and 14.

Geodetic Transform Database

GDT was the original specification for ECW version 1 and 2 files to store the georeferencing information, which was inherited from the ERDAS ER Mapper image processing application. The datum, projection and units described above are in the GDT format. You can convert these datum/projection pairs to an EPSG code for processing via other applications using the `NCSGetEPSGCode` or the `CNSFile.GetEPSGCode()` method on the file class.

ECW v3 files store this information using GeoTIFF tags and JPEG 2000 files generally store this information as XML using the ISO “GML in JP2” specification.

GDT File Formats

The GDT database stores all associated files in the “runtime/PROJ_DATA” directory. The files in this directory define all the datums and projections known to the SDK. These definition files are plain text ASCII format, with the .dat file extension. Data files in the PROJ_DATA directory have a similar format.

There is one logical record per line in the file. The first line of the file is an information line, describing the contents of each field in the file. For example, the first line of the file `mercator.dat`:

```
proj_name, false_north, false_east, scale_factor, centre_merid
```

This line tells us there are 5 fields in each record; the Projection Name (`proj_name`), the False Northing (`false_north`), the False Easting (`false_east`), the Scale Factor (`scale_factor`), and the Central Meridian (`centre_merid`).

The GDT database stores angular values as expressed in radians. For example, the first data record (found on the second line of the file) of the file `mercator.dat`:

```
MR1630N, 1000000.0, 1000000.0, 0.959078718808146, 0.692313937206194
```



This line tells us that the central meridian for projection MR1630N is 0.692313937206194 radians, which is equal to:

$$(0.692313937206194 \times 180) / \text{PI} = 39.6666666 \text{ degrees} = 39 \text{ degrees } 40 \text{ minutes East.}$$

How the SDK Stores Geocoding Information

The SDK represents registration, projection and datum information internally using fields in the `NCSFileInfo` structure. These include the world coordinates of the raster origin, the size of dataset cells in world units, the type of linear units used, the rotation of the raster dataset in degrees (shear transformations are unsupported) and the ERDAS ER Mapper style projection and datum strings.

Embedded Geography Markup Language (GML) Metadata

The Geography Markup Language (GML) is a set of XML schemas for recording and transferring geographic data. GML has been developed by the Open Geospatial Consortium (OGC) in consultation with members and the International Standards Organization. The JPEG 2000 working group have discussed a standard for storing OGC Geography Markup Language (GML) inside a JPEG 2000 file XML box. This standard defines GML metadata in a JPEG 2000 compatible JPX file with a .jp2 file extension.

A standard feature flag set at a value of 67 should signal the use of GML. This geolocating method requires a minimal set of GML to locate a JPEG 2000 image. A `JPEG_2000_GeoLocation` XML element stores the JPEG 2000 geolocation information. While the XML box may also contain additional GML elements, the first element must be the `JPEG_2000_GeoLocation`. There may also be additional XML boxes, containing additional XML elements. In any case, the decoder will use the first `JPEG_2000_GeoLocation` GML element found in the file.

The `JPEG2000_GeoLocation` element contains a `RectifiedGrid` construct. The `RectifiedGrid` has an `id` attribute of `JPEG2000_Geolocation_1`, with a `dimension` attribute equal to 2.

The standard requires an origin element, with an `id` attribute of `JPEG2000_Origin`. The `point` attribute specifies the coordinate of the bottom-left corner of the bottom-left cell in the image. The `srcName` attribute is an immediate EPSG code (recommended). Where an existing EPSG code is not available, `srsName` refers to a full `SpatialReferenceSystem` element definition within the same JPEG 2000 XML box.

A pair of `offsetVector` elements defines the vertical and horizontal cell step vectors. These vectors can include a rotation, but cannot include a shear.

Conformant readers will ignore any other elements found within a `JPEG2000_GeoLocation` element. The GML specification is available for reference at: <https://www.ogc.org/standards/gmljp2>

GML Examples

The following `JPEG_2000_GeoLocation` GML refers to a JPEG 2000 file with an EPSG code of 32610 (PCS_WGS84_UTM_zone_10N), origin 631333.108344E, 4279994.858126N, a cell size of X=4 and Y=4, and a 0.0 rotation.

```
<?xml version="1.0" encoding="UTF-8"?>
<JPEG_2000_GeoLocation>
```

```
<gml:RectifiedGrid xmlns:gml="http://www.opengis.net/gml" gml:id="
JPEG\2000_GeoLocation _1" dimension="2">
  <gml:origin>
    <gml:Point gml:id="JPEG 2000_Origin" srsName="epsg:32610">
      <gml:coordinates>631333.108344,4279994.858126
    </gml:coordinates>
    </gml:Point>
  </gml:origin>
  <gml:offsetVector gml:id="p1">0.0,4.0,0.0 </gml:offsetVector>
  <gml:offsetVector gml:id="p2">4.0,0.0,0.0 </gml:offsetVector>
</gml:RectifiedGrid>
</JPEG_2000_GeoLocation>
```

The following JPEG 2000_GeoLocation GML refers to a JPEG 2000 file with an EPSG code of 32610 (PCS_WGS84_UTM_zone_10N), origin 631333.108344E, 4279994.858126N, a cell size of X=4 and Y=4, and a rotation of 20.0 degrees clockwise.

```
<?xml version="1.0" encoding="UTF-8"?>
<JPEG_2000_GeoLocation>
  <gml:RectifiedGrid xmlns:gml="http://www.opengis.net/gml" gml:id="JPEG
2000_GeoLocation _1" dimension="2">
    <gml:origin>
      <gml:Point gml:id="JPEG 2000_Origin" srsName="epsg:32610">
        <gml:coordinates>631333.108344,
4279994.858126</gml:coordinates>
      </gml:Point>
    </gml:origin>
    <gml:offsetVector gml:id="p1">1.3680805733037027,
3.7587704831464577,0.0</gml:offsetVector>
    <gml:offsetVector gml:id="p2">3.7587704831464577,
-1.3680805733037027,0.0</gml:offsetVector>
  </gml:RectifiedGrid>
</JPEG_2000_GeoLocation>
```

The equivalent registration using a .jpw world file would be:

```
3.7587704831464577
1.3680805733037027
1.3680805733010719
-3.7587704831392297
631335.1083436138
4279992.8581256131
```

The following example C code demonstrates how to output a complete JPEG 2000_GeoLocation GML stream, given an upper-left image registration point, x and y cell sizes, rotation angle and image dimensions. Note that the registration point is the top-left corner of the top-left cell.

```
#define Deg2Rad(x) (x * M_PI / 180.0)

void OutputJPEG2000_GeoLocation(FILE *pFile, UINT32 nEPSGCode, double
dRegistrationX,
double dRegistrationY, double dCellSizeX, double dCellSizeY, double
dCWRotationDegrees,
UINT32 nImageWidth, UINT32 nImageHeight)
```

```
{
double p1[] = { (sin(Deg2Rad(dCWRotationDegrees)) * dCellSizeX),
(cos(Deg2Rad(dCWRotationDegrees)) * dCellSizeY),0.0 };
double p2[] = { (cos(Deg2Rad(dCWRotationDegrees)) * dCellSizeX), -
(sin(Deg2Rad(dCWRotationDegrees))*dCellSizeY), 0.0 };
fprintf(pFile, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\r\n");

fprintf(pFile, "<JPEG_2000_GeoLocation>\r\n"); fprintf(pFile,
" <gml:RectifiedGrid xmlns:gml=\"http://www.opengis.net/gml\" gml:id=\"JPEG
2000_GeoLocation_1\" dimension=\"2\">\r\n");

fprintf(pFile, " <gml:origin>\r\n");

fprintf(pFile, " <gml:Point gml:id=\"JPEG 2000_Origin\" srsName=\"epsg:%ld\">\r\
n", nEPSGCode);

fprintf(pFile, "<gml:coordinates>%lf,%lf
</gml:coordinates>\r\n", dRegistrationX - nImageHeight * p1[0], dRegistrationY -
nImageHeight * p1[1]);

fprintf(pFile, " </gml:Point>\r\n");
fprintf(pFile, " </gml:origin>\r\n");
fprintf(pFile, " <gml:offsetVector
gml:id=\"p1\">%lf,%lf,%lf</gml:offsetVector>\r\n", p1[0], p1[1], p1[2]);
fprintf(pFile, " <gml:offsetVector
gml:id=\"p2\">%lf,%lf,%lf</gml:offsetVector>\r\n", p2[0], p2[1], p2[2]);
fprintf(pFile, " </gml:RectifiedGrid>\r\n");
fprintf(pFile, "</JPEG_2000_GeoLocation>\r\n");
}
```

When the SDK opens a JPEG 2000 file containing GML metadata in the format above, the georeferencing information is automatically translated into the components of an `NCSFileInfo` data structure that can be queried from the open file using `NCSGetViewFileInfoEx` (via the C API) or `CNCSFile::GetFileInfo` (via the C++ API). The GML data itself is not exposed to the application programmer.

