ERDAS ECW JPEG2000

Software Development Kit (SDK)

September 2012

Table of Contents

Table of Contents i

Introduction 1

Viewing and Printing this Document 1
Code listings 2
Intended Audience 2
What's New in Version 4.3 2
What's New in Version 4.2 2
What's New in Version 4.1 2
What's New in Version 3.3 3
What's New in Version 3.0 3
What's New in Version 2.0 3
Contacting Us 4
Acknowledgements 4

About Image Compression 5

Introduction 5
Lossless or Lossy Compression 5
Wavelet Based Encoding 6
ECW Compression 6
Using Opacity Channels 8
JPEG 2000 Compression 8
Support for the NITF Standard 9

Licensing and Installation 11

System Requirements 11
Operating system 11
Platforms 11
Third Party Packages 11
Applications 11
Licensing 12
ECW JPEG2000 SDK Free Version 12
ECW JPEG2000 SDK Licensed Editions 12
Licensing the ECW JPEG2000 SDK 12
Installation 12
Directory Structure and Files 16
Subdirectories and Files 16

Development 19

PC Library and Include Files 19
Project Settings - Visual C++ 19
How Imagery is Accessed 20
How to Read a View 20
The SetFileView Concept 20
Viewing Areas Smaller than your Application Window Area 22
Requesting Odd-Aspect Views 22
Selecting Bands from an Image File to View 22

Blocking Reads Versus The Refresh Callback Interface 23
When to Use Blocking Reads 23
When to Use Refresh Reads 24
Blocking Reads 24
Refresh Callbacks 25
Canceling Reads 26
Multiple Image Views and Unlimited Image Size 26
Error Handling 26
Memory Management 26
Memory Usage 26
Caching 27
ECWP Persistent Local Cache 28
J2I Index Files 28
Transparent Proxying 29
Delivering Your Application 30
Creating Compressed Images 30
Preserving Image Quality 30
Optimizing the Compression Ratio 31
Compressing Previously Compressed Images 32
Compressing Hyperspectral Imagery 32
Image Size Limitations 33
Compression Directory Limitations 33
Testing Your Development 33

Examples 35

Compression Examples 35
Compression Example 1 35
Compression Example 2 37
Compression Example 3 40
Decompression Examples 43
Decompression Example 1 43
Decompression Example 2 44
Decompression Example 4 46
Decompression Example 5 49
Decompression Example 6 50

API Reference 53

Introduction 53
C API: Decompression Functions 53
NCScbmCloseFileView 53
NCScbmCloseFileViewEx 54
NCScbmGetViewFileInfo 54
NCScbmGetViewFileInfoEx 55
NCScbmGetViewInfo 56
NCScbmOpenFileView 56
NCScbmReadViewLineBGR 57
NCScbmReadViewLineBGRA 58
NCScbmReadViewLineBIL 58
NCScbmReadViewLineBILEx 59
NCScbmReadViewLineRGB 60
NCScbmReadViewLineRGBA 61
NCScbmSetFileView 61

NCScbmSetFileViewEx . . . 62
NCSecwSetConfig . . . 64
NCSecwSetIOCallbacks . . . 65
Decompression: Related Data Structures . . . 66
NCSFileViewFileInfo . . . 66
NCSFileViewFileInfoEx . . . 67
NCSFileBandInfo . . . 69
NCSFileViewSetInfo . . . 69
C API: Compression Functions . . . 70
NCSEcwCompressAllocClient . . . 71
NCSEcwCompressOpen . . . 71
NCSEcwCompress . . . 72
NCSEcwCompressClose . . . 72
NCSEcwCompressFreeClient . . . 73
NCSEcwCompressSetOEMKey . . . 73
Compression: Developer Defined Functions . . . 74
pReadCallback . . . 74
pStatusCallback . . . 75
pCancelCallback . . . 75
Compression: Related Data Structures . . . 76
NCSEcwCompressClient . . . 76
Information From The Application Developer . . . 77
C API: Utility Functions . . . 78
NCScbmGetFileMimeType . . . 79
NCScbmGetFileType . . . 79
NCSCopyFileInfoEx . . . 80
NCSDetectGDTPath . . . 80
NCSFreeFileInfoEx . . . 80
NCSGetEPSGCode . . . 81
NCSGetGDTPath . . . 82
NCSGetProjectionAndDatum . . . 82
NCSInitFileInfoEx . . . 83
NCSSetGDTPath . . . 83
NCSSetJP2GeodataUsage . . . 83
C++ API . . . 84
Class Reference: CNCSFile . . . 84
Constructor: . . . 85
Destructor: . . . 85
CNCSFile::AddBox . . . 85
CNCSFile::BreakdownURL . . . 85
CNCSFile::Close . . . 86
CNCSFile::ConvertDatasetToWorld . . . 86
CNCSFile::ConvertWorldToDataset . . . 87
CNCSFile::DetectGDTPath . . . 87
CNCSFile::FormatErrorText . . . 88
CNCSFile::GetBox . . . 88
CNCSFile::GetClientData . . . 89
CNCSFile::GetEPSGCode . . . 89
CNCSFile::GetEPSGCode . . . 89
CNCSFile::GetFile . . . 90
CNCSFile::GetFileInfo . . . 91
CNCSFile::GetFileMimeType . . . 91
CNCSFile::GetFileType . . . 91
CNCSFile::GetFileViewSetInfo . . . 92

CNCSFile::GetGDTPath 92
CNCSFile::GetNCSFileView 93
CNCSFile::GetNCSFileView 93
CNCSFile::GetParameter 93
CNCSFile::GetParameter 93
CNCSFile::GetPercentComplete 95
CNCSFile::GetPercentCompleteTotalBlocksInView 95
CNCSFile::GetProjectionAndDatum 95
CNCSFile::GetStream 96
CNCSFile::GetUUIDBox 96
CNCSFile::GetXMLBox 97
CNCSFile::Open 97
CNCSFile::Open 98
CNCSFile::ReadLineABGR 98
CNCSFile::ReadLineARGB 99
CNCSFile::ReadLineBGR 99
CNCSFile::ReadLineBGRA 100
CNCSFile::ReadLineBIL 100
CNCSFile::ReadLineBIL 100
CNCSFile::ReadLineBIL 101
CNCSFile::ReadLineBIL 101
CNCSFile::ReadLineBIL 101
CNCSFile::ReadLineBIL 102
CNCSFile::ReadLineBIL 102
CNCSFile::ReadLineBIL 102
CNCSFile::ReadLineBIL 103
CNCSFile::ReadLineBIL 103
CNCSFile::ReadLineBIL 103
CNCSFile::ReadLineBIL 104
CNCSFile::ReadLineRGB 104
CNCSFile::ReadLineRGBA 104
CNCSFile::RefreshUpdate 105
CNCSFile::RefreshUpdateEx 105
CNCSFile::SetClientData 106
CNCSFile::SetCompressClient 106
CNCSFile::SetFileInfo 106
CNCSFile::SetGDTPath 107
CNCSFile::SetOEMKey 107
CNCSFile::SetParameter 107
CNCSFile::SetParameter 108
CNCSFile::SetRefreshCallback 110
CNCSFile::SetView 110
CNCSFile::SetView 111
CNCSFile::SetView 112
CNCSFile::Write 113
CNCSFile::WriteCancel 113
CNCSFile::WriteLineBIL 114
CNCSFile::WriteReadLine 114
CNCSFile::WriteStatus 115
Class Reference: CNCSRenderer 115
Constructor 115
Destructor 116
CNCSRenderer::ApplyLUTs 116
CNCSRenderer::CalcHistograms 116

CNCSRenderer::DrawImage 116
CNCSRenderer::FreeJPEGBuffer 117
CNCSRenderer::GetHistogram 118
CNCSRenderer::GetTransparent 118
CNCSRenderer::ReadImage 118
CNCSRenderer::ReadImage 119
CNCSRenderer::ReadImage 120
CNCSRenderer::ReadLineBGR 120
CNCSRenderer::ReadLineBIL 121
CNCSRenderer::ReadLineRGB 121
CNCSRenderer::SetBackgroundColor 122
CNCSRenderer::SetTransparent 122
CNCSRenderer::WriteJPEG 122
CNCSRenderer::WriteJPEG 123
CNCSRenderer::WriteWorldFile 123
Class Reference: CNCSError 124
Constructor 124
Destructor 124
CNCSError::GetErrorMessage 124
CNCSError::GetErrorNumber 125
CNCSError::Log 125
CNCSError::operator= 125
CNCSError::operator== 126
CNCSError::operator!= 126
CNCSError::operator!= 127

Geocoding Information 129

Datum 129
Projection 129
Units 130
Registration Point 130
Geodetic Transform Database 131
GDT File Formats 132
How the ECW JPEG2000 SDK Reads Geocoding Information 132
Embedded Geography Markup Language (GML) Metadata 132
GML Examples 133
Embedded GeoTIFF Metadata 135
Support for World Files 136
Configuring the Use of Geocoding Data for JPEG 2000 Files 137
EPSG Codes 138

FAQ 141

What is Lossless Compression 141
What is Lossy Compression 141
What is Wavelet Compression 141
SDK Compresssion Rates 141
Maximum Output Bit Depth Per Image Component Supported by the SDK 141
Is the SDK 64 Bit Enabled 142
Which File Formats are Encoded by the ECW JPEG2000 SDK 142
How the SDK Manages Decompression Functions on the Opacity Channel 142
How the SDK Manages Different Sample Sizes and Component Bit Depths 143
What Happens if the Bit Depth Specified is Greater than the Maximum Bit Depth for the Cell Type . . . 143

How does the SDK Handle Optimal Block Sizes 143
What is JPEG 2000 143
What is the Extent of SDK Support for JPEG 2000 144
GeoJP2 Support in the SDK 144
Inconsistent Decompressiong Speeds with JPEG 2000 Files 144
What is GML 144
Support for Bi-level Imagery in the SDK 145
What is ECW 145
What is ECWP 145
What is ECWPS 145
What is GeoTIFF 145
What is NITF 145
NITF Support in the SDK 146
What is a Projection 146
How the SDK Handles Partially Georeferenced Datasets 146

Index 147

Introduction

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 (this is advantageous for imagery coming from an Image Web Server). You can include ECW or JPEG 2000 compressed images of any size (including terabytes or larger) within your application. The images can be from a local source (e.g. a hard disk, network server, CD or DVD-ROM). The images can also be from a remote source as delivered from an Image Web Server. 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.

Viewing and Printing this Document

This sections outlines the different format types of this documentation and the necessary programs needed to open them.

For on-line viewing we recommend that you select the following options from the Acrobat Reader® or Acrobat® View menu.

The free Acrobat® Reader, required to open PDF documents, is available for download at:

http://www.adobe.com/

The PDF version allows you to:

- Ability to print sections or the entire document.
- Easily navigate from the contents menu.
- Resize and zoom in on the document.
- Mark sections of the document.

You can access the PDF version of the documentation from the Start menu -> All Programs -> ERDAS 2010 -> ERDAS ECW JPEG2000 SDK after installation.

Code listings

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

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

Intended Audience

As with the SDK itself, this guide is intended for programmers with a good understanding of C and C++ programming concepts and techniques. This guide describes specific considerations and techniques for implementing the compression and decompression features of the SDK in application programming.

What’s New in Version 4.3

- 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.

What’s New in Version 4.2

- Many bug fixes and optimizations.

What’s New in Version 4.1

- 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 2005/2008 support (Windows).
- Many bug fixes and optimizations.

What’s New in Version 3.3

- Support for building on *NIX platforms using GNU autotools.
- Fixes to threading issues on *NIX 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.

What’s New in Version 3.0

- Added ISO standard JPEG 2000 support.

What’s New in Version 2.0

- Added many new features.
- Enhanced the ECW format (version 2).

- Added ECWP streaming.
- Added compression to the SDK.

Contacting Us

Any problems that you encounter when using this product should be reported to the email address below. This is the preferred method of communication.

support@erdas.com

Alternatively, there is a general developer forum for SDK's at the website below. You must first register for this free service.

http://developer.lggi.com/Home.aspx

Please include the following in your bug reports:

1. PRODUCT.
2. VERSION.
3. COMPUTER AND OPERATING SYSTEM DETAILS.
4. SUMMARY/DESCRIPTION.
5. STEPS TO REPRODUCE THE PROBLEM.
6. SAMPLE DATA (JPEGs for error snapshots, small ECW or ER Mapper files, etc).
7. SAMPLE CODE (IF AVAILABLE) THAT REPRODUCES THE ISSUE.
8. WHAT YOU HAVE DONE TO TRY TO FIX THE PROBLEM OR YOUR ANALYSIS.

Acknowledgements

We would like to acknowledge the following third part applications.

- Intel Thread Building Blocks.
- NSIS (Nullsoft Installation System).
- TinyXML.
- IJG libjpeg.

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, 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 datastore 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 color image such as an airphoto can be compressed to less than 5% of its original size (20:1 compression ratio or better). At 20:1 compression, a 10GB color image compresses down to 500MB in size. This size is small enough to be stored on a CD-ROM. 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, you can experience fast roaming and zooming on the imagery, even on slower media such as CD-ROM.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 uncompress to an identical copy of the original image. This perfect reconstruction is the advantage of lossless compression. The disadvantage of lossless compression is a ratio limit of approximately 2:1 compressed file size.

Lossy compression provides a compressed image that can uncompress to an approximate copy of the original image.

Lossy compression sacrifices some data fidelity, in order to achieve much higher compression rates than those available through lossless compression. This higher compression capability is the advantage of lossy compression.

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 analyzing 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, dequantized, and then an inverse DWT 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 (patent pending). 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 decompresed.

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.

Because the ECW technique does not require intermediate DWT files on disk during its compression process, ECW has the potential to provide further benefits. These onward benefits include:

- **Multiprocessor optimizations:** The ECW DWT compression and decompression engine leverages the power of multiprocessor systems to achieve up to 95% acceleration on dual CPU machines.
- **Guaranteed latency:** ECW provides guaranteed latency with its recursive algorithm pipeline technique, and guaranteed compression time with defined CPU performance. Although the ECW compression wizard reads uncompressed imagery from disk and writes compressed imagery to disk, this is only an implementation, not an architecture requirement. With equal facility, the ECW technique can take a line by line stream of uncompressed imagery data as input, compress it, and emit a compressed stream of imagery. Therefore, the ECW compression/decompression technique is available for a range of applications without disk storage. Such applications include HDTV signal transmission, real time compression of imagery on satellites for reduced down-link data rates, and compression of imagery on digital cameras.
- ECW is tightly integrated with other ERDAS products and functionality. Applications such as ER Mapper can be used to compress input from smart data algorithms, as well as directly from uncompressed imagery, to the .ecw file format. Other ERDAS APOLLO Image Web Server tools, such as the orthorectification, mosaicking and balancing wizards can be

used to prepare seamless mosaics, which can in turn be compressed to a fraction of their original size.

Using Opacity Channels

As of version 4.1 of the SDK, ECW files with an opacity channel can now be encoded and decoded. The work flow for compressing and decompressing ECW files is exactly the same to that of JPEG 2000.

For an example of how compress an ECW file with an opacity band, refer to Compression Example 1 and Compression Example 2.

Older versions of the ECW JPEG2000 SDK will ignore the opacity channel of an ECW file.

JPEG 2000 Compression

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 color and gray 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.

Support for the NITF Standard

NITF is an acronyn 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 codestreams compliant with the NPJE and EPJE profiles specified with NITF.

Licensing and Installation

This chapter has three main components. The first component summarizes the software and hardware requirements needed to run the application. The second component provides instructions on how to install and license the software. Finally the third component outlines the directory structure.

System Requirements

Below is a list of hardware and software needed to adequately run the SDK.

Operating system

The SDK is only available as pre-built binaries that require one of the following operating systems:

- Windows XP/Vista/7/2003 Server and 2008 Server.

All available operating system updates should be applied to ensure correct operation of the SDK libraries.

Platforms

- Intel Pentium (x86 and x64).
- AMD x86.

Third Party Packages

- TinyXML: A small XML parser. You can remove this dependency by removing NCSJPC_USE_TINYXML from the project settings. However, this will disable the GML Geolocation XML box support. TinyXML is available from www.sourceforge.net/projects/tinyxml.
- IJG libjpeg: The Independent JPEG Group's JPEG library. This is required by the NCSRenderer class, which supports writing out JPEG files. The IJG libjpeg library is available from www.ijg.org.

Applications

The SDK compression library has been specifically designed for applications created with:

- Microsoft Visual C++ Version 8, 9 (SP1 only), and 10.

Licensing

As of version 4.1 the SDK will now be distributed as a free version with limited functionality or licensed version.

ECW JPEG2000 SDK Free Version

The free version of the SDK will allow you to decompress both ECW and JPEG 2000 files. However, you will be unable to compress your input files to these formats.

ECW JPEG2000 SDK Licensed Editions

The licensed edition will enable you to compress and decompress to both ECW and JPEG 2000 file formats. The level of compression will be determined by the license tier that you have purchased. Each license tier is based on the amount of gigapixels you can compress from.

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.

Licensing the ECW JPEG2000 SDK

When you have obtained a license key please refer to the sections below for verification.

To validate a license when operating in the C environment please refer to the NCSEcwCompressSetOEMKey method.

To validate a license when operating in the C++ environment please refer to the CNCSFile::SetOEMKey method.

Before a compression can be made the ECW_COMPRESS_RW_SDK_VERSION must be defined in the pre-procesessor section of your project.

Installation

As of version 4.1 of the SDK there will be two separate installers, a Read Only (free) and Licensed version.