JPEG 2000 Registration Behaviour Change in version 5.2

GML in JP2 support was introduced in version 3.0 of the SDK. At this time the georeferencing information contained in ECW files in the `NCSFileInfo` structure interprets the origin as the top left corner of the top left dataset pixel. This was also applied to the GML metadata embedded in JP2 files when JPEG 2000 support was introduced. However, the more common interpretation of the GML in JPEG 2000 specification implies that the registration point represents the centre of the top left pixel.

As of version 5.2, when encoding a JPEG 2000 image, the GML written to the file is offset by half a pixel to account for the shift, and when reading the `NCSFileInfo` structure, a half pixel offset is applied in reverse accordingly to account for the shift.

In short, when reading registration information using the `NCSFileInfo` structure, the origin is the top left of the pixel. However when directly reading the GML embedded within a JPEG 2000 file, the origin is the centre of the top left pixel. These values therefore differ by half the cellsize.

Embedded GeoTIFF Metadata

Another proposed standard for embedding georeferencing information is to place a degenerate GeoTIFF file in a JPEG 2000 UUID box. This standard was originally proposed under the name GeoJP2 by Mapping Science, Inc.

GeoTIFF is a well-established standard for embedding georeferencing information in TIFF (Tagged Image File Format) using header tags, which in turn index a further level of metadata stored in *GeoKeys*. Linear map units, projection and datum information, pixel scales, coordinate transformations and dataset tie points are all examples of the kind of information that can be stored in a GeoTIFF file.

The SDK supports the reading of georeferencing information from a JPEG 2000 file stored in a UUID header box in the form of a degenerate 1 x 1 pixel GeoTIFF file. The information is processed into ERDAS ER Mapper style projection, datum, linear units and registration information. The table below indicates which GeoTIFF tags and *GeoKeys* are supported by the SDK.

For ECW version 3 files, the SDK supports GeoTIFF tags directly using the `NCSGetGeotiffKey/NCSGetGeotiffTag` C functions and methods on the `CNCSFile` class.

Supported GeoTIFF Tags

All standard GeoTIFF tags are supported, including:

- ModelTiePoint
- ModelPixelScale
- ModelTransformation
- GeoKeyDirectory,
- Geo-ASCIIParams
- GeoDoubleParams

Supported GeoTIFF GeoKeys

- GTRasterType
- GTModelType
- GeographicType
- ProjectedCSType
- GeogLinearUnits,
- ProjLinearUnits

Support for World Files

The SDK also supports image registration information for a JPEG 2000 file, stored as the matrix elements of an affine transformation in a six-valued world file located in the same directory as the input JPEG 2000 file.

These world files are a widely accepted standard for storing the geographic registration of an image.

The format of a world file is usually:

- X scaling factor
- Y rotation factor
- X rotation factor

- Y scaling factor
- X translation value
- Y translation value

Where the values are floating point numbers expressed in decimal format. Whilst the SDK makes some allowances for processing JPEG 2000 world files with variations on this format if you are experiencing problems with world file processing it is advisable to use a text editor to edit the file so it has the form above.

The six values provide the SDK with enough information to derive a rotation value, dataset cell sizes in world linear units, and a single registration point for the image. These can be queried from an `NCSFileInfo` structure (obtained in the same way as above in the sections on GML and GeoTIFF metadata).

World files are named according to a comparatively strict convention where the name of the file is the same as that of the image file for which it provides registration information, except that its 3 character extension is constructed by taking the first and third characters from that of the image file and appending the character *w*. The SDK only supports world files in tandem with JPEG 2000 files with file extension .jp2, .jpx, or .jpf, and these cases respectively correspond to world files with file extension .j2w, .jxw, and .jfw. You may need to rename world files produced by third party applications in order to meet this requirement.

Configuring the Use of Geocoding Data for JPEG 2000 Files

Given that there are three forms of geographic metadata supported by the SDK, some attention has been given allowing the application developer to configure which metadata is processed on input from a JPEG 2000 file or output to a JPEG 2000 file. Configuration is achieved using the `CNCSFile::SetParameter()` method.

This method is used to configure many different aspects of SDK usage, but in this case we are interested in the case where `eType` has the value `JP2_GEODATA_USAGE`. When this is the value of the first argument, there are sixteen valid values for the second argument `nValue`:

nValue	Processing of geographic metadata on file I/O
<code>JP2_GEODATA_USE_NONE</code>	No processing of metadata
<code>JP2_GEODATA_USE_WLD_ONLY</code>	World file only
<code>JP2_GEODATA_USE_GML_ONLY</code>	GML header box only
<code>JP2_GEODATA_USE_PCS_ONLY</code>	GeoTIFF UUID box only
<code>JP2_GEODATA_USE_WLD_GML</code>	World file, then GML box
<code>JP2_GEODATA_USE_WLD_PCS</code>	World file, then GeoTIFF box
<code>JP2_GEODATA_USE_GML_WLD</code>	GML box, then world file
<code>JP2_GEODATA_USE_GML_PCS</code>	GML box, then GeoTIFF box
<code>JP2_GEODATA_USE_PCS_WLD</code>	GeoTIFF box, then world file
<code>JP2_GEODATA_USE_PCS_GML</code>	GeoTIFF box, then GML box
<code>JP2_GEODATA_USE_WLD_GML_PCS</code>	World file, then GML, then GeoTIFF
<code>JP2_GEODATA_USE_WLD_PCS_GML</code>	World file, then GeoTIFF, then GML
<code>JP2_GEODATA_USE_GML_WLD_PCS</code>	GML, then world file, then GeoTIFF
<code>JP2_GEODATA_USE_GML_PCS_WLD</code>	GML, then GeoTIFF, then world file
<code>JP2_GEODATA_USE_PCS_WLD_GML</code>	GeoTIFF, world file, then GML
<code>JP2_GEODATA_USE_PCS_GML_WLD</code>	GeoTIFF, GML, then world file

The value chosen applies to processing both on opening any JPEG 2000 file and on compressing to a new JPEG 2000 file, and once set will apply to other files opened or compressed during the execution of an SDK application. Where a precedence is established in configuration (e.g. for `nValue` equal to



JP2_GEODATA_USE_WLD_GML_PCS), on input the metadata available will be established and the existing metadata that appears first in the order of precedence will be used to the exclusion of any other metadata.

On compression to JPEG 2000 output, all the currently configured types of metadata are written (e.g. for `nValue` equal to `JP2_GEODATA_USE_WLD_GML_PCS`, a world file will be written to the output directory, and GML and GeoTIFF header boxes will be written to the JPEG 2000 file). Because there is no explicit mapping between the data supported by each system of storing geographical information, there is no guarantee that the geographical metadata stored will be the same.

Choosing to store georeferencing information in one case in a world file, and in another in a GeoTIFF header box, may result in different interpretations of the stored information when the file is re-read by the SDK or by a third party application. It is up to the application developer to select the most appropriate use of geographical metadata for their SDK application.

EPSG Codes

The SDK uses ERDAS ER Mapper's georeferencing system internally, in which coordinate systems are specified using a pair of strings naming the projection and datum (e.g. projection `NUTM11`, datum `NAD27` for UTM Zone 11 using the North American Datum 1927).