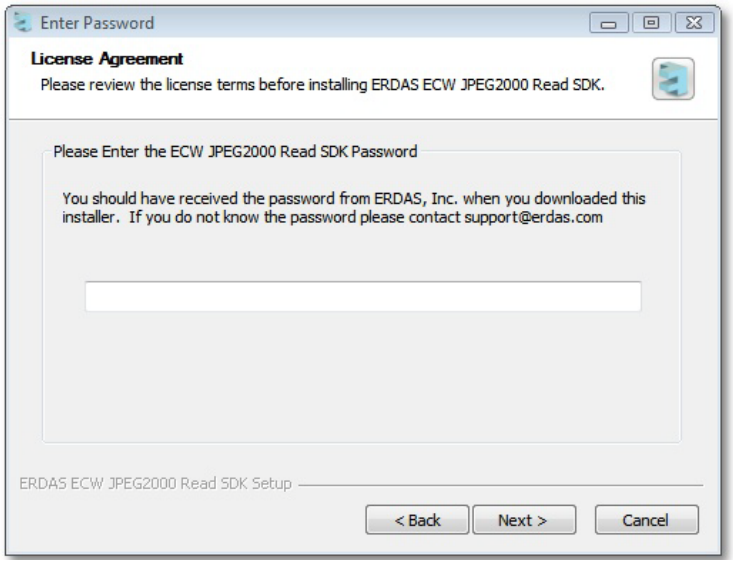
Furthermore, different files will be copied to your hard drive depending on which version you have installed. For example, if you have installed the read only version, there will be no compression examples.

You will need to complete the following procedure to install the SDK onto your workstation.

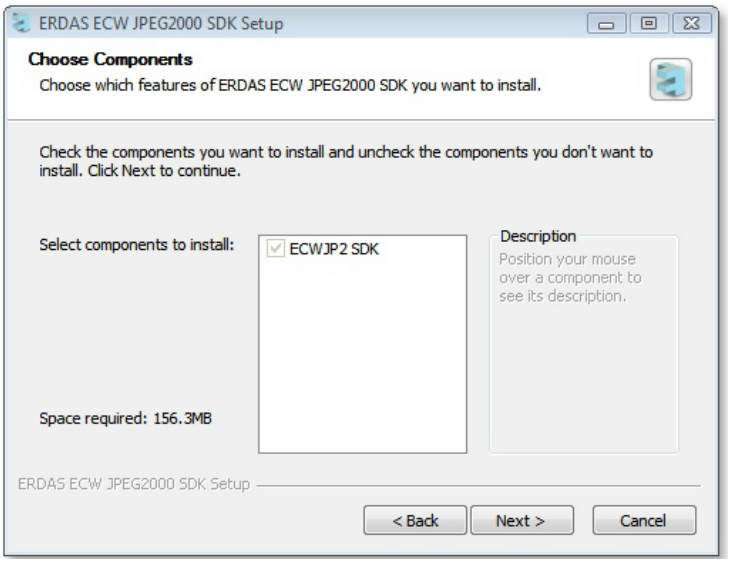
When installing either edition of the SDK you will need to obtain a password in order to unzip the installation files. Refer to your confirmation email to obtain this password.

1. Click on the Run button to start the installation.
2. If you agree to the License Agreement tick the I Agree box to proceed.

3. Enter the password in the text box that was supplied to you in the confirmation email.

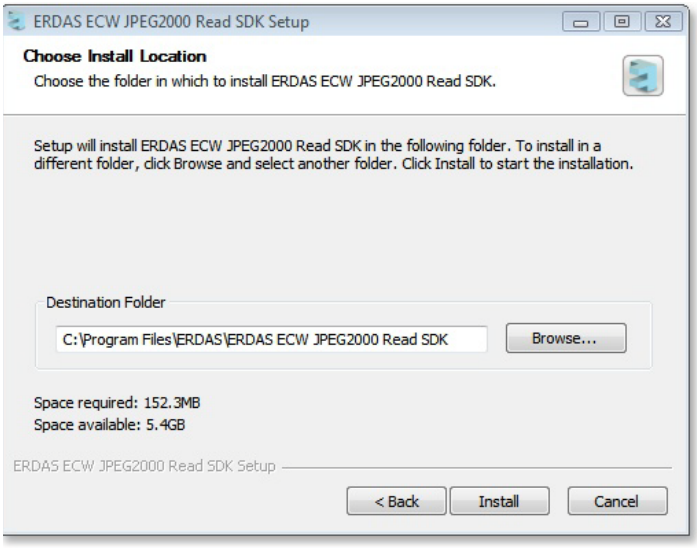


4. Choose the components you wish to install.

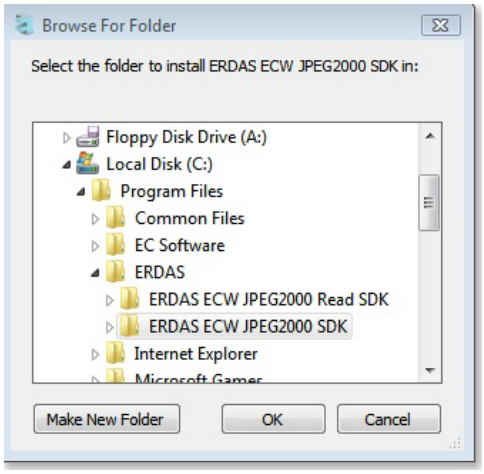


5. If you intend to install the application to it's default location simply click the Next button.

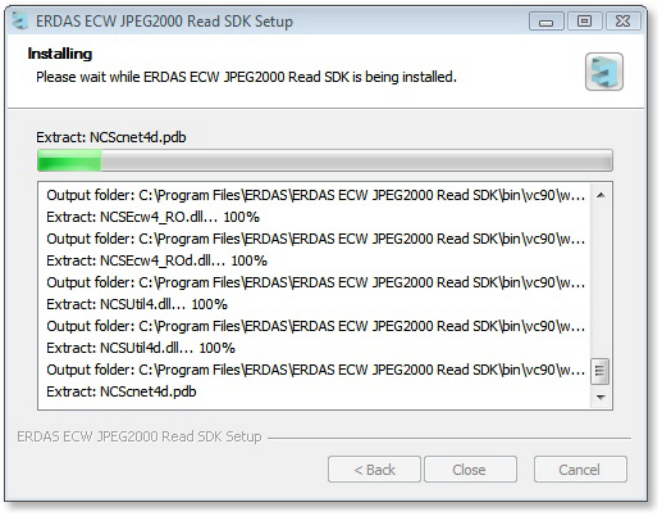
If you are installing the licensed version of the SDK, the default installation folder will be ERDAS ECW JPEG2000 SDK.



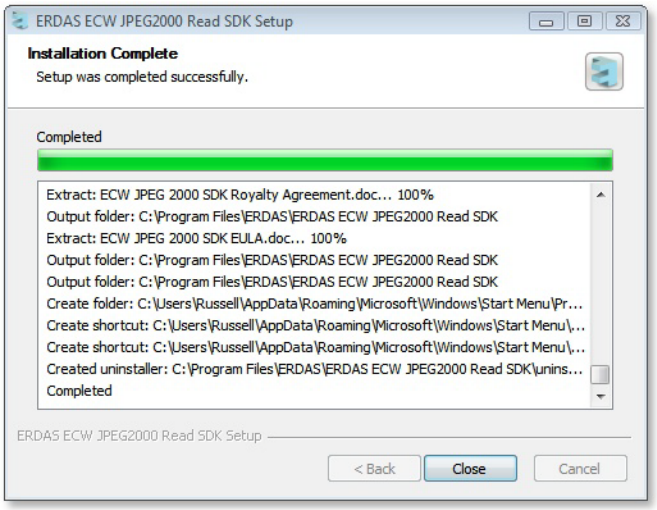
Selecting the browse button will enable you to select a specific file location to store the SDK setup files. If you click the Disk Cost button you are able to find out how much disk space you have available before completing the installation.



6. Click the next to initiate the installation.



7. The installation is complete, click the Close button to exit the installation.



Directory Structure and Files

During installation, new directories are created in the Program Files directory. A new ERDAS directory appears (if this is the first installation of ERDAS software). Within the ERDAS directory, a new folder named ERDAS ECW JPEG2000 SDK is created. This folder contains several subdirectories with all the constituent files. The following sections give an overview of each of these subdirectories and what is contained within them.

There is a slightly different manifest between the read only and licensed versions of the SDK.

Subdirectories and Files

The following section contains a list of all of the files that will be copied to the installation location.

Start Menu Items

Several items are placed in the Start -> All Programs -> ERDAS 2010 -> ERDAS ECW JPEG2000 SDK, these are:

- EULA - User License Agreement in Word format.
- Microsoft Visual Studio 2005 Examples Solution - Visual Studios 2005 file which will load all available examples.
- Microsoft Visual Studio 2008 Examples Solution - Visual Studios 2008 file which will load all available examples.

- Microsoft Visual Studio 2010 Examples Solution - Visual Studios 2010 file which will load all available examples.
- User Guide (PDF) - This is the SDK User Guide in PDF format.

License and Documentation Information

These files are located in the \ECW JPEG2000 SDK directory. Information for each of these files can be obtained below.

The \bin Directory

The **\bin** directory contains Dynamic Link Library (.dll), debug files and Program Database (.pdb) files.

There are two sub-folders located in the \bin directory. These are vc80 and vc90. Each of these files contain the same information. Use the vc80 folder if you are running Visual Studios 2005 and vc90 folder if you are running Visual Studios 2008. You have a choice of using 32 or 64 bit files.

The \examples Directory

The **\examples** directory contains source and project files for demonstrating how to compress and decompress files. Each example will contain Visual Studio Project .NET and C files.

If you are running the read only version of the SDK the compression examples will be omitted.

The\include Directory

The **\include** directory contains ECW header files for your applications.

The \lib Directory

The **\lib** directory is structured in the same manner as the \bin directory. Contained within this directory are the ECW static library files which are used for linking during the application build.

The \redistributable Directory

The **\redistributable** directory is structured in the same manner as the \bin directory. However, it only contains the necessary .dll files that you need to distribute the SDK with.

The\testdata Directory

The **\testdata** directory contains ECW and JPEG 2000 compressed images for testing your applications..

Development

This section describes the structure of the SDK and how to set up a Visual C++ development environment for your SDK implementations. These descriptions are set out for the PC platform. To use the ECW JPEG2000 decompression library, you will require Visual C++ Version 8.0, 9.0 (SP1 only) or 10.0, and the Microsoft Processor Pack, installed on your computer.

PC Library and Include Files

To install PC Library and Include files, complete the following procedure.

1. Run the downloaded SDK executable file to install the SDK directories and files.
2. Open Microsoft Developer Studio (MsDev) and select Options from the Tools menu.
3. Select the Directories tab in the Options dialog.
4. Select include files in the Show directories for: box and add the installdir\include directory to the list.
5. Click OK to close the dialogs.

Project Settings - Visual C++

Your Visual C++ projects must have the following settings to link your applications to the SDK library.

1. Select Settings from the MSDev Project menu.
2. Enter the following information in the Project Settings dialog:

- C, C++ Code Generation - Multithreaded DLL.
- Link/General/Object Files - Add NCSEcw4.lib, NCSEcwC4.lib and NCSUtil4.lib.
- Include the string `ECW_COMPRESS_RW_SDK_VERSION` into the pre-processor setings.

3. Click OK to close the Project Settings dialog.

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 `NCScbmOpenFileView()` function opens a view into an ECW JPEG2000 image file. This image file can also be served from a local or remote Image Web Server, in which case the image would be defined by its URL.
- Obtain information about the image that has been opened. The `NCScbmGetViewFileInfo()` 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 `NCScbmSetFileView()` and `NCScbmSetFileViewEx()` functions specify 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 `NCScbmReadViewLineBGR()` function returns data in an order that can be directly placed into a Windows bitmap.
- Close a view. You can close a view with the `NCScbmCloseFileView()` and `NCScbmCloseFileViewEx()` functions. There are some additional functions, particularly when using the `Refresh` callbacks interface into the library. However, the above outline gives the basic interface approach into the library.

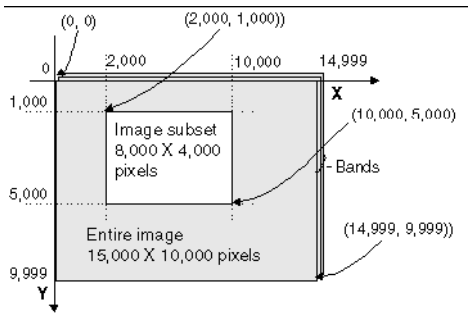
The SetFileView Concept

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 `NCScbmSetFileView()` 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 callback 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 =
NCScbmSetFileView(pView,nBands,pBandList,0,0,14999,999
9,500,300);
```

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 =
NCScbmSetFileView(pView,nBands,pBandList,2000,1000,100
00,5000,500,300);
```

For more accuracy you can call the `NCScbmSetFileViewEx()` *function instead of* `NCScbmSetFileView()`*. This allows you to specify the world coordinates of 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 `NCScbmGetViewFileInfo()` 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 (color) 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.

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 Versus The Refresh Callback Interface

There are two ways to access images:

- **Blocking reads:** You perform a `SetFileView`, read the view, perform another `SetFileView`, and so on. The library will block your reads from the view until image data is available.
- **Refresh callback interface:** A `SetFileView` can be performed whenever you choose, even if reading is not complete.

From time to time, the library will call your application back, using a callback function that you specify, to read new imagery for your application. You might get several callbacks for a single `SetFileView`, when the image is being served from a remote Image Web Server, as additional information comes in to update the image view.

This is progressive updating of the image view. To use blocking reads with the C API, you specify `NULL` as the callback function when performing the `NCScbmOpenFileView()`call.

To use refresh reads with the C API, you specify the name of a callback function when performing the `NCScbmOpenFileView()`call. The body of this function is defined in your application source code, and typically reads image scanlines from the current view using one of the SDK functions.

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 Image Web Server 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 (from a disk or CD-ROM), the delay will be very short. However, in the case of viewing an image from a remote Image Web Server 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 from a slow modem link. 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 `NCScbmGetViewInfo()` function call while in the refresh callback, to determine the actual view area and size currently available.

There is no direct correlation between `SetFileViews()` 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 an Image Web Server, 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 flush 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:

```
NCScbmOpenFileView() // open a file view
NCScbmSetFileView() // set a file view
   // _ Readimage _// Read the image using one of the
read line calls
NCScbmSetFileView() // Set another file view
      // _ Readimage _// Read the image using the new
view
_ continue until finished _// keep showing new views
NCScbmCloseFileView() // Close the file view
```

This will provide slightly better performance than doing the following:

```
NCScbmOpenFileView() // open a file view
NCScbmSetFileView() // set a file view
   // _ Readimage _// Read the image using one of the
read line calls
NCScbmCloseFileView() // Close the file view
NCScbmSetFileView() // Set another file view //
```

```
    _ Readimage _// Read the image using the new view
NCScbmCloseFileView() // Close the file view
... and so on _
```

This is because the SDK library can cache image information between `NCScbmSetFileViews`, 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.

Canceling 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`.

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.

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

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

Memory Usage

The SDK requires very little memory for image decompression 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 does perform a range of caching operations to speed access to compressed image files, although it will never default to using 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 using the call `NCSecwSetConfig(NCSCFG_CACHE_MAXMEM,nCacheSize)` where `nCacheSize` is a 32 bit unsigned integer specifying the desired cache size.

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 behavior is that if another view is immediately reopened it will be able to access the cached data, improving performance.

The default behavior 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 `NCScbmCloseFileViewEx(...,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:

```
NCScbmCloseFileViewEx(..., TRUE)
NCSecwSetConfig(NCSCFG_CACHE_MAXMEM, ...)
NCSecwSetConfig(NCSCFG_FORCE_FILE_REOPEN, ...)
NCSecwSetConfig(NCSCFG_CACHE_MAXOPEN, ...)
CNCSFile::Close(TRUE)
```

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

ECWP Persistent Local Cache

As of version 4.1 of the SDK, client side caching is available and can be deployed into 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 `NCSecwSetConfig` function (see "API Reference" chapter for details). The parameter options are `NCSCFG_ECWP_CACHE_ENABLED`, `NCSCFG_ECWP_CACHE_SIZE_MB` and `NCSCFG_ECWP_CACHE_LOCATION` respectively.

For security reasons, images secured via HTTPS are not cached.

Cache Limits

The maximum cache limit is 10GB. However, it is not recommended you set the cache to it's maximum value as this may result in the SDK crashing.

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

J2I Index Files

J2I index files are automatically created when a JPEG 2000 file stream is opened in the SDK. 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.*

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 chapter "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 `NCScbmGetViewFileInfo()`, which will return a pointer to an `NCSFileViewFileInfo` data structure. This data structure includes the following components:

`eCellSizeUnits`	These are units used for the cell size. This can be one of the following: Meters (or RAW) = 1, Degrees = 2, Feet = 3
`eSizeY`	This is the number of rows (cells down) in the image.
`eSizeX`	This is the number of columns (cells across) in the image.
`fCellIncrementX`, `fCellIncrementY`	This is the cell dimension sizes.
`fOriginX`, `fOriginY`	This is the world coordinates of registration (top-left) cell, in `CellSizeUnits`.
`szDatum`	This is an ER Mapper style Datum name string, e.g. `RAW` or `NAD27`.
`szProjection`	This is an ER Mapper style Projection name string, e.g. `RAW` or `WGS84`. This value will never be `NULL`.

Transparent Proxying

A further problem could occur where the connection to the Image Web Server is via a proxy. ECWP packets could be clocked 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.