The European Petroleum Survey Group (EPSG) produces a database of codes associated with particular geographical and projected coordinate systems, and these codes have been used in the GeoTIFF specification and also in various OGC specifications as a means of specifying the spatial reference of datasets.

When the SDK writes JPEG 2000 files, it has the option of creating the GML and GeoTIFF UUID (GeoJP2) header boxes. If the output data is spatially referenced by ERDAS ER Mapper projection and datum strings, the SDK converts these strings to a corresponding EPSG code that is embedded in the GML or GeoJP2 header boxes, and can subsequently be re-read by ERDAS ER Mapper and third party software.

The mapping between ERDAS ER Mapper projection and datum strings, and EPSG codes, is not entirely one-to-one, so at times it may be necessary for you to specify specific codes manually. You can do this in one of two ways:

- Using the shorthand value `EPSG:<code>` in your output projection and datum strings, which will cause the value to be embedded in output JPEG 2000 files e.g. `FileInfo.szProjection = "EPSG:32700"; FileInfo.szDatum = "EPSG:32700";`
- Creating a file called `PcsKeyProjDatum.dat` in which custom mappings between projection and datum strings are stored. The lines in the file should have the format: `<code>, <projection string>, <datum string>, <notes and comments>`

where `<code>` is the applicable PCS or GCS code, the projection and datum strings are those you wish to map to this code, and notes and comments allows you to briefly record the code's use,

e.g. `32700, CUSTPROJ, CUSTDAT`, output to our user-defined coordinate system.

Once you have created this file and the applicable content, you should submit its path (without the file name) to your SDK application using either the `NCSSetGDTPath` or the `CNCSTFile::SetGDTPath` calls, if your application uses the C or C++ APIs respectively.

ECW Version 3 Metadata

Metadata can be stored in ECW version 3 files, including RPC (Rapid Positioning Data), statistical data (average, mode, median, histograms etc.) and general file metadata (sensor name, acquisition date, contact details etc).

Decompression examples 9 to 14 demonstrate how to decode various types of metadata. Compression examples 9 to 12 demonstrate how to encode custom file metadata.

The metadata structures and fields that can be stored are listed in the tables below.

Table 1: RPC

Rapid Positioning Data (struct NCSRPCData)		
Field	Type	Description
ERR_BIAS	IEEE8	Error bias
ERR_RANDOM	IEEE8	Error random
HEIGHT_OFF	IEEE8	Geodetic height offset
HEIGHT_SCALE	IEEE8	Geodetic height scale (should be > 0)
LAT_OFF	IEEE8	Geodetic Latitude Offset, range from -90 to +90
LAT_SCALE	IEEE8	Geodetic Latitude Scale, range is 0 to 90
LINE_OFF	IEEE8	Line Offset, should be >= 0
LINE_SCALE	IEEE8	Line Scale, should be > 0
LONG_OFF	IEEE8	Geodetic Longitude Offset , range from -180 to +180
LONG_SCALE	IEEE8	Geodetic Longitude Scale, range is (0, 180]
SAMP_OFF	IEEE8	Sample Offset, should be >= 0
SAMP_SCALE	IEEE8	Sample Scale, should be > 0
LINE_DEN_COEFFS	IEEE8[20]	Line Denominator Coefficients
LINE_NUM_COEFFS	IEEE8[20]	Line Numerator Coefficients
SAMP_DEN_COEFFS	IEEE8[20]	Sample Denominator Coefficients
SAMP_NUM_COEFFS	IEEE8[20]	Sample Numerator Coefficients

Table 2: Statistical Metadata

File Statistics (struct NCSFileStatistics)		
Field	Type	Description
fMode	IEEE4	Statistical mode value
fMinVal	IEEE4	Statistical minimum value
fMaxVal	IEEE4	Statistical maximum value
fMeanVal	IEEE4	Statistical mean value
fMedianVal	IEEE4	Statistical median value
fStandardDev	IEEE4	Statistical standard deviation value
fMinHist	IEEE4	Minimum histogram value
fMaxHist	IEEE4	Maximum histogram value
nHistBucketCount	UINT32	Number of buckets in histogram
Histogram	UINT64*	Array of histogram values, size nHistBucketCount

Table 3: General File Metadata

File Metadata (struct NCSFileMetaData)		
Field	Type	Description
sClassification	wchar_t*	Image classification information
sAcquisitionDate	wchar_t*	Image acquisition date
sAcquisitionSensorName	wchar_t*	Image sensor name
sCompressionSoftware	wchar_t*	Software used to compress the image
sAuthor	wchar_t*	Author
sCopyright	wchar_t*	Copyright information
sCompany	wchar_t*	Company name
sEmail	wchar_t*	Email address
sAddress	wchar_t*	Street address
sTelephone	wchar_t*	Telephone contact number

Examples

Projects demonstrating the decompression and compression functionality can be found in the “examples” directory located in the root installation folder. They are grouped into 4 areas:

- C/C++
- Objective C
- Java (JNI)
- Android

For information on supported development platforms, compilers and runtime environments, refer to [System Requirements](#) chapter in this document.

C/C++

The C++ and C example are cross platform and can be built on Windows, Linux and Macintosh using standard tools and compilers. They require a modern C++ 11 compatible compiler environment for C++ APIs. Most standard “C” compilers can be used for the plain “C” examples. These are the basis for which other language bindings are created. The examples can be found in the following folders:

```
[ROOT]/examples/compression  
[ROOT]/examples/decompression
```

Objective C

The SDK includes a set of Objective C bindings for MacOS X native and iOS development. The examples can be run in a supported version of XCode, and can be found in the following folder:

```
[ROOT]/examples/objectivec
```

The Objective C bindings only support the decompression APIs.

Java (JNI)

The SDK includes a set of Java JNI bindings, for a simple decoder that can be incorporated into a Java application for viewing ECW and JPEG 2000 images. The samples can be build using the Ant build system, and can be found in the following folder:

```
[ROOT]/examples/jni
```

The Java JNI bindings only support the decompression APIs.

Android

The SDK includes a set of Java JNI bindings for Android development, for a simple decoder that can be incorporated into a Java application. Android Studio is the recommended way to build the projects. The samples can be found in the following folder:

```
[ROOT]/android/examples
```

The Java bindings for Android only support the decompression APIs.

FAQ

Licensing

Q: Is a license required for each platform?

No. The license includes all available platforms for that license tier. For example, people with a license for desktop read-write (100 gigapixel) can deploy the ERDAS ECW/JP2 SDK within their third-party product on Windows®, Linux, or MacOS® X using the one SDK license and identical OEM key. Any additional platforms added to the ERDAS ECW/JP2 SDK Version 5 will also be available at no additional charge. An additional license is only required if the application extends into server or mobile use.

The only exception is for those with the server read-only end-user license. Since this product is licensed per server instance, it only provides a license for one of the supported platforms.

Q: How do you define a gigapixel?

A gigapixel is the size of the input expressed in the total number of pixels, width by height, where 1 gigapixel represents 10^9 pixels. For example, an image with dimensions of 150,000 x 150,000 pixels is 22.5 gigapixels.

For simplicity, we do not consider the number of bands, bit depth, or amount of null or empty area in the input. For example, a one-gigapixel image can be a greyscale 1-band uint8 or a 255-band uint16 dataset; they are treated the same way in terms of the size restriction.

Q: Are any licenses free?

The desktop read-only license is still free and includes redistribution rights. This is the only license tier that is available at no charge. Please refer to the EULA for more information about the terms and conditions. You will always be able to convert from ECW and JPEG2000 to any other format using our SDK, for free in Desktop environments.

Q: Are there any limits on the amount or size of datasets that can be read or decoded?

There are no limits on the Desktop or Server license tiers and they include unlimited read capabilities. You may download and use the desktop read-only license to decompress terabytes of data stored as ECW or JPEG2000 for free. For the Mobile tier, decoding of locally stored images greater than 1 gigapixel requires an OEM key however ECWP access has no such limits.

Q: What happens on Mobile when I open a file larger than the free 1 gigapixel limit without an OEM Key?