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 `NCSUtil.dll`, `NCSecw.dll` and `NCScnet.dll` files should be installed in a bin directory. The `NCSecw.dll` file is self-registering, becoming available to applications after registration. Keeping these files in the same directory makes all these libraries available to the system.

Creating Compressed Images

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

- The string `ECW_COMPRESS_RW_SDK_VERSION` must be defined.
- 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 color (RGB) file, or as a multi-band file.
- The desired compression ratio to use; typically between 20:1 to 50:1 for color 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.

Preserving Image Quality

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.

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

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 colordraped 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.

Compressing Previously Compressed Images

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 roam and zoom 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.

Compressing Hyperspectral Imagery

The SDK is unique in that it will allow you to compress multi-band hyperspectral imagery to one of two popular formats. To do this you must specify the `MULTIBAND` compression format option before starting the compression process.

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, but the free ECW JPEG2000 SDK Compressor, and applications created with the Limited (free) version of the SDK are limited to compressing images with file sizes that are no larger than 500MB.

Compression Directory Limitations

The ECW JPEG2000 SDK compression creates .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 both the compression and the operating system. A reboot may be required to recover the system. Tiled images are particularly susceptible because the number of .tmp files generated is proportional to the number of tiles in the image.

For compression, 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 detrimentally. In such cases, the process of creating and deleting an excessive number of .tmp files could hinder the compression substantially.

Testing Your Development

If you are wanting to test your development efforts you can do so by utilizing the content in the http://iws.erdas.com/ website. You can also use ERDAS' free application known as ER Viewer to open or stream compressed ECW or JPEG 2000 fies. ER Viewer can be downloaded from the ERDAS website (below).

http://www.erdas.com/products/ERDASERMapper/ERDASERViewer/Details.aspx

Examples

Compression Examples

This section contains three different ecompression related examples. These examples are located in the compression folder which is a sub directory of the examples folder. It is also possible to load all of the examples by selecting the example_projects-msvc80 (90 or 10).sln. This is located in the examples folder or via the start menu.

The compression examples will only work if you have a licensed version of the SDK. Refer to the "Licensing and Installation" *chapter for more information.*

Compression Example 1

The first example program does not open any specific image for compression. Instead, the `ReadCallback()` function assigns cell values of 0 or 255 to the `ppInputArray[]` variable, thus creating a checker-board pattern. An opacity band is also added before the image is compressed which displays the background the image layer sits on. It then compresses this image to a file e.g output1.ecw. The compression parameters, entered into the compression client structure, `*pClient` are:

Number of Bands:	`nInputBands`	3
Image Width:	`nInOutSizeX`	512 cells
Image Height:	`nInOutSizeY`	512 cells
Compression Format:	`eCompressFormat`	Set by number of bands: 1 band = Grayscale 2 band = Grayscale plus opacity 3 band = RGB 4 band = RGBA Other = Multi-band In this example, the band number is set to 4, so the format will be RGBA.
Target Compression Ratio:	`fTargetCompression`	Set by compression format: Grayscale = 20 RGB = 10 Multi-band = 20 In this example, the target compression ratio will be 10.

Output File Name:	`szOutputFileName`	Specified by the user.

The `pClient` structure also has pointers to the following callback functions. These are called by the ECW Compression Library function; `NCSEcwCompress()`.

- **ReadCallback()** - For each line and band of the image, the `ReadCallback()` function assigns cell values of 0 or 1000, creating a checkerboard pattern.
- **StatusCallback()** - This function determines which image line is being processed, and displays this as the percentage complete.
- **CancelCallback()** - This function always returns a value of `FALSE`, so that the compression is not cancelled.

At the end of the compression, the `NCSEcwCompressClose()` function enters the compression statistics into the pClient structure:

- **Actual compression ratio:** `fActualCompression`
- **Compression time in seconds:** `fCompressionSeconds`
- **Size of output compressed image:** `nOutputSize`
- **Compression rate in MB per second:** `fCompressionMBSec`

SDK Decompression Library Functions Called

- `NCSEcwCompressAllocClient(void)`
- `NCSEcwCompressOpen(NCSEcwCompressClient *pInfo,`

  `BOOLEAN bCalculateSizesOnly)`
- `NCSEcwCompress(NCSEcwCompressClient *pInfo)`
- `NCSEcwCompressClose(NCSEcwCompressClient *pInfo)`
- `NCSEcwCompressFreeClient(NCSEcwCompressClient *pInfo)`
- `NCSEcwCompressSetOEMKey(szLicensee, szOEMKey);`

Developer-Defined Functions Called

- `ReadCallback(NCSEcwCompressClient *pClient UINT32`

  `nNextLine, IEEE4 **ppInputArray)`
- `StatusCallback(NCSEcwCompressClient *pClient,`

  `UINT32 nCurrentLine)`
- `CancelCallback(NCSEcwCompressClient *pClient)`

Program Flow

1. Allocate a client structure and insert input dimensions and compression required:

```
if(pClient = NCSEcwCompressAllocClient())
```

You have the option of inputting an opacity channel for RGB by setting `TEST_NR_BANDS` to 4 or setting the number of bands to 3 for a normal RGB. If you intend to compress to a grayscale image with any opacity channel you will need to specify 2 bands.

```
if(pClient->nInputBands == 1) {
         pClient->eCompressFormat = COMPRESS_UINT8;
         pClient->fTargetCompression = 20.0f;
      } else if(pClient->nInputBands == 4) {
         pClient->eCompressFormat = COMPRESS_RGB;
         pClient->fTargetCompression = 10.0f;
      } else {
         pClient->eCompressFormat = COMPRESS_MULTI;
         pClient->fTargetCompression = 20.0f;
      }
```

2. Specify the callback functions and client data pointers:

```
pClient->pReadCallback = ReadCallback;
pClient->pStatusCallback = StatusCallback;
pClient->pCancelCallback = CancelCallback;
pClient->pClientData = (void*)&RI;
```

3. Open the compression:

```
eError = NCSEcwCompressOpen(pClient, FALSE);
```

4. Execute the compression:

```
eError = NCSEcwCompress(pClient);
```

5. Call the developer-defined callback functions for every input image line.
6. Close the compression and display the compression output statistics.

```
NCSEcwCompressClose(pClient);
```

7. Free the compression client structure:

```
NCSEcwCompressFreeClient(pClient);
```

Compression Example 2

This example program accepts an ECW or JPEG 2000 compressed image as input. The program decompresses the image and then compresses it again. The compression parameters are extracted from the `NCSFileViewFileInfo` structure `*pNCSFileInfo`, and entered into the compression client structure. The Target Compression Ratio can be entered by the user as an argument or it will default to that of the original compressed image.

Number of Bands:	`nInputBands`	Same as input compressed image
Image Width:	`nInOutSizeX`	Same as input compressed image
Image Height:	`nInOutSizeY`	Same as input compressed image
Compression Format:	`eCompressFormat`	Set by the number of bands in the compressed image format: 1 band = grayscale 3 bands = RGB Other= multi-band
Target Compression Ratio:	`fTargetCompression`	Same as that of the original compressed image unless the `fTargetCompressionOverride` value is entered by the user.
Output File Name:	`szOutputFileName`	As entered by the user.

The `pClient` structure also has pointers to the following callback functions. These are called by the ECW Compression Library function; `NCSEcwCompress()`.

• **ReadCallback()** - For each line and band of the image, the `ReadCallback()` function reads a line from the input image.

• **StatusCallback()** - This function determines which image line is being processed, and displays this as the percentage complete.

• **CancelCallback()** - This function always returns a value of `FALSE`, so that the compression is not cancelled.

At the end of the compression, the `NCSEcwCompressClose()` function enters the compression statistics into the pClient structure:

- **Actual compression ratio:** `fActualCompression`
- **Size of output compressed image:** `nOutputSize`
- **Compression time in seconds:** `fCompressionSeconds`
- **Compression rate in MB per second:** `fCompressionMBSec`

SDK Compression Library Functions Called

- `NCScbmReadViewLineBIL(`

```
pReadInfo->pNCSFileView,
pReadInfo->ppInputBandBufferArray)
```

- `NCScbmOpenFileView(`

  `szInputFilename, &pNCSFileView, NULL)`

- `NCScbmGetViewFileInfo(pNCSFileView, &pNCSFileInfo)`

- `NCScbmSetFileView(`

  ```
  pNCSFileView, pNCSFileInfo->nBands,
  pBandList, 0, 0, pNCSFileInfo-> nSizeX-1,
  pNCSFileInfo->nSizeY-1, pNCSFileInfo->nSizeX,
  pNCSFileInfo->nSizeY)
  ```

- `NCSEcwCompressSetOEMKey(szLicensee, szOEMKey);`

SDK Decompression Library Functions Called

- `pClient = NCSEcwCompressAllocClient()`

- `NCSEcwCompressOpen(pClient, FALSE)`

- `NCSEcwCompress(pClient)`

- `NCSEcwCompress(pClient)`

- `NCSEcwCompressFreeClient(pClient)`

Other SDK Library Functions Called

- `ReadCallback(NCSEcwCompressClient *pClient,`

  ```
  UINT32 nNextLine, IEEE4
  **ppOutputBandBufferArray)
  ```

- `StatusCallback(NCSEcwCompressClient *pClient,`

  `UINT32 nCurrentLine)`

- `CancelCallback(NCSEcwCompressClient *pClient)`

Program Flow

1. Open an existing compressed image file:

   ```
   eError = NCScbmOpenFileView(
         szInputFilename, &pNCSFileView, NULL);
   ```

2. Get information from the compressed image file to set up a band list:

   ```
   eError = NCScbmGetViewFileInfo(
         pNCSFileView, &pNCSFileInfo);
   ```

3. Set the decompressed file view to encompass the whole image:

   ```
   eError = NCScbmSetFileView(pNCSFileView,
   ```

```
      pNCSFileInfo->nBands, pBandList, 0, 0,
      pNCSFileInfo->nSizeX-1, pNCSFileInfo->nSizeY-1,
      pNCSFileInfo->nSizeX, pNCSFileInfo->nSizeY);
```

4. Allocate a client compression structure:

   ```
   if(pClient = NCSEcwCompressAllocClient())
   ```

5. Insert input dimensions from the decompressed image information, and the required compression.

6. Specify the callback functions and client data pointers:

   ```
   pClient->pReadCallback = ReadCallback;
   pClient->pStatusCallback = StatusCallback;
   pClient->pCancelCallback = CancelCallback;
   ```

7. Set up client data for the read callback function:

   ```
   pClient->pClientData = (void *)&CompressReadInfo;
   ```

8. Open the compression:

   ```
   eError = NCSEcwCompressOpen(pClient, FALSE);
   ```

9. Do the compression:

   ```
   eError = NCSEcwCompressOpen(pClient, FALSE);
   ```

10. For every input image line call the developer-defined callback functions. The read callback function calls the decompression library Read function to enter the raster data into an array of input buffers:

    ```
    eReadStatus = NCScbmReadViewLineBIL(
          pReadInfo->pNCSFileView,
          pReadInfo->ppInputBandBufferArray);
    ```

11. Convert the raster data to `IEE4` and store it in an array of output buffers:

    ```
    for (nCell = 0; nCell < pClient->nInOutSizeX; nCell++)
    {
       *pOutputValue++ = (IEEE4)*pInputValue++;
    }
    ```

12. Close the compression and display the compression output statistics:

    ```
    NCSEcwCompressClose(pClient);
    ```

13. Free the compression client structure:

    ```
    NCSEcwCompressFreeClient(pClient);
    ```

Compression Example 3

This example program recompresses an input ECW or JPEG 2000 file to lossless JPEG 2000 or near lossless ECW using the C++ compression API. It provides a useful demonstration of the ease of use of this API for configuring JPEG 2000 and ECW output.

Number of Bands:	`nInputBands`	Same as input compressed image
Image Height:	`nInOutSizeY`	Same as input compressed image
Compression Format:	`eCompressFormat`	Set by number of bands: 1 band = Grayscale 2 band = Grayscale plus opacity 3 band = RGB 4 band = RGBA Other = Multi-band In this example, the band number is set to 4, so the format will be RGBA.
Target Compression Ratio:	`fTargetCompression`	1:1 forcing lossless JPEG 2000 output if a .jp2 output filename is selected.
Output File Name:	`szOutputFileName`	As entered by the user - use .jp2 extension to choose JPEG 2000 output.

The recommended methodology when using the C++ API of the SDK is to subclass the `CNCSFile` or `CNCSRenderer` classes and override certain functions to meet the specific needs of your application. In compression Example 3, the `CNCSFile` class is subclassed by `CLosslessCompressor`, which overrides the methods of `CNCSFile` associated with a compression process to do its job.

CNCSFile Methods Reimplemented By CLosslessCompressor

```
CNCSFile::WriteReadLine(
      UINT32 nNextLine,
      void **ppInputArray);
```

The overridden version of `WriteReadLine` is the cornerstone of the `CLosslessCompressor` class. `WriteReadLine` is the function called by the SDK to obtain new input data for compression during the execution of a compression task. In this case, the input data is being obtained from an ECW or JPEG 2000 file using the SDK itself, so only a simple call to `ReadLineBIL` on the input file is required.

```
CNCSFile::WriteStatus(UINT32 nCurrentLine);
CNCSFile::WriteCancel();
```

The two methods `WriteStatus` and `WriteCancel` can be overridden to manage the interaction of an SDK application with a compression task. In the case of `WriteStatus`, the SDK calls the method for each scanline of input data read during compression, and since the scanline number is provided as an argument to the method, it can be used to update progress bars or other status indicators in your program.

Other CNCSFile Methods Used

- `CNCSFile::Open(`

  ```
  char *pFilename,
  BOOLEAN bProgressive,
  BOOLEAN bWrite);
  ```

- `CNCSFile::GetFileInfo();`

- `CNCSFile::SetView(`

  ```
  UINT32 nBands, INT32 *pBandList,
  INT32 nSizeX, INT32 nSizeY,
  IEEE8 fStartX, IEEE8 fStartY,
  IEEE8 fEndX, IEEE8 fEndY);
  ```

- `CNCSFile::SetFileInfo(NCSFileViewFileInfoEx &pInfo);`

- `CNCSFile::Write();`

- `CNCSFile::Close(BOOLEAN bFreeCachedFile);`

- `CNCSFile::SetOEMKey(szLicensee, szOEMKey);`

Program Flow

Most of the work to code this simple SDK application can be seen in the `CLosslessCompressor::Recompress` method. The main function simply parses input arguments (in this case, input and output filenames), instantiates a `CLosslessCompressor` object, and calls its `Recompress` method. Within `Recompress` we can see the following chain of method calls:

1. The `Open` method is called on the `m_Src` member of the `CLosslessCompressor` object, which is itself an instance of `CNCSFile` used to handle input from another ECW or JPEG 2000 file.
2. The `GetFileInfo` method is called on `m_Src` to query it for metadata, allowing us to set up our output parameters correctly.
3. The file metadata obtained is used to set a view of maximal extents on the input file, so that its entire contents can be read line by line into the output file. These extents are used as arguments to the corresponding call to `SetView` on `m_Src`.
4. The output target compression ratio is set to 1, specifying lossless JPEG 2000 in the case of JPEG 2000 output or near lossless in the case of ECW.
5. Since there are no other changes required to the output metadata, a call to `SetFileInfo` is made on the `CLosslessCompressor` object, specifying output parameters. If `m_bGenNulls` is set, an extra opacity band will be created in the output dataset.
6. The `CLosslessCompressor` is opened with a call to `Open`. Note the arguments of `Open` which indicate that the output file should be opened in non-progressive mode for writing.

7. Output to file begins with a call to `Write` on the `CLosslessCompressor` object. This initiates a series of calls to `WriteReadLine`, `WriteStatus` and `WriteCancel` on the `CLosslessCompressor` object (one for each input scanline) as the SDK creates compressed output.
8. The output file is closed with a call to `Close`.

Decompression Examples

This section contains five different decompression related examples. These examples are located in the decompression folder which is a sub directory of the examples folder. It is also possible to load all of the examples by selecting the example_projects-msvc80 (90 or 10).sln. This is located in the examples folder or via the start menu.

Decompression Example 1

The dexample1.exe program uses the Blocking Reads Interface to the ECW library. To build the program, ensure that your Visual C++ option settings are correct.

SDK Decompression Library Functions Called

- `NCScbmOpenFileView(`

  `szInputFilename, &pNCSFileView, NULL)`

- `NCScbmGetViewFileInfo(pNCSFileView, &pNCSFileInfo)`
- `NCScbmSetFileView(`

  ```
  pNCSFileView, nBands,
  band_list, start_x,
  start_y, end_x, end_y,
  number_x, number_y)
  ```

- `NCScbmReadViewLineBIL(pNCSFileView, p_p_output_line)`
- `NCScbmCloseFileView(pNCSFileView)`
- `NCScbmCloseFileViewEx(pNCSFileView, FALSE)`

Program Flow

1. Open the file view and get image information:

```
NCScbmOpenFileView();
      // open a view into a file
NCScbmGetViewFileInfo();
      // get image size, map projection
      //info etc,
```

2. Repeat the following routine as many times as required for different views:

```
while(/* read a new view */) {
   NCScbmSetFileView();
   // set a view bands, extents, and window size
   while(lines--) {
      // read lines from the view
      // you can abort at any time if you don't
      // want to read all lines
      // (perhaps because the user wants a new
      // view)
      NCScbmReadViewLineRGB(); // or BIL or BGR read
      /* and give the line back to the application */
   }
}
```

3. Finish working with this file:

```
NCScbmCloseFileViewEx() // all done
```

Decompression Example 2

The dexample2.exe demonstrates the CALLBACK interface into the NCSECW Client Library. The application opens the view, then sets the view whenever appropriate (these may be done at any time). The example can be loaded by opening the file named dexample2.c. This is located in the example2 folder.

SDK Decompression Library Functions Called

- `NCSSleep(NCSTimeStampMs tsSleepTime);`
- `NCSecwInit();`
- `NCScbmOpenFileView(`

```
   pMyView->szInputFilename,
   &pMyView->pNCSFileView,
   ShowViewCallback);
```

- `NCScbmGetViewFileInfo(`

```
   pMyView->pNCSFileView,
   &pMyView->pFileInfo);
```

- `NCScbmGetViewInfo(`

```
   pMyView->pNCSFileView,
   &pMyView->pViewInfo);
```

- `NCScbmSetFileView(`

```
   pMyView->pNCSFileView,
```

```
   nBands,pMyView->pBandList,
   pMyView->nFromX, pMyView->nFromY,
   pMyView->nToX, pMyView->nToY,
   pMyView->nViewSizeX, pMyView->nViewSizeY);
```

- `NCScbmCloseFileView(`

```
   pMyView->pNCSFileView);
```

- `NCScbmReadViewLineRGB(`

```
   pNCSFileView,
   pRGBTriplets);
```

- `NCSecwShutdown();`

Program Flow

1. Initialize the library.

```
NCSecwInit();
```

2. Get input file name to display.

```
if (argc != 2) {
     printf("Usage:  %s file.ecw\n", argv[0]);
     return(1);
   }
```

3. Set the `pMyView` defaults.

```
   pMyView->szInputFilename = argv[1];
   pMyView->pReadImage = &pMyView->Image0;
   pMyView->pDisplayImage = &pMyView->Image1;
   pMyView->nViewSizeX = MAX_WINDOW;
   pMyView->nViewSizeY = MAX_WINDOW;
```

4. Call `ShowViews`.

```
   if( ShowViews(pMyView, 20) ) {
      printf("ShowViews on %s returned with an
error\n", pMyView->szInputFilename);
      return(1);
   }
```

5. Program flow of `ShowViews`.

   a. Open the View and get file size details.

```
NCScbmOpenFileView(pMyView->szInputFilename,
   &pMyView->pNCSFileView,
   ShowViewCallback)
```

   b. Get the `viewfileinfo`.

```
nError = NCScbmGetViewFileInfo(
   pMyView->pNCSFileView, &pMyView->pFileInfo);
```

c. Get the view specific information.

```
nError = NCScbmGetViewInfo(
   pMyView->pNCSFileView, &pMyView->pViewInfo);
```

d. For nSetViews (simulating user input):

Do a SetView.

```
while(nSetViewsToDo--) {
   SetViewExtents(pMyView);
   nError = NCScbmSetFileView(
      pMyView->pNCSFileView,
      nBands,pMyView->pBandList,
      pMyView->nFromX, pMyView->nFromY,
      pMyView->nToX, pMyView->nToY,
      pMyView->nViewSizeX, pMyView->nViewSizeY);
```

e. Sleep for a while.

```
NCSSleep((5000 * rand() / RAND_MAX));
```

f. Close the View and return.

```
NCScbmCloseFileView(pMyView->pNCSFileView);
```

6. Shutdown the library.

```
NCSecwShutdown();
```

7. Open the View and get file size details.

```
nError = NCScbmOpenFileView(pMyView->szInputFilename,
   &pMyView->pNCSFileView,
   ShowViewCallback);
```

8. Close the View and return.

```
nError = NCScbmCloseFileView(pMyView->pNCSFileView);
```

The routine that actually reads the view is the `ShowViewRefresh()` function.

Decompression Example 4

This example demonstrates the use of the `NCSecwSetIOCallbacks()` interface into the NCSECW library. This example is made up of three different parts. The example file is named dexample4.c which is located in the example4 folder.

1. The first part demonstrates the `BLOCKING` interface into the `NCSECW` library. The application opens the view, reads the view and then closes the view.
2. The second part uses the RGB read call.
3. The final part demonstrates the use of the `NCSecwSetIOCallbacks()` call to provide custom read routines.

SDK Decompression Library Functions Called

- `NCSecwInit();`
- `NCSecwSetIOCallbacks(`

  ```
  FileOpenCB, FileCloseCB,
  FileReadCB, FileSeekCB, FileTellCB);
  ```

- `NCScbmOpenFileView(argv[1], &pNCSFileView, NULL);`
- `NCScbmGetViewFileInfo(pNCSFileView, &pNCSFileInfo);`
- `NCScbmSetFileView(`

  ```
  pNCSFileView,
  nBands, Bands,
  nTLX, nTLY,
  nBRX, nBRY,
  nWidth, nHeight);
  ```

- `NCScbmReadViewLineRGB(pNCSFileView, pRGBTriplets);`
- `NCScbmCloseFileView(pNCSFileView);`
- `NCSecwShutdown();`

Callback Functions Used

- Tell File Callback - Get the current file offset.

  ```
  static NCSError NCS_CALL FileTellCB(void *pClientData,
  UINT64 *pOffset)
  ```

- Seek File Callback - Seek file to given offset.

  ```
  static NCSError NCS_CALL FileSeekCB(void *pClientData,
  UINT64 nOffset)
  ```

- Read File Callback - Read given length from current offset into buffer.

  ```
  static NCSError NCS_CALL FileReadCB(void *pClientData,
  void *pBuffer, UINT32 nLength)
  ```

- Close File Callback - Close file.

  ```
  static NCSError NCS_CALL FileCloseCB(void *pClientData)
  ```

- Open File Callback - open the file.

```
static NCSError NCS_CALL FileOpenCB(char *szFileName,
void **ppClientData)
```

Program Flow

1. Initialize the library.

   ```
   NCSecwInit();
   ```

2. Set up the callbacks.

   ```
   NCSecwSetIOCallbacks(FileOpenCB, FileCloseCB,
   FileReadCB, FileSeekCB, FileTellCB);
   ```

3. Open the input `NCSFileView`.

   ```
   NCScbmOpenFileView(argv[1], &pNCSFileView, NULL);
   ```

4. Get the file information.

   ```
   NCScbmGetViewFileInfo(pNCSFileView, &pNCSFileInfo);
   ```

5. Set `nBands` - 1 or 3 bands only (Grayscale or RGB).

   ```
   nBands = pNCSFileInfo->nBands < 3 ? 1 : 3;
   ```

6. Setup view dimensions.

   ```
   nTLX = 0;
      nTLY = 0;
      nBRX = pNCSFileInfo->nSizeX - 1;
      nBRY = pNCSFileInfo->nSizeY - 1;
      nWidth = MIN((INT32)pNCSFileInfo->nSizeX - 1,
   atoi(argv[2]));
      nHeight = MIN((INT32)pNCSFileInfo->nSizeY - 1,
   atoi(argv[3]));
   ```

7. Allocate scanline RGB triplet buffer.

   ```
   pRGBTriplets = (UINT8 *) malloc(nWidth * 3);
   ```

8. Set the view, using the dimensions calculated above.

   ```
   eError = NCScbmSetFileView(pNCSFileView,
                              nBands, Bands,
                              nTLX, nTLY,
                              nBRX, nBRY,
                              nWidth, nHeight);

            if(eError == NCS_SUCCESS) {
               INT32 nLine;
   ```

9. Read all scanlines.

   ```
   for(nLine = 0; nLine < nHeight; nLine++) {
                  NCSEcwReadStatus eStatus;

                  eStatus =
   NCScbmReadViewLineRGB(pNCSFileView, pRGBTriplets);
                  if(eStatus == NCSECW_READ_OK) {
                     INT32 nCell;
   ```

If a read fails the loop will be aborted.

```
eError = NCS_FILEIO_ERROR;
```

10. Dump RGB triplets to `stdout` as HEX.

```
for(nCell = 0; nCell < nWidth; nCell++) {
                    fprintf(stdout, "(%lx,%lx,%lx)",
                          (unsigned
long)pRGBTriplets[nCell * 3],
                          (unsigned
long)pRGBTriplets[nCell * 3 + 1],
                          (unsigned
long)pRGBTriplets[nCell * 3 + 2]);
                }
```

11. Free RGB triplet buffer.

```
free(pRGBTriplets);
```

12. Close file view.

```
NCScbmCloseFileView(pNCSFileView);
```

13. Shutdown library.

```
NCSecwShutdown();
```

Decompression Example 5

This example demonstrates how to read an entire image at dataset resolution.

The example is located in the example5 folder. Launch the dexample5.cpp file to access the example.

SDK Decompression Library Functions Called

- `NCSecwSetConfig(NCSCFG_CACHE_MAXMEM, 1000000000);`
- `NCSGetTimeStampMs();`
- `File.Open(FileName, false, false);`
- `File.GetFileInfo();`
- `File.SetView(`

```
   pInfo->nBands, pBandList, 0, 0,
   pInfo->nSizeX-1, pInfo->nSizeY-1, pInfo->nSizeX,
   pInfo->nSizeY);
```

- `ReadLineBIL(pLine);`
- `NCSFree(pLine[nBand]);`
- `NCSMalloc(pInfo->nBands * sizeof(UINT32), FALSE);`

- `File.Close(true);`

Program Flow

1. Set an ECW decompression configuration parameter.

   ```
   NCSecwSetConfig(NCSCFG_CACHE_MAXMEM, 1000000000);
   ```

2. Start the clock timer.

   ```
   NCSTimeStampMs tsStart = NCSGetTimeStampMs();
   ```

3. Open the file.

   ```
   if((Error = File.Open(sFileName, false, false)) ==
   NCS_SUCCESS)
   ```

4. Get file information.

   ```
   NCSFileViewFileInfoEx *pInfo = File.GetFileInfo();
   ```

5. Set the view.

   ```
   if((Error = File.SetView(pInfo->nBands, pBandList,
         0, 0,
         pInfo->nSizeX-1, pInfo->nSizeY-1,
         pInfo->nSizeX, pInfo->nSizeY));
   ```

6. Read each line in the file, and print a status message

   ```
   for(UINT32 nLine = 0; nLine < pInfo->nSizeY && Error ==
   NCS_SUCCESS; nLine++) {
      if(File.ReadLineBIL(pLine) == NCSECW_READ_OK) {
         fprintf(stdout, "Read: %d%% Complete\r",
   (int)(((((float)nLine) / pInfo->nSizeY) * 100) + 0.5));
   ```

7. The file is then closed.

   ```
   File.Close(true);
   ```

Decompression Example 6

This example demonstrates how to use the header editor interface.

The example is located in the example6 folder. Launch the dexample.cpp file to access the example.

SDK Decompression Library Functions Called

- `HeaderEditor.SetFile((char*)sFileName.a_str()))`
- `NCSEcwEditCopyInfo(HeaderEditor.GetEditInfo(),`

  `&pOriginalInfo);`
- `HeaderEditor.ApplyHeaderChanges()`

Program Flow

1. Set `headereditor` for the file to be edited (i.e. open file for editing via `HeaderEditor.SetFile()`).

```
NCS::CHeaderEditor HeaderEditor;
   if((Error =
HeaderEditor.SetFile((char*)sFileName.a_str())) !=
NCS_SUCCESS) {
      printf("Failed to open the file.\n\n%s\n\n",
NCSGetErrorText(Error));
      exit(1);
   }
```

2. Make a copy of headereditor information for editing.

```
NCSEcwEditCopyInfo(HeaderEditor.GetEditInfo(),
&pOriginalInfo);
```

3. Parse arguments (ie update/edit header editor information with parsed arguments).

```
   if(ParseArgs(argc, argv, &HeaderEditor)) {
      if((Error = HeaderEditor.ApplyHeaderChanges())
!= NCS_SUCCESS) {
         printf("Failed to write to the
file.\n\n%s\n\n", NCSGetErrorText(Error));
         exit(1);
```

4. Print comparison.

```
void PrintComparison(NCSEcwEditInfo *pOld,
NCSEcwEditInfo *pNew)
```

API Reference

Introduction

This section contains descriptions of the essential functions and classes you will be working with in the SDK. There are two methodologies that can be employed when writing applications based on the SDK, one of which uses C language functions in a procedural way, and the other of which uses file-oriented C++ objects. Throughout this section these two paradigms are referred to using the terms C API and C++ API. In addition to the functions, classes, methods and data structures documented here that fall under the umbrellas of the C and C++ APIs, some further utility functions that will be of benefit to you when programming with the SDK are also documented.

C API: Decompression Functions

The ECW JPEG2000 decompression library (`NCSEcw.lib`) has been designed for simplicity and ease of use. They allow you to open a view into an image file, set the extents of the view interactively, read the image data in a variety of different ways and close the view at the end.

You can also select either of two interfaces to the library;

- **Blocking:** when using the blocking interface, the application simply opens the view, reads the view, reads another view, and so on until it closes the view.
- **Refresh Callback:** when using the refresh callback interface, the application opens the view, and the view extents are then reset whenever appropriate (e.g. in response to user input). The library calls back to the application whenever new image data is available for reading.

There are separate functions to call depending on whether your application will read each line of every band in the image (in BIL format) or will return a straight RGB or BGR or other image regardless of what is in the source file the C functions required to decompress image data from ECW and JPEG 2000 files are documented below.

NCScbmCloseFileView

```
NCSError NCScbmCloseFileView(
      NCSFileView *pNCSFileView)
```

Remarks:

Closes an open view. You can do this at any time after a call to `NCScbmOpenFileView()`.

Parameters:

- NCSFileView *pNCSFileView - The file view to close.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
nError = NCScbmCloseFileView(pMyView->pNCSFileView);
if (nError != NCS_SUCCESS)
   printf("Error = %s\n", NCSGetErrorText(nError));
```

NCScbmCloseFileViewEx

```
NCSError NCScbmCloseFileViewEx(
      NCSFileView *pNCSFileView,
      BOOLEAN bFreeCachedFile)
```

Remarks

This function is similar to `NCScbmCloseFileView()` in that it also closes a file view. The only difference between this and the non-Ex call is that you can force it to close the file by passing in `TRUE` for `bFreeCachedFile`. It behaves exactly the same as `NCScbmCloseFileView()` if you pass in `FALSE` for `bFreeCachedFile`. The file only closes when all the views for that file are closed (a single file may have multiple views open on it at any given time).

Parameters:

- `NCSFileView *pNCSFileView` - The file view to close.
- `BOOLEAN bFreeCachedFile` - Set to `TRUE` to force closure of the file and the release of associated memory.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

NCScbmGetViewFileInfo

```
NCSError NCScbmGetViewFileInfo(
      NCSFileView *pNCSFileView,
      NCSFileViewFileInfo **ppNCSFileViewFileInfo)
```

Remarks:

Obtains generic information about the image file associated with an open file view. No information specific to a call of `NCScbmSetFileView` is available via this call. To obtain such information use `NCScbmGetViewInfo()`. Use this call in conjunction with a view on an open ECW file.

Parameters:

- `NCSFileView *pNCSFileView` - The file view to close.
- `NCSFileViewFileInfo **ppNCSFileViewFileInfo` - Non view specific file information.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
nError = NCScbmGetViewFileInfo(
      pMyView->pNCSFileView,&pMyView->pFileInfo);
if (nError != NCS_SUCCESS)
   printf("Error = %s\n", NCSGetErrorText(nError));
```

NCScbmGetViewFileInfoEx

```
NCSError NCScbmGetViewFileInfoEx(
      NCSFileView *pNCSFileView,
      NCSFileViewFileInfoEx **ppNCSFileViewFileInfo)
```

Remarks:

Obtains generic information about the image file associated with an open file view. No information specific to a call of `NCScbmSetFileView` is available via this call. To obtain such information use `NCScbmGetViewInfo()`. Use this call in conjunction with a view on an open JPEG 2000 file.

Parameters:

- `NCSFileView *pNCSFileView` - The file view to close.
- `NCSFileViewFileInfoEx **ppNCSFileViewFileInfoEx` - Non view-specific file information.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
nError = NCScbmGetViewFileInfoEx(
```

```
        pMyView->pNCSFileView, &pMyView->pFileInfo);
if (nError != NCS_SUCCESS)
   printf("Error = %s\n", NCSGetErrorText(nError));
```

NCScbmGetViewInfo

```
NCSError NCScbmGetViewInfo(
      NCSFileView *pNCSFileView,
      NCSFileViewSetInfo **ppNCSFileViewSetInfo)
```

Remarks:

Gets view information about the view data currently being processed.

Parameters:

- `NCSFileView *pNCSFileView` - The file view to close.
- `NCSFileViewSetInfo **ppNCSFileViewSetInfo` - Current `Setview` information.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
NCScbmGetViewInfo(
      pMyView->pNCSFileView, &pMyView->pViewInfo);
```

NCScbmOpenFileView

```
NCSError NCScbmOpenFileView(
      char *szUrlPath,
      NCSFileView **ppNCSFileView,
      NCSEcwReadStatus (*pRefreshCallback)(NCSFileView
      *pNCSFileView))
```

Remarks:

Opens a file view. After doing this, you can call `NCScbmGetViewFileInfo()` to get the file details.

Parameters:

- `char *szUrlPath` - Name of the file to be opened. This can also be an ecwp:// URL.
- `NCSFileView *pNCSFileView` - The file view.