The SDK will return the error code `NCS_FILE_SIZE_LIMIT_REACHED`, see `NCS_Error` enum. When a valid OEM Key is set, Mobile platforms will be able to decode unlimited sized imagery. There is no further licensing tiers.

Q: Are evaluations available for the encoding or writing capabilities?

No, evaluations are not available. Organizations who want to test performance or general compression capabilities should contact their local Hexagon representative to discuss their requirements.



Q: Will I have to pay royalties?

No. The ERDAS ECW/JP2 SDK license fees are one-time payments to Hexagon for each major release version. We do not limit the amount of data that can be compressed over time, for example have a finite resource that is consumed per compression job nor charge additional royalties per end-user seat. If the gigapixel limit is appropriate for your customer requirements, you can compress unlimited amounts of data.

Q: What happens if I do not have a valid OEM Key?

The call to the Write() function will fail. An OEM key is invalid if:

1. The key does not match the current minor version
 - a. Existing customers need an updated OEM key for each minor release version. For example, the OEM key that worked for version 4.3 will not work with version 5.0 and version 5.0 keys will not work with version 5.5.
2. The key is not valid for the input image size in gigapixels
 - a. Licensees can create output files up to the specified 1, 10, 100, or 1000 gigapixel limits only. For example, if you have a license for one gigapixel, compressing an input size of 32,000 x 32,000 pixels (1.02 gigapixels) would fail, but an input size of 31,000 x 31,000 pixels (0.96 gigapixels) would succeed.
3. The OEM key does not match the OEM company name

Q: I have a redistributable license. Can I redistribute my OEM key?

No, this is expressly prohibited. Per the EULA, the redistributable rights only allow you to redistribute the ERDAS ECW/JP2 SDK within a third-party product. The OEM key must be embedded within your product in a way that end users cannot extract and reuse that key outside of your licensed application.

Q: What is the server read-only end-user license for?

This license is for users who use server-based software that do not or cannot license redistributable rights directly from Hexagon. This is most commonly seen in open-source server applications, so this product allows end users to purchase that right directly. The license is purchased per server instance and does not include redistributable rights.

Q: I need to compress more than 1,000 gigapixels. What can I do?

Images larger than one terapixel are rare, but if required please contact Hexagon with your requirements. The ERDAS ECW/JP2 SDK has successfully compressed ECW files up to 15,000 gigapixels, so the omission of this license type is not due to technical limitations and can be provided only upon request.

Q: Are development or test licenses available?

No. Hexagon issues ERDAS ECW/JP2 SDK licenses per product and does not differentiate between production, test, or development use.



Platform Support and Build Environment

Q: Is the ERDAS ECW/JP2 SDK thread-safe?

Yes, it is fully thread-safe. Multiple file-view instances can be used in separate threads without issue, but a single file-view instance should not be shared between multiple threads.

The global configuration options (e.g. NCSSetConfig) are not thread-safe and should only be called from a single thread.

Q: Is source code available?

No, ERDAS ECW/JP2 SDK only includes static and dynamic binary libraries.

Q: My organization uses a platform for which support is not currently available. What can I do?

Although the latest version of the ERDAS ECW/JP2 SDK has greatly expanded platform compatibility, we understand that some customers still may require unique or non-standard libraries such as specific GCC versions, compilation options, or even different compilers like LLVM. If this is the case, please contact your local Hexagon sales representative to discuss your needs. It is in Hexagon's best interest to ensure ERDAS ECW/JP2 SDK supports as many platforms as possible, but custom builds will only be provided to paid licensees unless there is large general demand from the community.

Q: Why is GCC 4.4 the minimum supported version on Linux?

ERDAS ECW/JP2 SDK version 5 requires specific template features that are only available in GCC 4.4 or higher. However, the library is forward compatible and can compile projects built on newer GCC versions.

Q: What happened to Solaris?

Solaris was supported in ERDAS ECW/JP2 SDK, version 3.x. The platform has not been maintained, primarily due to declining demand. We have no current plans to support this platform.

Q: Does ERDAS ECW/JP2 SDK use hardware acceleration like OpenCL or NVIDIA® CUDA™?

We do not currently implement OpenCL or CUDA GPU acceleration. We do make extensive use of CPU-based SIMD hardware acceleration using SSE (x86) and NEON (ARM) instructions. We will continue hardware optimization in future releases to further enhance SDK performance for our customers. This is an ongoing exercise for the product team. If you have particular performance needs or this area is of particular interest to you, please contact Hexagon.

Q: What happened to NCSEcw*4.dll?

With ERDAS ECW/JP2 SDK version 4.x, there were three separate DLLs that had to be redistributed - NCSUtil4.dll, NCSEcw4.dll, and NCScnet4.dll - with two variations depending on whether the version was read-only or read-write. In ERDAS ECW/JP2 SDK version 5, the library is unified into a single NCSEcw.dll (Windows) libNCSEcw.so (Linux) or framework (Mac) file and there is no dependency on the TBB library.

Q: Why is the Windows installer so much larger than the Linux installer?

Windows includes three compiler platforms - VC90, VC100 and VC110 - and Linux only includes one built on GCC 4.4. Windows debug libraries are also unusually large, but this won't impact third-party users since they



will not be using this version. Linux and Windows both include x86-64 (64-bit) and x86 (32-bit) builds. Use of the static library will also reduce the size of the third-party application.

Software Maintenance (SWM)

Q: Why does Hexagon Geospatial not assume responsibility for “achieving the intended result”? Is support not available?

Support is available and includes bug fixes, platform improvements, enhancements, and minor upgrades. The EULA clause is to prevent third parties from assuming that ERDAS ECW/JP2 SDK version 5 will work with all third-party software with minimal effort. We will assist any customer who requires API clarification or has general questions, but implementation remains the responsibility of the customer. We highly recommend that prospective customers select the free desktop read-only license to become familiar with the product before acquiring a license.

Q: I am using a free license. Can I still give feedback?

We encourage general feedback or bug reports to be submitted via the [Hexagon Community Forum](#). Product team members will contribute to the discussion, but customers with software maintenance contracts should submit support tickets via [Hexagon's support system](#) so the product team can commit to defect resolution and enhancement requests. Bugs posted to the Hexagon Community Forum may be resolved in future releases, but we make no guarantees. Priority is always given to those with software maintenance contracts.

Q: What is the planned release cycle for the v5 release?

Based on the product's time line we typically release a new major version every 3 to 4 years with minor releases in between based on customer feedback.

The SDK, although a part of the Hexagon Geospatial group, does not release major versions yearly ensuring licenses will hold their value for future years and offer great returns on software maintenance.

Q: What if I choose not to purchase software maintenance?

OEM keys will not be regenerated for future minor releases and you will not be able to submit support tickets or shape the direction of future releases.

Q: I am an end user of a product that uses ERDAS ECW/JP2 SDK and I have a problem reading or writing ECW and/or JPEG2000. Do I contact the maker of the product or the maker of the SDK?

Please contact the support team for the product as they are responsible for the implementation of the SDK when building their product. If they believe the problem is within the ERDAS ECW/JP2 SDK, the developers should contact Hexagon Geospatial directly.

General

Q: I am an end user and I just want to compress some imagery. Do I need the ERDAS ECW/JP2 SDK?

ERDAS ECW/JP2 is a software development kit (SDK) that developers can use to build products for end users who want to work with ECW or JPEG2000 imagery. The SDK is not aimed directly at the end users.

Hexagon's Geospatial division does have multiple applications, such as **ERDAS IMAGINE®**, that are aimed at end users and can compress imagery and perform many other image processing tasks. Your local Hexagon Geospatial sales representative can tell you more.

Q: Can I publish benchmarks that compare ERDAS ECW/JP2 SDK to other SDKs?

Yes, but we would like to be contacted before you publish them so we can reply to the methodology. We value the information from benchmarking exercises because they help us improve our software. Please post any findings to the Hexagon Geospatial Community Forum so all can benefit from the results.

Q: I use GDAL version 1.x. Is this version supported?