- `NCSEcwReadStatus (*pRefreshCallback) (NCSFileView *pNCSFileView)-` Routine called by the library whenever the view needs to be refreshed, based on the available imagery information. Set this to `NULL` if you are not using the refresh callback interface.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
//Simple Read Region Interface
eError = NCScbmOpenFileView(
      szInputFilename, &pNCSFileView, NULL);
if (nError != NCS_SUCCESS)
   printf("Error = %s\n", NCSGetErrorText(nError));
//Interactive Callback interface
NCScbmOpenFileView(
      pMyView->szInputFilename,
      &pMyView->pNCSFileView,ShowViewCallback);
```

NCScbmReadViewLineBGR

```
NCSEcwReadStatus NCScbmReadViewLineBGR(
      NCSFileView *pNCSFileView,
      UINT8 *pRGBTriplets)
```

Remarks:

Reads line by line in BGR format.

Parameters:

- `NCSFileView *pNCSFileView` - The file view from which to read.
- `UINT8 *pRGBTriplets` - A buffer for the RGB triplets being read.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful.
- `NCSECW_READ_FAILED = 1`, read failed due to an error.
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
NCScbmReadViewLineBGR(pNCSFileView, pRGBTriplets);
```

NCScbmReadViewLineBGRA

```
NCSEcwReadStatus NCScbmReadViewLineBGRA(
      NCSFileView *pNCSFileView, UINT32 *pRGBA)
```

Remarks:

Reads line by line in BGRA format, packed into 32bits with 8 bits each for Red, Green, Blue and Alpha. The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT32 *pBGRA` - A pointer to a UINT32 buffer for BGRA values being read.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful.
- `NCSECW_READ_FAILED = 1`, read failed due to an error.
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
NCScbmReadViewLineBGRA(pNCSFileView, pBGRA);
```

NCScbmReadViewLineBIL

```
NCSEcwReadStatus NCScbmReadViewLineBIL(
      NCSFileView *pNCSFileView,
      UINT8 **ppOutputLine)
```

Remarks:

Reads line by line in BIL format.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT8 **ppOutputLine` - A buffer passed in to store the BIL data read by the call.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful.
- `NCSECW_READ_FAILED = 1`, read failed due to an error.
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
eReadStatus = NCScbmReadViewLineBIL(
      pNCSFileView,ppOutputLine);
if (eReadStatus != NCSECW_READ_OK)
   printf("Status code = %e\n", eReadStatus);
```

NCScbmReadViewLineBILEx

```
NCSEcwReadStatus NCScbmReadViewLineBILEx(
      NCSFileView *pNCSFileView,
      NCSEcwCellType eType,
      void **ppOutputLine)
```

Remarks:

`Read line by line in BIL format to buffers with different cell types. This extended version allows the client program to read in view lines made up of cells with sample bitdepth other than 8 bit.`

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `NCSEcwCellType eType` - The sample type of the buffer into which to read BIL pixel data.
- `void **ppOutputLine` - A buffer passed in to store the BIL data read by the call.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful.
- `NCSECW_READ_FAILED = 1`, read failed due to an error.
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
eReadStatus = NCScbmReadViewLineBILEx(
      pNCSFileView, NCSCT_UINT16, ppOutputLine);
if (eReadStatus != NCSECW_READ_OK)
   printf("Status code = %e\n", eReadStatus);
```

NCScbmReadViewLineRGB

```
NCSEcwReadStatus NCScbmReadViewLineRGB(
      NCSFileView *pNCSFileView,
      UINT8 *pRGBTriplets)
```

Remarks:

Reads line by line in RGB format.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT8 *pRGBTriplets RGB -` Triplets being read.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful
- `NCSECW_READ_FAILED = 1`, read failed due to an error
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
eReadStatus = NCScbmReadViewLineRGB(
      pNCSFileView, pRGBTriplets);
if(eReadStatus == NCSECW_READ_CANCELLED )
   printf("*** Read was cancelled.\n");
```

NCScbmReadViewLineRGBA

```
NCSEcwReadStatus NCScbmReadViewLineRGBA(
      NCSFileView *pNCSFileView,
      UINT32 *pRGBA)
```

Remarks:

Reads line by line in RGBA format, packed into 32 bits with 8 bits each for Red, Green, Blue and Alpha. The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT32 *pRGBA` - A pointer to `UINT32` buffer for RGBA values being read.

Returns:

`NCSEcwReadStatus`, one of:

- `NCSECW_READ_OK = 0`, read was successful.
- `NCSECW_READ_FAILED = 1`, read failed due to an error.
- `NCSECW_READ_CANCELLED = 2`, read was cancelled, due to:
  - The application moving to process a new `SetView`, or
  - library shutdown in progress.

Example:

```
NCScbmReadViewLineRGBA(pNCSFileView, pRGBA);
```

NCScbmSetFileView

```
NCSError NCScbmSetFileView(
      NCSFileView *pNCSFileView,
      UINT32 nBands,
      UINT32 *pBandList,
      UINT32 nTLX, UINT32 nTLY,
```

```
                UINT32 nBRX, UINT32 nBRY,
                UINT32 nSizeX, UINT32 nSizeY)
```

Remarks:

Sets the extents and image components associated with an open file view. You can do this at any time after a call to `NCScbmOpenFileView`. Multiple `SetFileViews` can be done, even if previous `SetFileViews` have not finished processing yet. After the call to `NCScbmSetFileView` is made, you can free the bandlist (`pBandList`) if you wish (it is used only during the call, and not afterwards). You can specify the view to be part of the image by setting the required number of bands and the top left and bottom right coordinates of an area within the image.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT32 nBands` - The number of bands to read.
- `UINT32 *pBandList` - The index into the actual band numbers from the source file. Band numbering starts at 0.
- `UINT32 nTLX` - The top left X of the view in dataset coordinates.
- `UINT32 nTLY` - The top left Y of the view in dataset coordinates.
- `UINT32 nBRX` - The bottom right X of the view in dataset coordinates.
- `UINT32 nBRY` - The bottom right Y of the view in dataset coordinates.
- `UINT32 nSizeX` - The view size X in dataset cells.
- `UINT32 nSizeY` - The view size Y in dataset cells.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
eError = NCScbmSetFileView(
      pNCSFileView,
      nBands,
      pBandList,
      nTLX, nTLY
      nBRX, nBRY
      nSizeX, nSizeY);
if(eError != NCS_SUCCESS)
   printf("Error = %s\n", NCSGetErrorText(nError));
```

NCScbmSetFileViewEx

```
NCSError NCScbmSetFileViewEx(
```

```
NCSFileView *pNCSFileView,
UINT32 nBands,
UINT32 *pBandList,
UINT32 nTLX, UINT32 nTLY,
UINT32 nBRX, UINT32 nBRY,
UINT32 nSizeX, UINT32 nSizeY,
IEEE8 fTLX, IEEE8 fTLY,
IEEE8 fBRX, IEEE8 fBRY))
```

Remarks:

This is similar to the `NCScbmSetFileView` function, with the added possibility of passing in the real world coordinates, so that you know what they are in your refresh callback function. This is necessary where dataset cells are rounded based on the current scale, but you need the exact world coordinates in the callback function for some reason.

You can specify the view to be part of the image by setting the required number of bands and the top left and bottom right coordinates of an area within the image.

Parameters:

- `NCSFileView *pNCSFileView` - The file view.
- `UINT32 nBands` - The number of bands to read.
- `UINT32 *pBandList` - The index into the actual band numbers from the source file. Band numbering starts at 0.
- `UINT32 nTLX` - The top left X of the view in dataset coordinates.
- `UINT32 nTLY` - The top left Y of the view in dataset coordinates.
- `UINT32 nBRX` - The bottom right X of the view in dataset coordinates.
- `UINT32 nBRY` - The bottom right Y of the view in dataset coordinates.
- `UINT32 nSizeX` - The view size X in raster cells.
- `UINT32 nSizeY` - The view size Y in raster cells.
- `IEEE8 fTLX` - The top left X of the view in world coordinates.
- `IEEE8 fTLY` - The top left Y of the view in world coordinates.
- `IEEE8 fBRX` - The bottom right X of the view in world coordinates.
- `IEEE8 fBRY` - The bottom right Y of the view in world coordinates.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
eError = NCScbmSetFileViewEx(
      pNCSFileView,
      nBands,
      pBandList,
      nTLX, nTLY,
      UINT32 nBRX, nBRY,
      nSizeX, nSizeY
      fTLX, fTLY,
      fBRX, fBRY);
```

NCSecwSetConfig

```
NCSError NCSecwSetConfig(NCSEcwConfigType eType, ...)
```

Remarks

Set an ECW decompression configuration parameter.

Parameters:

- `NCSEcwConfigType eType` - The ECW configuration parameter to set.

The desired value(s) of configuration parameters, as below:

Parameter	Argument Type	Notes
NCSCFG_TEXTURE_DITHER	BOOLEAN	Apply texture dither to decompressed image for ECW files. Default is TRUE.
NCSCFG_FORCE_FILE_REOPEN	BOOLEAN	Force each individual view to open a separate file/connection. Default is FALSE.
NCSCFG_CACHE_MAXMEM	UINT32	TARGET maximum memory to use for ECW cache, in bytes. Default is ¼ of RAM, up to 2GB.
NCSCFG_CACHE_MAXOPEN	UINT32	TARGET maximum number of open files to use for ECW cache.
NCSCFG_USE_BUFFERED_IO_STREAM	BOOLEAN	Enable/disable buffered IO in the JPEG 2000 reader (use `CBufferedIOStream` instead of `CFileIOStream`). Default is FALSE.
NCSCFG_JP2_FILEIO_CACHE_MAXMEM	UINT32	Buffer size (bytes) to control the block size per IO reading when `CBufferedIOStream` is used.
NCSCFG_ECWP_CACHE_ENABLED	BOOLEAN	Enable the persistant local ECWP cache. Default is FALSE.

NCSCFG_ECWP_CACHE_SIZE_MB	INT32	The maximum size of the local cache, in megabytes.
NCSCFG_ECWP_CACHE_LOCATION	char*	The local directory where the cache is stored, as a C string. A folder named “ecwpcache” will be created within the specified folder. The default is the users temp directory.
NCSCFG_JP2_AUTOGEN_J2I	BOOLEAN	Enables automatic generation of J2I index files. Default is TRUE.
NCSCFG_OPTIMIZE_USE_NEAREST_NEIGHBOUR	BOOLEAN	Specifies nearest neighbour resampling (TRUE, faster but less quality) or bilinear resampling (FALSE, slower, but better quality) for JPEG 2000 files. Default is FALSE.
NCSCFG_RESILIENT_DECODING	BOOLEAN	Enables resilient decoding for ECW files. When TRUE, the SDK will attempt to decode corrupt ECW files, and return and image composed of only the valid data. If FALSE the SetView or ReadLineXXX functions will throw an error. The default is TRUE.

Returns:

An NCSError value to use for error checking. The return value is NCS_SUCCESS if the operation succeeds.

Example:

```
NCSecwSetConfig(
      NCSCFG_FORCE_FILE_REOPEN, (BOOLEAN)TRUE);
```

NCSecwSetIOCallbacks

```
NCSError NCSecwSetIOCallbacks(
         NCSError (NCS_CALL *pOpenCB)
         (char *szFileName, void **ppClientData),
   NCSError (
         NCS_CALL *pCloseCB)(void *pClientData),
         *pClientData),
   NCSError (
         NCS_CALL *pReadCB)(void *pClientData,
         void *pBuffer, UINT32 nLength),
   NCSError (
         NCS_CALL *pSeekCB)(void *pClientData, UINT64
         nOffset),
   NCSError (
         NCS_CALL *pTellCB)(void *pClientData, UINT64
         *pOffset))
```

Remarks:

An API call which allows an application to specify callback functions to be used for ECW file I/O instead of the built-in functions. This allows embedding of ECW files into another file, a database etc., without requiring extracting to a temporary file before opening via the SDK.

Callback routines should be coded to handle 64-bit file sizes.

Parameters:

- `pOpenCB` - Callback function for opening files.
- `pCloseCB` - Callback for closing files.
- `pReadCB` - Callback for reading from files.
- `pSeekCB` - Callback for seeking in files.
- `pTellCB` - Callback for determining the location of the current file pointer in the file.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Decompression: Related Data Structures

When decompressing data from an ECW or JPEG 2000 file, it is a natural requirement to be able to request information about the file, such as its size in pixels, number of components, or geographic location. This information allows an application to correctly allocated resources for the decompressed data and process it in an appropriate way.

The SDK makes two kinds of information available to a client program upon opening a view into an ECW or JPEG 2000 file. The first kind is generic information about the image compressed in the file, which remains constant regardless of the characteristics of the current view being processed. The second kind is information specific to the view currently being processed, such as its extents, and the amount of view data available. Different functions are provided in the C and C++ APIs for accessing the two different kinds of information. The information returned to the client program is stored in data structures called `NCSFileViewFileInfo`, `NCSFileViewFileInfoEx`, and `NCSFileViewSetInfo`.

NCSFileViewFileInfo

```
typedef struct
{
   UINT32 nSizeX, nSizeY;
```

```
   UINT16 nBands;
   UINT16 nCompressionRate;
   CellSizeUnits eCellSizeUnits;
   IEEE8 fCellIncrementX;
   IEEE8 fCellIncrementY;
   IEEE8 fOriginX;
   IEEE8 fOriginY;
   char *szDatum;
   char *szProjection;
}NCSFileViewFileInfo
```

Remarks:

The `NCSFileViewFileInfo` structure contains all the static file information associated with an open ECW file view. The `NCScbmGetViewFileInfo()` function returns a pointer to this structure.

Parameters:

- `nSizeX, nSizeY` - Image size in number of raster cells.
- `nBands` - Number of bands in the file, e.g. 3 for an RGB image file.
- `nCompressionRate` - Approximate compression rate. May be zero. e.g. 20 = 20:1 compression.
- `eCellSizeUnits` - Units used for raster cell size. This can be one of the following (set to `ECW_CELL_UNITS_METERS` for `RAW` type files):

  ```
  ECW_CELL_UNITS_INVALID = 0,
  ECW_CELL_UNITS_METERS = 1,
  ECW_CELL_UNITS_DEGREES = 2,
  ECW_CELL_UNITS_FEET = 3
  ```

- `fCellIncrementX` - Cell size across in `CellSizeUnits`. May be negative, but never zero.
- `fCellIncrementY` - Cell size down in `CellSizeUnits`. May be negative, but never zero.
- `fOriginX` - World X coordinate for topleft corner of top left cell, in `CellSizeUnits`.
- `fOriginY` - World Y coordinate for topleft corner of top left cell, in `CellSizeUnits`.
- `szDatum` - ER Mapper style datum name string, e.g. `RAW` or `NAD27`. Will never be `NULL`.
- `szProjection` - ER Mapper style projection name string, e.g. `RAW` or `WGS84`. Will never be `NULL`.

NCSFileViewFileInfoEx

```
typedef struct
{
```

```
   //Members of NCSFileViewFileInfo ...
   //Additional data
   IEEE8 fCWRotationDegrees;
   NCSFileColorSpace eColorSpace;
   NCSEcwCellType eCellType;
   NCSFileBandInfo *pBands;
}
NCSFileViewFileInfoEx;
```

Remarks:

The `NCSFileViewFileInfoEx` structure contains static file information about an open JPEG 2000 file view. This information is obtained from a call to `NCScbmGetViewFileInfoEx()` in the C API. The structure contains similar components to `NCSFileViewFileInfo`, with some additional data corresponding to the added flexibility of the JPEG 2000 file format.

Parameters:

- `fCWRotationDegrees` - Clockwise rotation of the image in world coordinate space, expressed in degrees.
- `eColorSpace` - Color space of the image, one of:
  - `NCSCS_NONE = 0,`
  - `NCSCS_GREYSCALE = 1,`
  - `NCSCS_YUV = 2,`
  - `NCSCS_MULTIBAND = 3,`
  - `NCSCS_sRGB = 4,`
  - `NCSCS_YCbCr = 5`
- `eCellType` - Data type of the sample values in each image component at each pixel, one of:
  - `NCSCT_UINT8,`
  - `NCSCT_UINT16,`
  - `NCSCT_UINT32,`
  - `NCSCT_UINT64,`
  - `NCSCT_INT8,`
  - `NCSCT_INT16,`
  - `NCSCT_INT32,`
  - `NCSCT_INT64,`
  - `NCSCT_IEEE4,`
  - `NCSCT_IEEE8`
- `pBands` - Pointer to an array of `NCSFileBandInfo` structures describing the data content of each band in the image. See the description of the `NCSFileBandInfo` structure below for details.

NCSFileBandInfo

```
typedef struct
{
   UINT8 nBits;
   BOOLEAN bSigned;
   char *szDesc;
}
NCSFileBandInfo;
```

Remarks:

The `NCSFileBandInfo` struct contains details about the bit depth and signedness of the data in each band of a JPEG 2000 file, and also a short ASCII description of the band, e.g. `Red` or `Band #1`, which is automatically created by the SDK for you based on the probable content of the file.

NCSFileViewSetInfo

```
typedef struct
{
   void *pClientData;
   UINT32 nBands;
   UINT32 *pBandList;
   UINT32 nTLX, nTLY;
   UINT32 nBRX, nBRY;
   UINT32 nSizeX, nSizeY;
   UINT32 nBlocksInView;
   UINT32 nBlocksAvailable;
   UINT32 nBlocksAvailableAtSetView;
   UINT32 nMissedBlocksDuringRead;
   IEEE8 fTLX, fTLY;
   IEEE8 fBRX, fBRY;
}
NCSFileViewSetInfo;
```

Remarks:

The NCSFileViewSetInfo structure contains information specific to the processing of data from the current view, including the view extents, active band list, the amount of view data available, and client data corresponding to the developer's application. The NCScbmGetViewSetInfo() function returns a pointer to this structure.

Parameters:

- `pClientData` - Client data.
- `nBands` - Number of bands to read.
- `pBandList` - Array of band numbers being read.