Hexagon Geospatial submitted a series of improvements (tracked under <https://trac.osgeo.org/gdal/ticket/5029>) that include support for ERDAS ECW/JP2 SDK version 5. These changes were accepted and implemented in GDAL v1.10 and above.

Please see the GDAL driver documentation, <https://gdal.org/drivers/raster/ecw.html> and <https://gdal.org/drivers/raster/jp2ecw.html> for more details. Customers using recent versions of GDAL will find compilation of the version 5 SDK much simpler than previous versions. GDAL-specific issues should be directed to the GDAL mailing lists unless the problem is SDK-specific.

Q: Is this SDK any different than the version used by Hexagon Geospatial's own products?

No, there are no technical differences. We do not artificially limit the performance or functionality in any way. Hexagon Geospatial and other Hexagon technology partners have priority access however these improvements will continue to flow through to the publicly available releases.

Q: I have a product that currently implements ERDAS ECW/JP2 SDK version 3.x or version 4.x. Has the API changed?

Although the general API structure is unchanged, the names of some functions have changed so you should expect some work to implement the new version. It is not a drop-in binary compatible replacement.

Q: What happened to the free 500-megabyte compression limit?

Unfortunately, the ability to compress limited amounts of imagery to ECW and JPEG2000 was routinely abused. Hexagon Geospatial had no choice but to remove this capability with the ERDAS ECW/JP2 version 4 release in 2008 and it has not been reintroduced.

Q: Will the ECW and JPEG2000 files created by ERDAS ECW/JP2 SDK version 5 work with older ERDAS ECW/JP2 SDK versions?

Yes, but with one exception. ERDAS ECW/JP2 SDK version 5 introduces the new ECW version 3 format. Files in ECW version 3 format cannot be read by previous SDKs, but any ECW version 2 and JPEG2000 files can be. ECW version 2 remains the default version created by ERDAS ECW/JP2 SDK.

Q: Can other libraries read the JPEG2000 files produced by ERDAS ECW/JP2 SDK?

Yes. All JPEG2000 files we produce are compliant with the ISO standard (ISO/IEC 15444). However, some third-party libraries have limitations in their SDKs so that they cannot easily open massive geospatial JPEG2000 files. This is a limitation of those SDK's rather than an invalid file produced by our software. If unsure please contact Hexagon Geospatial.

Q: Is your compression technology relevant outside of the geospatial industry?

Although the ERDAS ECW/JP2 SDK is made for geospatial purposes, there are many third parties successfully using the same technology for other purposes, such as medicine and imagery archival.

SDK Concepts

Q: What is a compression ratio?

A compression ratio expresses the file-size differences between the original image and compressed image. A compression ratio of 2:1 means the original image was reduced by a factor of 2. ERDAS ECW/JP2 SDK always expresses the relationship against a value of 1, which means values can easily be calculated as a percentage. If a compression ratio is 20:1, you could also say that the output image is 95% smaller than the original.

Q: What is the difference between the target ratio and actual compression ratio?

The target ratio is supplied as part of the compression call to the SDK. This value is used as a goal, but the SDK will also maintain the quality. So, depending on the input image characteristics, when you compress with a 20:1 target, the actual compression rate may be higher or lower. This is extremely important to ensure a 20:1 target compressed image matches the quality of other 20:1 target images, even though the actual rates may vary.

Q: What is wavelet compression?

The most effective form of compression today is wavelet-based image encoding. Wavelet compression analyzes an uncompressed image recursively. This analysis produces a series of sequentially higher-resolution images, each augmenting the information in the lower-resolution images. Wavelet compression is very effective at retaining data accuracy within highly compressed files. Unlike JPEG, which uses a block-based Discrete Cosine Transformation (DCT) technique on blocks across the image, modern wavelet compression techniques enable compressions of 20:1 or greater without visible degradation of the original image. Wavelet compression can also be used to generate lossless compressed imagery, at ratios of around 2:1.

Q: What is ECW?

ECW is an acronym for Enhanced Compressed Wavelet, a popular standard for compressing and using very large images. ECW is a proprietary, patented format developed by Hexagon Geospatial. ECW only supports an irreversible wavelet filter, making it suitable only for visually lossless requirements.

Q: What is ECWP?

ECWP is an acronym for Enhanced Compression Wavelet Protocol, which is used to transmit images compressed with ECW or JPEG2000 over computer networks. ECWP offers the fastest possible access to



large ECW and JPEG 2000 images, and is fully supported by ERDAS APOLLO. ECWP enables progressive decoding to give end users the fastest possible display speed.

Q: What is ECWPS?

ECWPS is the version of ECWP that includes security via SSL. ECWPS enables private and secure encrypted streaming of image data over computer networks.

Q: What is JPEG 2000?

JPEG 2000 is an ISO standard (ISO/IEC 15444) for compressing, storing, and transmitting images of all types. It uses wavelet compression to achieve scalable compression ranging from numerically lossless to arbitrarily lossy while retaining unprecedented decompressed image quality.

Q: What is the extent of SDK Support for JPEG 2000?

Hexagon Geospatial is the only vendor in the geospatial industry to commit to developing its own JPEG 2000 implementation to provide superior solutions for multi-terabyte JPEG 2000 images. This includes compliance class 2 JPEG2000 and NITF decompression, input and output of geographical metadata in three formats (embedded GML, embedded GeoTIFF UUID box and six-value world file), easy to use API and customizable compression parameters.

Q: I see wild variations in decompression speeds with JPEG 2000 files. Is this normal?

JPEG 2000 is an extremely robust, complex, and highly customizable specification. The JPEG 2000 standard supports many different compression formats, and some of these are better optimized for quick loading. As a result, the performance of ERDAS ECW/JP2 SDK varies across the range of differently structured JPEG2000 files. It decompresses JPEG 2000 files as well as or better than other decoder implementations. If you believe a particular JPEG2000 profile is unusually slow or inefficient, please log a support ticket.

Q: What is GeoTIFF?

GeoTIFF is a version of the popular Tagged Image File Format (TIFF) that includes geo-referencing. GeoTIFF files are standard TIFF 6.0 files, with geo-referencing encoded in several reserved TIFF tags. ERDAS ECW/JP2 SDK fully supports GeoTIFF metadata in compressed and decompressed image files.

Q: What is NITF?

[National Imagery Transmission Format Standard](#) (NITF) is a set of combined government standards for the formatting, storage, and transmission of digital imagery. Originally developed for United States military and government agencies, NITF has been accepted through standardization agreements by NATO and other international defense organizations.

ERDAS ECW/JP2 SDK can encode and decode NITF/NSIF BIIF NPJE, EPJE compliant code-streams.

Q: What is GML?

[Geography Markup Language](#) is an XML grammar and schema for recording and transferring geographic data. GML has been developed by the Open Geospatial Consortium (OGC®) in consultation with members and the International Standards Organization. Geospatial information is available as OGC GML in an XML header box as specified in part 2 of the ISO JPEG 2000 standard (ISO/IEC 15444-2).

Q: Does ERDAS ECW/JP2 SDK support GeoJP2?

GeoJP2 standard for embedding geospatial information in JPEG2000 files inserts a degenerate GeoTIFF file in a UUID box in the JPEG 2000 file, providing coordinate system information and a mapping between pixel coordinates and geo-referenced coordinates. Although it is a somewhat inelegant solution to the problem of embedding geographic metadata in a JPEG 2000 file, it is supported by the ERDAS ECW/JP2 SDK.

ERDAS ECW/JP2 SDK also supports the inclusion of geo-referencing information as Open GIS Consortium Geography Markup Language (OGC GML), continuing our commitment to open standards and interoperability. Developers using the SDK can select which forms of geographical metadata are processed on input and output to and from a JPEG 2000 image file.

Q: Can I target a compressed file size rather than a compression ratio?

No. Currently, only a target compression ratio is supported, which allows an estimated output size to be calculated. This is done primarily to maintain quality, as targeting a file size will yield vastly different qualities depending on the image characteristics.

Advanced Concepts

Q: Can ERDAS ECW/JP2 SDK support bi-level imagery?