- `nTLX, nTLY` - Top left of view in image cell coordinates.
- `nBRX, nBRY` - Bottom right of view in image cell coordinates.
- `nSizeX, nSizeY` - Size of view in pixels.
- `nBlocksInView` - Total number of blocks that cover the view area.
- `nBlocksAvailable` - Number of blocks available at this instant.
- `nBlocksAvailableAtSetView` - Number of blocks that were available at the time the view was last set.
- `nMissedBlocksDuringRead` - Number of blocks that were not present during a view read.
- `fTLX, fTLY` - Top left of the view in world coordinates as set by `NCScbmSetFileViewEx`.
- `fBRX, fBRY` - Bottom right of the view in world coordinates.

If you use `NCScbmSetFileView()` instead of `NCScbmSetFileViewEx()`, you get the dataset coordinates (and not the world coordinates) returned in `fTLX`, `fTLY`, and `fBRX`, `fBRY`.

If you want to determine the progress of a view download, you can use the `NCScbmGetViewInfo()` function to set up an `NCSFileViewSetInfo` structure for you. You can then calculate the progress from the `nBlocksInView` and `nBlocksAvailable` fields.

C API: Compression Functions

The C interface for compressing to the ECW and JPEG 2000 file formats has been designed for simplicity and ease of use. There are six associated library functions and three callback functions which must be supplied by the developer for reading the data to be compressed, checking the status of the compression process, and cancelling the compression process if necessary. All the data, including the names of the callback functions, are contained in a single compression client data structure.

An application will generally follow this schedule when compressing an image file via the C API:

- Read, status and cancel callbacks are defined in separate functions.
- A compression client data structure is created using `NCSEcwCompressAllocClient`.
- The fields of the client data structure are populated according to the purpose of the application.
- `NCSEcwCompressOpen` is called to initialize a new compression process.
- `NCSEcwCompress` is called to commence compressing data. As new data is required for compression the developer-defined read callback function acquires it from other data resources available to the application

(for example an uncompressed buffer of image data loaded from another image file).

- After compression completes, `NCSEcwCompressClose` is called to release the resources used during compression and clean up.
- Finally `NCSEcwCompressFreeClient` is called to release the memory allocated to the compression client data structure used to configure the compression task.

ECW compression is a recursive process and requires a large stack space to prevent stack overflow, especially when compressing large images. This can be a problem when using the ECW JPEG2000 compression SDK inside an ATL control as they have limited stack space. A solution is to call the `NCSecw*()` *calls from a thread you create inside the control (with a reasonable sized stack), rather than directly from the control's exported methods.*

NCSEcwCompressAllocClient

```
NCSEcwCompressClient *NCSEcwCompressAllocClient(void)
```

Remarks:

Allocates a new Compression Client structure, `NCSEcwCompressClient`, and fills in default values.

Parameters:

None.

Returns:

A new `NCSEcwCompressClient` structure containing default values.

Example:

```
pClient = NCSEcwCompressAllocClient();
```

NCSEcwCompressOpen

```
NCSError NCSEcwCompressOpen(
      NCSEcwCompressClient *pInfo,
      BOOLEAN bCalculateSizesOnly)
```

Remarks:

Opens the compression, initializing the process using data given in the `NCSEcwCompressClient` structure.

Parameters:

- `NCSEcwCompressClient *pInfo` - This is a pointer to the compression client structure, which contains compression information and requirements.
- `BOOLEAN bCalculateSizesOnly` - This is set to `TRUE` only if an estimate of the size of the output compressed file is required without doing the compression.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
NCSEcwCompressOpen(pInfo, (BOOLEAN)FALSE);
```

NCSEcwCompress

```
NCSError NCSEcwCompress(NCSEcwCompressClient *pInfo)
```

Remarks:

This runs the compression process, which calls the developer-defined callback functions referenced within the compression client structure by the `pReadCallback()`, `pStatusCallback()` and `pCancelCallback()` pointers.

Parameters:

- `NCSEcwCompressClient *pInfo` - This is a pointer to the compression client structure which contains compression information and requirements, including the `pReadCallback()`function.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
eError = NCSEcwCompress(pClient);
```

NCSEcwCompressClose

```
NCSError NCSEcwCompressClose(
      NCSEcwCompressClient *pInfo)
```

Remarks:

This closes a compression process and releases the associated resources.

Parameters:

- `NCSEcwCompressClient *pInfo` - This is a pointer to the compression client structure which contains information and requirements for the process.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
NCSEcwCompressClose(pClient);
```

NCSEcwCompressFreeClient

```
NCSError NCSEcwCompressFreeClient(
      NCSEcwCompressClient *pInfo)
```

Remarks:

This frees the compression client structure.

Parameters:

- `NCSEcwCompressClient *pInfo -` This is a pointer to the compression client structure that is being freed.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
NCSEcwCompressFreeClient(pClient);
```

NCSEcwCompressSetOEMKey

```
void NCSEcwCompressSetOEMKey(
      char *szCompanyName,
      char *szKey);
```

Remarks:

This function, which is only available in the read/write version of the SDK, allows the developer to set their license key which enables the compression feature of the ECW JPEG2000 SDK.

Parameters

- `char *szCompanyName` - The name of the company for which the key has been generated for.
- `char *szKey` - The generated key code provided by ERDAS.

Returns:

None.

The company name needs to be the same that was provided to ERDAS when generating the key. The key needs to be set by either the C or C++ function call before a file can be opened for write.

Compression: Developer Defined Functions

To implement the SDK's ECW and JPEG 2000 compression scheme using the C API, you must provide code for a mandatory callback function for reading input image data, and may also provide status and cancellation callback functions.

These functions are called by the library function `NCSEcwCompress()` during the compression process.

- `pReadCallback` - mandatory
- `pStatusCallback` - optional
- `pCancelCallback` - optional

If you do not wish to specify a status callback or a cancel callback, you can leave the values of the associated function pointers in your `NCSEcwCompressClient` data structure with value `NULL`. Any callback functions you define must also be threadsafe because they are called from a different thread to that from which `NCSEcwCompress` is called.

pReadCallback

```
BOOLEAN *pReadCallback(
      struct NCSEcwCompressClient *pClient,
      UINT32 nNextLine,
      IEEE4 **ppInputArray)
```

Remarks:

This mandatory developer-created function is called by `NCSEcwCompress()` for every line of the input image, designated by the `nNextLine` argument, and reads in the cell values for each band into the variable `ppInputArray`.

Parameters:

- `NCSEcwCompressClient *pClient` - This is a pointer to the compression client structure which contains information about the compression task.
- `UINT32 nNextLine` - This is the number of the next image scan line to be read in.
- `IEEE4 **ppInputArray` - This is the array into which the callback function must load the input cell values for each cell of each band.

Returns:

TRUE if successful, FALSE if in error.

pStatusCallback

```
void *pStatusCallback(
      struct NCSEcwCompressClient *pClient,
      UINT32 nCurrentLine)
```

Remarks:

This optional developer-created function is called by `NCSEcwCompress()` for every line of the input image, designated by the `nCurrentLine` argument. It provides status information on the compression progress, which can be used to keep a user of your SDK application advised of the rate at which compression is occurring.

Parameters:

- `NCSEcwCompressClient *pClient` - This is a pointer to the compression client structure which contains information about the compression task.
- `UINT32 nCurrentLine` - This is the number of the line of the input data currently being read.

Returns:

None.

pCancelCallback

```
BOOLEAN *pCancelCallback(
      struct NCSEcwCompressClient *pClient)
```

Remarks:

This optional developer-created function is called by `NCSEcwCompress()` during the compression. If it returns a value of `TRUE`, the compression process is aborted.

Parameters:

- `NCSEcwCompressClient *pClient` - This is a pointer to the compression client structure which contains information about the compression task.

Returns:

FALSE to continue compression, TRUE to cancel.

Compression: Related Data Structures

There is one main data structure that contains all the information pertinent to the compression.

NCSEcwCompressClient

```
typedef struct NCSEcwCompressClient
{
   char szInputFilename[MAX_PATH];
   char szOutputFilename[MAX_PATH];
   IEEE4 fTargetCompression;
   CompressFormat eCompressFormat;
   CompressHint eCompressHint;
   UINT32 nBlockSizeX;
   UINT32 nBlockSizeY;
   UINT32 nInOutSizeX;
   UINT32 nInOutSizeY;
   UINT32 nInputBands;
   UINT32 nOutputBands;
   UINT64 nInputSize;
   IEEE8 fCellIncrementX;
   IEEE8 fCellIncrementY;
   IEEE8 fOriginX;
   IEEE8 fOriginY;
   CellSizeUnits eCellSizeUnits;
   char szDatum[ECW_MAX_DATUM_LEN};
   char szProjection[ECW_MAX_PROJECTION_LEN];
   BOOLEAN (*pReadCallback) (
      struct NCSEcwCompressClient *pClient,
      UINT32 nNextLine,
      IEEE4 **ppInputArray);
   void (*pStatusCallback) (
      struct NCSEcwCompressClient *pClient,
      UINT32 nCurrentLine);
   BOOLEAN (*pCancelCallback) (
         struct NCSEcwCompressClient *pClient);
   void pClientData;
   struct EcwCompressionTask pTask;
```

```
    //These are filled in by NCSEcwCompressClose()
    IEEE4 fActualCompression;
    IEEE8 fCompressionSeconds;
    IEEE8 fCompressionMBSec;
    UINT64 nOutputSize;
}
NCSEcwCompressClient;
```

Remarks:

The `NCSEcwCompressClient` compression client structure contains all the compression information. Some data must be defined by the application developer prior to commencement, the compression client inserts some of the component values, while others are inserted by the compression library functions.

Information From The Application Developer

Each of the following data items must be specified by the application developer prior to commencing a compression process. If they are not specified the default values will be used where they exist, or an error will occur.

- `szInputFilename[]` - This is the path and file name of the image being compressed. It is optional. If no output file is specified, a default output file name will be derived from the input file name.
- `szOutputFilename[]` - This is the path and file name of the resultant compressed image. This is mandatory unless the input file name has been specified.
- `fTargetCompression` - This is the target compression rate sought. This rate will usually be larger than the compression rate actually achieved.
- `eCompressFormat` - This specifies the required compression format. Valid format codes are as follows:
  - `0 = COMPRESS_NONE` (no compression).
  - `1 = COMPRESS_UINT8` (single band, grayscale, UINT8 compression).
  - `2 = COMPRESS_YUV` (RGB images in YUV digital format, e.g. JPEG standard YUV).
  - `3 = COMPRESS_MULTI` (Multiband).
  - `4 = COMPRESS_RGB` (RGB conv. to YUV, format set internally to COMPRESS_YUV).
- `eCompressHint` - This specifies the required compression type. Valid compression type codes are as follows:
  - `0 = COMPRESS_HINT_NONE` (no compression).
  - `1 = COMPRESS_HINT_FAST` (perform the fastest possible compression).

- `2 = COMPRESS_HINT_BEST` (Perform the best possible compression).
- `3 = COMPRESS_HINT_INTERNET` (Default: optimize for Internet use).

- `nBlockSizeX, nBlockSizeY` - These specify the dimensions of the compressed image block size. `nBlockSizeX` can be 64, 128, 256, 512, 1024 or 2048.`nBlockSizeY` can be 64, 128, 256 or 512. The default value for both is 64.

The SDK 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 backwards compatability with older SDK applications.

- `nInOutSizeX,nBlockSizeY` - This specifies the number of cells in the input and compressed image, in X and Y directions.
- `nInputBands` - This is the number of bands in the input range.
- `nOutputBands` - This is the number of bands in the output file (this should not generally be specified).
- `nInputSize` - The size of the input file in bytes. This field is determined automatically and should not be specified.
- `fCellIncrementX,fCellIncrementY` - This is the input image cell size in cell size units.
- `fOriginX,fOriginY` - These are the world coordinates of the input image registration cell.
- `eCellSizeUnits` - This is the cell size units. This can be one of:
  - `0 = ECW_CELL_UNITS_INVALID`
  - `1 = ECW_CELL_UNITS_METERS` (default setting for `RAW` type images)
  - `2 = ECW_CELL_UNITS_DEGREES`
  - `3 = ECW_CELL_UNITS_FEET`
- `szDatum[]` - This is the image datum (ER Mapper GDT format).
- `szProjection[]` -This is the image projection (ER Mapper GDT format).

C API: Utility Functions

The following utility functions can be used via the C API to streamline your SDK application code, providing support for several common tasks.

NCScbmGetFileMimeType

```
char *NCScbmGetFileMimeType(NCSFileView *pNCSFileView)
```

Remarks:

Given an open file view, returns the MIME type of the underlying file as a string. If the file is an ECW file, this will be **x-image/ecw**. If the file is a JPEG 2000 file, it will be **image/jp2**.

Parameters:

- `pNCSFileView` - The file view to query.

Returns:

MIME type string.

Example:

```
char *szMIME;
NCSFileView *pView = NCScbmOpenFileView(
      "c:\\foo.ecw", &pView, NULL);
szMIME = NCSGetFileMimeType(pView);
```

NCScbmGetFileType

```
NCSFileType NCScbmGetFileType(
      NCSFileView *pNCSFileView )
```

Remarks:

Given an open file view, returns the type of the underlying file, either ECW, JPEG 2000 or unknown.

Parameters:

- `pNCSFileView` - The file view to query.

Returns:

`NCSFileType` enum value, either:

- `NCS_FILE_ECW,`
- `NCS_FILE_JP2,` or
- `NCS_FILE_UNKNOWN`

Example:

```
NCSFileView *pView = NCScbmOpenFileView(
      "c:\\foo.ecw", &pView, NULL);
NCScbmGetFileType(pView);
```

NCSCopyFileInfoEx

```
void NCSCopyFileInfoEx(
      NCSFileViewFileInfoEx *pDst,
      NCSFileViewFileInfoEx *pSrc)
```

Remarks:

Copy the contents of one `NCSFileViewFileInfoEx` struct to another. This will duplicate dynamically allocated resources associated with the source struct, e.g. projection and datum strings.

Parameters:

- `pDst` - Struct to copy values to.
- `pSrc` - Struct to copy values from.

Returns:

`None.`

Example:

```
NCSCopyFileInfoEx(&Info, File.GetFileInfo())
```

NCSDetectGDTPath

```
void NCSDetectGDTPath()
```

Remarks:

Try and detect GDT files on the machine currently in use and set the value of the GDT path accordingly. This looks in GDT locations commonly used by various ERDAS applications, such as ER Mapper and Image Web Server.

Parameters:

None.

Returns:

None.

Example:

```
NCSDetectGDTPath();
```

NCSFreeFileInfoEx

```
void NCSFreeFileInfoEx(NCSFileViewFileInfoEx *pDst)
```

Remarks:

Free the resources allocated to an `NCSFileViewFileInfoEx`. This will free any dynamically allocated resources associated with the structure (e.g. the memory allocated to projection, datum and band description strings) as well.

Parameters:

- `pDst` - Pointer to the struct.

Returns:

None.

Example:

```
NCSFileViewFileInfoEx *pInfo = (
      NCSFileViewFileInfoEx*)
      NCSMalloc(sizeof(NCSFileViewFileInfoEx),TRUE);
NCSInitFileInfoEx(pInfo);
NCSFreeFileInfoEx(pInfo);
```

NCSGetEPSGCode

```
NCSError NCSGetEPSGCode(
      char*szProjection, char *szDatum
      INT32 *pnEPSG)
```

Remarks:

Translates ER Mapper projection and datum strings into an EPSG PCS (European Petroleum Survey Group Projected Coordinate System) code, if the SDK can find an appropriate code. Otherwise, returns 0.

Parameters:

- `szProjection` - ER Mapper projection string.
- `szDatum` - ER Mapper datum string.
- `pnEPSG` - Pointer to returned EPSG code.

Returns:

An `NCSError` value to use for errorchecking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
INT32 nEPSGCode = 0;
NCSGetEPSGCode("NUTM11","NAD27",&nEPSGCode);
```

NCSGetGDTPath

```
char *NCSGetGDTPath(void)
```

Remarks:

Obtain the location which the SDK currently searches for custom EPSG code mappings. This may or may not be a valid location, depending on previous usage of `NCSSetGDTPath`.

Parameters:

None.

Returns:

Current path of GDT data.

Example:

```
char *szPath = NCSGetGDTPath();
```

NCSGetProjectionAndDatum

```
NCSError NCSGetProjectionAndDatum(
      INT32 nEPSG,
      char **pszProjection,
      char **pszDatum)
```

Remarks:

Translates an EPSG code into ER Mapper projection and datum strings, if the SDK can find an appropriate correspondence.

Parameters:

- `nEPSG` - EPSG code to search against.
- `pszProjection` - Pointer to returned projection string.
- `pszDatum` - Pointer to returned datum string.

Returns:

An `NCSError` value to use for errorchecking. The return value is `NCS_SUCCESS` if the operation succeeds.

Example:

```
char *szProjection;
char *szDatum;
NCSGetProjectionAndDatum(4326,&szProjection,&szDatum);
```

NCSInitFileInfoEx

```
void NCSInitFileInfoEx(NCSFileViewFileInfoEx *pDst)
```

Remarks:

Initializes the members of an `NCSFileViewFileInfoEx` struct.

Parameters:

- `pDst` - Pointer to an `NCSFileViewFileInfoEx` struct

Returns:

None.

Example:

```
NCSFileViewFileInfoEx Info;
InitFileInfoEx(&Info);
```

NCSSetGDTPath

```
void NCSSetGDTPath(char *szPath)
```

Remarks:

Set the location of the GDT data you want the SDK to use when translating between EPSG codes and projection and datum strings.

Parameters:

- `szPath` - The fully qualified path of the GDT data director.

Returns:

None.

Example:

```
NCSSetGDTPath(
      "c:\\development\\erm\\ermapper_dev\\GDT_DATA");
```

NCSSetJP2GeodataUsage

```
void NCSSetJP2GeodataUsage(GeodataUsage nGeodataUsage)
```

Remarks:

Set the usage of geographical metadata with JPEG 2000 input and output. The precedence of metadata controls which type of metadata (GML box, GeoTIFF box, or world file) will be used by preference when reading JPEG 2000 files, and the types of metadata control which metadata will be written when JPEG 2000 files are created.

Parameters:

- `nGeodataUsage GeodataUsage` - Enumeration value specifying the preferred metadata usage.

Returns:

None.

Example:

```
NCSSetJP2GeodataUsage(USE_GML_WLD);
```

See Geocoding Information *for more details.*

C++ API

The C++ API to the SDK provides a file-oriented means of opening, configuring and compressing ECW and JPEG 2000 files. With the release of the SDK v3.0, the C++ API has become the preferred entry point to the functionality provided by the SDK, giving application developers slightly more control than the C API. Although the SDK has an internal structure that includes many complex data objects, only three are relevant to SDK users and documented here, namely the `CNCSFile`, `CNCSRenderer`, and `CNCSError` classes.

Full class documentation describing all the functionality made available through these objects are provided.

Class Reference: CNCSFile

This class provides a file oriented object to access and create ECW and JPEG 2000 images. The principal methods for use by an application programmer are `Open`, `Close`, `GetFileInfo`, `SetFileInfo`, `SetParameter`, `Write`, `WriteLineBIL`, `WriteReadLine`, `RefreshUpdateEx` and the various `ReadLine` methods. This class is the main access point for SDK functionality using the C++ API.

`CNCSFile` inherits from `CNCSJP2FileView` and is the parent class of `CNCSRenderer`. A standard way to use the `CNCSFile` class is to create a class in your application that inherits from `CNCSFile` and overrides its methods, `Refresh-UpdateEx` and `WriteReadLine` for decompression and compression respectively.

Constructor:

```
CNCSFile::CNCSFile()
```

Remarks:

This is the default constructor. It initializes all members of the class and leaves it ready to handle new input or output tasks.

Destructor:

```
virtual CNCSFile::~CNCSFile()[virtual]
```

Remarks:

The destructor of `CNCSFile` is declared virtual so that it can be overridden in subclasses to release any additional resources they may acquire.

CNCSFile::AddBox

```
virtual CNCSError CNCSFile::AddBox(CNCSJP2Box *pBox)
```

Remarks:

Add a header box to an output JPEG 2000 file, which will be written out when the file is compressed. Normally the box would be a subclass of one of the two metadata boxes `CNCSJP2XMLBox` and `CNCSJP2UUIDBox`. When the file is written, the box's `UnParse()` method will be called. Ensuring the validity of the resulting output is the responsibility of the application developer.

Parameters:

- `pBox` - Pointer to the box to be written

Returns:

`CNCSError` object providing information about any error that occurred.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, true);
CMyMetadataBox Box;
File.AddBox((CNCSJP2Box *)&Box);
```

CNCSFile::BreakdownURL

```
BOOLEAN CNCSFile::BreakdownURL(
      char** pURLPath,
      char** ppProtocol,
      char** ppHost,
      char** ppFilename)[static]
```

Remarks:

Utility Function, breaks down a URL string into protocol, hostname and filename components.

Parameters:

- `[in] pURLPath` - The URL to be broken down and analyzed.
- `[out] ppProtocol` - A pointer to the protocol string resulting from the URL decomposition.
- `[out] ppHost` - A pointer to the hostname resulting form the URL decomposition.
- `[out] ppFilename` - A pointer to the filename resulting from the URL decomposition.

Returns:

`BOOLEAN` value, whether the input URL is a remote file.

CNCSFile::Close

NCSError CNCSFile::Close(BOOLEAN bFreeCache = TRUE)

Remarks:

Close the file.

Parameters:

- `[in] bFreeCache` - Specify whether or not to free the memory cache that is associated with the file after closing it.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSFile::ConvertDatasetToWorld

```
NCSError CNCSFile::ConvertDatasetToWorld(
      INT32 nDatasetX,
      INT32 nDatasetY,
      IEEE8* pdWorldX,
      IEEE8* pdWorldY)
```

Remarks:

Performs a rectilinear conversion from dataset coordinates to world coordinates.

Parameters:

- `[in] nDatasetX` - The dataset X coordinate.
- `[in] nDatasetY` - The dataset Y coordinate.
- `[out] pdWorldX` - A buffer for the output world X coordinate.
- `[out] pdWorldY` - A buffer for the output world Y coordinate.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSFile::ConvertWorldToDataset

```
NCSError CNCSFile::ConvertWorldToDataset(
      INT8 dWorld,
      IEE8 dWorldY,
      INT32* pnDatasetX,
      INT32* pnDatasetY)
```

Remarks:

Performs a rectilinear conversion from world coordinates to dataset coordinates.

Parameters:

- `[in] dWorldX` - The world X coordinate.
- `[in] dWorldY` - The world Y coordinate.
- `[out] pnDatasetX` - A buffer for the output dataset X coordinate.
- `[out] pnDatasetY` - A buffer for the output dataset Y coordinate.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSFile::DetectGDTPath

```
static void DetectGDTPath()
```

Remarks:

Try and detect GDT files on the machine currently in use and set the value of the GDT path accordingly. This looks in GDT locations commonly used by various ERDAS applications, such as ER Mapper and Image Web Server.

Parameters:

None.

Returns:

None.

Example:

```
CNCSFile::DetectGDTPath();
```

CNCSFile::FormatErrorText

```
const char* CNCSFile::FormatErrorText(
      NCSError nErrorNum) [static]
```

Remarks:

Obtains meaningful error text from a returned error code.

Parameters:

- `[in] nErrorNum` - Error code

Returns:

`const char*`, an explanatory ASCII string for the error code.

CNCSFile::GetBox

```
virtual CNCSJP2Box* CNCSFile::GetBox(
      UINT32 nTBox, CNCSJP2Box *pLast = NULL)
```

Remarks:

Return the next box of the specified type from an open JPEG 2000 file.

Parameters:

- `nTBox` - Unsigned 32 bit value representing the box type. This is usually a string of four bytes with a mnemonic value such as jp2h, uuid, colr, etc.
- `pLast` - Last box of this type found, if applicable

Returns:

Pointer to the next box of this type found in the file.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, false);
```

```
UINT8 nTypeChars[4] = {'u','u','i','d'};
UINT32 nTBox = *((UINT32 *)nTypeChars);
CNCSJP2Box pBox = File.GetBox(nTBox);
```

CNCSFile::GetClientData

```
void *CNCSFile::GetClientData()
```

Remarks:

Get any client data that has been established by the SDK application. This method is generally called by a subclass of `CNCSFile` in an overridden version of `CNCSFile::RefreshUpdateEx`.

Parameters:

None.

Returns:

`void` pointer to client data.

CNCSFile::GetEPSGCode

```
INT32 CNCSFile::GetEPSGCode()
```

Remarks:

Return the EPSG code associated with an open ECW or JPEG 2000 file's coordinate system, if any.

Parameters:

None.

Returns:

The applicable EPSG code, or 0.

Example:

```
CNCSFile File;
File.Open("C:\\georeferenced.ecw",false, false);
INT32 nEPSG = File.GetEPSGCode;
```

CNCSFile::GetEPSGCode

```
CNCSError CNCSFile::GetEPSGCode(
      char *szProjection,
      char *szDatum,
      UINT32 *nEPSGCode) [static]
```

Remarks:

This function returns a European Petroleum Survey Group (EPSG) code for the projected coordinate system to which the open image file corresponds, given ER Mapper style projection and datum strings. Where the image is not georeferenced an error will be returned.

Parameters:

- `[in] szProjection` - ER Mapper style projection string.
- `[in] szDatum` - ER Mapper style datum string.
- `[out] nEPSGCode` - Reference to an integer variable in which to store the corresponding EPSG code.

Returns:

`CNCSError` object providing information about any error that occurred.

Example:

```
INT32 nEPSG;
CNCSFile::GetEPSGCode("NUTM11", "NAD27", &nEPSG);
if (nEPSG != 0)
   printf("Associated EPSG code:%d\r\n",nEPSG);
else
   printf("No associated EPSG code found.\r\n");
```

CNCSFile::GetFile

```
class CNCSJP2File* CNCSFile::GetFile()
```

Remarks:

Retrieve the underlying `CNCSJP2File` object from an open JPEG 2000 file view.

Parameters:

None.

Returns:

Pointer to the underlying file.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, false);
CNCSJP2File *pJP2File = File.GetFile();
```

CNCSFile::GetFileInfo

```
const NCSFileViewFileInfoEx *CNCSFile::GetFileInfo()
```

Remarks:

Get the `NCSFileViewFileInfoEx` structure corresponding to open file.

Parameters:

None.

Returns:

Pointer to the file information structure.

CNCSFile::GetFileMimeType

```
char* CNCSFile::GetFileMimeType()
```

Remarks:

Returns the MIME type of the currently open file. This will either be `x-image/ecw` or `image/jp2`.

Parameters:

None.

Returns:

MIME type as string.

Example:

```
char *szMime = File.GetFileMimeType();
```

CNCSFile::GetFileType

```
NCSFileType CNCSFile::GetFileType()
```

Remarks:

Returns the type of the file associated with an open file view. This is either:

- `NCS_FILE_JP2`,
- `NCS_FILE_ECW`, or
- `NCS_FILE_UNKNOWN`.

Parameters:

None.

Returns:

`NCSFileType` enum value.

Example:

```
CNCSFile File;
File.Open("C:\\georeferenced.jp2",false,false);
if (File.GetFileType() == NCS_FILE_ECW)
   printf("Open file is actually an ECW\r\n");
```

CNCSFile::GetFileViewSetInfo

```
const NCSFileViewSetInfo
      *CNCSFile::GetFileViewSetInfo()
```

Remarks:

Get current `NCSFileViewSetinfo` structure.

Parameters:

None

Returns:

Pointer to the current `SetViewInfo`.

CNCSFile::GetGDTPath

```
static char* CNCSFile::GetGDTPath()
```

Remarks:

Obtain the location which the SDK currently searches for custom EPSG code mappings. This may or may not be a valid location, depending on previous usage of `NCSSetGDTPath`.

Parameters:

None.

Returns:

Current path of GDT data.

Example:

```
char *szPath = CNCSFile::GetGDTPath()
```

CNCSFile::GetNCSFileView

```
NCSFileView* CNCSFile::GetNCSFileView()
```

Remarks:

Retrieve the underlying NCSFileView struct from an open ECW file view.

Parameters:

None

Returns:

Pointer to the associated `NCSFileView` structure.

Example:

```
CNCSFile File;
File.Open("C:\\foo.ecw", false, false);
NCSFileView *pView = File.GetNCSFileView();
```

CNCSFile::GetNCSFileView

```
NCSFileView *CNCSFile::GetNCSFileView()
```

Remarks:

Get underlying `NCSFileView` pointer, where it exists.

Parameters:

None.

Returns:

Pointer to the `NCSFileView` instance.

CNCSFile::GetParameter

```
void CNCSFile::GetParameter(Parameter eType, ...)
```

Remarks:

Do not use. This method is deprecated and exists for backwards compatibility only.

CNCSFile::GetParameter

```
void CNCSFile::GetParameter(
      char *sName,
      ...);
```

Remarks:

This function is used to get metadata from the file view. This method is a `var_args` style function which takes two arguments, the second argument being the address of an appropriately typed variable to receive the output parameter.

E.g. To get the profile type of a JPEG 2000 file, the following would be used:

```
UINT32 profileType = 0;
CNCSFileView view.GetParameter(
      "JP2:COMPLIANCE:PROFILE:TYPE", &profileType);
```

Parameters:

- `sName` - The parameter name as a C string.
- `...` - A pointer to a variable where the parameter will be returned (usually `BOOLEAN, UINT31,` or none).

Returns:

None.

The value of `nType` after the call will be the value in the CA (capabilities word) in the `SIZ` marker in the JPEG 2000 main header. The values corresponds according to the table below.

Profile	Values
Profile 0	1
Profile 1	2
Profile 2	0 3 (NITF NPJE profile) 4 (NITF EPJE profile)

The table below lists and describes the parameters available to the `GetParameter` method.

Parameter	Value Type	Notes
`JP2:DECOMPRESS:LAYERS`	`UINT32`	Sets the maximum number of quality layers.
`JP2:COMPLIANCE:PROFILE:TYPE`	`UINT32`	Retrieves the compliance profile of the JPEG 2000 fiile.

CNCSFile::GetPercentComplete

```
INT32 CNCSFile::GetPercentComplete()
```

Remarks:

Return the percentage of image remaining to be downloaded.

Parameters:

None.

Returns:

The percentage complete value; a number from 0 to 100 indicating the proportion of the image that remains to be downloaded.

CNCSFile::GetPercentCompleteTotalBlocksInView

```
INT32 CNCSFile::GetPercentCompleteTotalBlocksInView()
```

Remarks:

Return the percentage of the total blocks in the view that have been downloaded.

Parameters:

None.

Returns:

A number from 0 to 100 representing the total proportion of blocks in the view that have been downloaded.

CNCSFile::GetProjectionAndDatum

```
static CNCSError CNCSFile::GetProjectionAndDatum(
      const INT32 nEPSGCode,
      char **ppProjection,
      char **ppDatum)
```

Remarks:

Convert an EPSG PCS code to ER Mapper projection and datum strings if a mapping is available to the SDK.

Parameters:

- `[in] EPSGCode` - Input EPSG code.
- `[out] ppProjection` - Output ER Mapper projection string, e.g. `NUTM11`.

- `[out] ppDatum` - Output ER Mapper datum string, e.g. `NAD27`.

Returns:

`CNCSError` object providing information about any error that occurred.

Example:

```
char *szDatum = NULL;
char *szProjection = NULL;
CNCSFile::GetProjectionAndDatum(
      4326, &szProjection, &szDatum);
```

CNCSFile::GetStream

```
CNCSJPCIOStream* CNCSFile::GetStream()
```

Remarks:

Return a pointer to the underlying `CNCSJPCIOStream` object being used for input or output with a JPEG 2000 file.

Parameters:

None.

Returns:

Pointer to the disk, memory or ECWP IO stream.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, false);
CNCSJPCIOStream *pStream = File.GetStream();
```

CNCSFile::GetUUIDBox

```
virtual CNCSJP2Box* CNCSFile::GetUUIDBox(
      NCSUUID uuid,
      CNCSJP2Box *pLast = NULL)
```

Remarks:

Return the next UUID box in an open JPEG 2000 file, with a UUID matching the argument.

Parameters:

- `uuid` - 16-byte UUID value to search for in the open file.
- `pLast` - Last such UUID box found, if applicable.

Returns:

Pointer to the next such UUID box found.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, false);
NCSUUID uuid = {
      0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
      0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf};
CNCSJP2Box *pBox = File.GetUUIDBox(uuid);
```

CNCSFile::GetXMLBox

```
virtual CNCSJP2Box* CNCSFile::GetXMLBox(
      CNCSJP2Box *pLast = NULL)
```

Remarks:

Return the next XML box in an open JPEG 2000 file.

Parameters:

- `pLast` - Last XML box found, if applicable.

Returns:

Pointer to the next XML box found.

Example:

```
CNCSFile File;
File.Open("C:\\foo.jp2", false, false);
CNCSJP2Box *pBox = NULL;
while ((pBox = File.GetXMLBox(pBox)) != NULL)
{
   printf("New XML box found!\r\n");
}
```

CNCSFile::Open

```
CNCSError CNCSFile::Open(
      char* pURLPath,
      BOOLEAN bProgressiveDisplay,
      BOOLEAN bWrite = FALSE)
```

Remarks:

Open a file for input or output.

Parameters:

- `[in] pURLPath` - The location of the file - if for input, can be a remote file. Can be a UNC location.
- `[in] bProgressiveDisplay` - Selects whether the file will be opened in progressive mode if for input.
- `[in] bWrite` - Selects whether the file is being opened for output.

Returns:

`CNCSError` object providing information about any error that occurred.

Example:

```
CNCSFile File;
File.Open("ecwp://localhost/SampleIWS/
      images/usa/sandiego3i/ecw, true, false);
```

CNCSFile::Open

```
virtual CNCSError CNCSFile::Open(
      CNCSJPCIOStream *pStream,
      bool bProgressiveDisplay = false)
```

Remarks:

Open a JPEG 2000 file parsing input from the specified input stream.

Parameters:

- `pStream` - Input stream on which to open the file.
- `bProgressiveDisplay` - Whether or not to open in progressive read mode.

Returns:

`CNCSError` object providing information about any error that occurred.

Example:

```
CMyIOStream Stream;
Stream.Open("c:\\foo.nitf", false);
CNCSFile File;
((CNSJP2FileView&)File).Open(&Stream);
```

CNCSFile::ReadLineABGR

```
NCSEcwReadStatus CNCSFile::ReadLineABGR (
      UINT32 *pABGR)[inline, virtual]
```

Remarks:

Read the next line in ABGR UINT32 format from the current view into the file. The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- pABGR - Pointer to UINT32 buffer to receive ABGR data.

Returns:

NCSEcwReadStatus read status code.

CNCSFile::ReadLineARGB

```
NCSEcwReadStatus CNCSFile::ReadLineARGB(
      UINT32 *pARGB) [inline, virtual]
```

Remarks:

Read the next line in ARGB UINT32 format from the current view into the file.The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- pARGB - Pointer to UINT32 buffer to receive ARGB data.

Returns:

NCSEcwReadStatus read status code.