Bi-level images are an important subset of the images that can conform to the JPEG 2000 standard. The standard supports bit depths from 1-31, allowing a maximum level of flexibility. ERDAS ECW/JP2 SDK is able to decode compressed bi-level .jp2 files (with a bit depth of 1) since it is fully compliant with the standard. Also, 1-bit .jp2 compression is supported by the SDK, and users are still able to compress bi-level data to grayscale ECW images.

Q: How does ERDAS ECW/JP2 SDK handle partially geo-referenced datasets?

ERDAS ER Mapper uses a datum and projection pair called WGS84/LOCAL to represent the coordinate system of datasets that have a geographic registration, but no formal coordinate system or geocoding. This allows such datasets to be accurately overlaid (similar to the use of world files for this purpose).

An ECW file that is listed in WGS84/LOCAL is processed with vertical values inverted in ERDAS ER Mapper as compared to a RAW/RAW dataset to account for the treatment of location as eastings/ northings rather than agnostic dataset coordinates. Sometimes, ERDAS ECW/JP2 SDK can compress a partially geo-referenced dataset, e.g. one with a registration but no projection or datum, or, in the case of JPEG 2000 files only, a registration and a projection/datum pair that has no corresponding EPSG code. In the case of compression to ECW files, the projection and datum are stored in the file as listed in the output metadata. In the case of JPEG 2000 files, a partially geo-referenced dataset is compressed without a stored EPSG code, and when it is reloaded, it is loaded with projection and datum WGS84/LOCAL to account for the registration information present (which indicates the dataset has a geographic purpose). This behavior can be tested using utility functions provided in the C API, and worked around in client code if it is considered undesirable for some reason.

Q: What is the maximum output bit depth supported by ERDAS ECW/JP2 SDK?

Its JPEG 2000 encoder uses a 32-bit wide encoding pipeline. The maximum effective bit depth per component is currently 28 bits, due to the need to reserve one or more bits as "guard" bits and one bit as a "carry" bit. Compression to a bit depth less than or equal to 28 bits is recommended. You can compress to bit depths that



are not multiples of 8. Therefore, if image quality is a high priority, you could, for example, compress to 26 bits of depth per component and later extract the image data into 32-bit pixel buffers.

Any files (lossy or lossless) compressed to more than 28 bits of depth would be unreadable in all vendor implementations of the JPEG 2000 codec of which we are aware. This restriction is currently the same across all known vendor implementations. A wider pipeline of 64 bits would increase the possible bit depth per component to allow the full 1-38 bit depth range specified by the JPEG 2000 standard.

However, even with this enhancement, there will be no way to do lossless compression with more than 32 bits of depth due to some rather obscure restrictions in the format of the JPEG 2000 code stream. A 64-bit pipeline should allow lossy compression of 33-38 bits.

Q: What file formats are encoded by the ECW JPEG2000 SDK version 5?

It compresses to the ECW (version 2 and 3) and JPEG 2000 (ISO/IEC 15444-1/2) file formats. The JP2 files compressed by the SDK will be backward-compatible JPX files from part 2 of the ISO JPEG 2000 standard, allowing third-party applications to embed geo-referencing information in header boxes in the files to support GIS applications. A JPEG 2000 decoder that complies with part 1 of the standard will be able to decompress these files by default since it will ignore these header boxes.

Q: How does ERDAS ECW/JP2 SDK manage decompression on the opacity channel?

An opacity channel is decoded the same as any other band. Opacity bands are usually encoded as 1 bit. When reading using the ReadLineRGBA family of calls, the opacity band is usually scaled up to return 0 or 255. When using the ReadLineBIL family of functions, the opacity band will be 0 or 1 and the application should scale the data as necessary.

Q: How does ERDAS ECW/JP2 SDK manage different sample sizes and component bit depths?

If the user specifies three bands, cell type NCSCT_UINT32, and bit depth 17 in each of the NCSFileBandInfo structs in pBands, does this mean the compression process will read data in 32-bit chunks for each?

The data type read or the compression input buffer is determined by the NCSCellType specified in the SetFileInfo () call. Out of this, the compressor assumes the data is within the valid range. Currently, it does clip IEEE4 buffers for compatibility with the C API to the specified bit depth, but for performance reasons, it does not clip other types.

It is currently up to the application to guarantee that the bit depth specified is sufficient to handle whatever is passed into the buffer, and that the buffer is big enough to hold it. For example, an INT16 in a UINT8 buffer will not work.

Q: How do I encode to a bit depth less than a standard cell type?

If, for example, you have an image whose range is only 12 bits that you want to compress, encoding this data as a 16 bit data type is not very efficient as there are 4 bits that are unused. If you inform the SDK (via the nBits setting in the compressor structure) of the actual data range, the compressor will be much more efficient and use that information to more effectively compress the image. The decoded data will be in the same range as the original image. To display this data (as with any data greater than 8 bit), you may need to apply a scaling function or a transform to stretch the data to the full output range to make a more dynamic image. The SDK supplies functions for this purpose, however you may choose to do this in the application display logic itself. Refer to the sections on [Data Scaling](#) and [Dynamic Range Calculation](#) for more information.



Q: How does ERDAS ECW/JP2 SDK calculate the optimal block size?

It calculates an optimal block size internally instead of allowing the application developer to change it manually. However, the architecture for compression remains the same for backward-compatibility with old SDK applications.

Appendix A: JPEG 2000 Conformance

ISO/IEC 15444

JPEG 2000 is an image compression standard and coding system created by the Joint Photographic Experts Group committee in 2000 and is standardized by the ISO under specification ISO/IEC 15444. The specification has multiple parts and the ECW JPEG 2000 SDK primarily implements part 1 “Core Coding System” ([ISO 15444-1](#)) and some of part 2 “Extensions” ([ISO 15444-2](#)). The SDK does not implement any other parts of the standard such as “Motion JPEG” (Part 3) or the “Compound Image format” (Part 6). Part 7 “JPEG Interactive Protocol (JPIP)” is also not implemented in the SDK. Alternatively, the SDK offers its own proprietary and more efficient [ECWP](#) protocol for internet streaming. The table below shows the parts implemented by the SDK.

Table 4: Supports Parts of ISO/IEC 15444 Specification

Supported	Part	Specification	Description
✓	1	ISO/IEC 15444-1	Core coding system
<i>Partial</i>	2	ISO/IEC 15444-2	Extensions
✗	3	ISO/IEC 15444-3	Motion JPEG 2000
✗	4	ISO/IEC 15444-4	Conformance testing
✗	5	ISO/IEC 15444-5	Reference software
✗	6	ISO/IEC 15444-6	Compound image file format
✗	7	ISO/IEC 15444-7	<i>Abandoned</i>
✗	8	ISO/IEC 15444-8	Secure JPEG 2000
✗	9	ISO/IEC 15444-9	Interactivity tools (JPIP), APIs and protocols
✗	10	ISO/IEC 15444-10	Extensions for three-dimensional data
✗	11	ISO/IEC 15444-11	Wireless
✗	12	ISO/IEC 15444-12	ISO base media file format
✗	13	ISO/IEC 15444-13	An entry level JPEG 2000 encoder
✗	14	ISO/IEC 15444-14	XML structural representation and reference

Supported Extensions from Part 2

Part 2 provides extensions to the core coding system, and the SDK provides support for some of these. The following table lists the extensions and their status in the SDK.

Table 5: Supported extensions in ISO/IEC 15444-2

Supported	Extension	Description	Comments
✓	1	Syntax	SIZ supported
✗	2	Variable DC offset	Not supported
Partial	3	Variable scalar quantization	QCD, QCC, SOT supported; QPD QPC not supported
Partial	4	Trellis coded quantization	QCD, QCC, SOT supported; QPD QPC not supported
✗	5	Visual masking	Not supported
Partial	6	Arbitrary decomposition	COD, COC supported; DFS, ADS not supported
Partial	7	Arbitrary wavelet transformation	COD, COC supported; ATK not supported
✗	8	Single sample overlap discrete wavelet transformations	SSO not supported
Partial	9	Multiple component transformations	COD, MCT supported; CBD, MCC, MCO not supported
✗	10	Non-linear transformation	NLT not supported
✓	11	Region of interest	RGN supported
Partial	12	File format (JPX)	Only baseline JPX supported (no extensions)
✗	13	Metadata definitions	No Supported

- Note: Only the “Baseline JPX” profile is supported for Extension 12. This means that only JPX files that conform to Part 1 of the standard are supported for decoding and cannot contain other other extensions or features from Part 2.
- Note: The SDK does not support encoding of JP2 or JPX files with any of the options from part 2, where supported, they are only supported when decoding.

Default JPEG 2000 Encoding Format

The default values for the standard parameters when encoding are shown in the table below:

Table 6: Default JPEG 2000 encoding parameter values

Property	Value	Comment
PROFILE	PROFILE1	Only features in Part 1 of the standard
CAP	NO	No features beyond Part 1 or Part 2
EXTENSIONS	NO	No features from Part 2
SIZE	X,Y	Always the size of the entire image
ORIGIN	0,0	Always the top left of the image
TILES	X,Y	Always the same as the SIZE parameter (single tile)
TILE_ORIGIN	0,0	Always the top left of the image

LAYERS	1	Always 1 quality layer
MCT	NO	Do not use MCT
SOP	NO	Do not use SOP markers
EPH	YES	Include EPH markers
ORDER	RPCL	Default order always Resolution, Position, Component, Layer

While these parameters are able to be modified when encoding, they should be left at default as they are best suited for large imagery typically used in GIS applications (gigabytes or terabytes). Changing these may decrease decoding performance, increase memory usage, decrease internet streaming performance, and may make them unreadable in other JPEG 2000 libraries.

GML in JP2

Additionally, the Open Geospatial Consortium (OGC) has defined a metadata standard for georeferencing JPEG 2000 images with embedded XML using Geography Markup Language (GML), called "[GML in JPEG 2000 for Geographic Imagery Encoding \(GMLJP2\)](#)," version 1.0 in 2006. The SDK supports both reading and writing of version 1.0 of this standard.

NITF 2.1

The SDK supports both encoding and decoding of the recommended NPJE and EPJE profiles for JPEG 2000 code streams in NITF 2.1 files. However, the SDK does not support the NITF file format and will not open these files directly. Use must use a container implementation, such as [GDAL](#), which has built in support for NITF files using the ECW JPEG 2000 SDK.

Appendix B: List of View Parameters

Further advanced information about ECW and JPEG 2000 files can be found from the view class `CNCS::View` methods `GetParameter` and `SetParameter`. In the C API these functions are `NCSGetViewParameter` and `NCSSetViewParameter`.

Parameter Name	Description	Data type	ECW/JP2
<code>NCS:REFRESH:TIME:MS</code>	Time between set view updates in the callback thread in milliseconds.	INT32	ECW/JP2
<code>SDK:ECWPSTREAM:VERSION</code>	The ECWP version (2 or 3).	INT32	ECW/JP2
<code>ECW:HUFFMANEDCODE:FLAG</code>	Huffman flag for ECW Compression	INT32	ECW
<code>JPC:COMPRESSMEM:LEVEL</code>		INT32	JP2
<code>JP2:TRANSFORMATION:TYPE</code>	DWT transformation type, <code>IRREVERSIBLE_9x7</code> or <code>REVERSIBLE_5x3</code>	TransformationType	JP2
<code>JPC:DECOMPRESS:RECONSTRUCTION:PARAMETER</code>	<code>IRREVERSIBLE_9x7</code> or <code>REVERSIBLE_5x3</code>	float	JP2
<code>JP2:COMPLIANCE:PROFILE:TYPE</code>	Profile 0, 1 or 2.	UINT32	JP2
<code>JP2:GML:JP2:BOX:EXISTS</code>	Determines existence of the "GML in JP2" georeferencing box.	BOOLEAN	JP2
<code>JP2:GEODATA:USAGE</code>	Control the precedence of georeferencing metadata from world files and embedded GML XML boxes and PCS UUID boxes	UINT32	JP2
<code>JP2:GEODATA:PRECISION:EPSILON</code>			
<code>JP2:DECOMPRESS:LAYERS</code>	Number of quality layers to decode (1- n where n is the maximum number of quality layers). A value of 0 has a special meaning of "auto", where the number of layers is calculated based on the resolution of the current view, where: $nDecompressionLayers = nResolutionLevel^2 + 2;$ and $nResolutionLevel$ is 0 on lowest resolution and increases towards dataset resolution level.	UINT32	JP2
<code>JPC:DECOMPRESS:AUTOSCALE:UP</code>	When decoding, scale the input data range to the output data range.	BOOLEAN	ECW/JP2
<code>JP2:COMPRESS:TILE:WIDTH</code>	Tile Width (default to file width e.g. 1 tile)	UINT32	JP2
<code>JP2:COMPRESS:TILE:HEIGHT</code>	Tile Height (default to file width e.g. 1 tile)	UINT32	JP2
<code>JP2:COMPRESS:PROGRESSION:RPCL</code>	JP2 RPCL Progression Order	char *	JP2
<code>JP2:COMPRESS:PROGRESSION:RLCP</code>	JP2 RLCP Progression Order	char *	JP2
<code>JP2:COMPRESS:PROGRESSION:LRCP</code>	JP2 LRCP Progression Order	char *	JP2
<code>JP2:COMPRESS:PROFILE:NITF:BIIF:NPJE</code>	JP2 NITF NSIF BIIF NPJE Profile	char*	JP2
<code>JP2:COMPRESS:PROFILE:NITF:BIIF:EPJE</code>	JP2 NITF NSIF BIIF EPJE Profile	char*	JP2
<code>JP2:COMPRESS:PROFILE:BASELINE:2</code>	JP2 Profile 2	char*	JP2
<code>JP2:COMPRESS:PROFILE:BASELINE:1</code>	JP2 Profile 1	char*	JP2
<code>JP2:COMPRESS:PROFILE:BASELINE:0</code>	Default, JP2 Profile 0	char*	JP2
<code>JP2:COMPRESS:PRECINCT:WIDTH</code>	Precinct Width (default 64)	UINT32	JP2

JP2:COMPRESS:PRECINCT:HEIGHT	Precinct Height (default 64)	UINT32	JP2
JP2:COMPRESS:MT:READ	Enable compression threaded reading (default true)	int	JP2
JP2:COMPRESS:LEVELS	Levels in pyramid (calculated so r=0 <= 64x64)	UINT32	JP2
JP2:COMPRESS:LAYERS	Quality layers (default is 1)	UINT32	JP2
JP2:COMPRESS:INCLUDE:SOP	Output Start of Packet Marker (default false)	int	JP2
JP2:COMPRESS:INCLUDE:EPH	Output End of Packet Header Marker (default true)	int	JP2
JP2:COMPRESS:CODESTREAM:ONLY	Write only j2c codestream	int	JP2
ECW:BACKGROUND:COLOR	Get the background color used when decoding NULL blocks. Format is "red, green blue", e.g. "255,255,255" for white.	char*	ECW v3

Appendix C: ECW Header Editor CLI Parameters

The ECW header editor allows you to edit the georeferencing information contained in an ECW or JP2 file header without recompressing the file.

To display the help for the generic parameters for ECW or JP2 files use the following command line:

```
> ECWHeaderEditorCLI.exe --header -help

Header editor operation:
  --help                Display header editor help message
  --input arg           Path of the dataset to update
  --projection arg      The new projection string
  --datum arg          The new datum string
  --origin-x arg        The new X coordinate of the top-left
                        corner of the top-left pixel
  --origin-y arg        The new Y coordinate of the top-left
                        corner of the top-left pixel
  --cell-increment-x arg The new world units per pixel in the X
                        axis
  --cell-increment-y arg The new world units per pixel in the Y
                        axis
  --update-ers arg (=0) Should a .ers side-car file also be
                        updated
  --allow-invalid-datum-projection-pairs arg (=1) Allow any datum and projection
                        combinations in the dataset
```

To display the help for the specific parameters for JP2 files use the following command line:

```
> ECWHeaderEditorCLI.exe --jp2 --help

JP2 advanced operations:
  --help                Display JP2 advanced operations help
                        message
  --input arg           Path of the JP2 dataset to update
  --show-cdef-box       Show the cdef box contents
  --validate-cdef-box   Validate that the cdef box contents are
                        correct
  --correct-cdef-box-rgb-in-multiband Correct the cdef box where RGB channels
                        are associated to bands for non RGB
                        datasets
  --backup-dataset arg (=1) For operations that modify the input
                        backup the file first
```

Appendix D: Screenshots

Figure 3 - HTML API Documentation

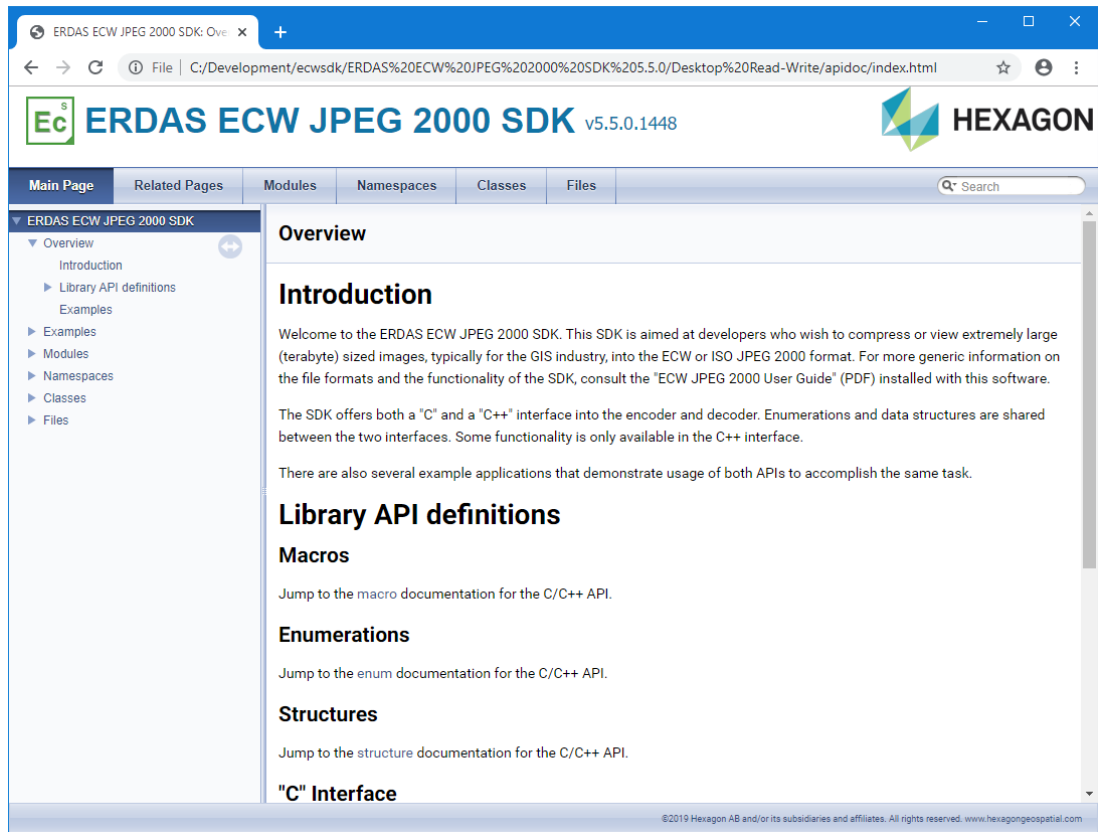
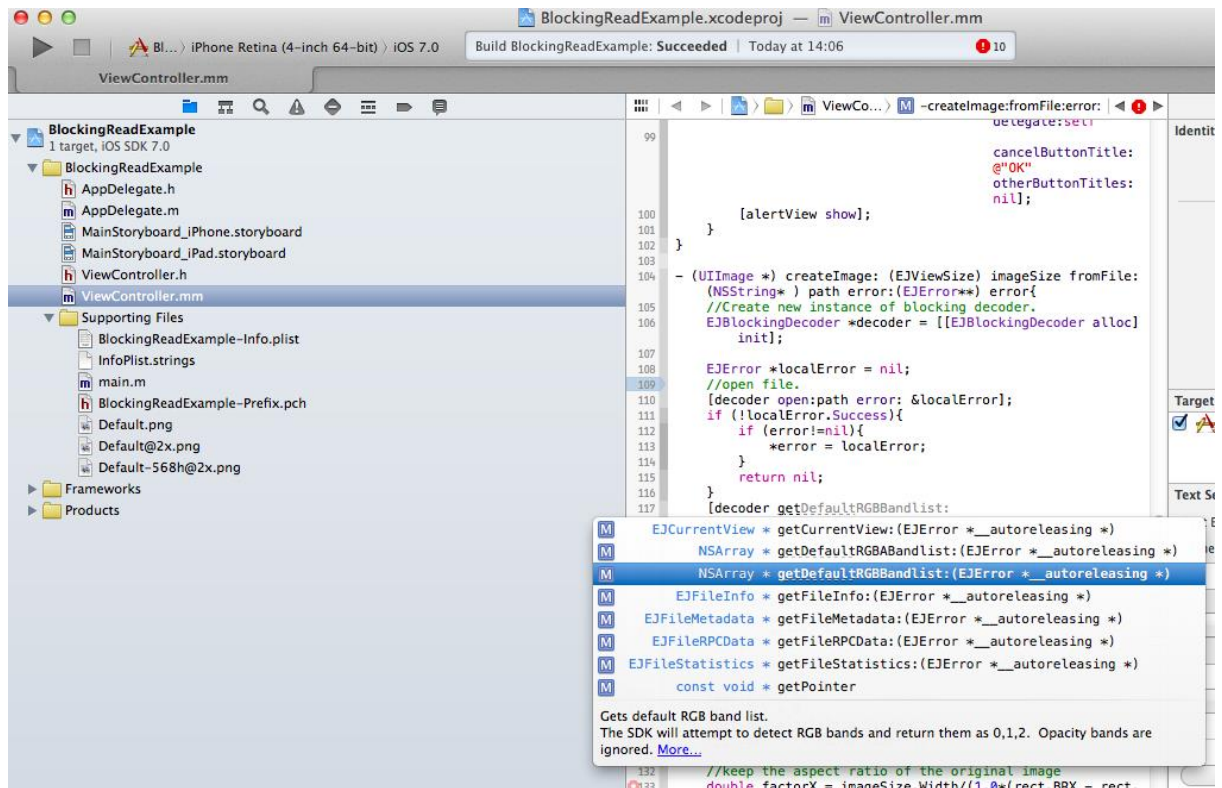


Figure 4 - MacOSX API Xcode embedded documentation





Support

ERDAS ECW JP2 SDK product support is available to all customers with an active subscription or Software Maintenance contract. See the [Hexagon Geospatial Support](#) webpage for more information on how to raise support requests.

When reporting problems, please include all relevant log files, stack traces, function calls and compiler and platform information that will help diagnose possible causes. Input data may be required to reproduce.



About Hexagon

Hexagon is a global leader in sensor, software and autonomous solutions. We are putting data to work to boost efficiency, productivity, and quality across industrial, manufacturing, infrastructure, safety, and mobility applications.

Our technologies are shaping urban and production ecosystems to become increasingly connected and autonomous — ensuring a scalable, sustainable future.

Hexagon's Geospatial division creates solutions that deliver a 5D smart digital reality with insight into what was, what is, what could be, what should be, and ultimately, what will be.

Hexagon (Nasdaq Stockholm: HEXA B) has approximately 20,000 employees in 50 countries and net sales of approximately 4.3bn USD. Learn more at [hexagon.com](https://www.hexagon.com) and follow us @HexagonAB.

© 2021 Hexagon AB and/or its subsidiaries and affiliates. All rights reserved. Hexagon and the Hexagon logo are registered trademarks of Hexagon AB or its subsidiaries. All other trademarks or service marks used herein are property of their respective owners.