CNCSFile::ReadLineBGR

```
NCSEcwReadStatus CNCSFile::ReadLineBGR(
      UINT8 *pBGRTriplet) [inline, virtual]
```

Remarks:

Read the next line in BGR UINT8 triplet format from the current view into the file.

Parameters:

- pBGRTriplet - Pointer to UINT8 buffer to receive BGR data.

Returns:

NCSEcwReadStatus read status code.

CNCSFile::ReadLineBGRA

```
NCSEcwReadStatus CNCSFile::ReadLineBGRA (
      UINT32 *pBGRA) [inline, virtual]
```

Remarks:

Read the next line in `BGRA UINT32` format from the current view into the file. The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- `pBGRA` - Pointer to `UINT32` buffer to receive `BGRA` data.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      NCSEcwCellType eType,
      UINT16 nBands,
      void **ppOutputLine,
      UINT32 *pLineSteps = NULL)
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `eType` - Preferred sample type for the output buffer.
- `nBands` - Number of bands in the output buffer.
- `ppOutputLine` - Array of buffer pointers, one buffer for each band.
- `pLineSteps` - Line steps, in dataset cells.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      UINT8 **ppOutputLine) [inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      UINT16**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` Read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      UINT32**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` Read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      UINT64**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      UINT8**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      INT8**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      INT16**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      INT32**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      INT64**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      IEEE4**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` - Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineBIL

```
NCSEcwReadStatus CNCSFile::ReadLineBIL(
      IEEE8**ppOutputLine)[inline, virtual]
```

Remarks:

Read the next line in `BIL` format from the current view into the file.

Parameters:

- `ppOutputLine` Array of buffer pointers, one buffer for each band.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineRGB

```
NCSEcwReadStatus CNCSFile::ReadLineRGB(
      UINT8 *pRGBTriplet) [inline, virtual]
```

Remarks:

Read the next line in `RGB UINT8` triplet format from the current view into the file.

Parameters:

- `pRGBTriplet` - Byte buffer into which `RGB` triplets can be read.

Returns:

`NCSEcwReadStatus` read status code.

CNCSFile::ReadLineRGBA

```
NCSEcwReadStatus CNCSFile::ReadLineRGBA(
      UINT32 *pRGBA) [inline, virtual]
```

Remarks:

Read the next line in RGBA UINT32 format from the current view into the file. The alpha band contains only zero values, and this function is provided solely for interoperability with graphics software that uses alpha blending and a compatible pixel value data structure.

Parameters:

- pRGBA Pointer to UINT32 buffer to receive RGBA data.

Returns:

NCSEcwReadStatus read status code.

CNCSFile::RefreshUpdate

```
virtual void CNCSFile::RefreshUpdate(
      NCSFileViewSetInfo *pViewSetInfo)[virtual]
```

Remarks:

More data has become available and a refresh update should be done. This function is deprecated and you should generally use CNCSFile::RefreshUpdateEx instead.

Parameters:

- pViewSetInfo - This is a pointer to a SetViewInfo containing details about the view the update is from.

Returns:

None

CNCSFile::RefreshUpdateEx

```
virtual NCSEcwReadStatus CNCSFile::RefreshUpdateEx(
      NCSFileViewSetInfo* pViewSetInfo) [virtual]
```

Remarks:

More data has become available and a refresh update should be done.

Parameters:

- pViewSetInfo - This is a pointer to SetViewInfo containing details about the view for which the update is intended.

Returns:

Returns the `NCSEcwReadStatus` read status code from the `ReadLine*()` call. This is reimplemented from `CNCSJP2FileView`.

CNCSFile::SetClientData

```
void CNCSFile::SetClientData(void *pClientData)
```

Remarks:

This method allows the SDK user to set a private data structure the state of which can safely be queried in an overridden version of `CNCSFile::RefreshUpdateEx` in some developer-defined subclass.

Parameters:

- `pClientData` - Void pointer to private data structure.

Returns:

None

CNCSFile::SetCompressClient

```
CNCSError CNCSFile::SetCompressClient(
      struct NCSEcwCompressClient *pCompressClient)
```

Remarks:

Internal func for C API support only.

Parameters:

- `pCompressClient` - ECW Compress Client structure.

Returns:

`CNCSError` object providing information about any error that occurred.

CNCSFile::SetFileInfo

```
CNCSError CNCSFile::SetFileInfo(
      NCSFileViewFileInfoEx &Info) [inline, virtual]
```

Remarks:

Set `FileInfo` structure.

Parameters:

- `Info` - Used to specify file info for compression.

Returns:

`CNCSError` object providing information about any error that occurred.

CNCSFile::SetGDTPath

```
static void CNCSFile::SetGDTPath(const char *szPath)
```

Remarks:

Set the location of the GDT data you want the SDK to use when translating between EPSG codes and ER Mapper projection and datum strings.

Parameters:

- `szPath` - The fully qualified path of the GDT data directory.

Returns:

None.

Example:

```
CNCSFile::SetGDTPath(
      "c:\\development\\erm\\ermapper_dev\\GDT_DATA");
```

CNCSFile::SetOEMKey

```
void CNCSFile::SetOEMKey(
      char *szCompanyName,
      char *szKey);
```

Remarks:

This function, which is only available in the read/write version of the SDK, allows the developer to set their license key which enables the compression feature of the ECW JPEG2000 SDK.

Returns

None.

The company name needs to be the same that was provided to ERDAS when generating the key. The key needs to be set by either the C or C++ function call before a file can be opened for write.

CNCSFile::SetParameter

```
void CNCSFile::SetParameter(
      Parameter eType,
```

```
...)
```

Remarks:

Do not use. This method is deprecated and exists for backwards compatibility only.

CNCSFile::SetParameter

```
void CNCSFile::SetParameter(
      Parameter eType,
      IEEE4 fValue)
```

Remarks:

This function is used to set metadata for the file view (usually when compressing JPEG 2000 files). This method is a `var_args` style function which takes two arguments, the second argument being the appropriately typed variable which contained the parameter to set.

E.g. To set the profile type of a JP2 file, the following would be used:

```
UINT32 profileType = 1;
CNCSFileView view.SetParameter(
      "JP2:COMPLIANCE:PROFILE:TYPE", profileType);
```

Parameters:

- `sName` - The parameter name as a C string
- `...` - A variable containing the parameter to set (usually `BOOLEAN`, `UINT32` or none).

Returns:

None.

Parameter	Value Type	Notes
`JP2_COMPRESS_PROFILE_BASELINE_0`	None	Compress for Class 0 compliant decoders.
`JP2_COMPRESS_PROFILE_BASELINE_1`	None	Compress for Class 1 compliant decoders.
`JP2_COMPRESS_PROFILE_BASELINE_2`	None	Compress for Class 2compliant decoders.
`JP2_COMPRESS_PROFILE_NITF_BIIF_NPJE`	None	Compress according to NITF/BIIF NPJE compression profile.

JP2_COMPRESS_PROFILE_NITF_BIIF_EPJE	None	Compress according to NITF/BIIF EPJE profile.
JP2_COMPRESS_LEVELS	UINT32	Default calculated so that (r == 0) has dimensions less than or equal to 64 x 64.
JP2_COMPRESS_LAYERS	UINT32	Default 1.
JP2_COMPRESS_PRECINCT_WIDTH	UINT32	Default 64 or larger depending on output size.
JP2_COMPRESS_PRECINCT_HEIGHT	UINT32	Default 64 or larger depending on output size.
JP2_COMPRESS_TILE_WIDTH	UINT32	Default to image width specified in `SetFileInfo`.
JP2_COMPRESS_TILE_HEIGHT	UINT32	Default to image height specified in `SetFileInfo`.
JP2_COMPRESS_INCLUDE_SOP	BOOLEAN	Specify whether to include start-of-packet markers - default is false.
JP2_COMPRESS_INCLUDE_EPH	BOOLEAN	Specify whether to include end-of-packetheader markers - default is true.
JPC:DOWNSCALING:ENABLED	BOOLEAN	Enables downscaling of JPEG 2000 images from 16 bit to 8 bit. By default this is set to true.

JPC:DOWNSCALING:MAX
JPC:DOWNSCALING:MIN
	UINT64	Specify a maximun and minimum range for downscaling JPEG 2000 images. E.g:
		`INT64 nMax = 65528;`
		`View.SetParameter("JPC:DOWNSCALING:MAX", nMax);`

JP2_COMPRESS_PROGRESSION_LRCP	None	Default progression order - layer, resolution, component, precinct.
JP2_COMPRESS_PROGRESSION_RLCP	None	Specify resolution, layer, component, precinct progression order instead if desired.
JP2_GEODATA_USAGE	UINT32	Control the usage and precedence of georeferencing metadata - see chapter ‘Geocoding Information’.
JP2_DECOMPRESS_LAYERS	UINT32	Default behaviour is to decompress all layers.
JP2_DECOMPRESS_RECONSTRUCTION_PARAMETER	IEEE4	Dequantization parameter 0.0 <= r < 1.0, default is 0.0.

JP2:DECOMPRESS:LAYERS	UINT32	By default this will decode all target quality layers. However if used, a specified number of target quality layers will be decoded.

The JP2_GEODATA_USAGE *parameter corresponds to a static variable and thus once set the new value will apply to all instances of* CNCSFile *and its subclasses on input and output. All other configuration parameters correspond to dynamic data that is associated with a single instance of* CNCSFile *or one of its subclasses.*

CNCSFile::SetRefreshCallback

```
CNCSError CNCSFile::SetRefreshCallback(
   NCSEcwReadStatus (*pCallback)(NCSFileView *))
```

Remarks:

Set refresh callback function.

Parameters:

- pCallback - Refresh callback function to use.

Returns:

CNCSError object providing information about any error that occurred.

CNCSFile::SetView

```
CNCSError CNCSFile::SetView(
      INT32 nBands,
      INT32* pBandList,
      INT32 nWidth,
      INT32 nHeight,
      INT32 dDatasetTLX,
      INT32 dDatasetTLY,
      INT32 dDatasetBRX,
      INT32 dDatasetBRY)
```

Remarks:

Set the view on the open file. This version takes dataset coordinates as input.

Parameters:

- nBands - The number of bands to include in the view being set.

- `pBandList` - An array of band indices specifying which bands to include and in which order.
- `nWidth` - The width of the view to construct in dataset cells.
- `nHeight` - The height of the view to construct in dataset cells.
- `dDatasetTLX` - The top left X of the view to construct, specified in dataset coordinates.
- `dDatasetTLY` - The top left Y of the view to construct, specified in dataset coordinates.
- `dDatasetBRX` - The top right X of the view to construct, specified in dataset coordinates.
- `dDatasetBRY` - The bottom right Y of the view to construct, specified in dataset coordinates.

Returns:

`CNCSError` object providing information about any error that occurred.

Notes:

This is implemented in `CNSRenderer`.

CNCSFile::SetView

```
NCSError CNCSFile::SetView(
      INT32 nBands,
      INT32* pBandList,
      INT32 nWidth,
      INT32 nHeight,
      IEEE8 dWorldTLX,
      IEEE8 dWorldTLY,
      IEEE8 dWorldBRX,
      IEEE8 dWorldBRY)
```

Remarks:

Set the view on the open file. This version takes world coordinates as input.

Parameters:

- `nBands` - The number of bands to include in the view being set.
- `pBandList` - An array of band indices specifying which bands to include and in which order.
- `nWidth` - The width of the view to construct in dataset cells.
- `nHeight` - The height of the view to construct in dataset cells.

- `dWorldTLX` - The top left X of the view to construct, specified in world coordinates.
- `dWorldTLY` - The top left X of the view to construct, specified in world coordinates.
- `dWorldBRX` - The bottom right X of the view to construct, specified in world coordinates.
- `dWorldBRY` - The bottom right Y of the view to construct, specified in world coordinates.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Notes:

This is reimplemented in `CNSRenderer`.

CNCSFile::SetView

```
CNCSError CNCSFile::SetView(
      INT32 nBands,
      INT32* pBandList,
      UINT32 nDatasetTLX,
      UINT32 nDatasetTLY,
      UINT32 nDatasetBRX,
      UINT32 nDatasetBRY,
      UINT32 nWidth,
      UINT32 nHeight,
      IEEE8 dWorldTLX = 0.0,
      IEEE8 dWorldTLY = 0.0,
      IEEE8 dWorldBRX = 0.0,
      IEEE8 dWorldBRY = 0.0)[inline, virtual])
```

Remarks:

Set a view into the open file for reading. The world coordinates are informative only.

Parameters:

- `nBands` - The number of bands in pBandList to read.
- `pBandList` - An array of band indices to read.
- `nDatasetTLX` - Top left X dataset coordinate of view.
- `nDatasetTLY` - Top left Y dataset coordinate of view.
- `nDatasetBRX` - Bottom right X dataset coordinate of view.

- `nDatasetBRY` - Bottom right Y dataset coordinate of view.
- `nWidth` - Width of the view in pixels.
- `nHeight` - Height of the view in pixels.
- `dWorldTLX` - Top left X world coordinate of view (informative only).
- `dWorldTLY` - Top left Y world coordinate of view (informative only).
- `dWorldBRX` - Bottom right X world coordinate of view (informative only).
- `dWorldBRY` - Bottom right Y world coordinate of view (informative only).

Returns:

`CNCSError` object providing information about any error that occurred.

CNCSFile::Write

```
CNCSError CNCSFile::Write() [virtual]
```

Remarks:

This method starts compressing output data to your output file. The output filename is the filename associated with the currently open file object, and the output file format is determined according to the extension of this filename (for example, if the filename has a .ecw extension, compression will be to the ECW format, and if the extension is .jp2, compression will be to the JPEG 2000 format. Once the compression process begins callback functions are used to read data, abort the process, and check on its progress.

Parameters:

None.

Returns:

`CNCSError` value indicating the success or otherwise of the compression task.

CNCSFile::WriteCancel

```
bool CNCSFile::WriteCancel(void) [virtual]
```

Remarks:

This function is called by the SDK each time a scanline is written from input data to an output ECW or JPEG 2000 file. Override this virtual callback function to return true in response to a cancel compression event from your application.

Parameters:

None.

Returns:

TRUE if the compression should be cancelled.

CNCSFile::WriteLineBIL

```
CNCSError CNCSFile::WriteLineBIL(
      NCSEcwCellType eType,
      UINT16 nBands,
      void **ppOutputLine,
      UINT32 *pLineSteps = NULL) [inline, virtual]
```

Remarks:

Write the next line in BIL format into the JPEG 2000 file.

Parameters:

- `eType` - Output buffer cell type.
- `nBands` - Number of output bands.
- `ppOutputLine` - Array of scanline buffer pointers, one buffer for each band.
- `pLineSteps` - Line steps, in cells.

Returns:

`CNCSError` object providing information about any error that occurred.

CNCSFile::WriteReadLine

```
CNCSError CNCSFile::WriteReadLine(
      UINT32 nNextLine,
      void **ppInputArray) [virtual]
```

Remarks:

In pull-through write mode this callback method is called once by the SDK for every image line output to an ECW or JPEG 2000 file by this `CNCSFile` object. You should override this method to read uncompressed image data from another resource available to your SDK application and load it into the input buffer.

Parameters:

- `nNextLine` - The next line of uncompressed input to load.
- `ppInputArray` - The BIL-formatted input buffer into which to load your uncompressed input.

Returns:

`CNCSError` object providing information about any error that occurred.

CNCSFile::WriteStatus

```
void CNCSFile::WriteStatus(
      UINT32 nCurrentLine)[virtual]
```

Remarks:

This callback function is called once by the SDK for every image line output to an ECW or JPEG 2000 file during compression. You should override this function to advise you of the progress of compression by printing to the standard output or your application's user interface. You can use the `nCurrentLine` parameter to determine the progress of compression based on the total number of scanlines in the output. If updating a GUI progress bar in your application, it is wise to fix the total number of times the GUI is updated by testing `nCurrentLine` relative to the total number of scanlines, since in a large compression process performance may deteriorate if it is necessary to constantly perform a comparatively expensive update routine.

Parameters:

- `nCurrentLine` - The current line being written to the output image.

Returns:

None.

Class Reference: CNCSRenderer

This class is used to render ECW and JPEG 2000 imagery to a device context. It inherits from the `CNCSFile` class. A simple `SetExtents()` call is used to adjust the view extents appropriately and then imagery can be drawn. The extents do not have to lie within the boundary of the dataset (as in the case of the `CNCSFile` class). It will clip and draw intersection regions accordingly. It can be transparent or opaque. In opaque mode a background color can be set.

`CNCSRenderer` is a great example of the flexibility and ease of use of the SDK technology and the best option for you to get ECW and JPEG 2000 imagery rapidly into your application.

The additional public members of `CNCSRenderer`, and those whose behavior is changed from that in the parent class `CNCSFile`, are documented in this section.

Constructor

```
CNCSRenderer::CNCSRenderer()
```

This is the default constructor. It initializes all members of the class and leaves it ready to handle new input or output tasks. The background color is initially set to the system default.

Destructor

```
virtual CNCSRenderer::~CNCSRenderer()[virtual]
```

The destructor of `CNCSRenderer` is declared virtual so that it can be overridden in subclasses to release any additional resources they may acquire.

CNCSRenderer::ApplyLUTs

```
BOOLEAN CNCSRenderer::ApplyLUTS(BOOLEAN bEnable)
```

Remarks:

Sets whether or not to apply look up tables before rendering to the device context.

Parameters:

- `bEnable` - Whether or not to apply LUTs.

Returns:

TRUE or FALSE.

CNCSRenderer::CalcHistograms

```
BOOLEAN CNCSRenderer::CalcHistograms(BOOLEAN bEnable)
```

Remarks:

Sets whether or not to perform histogram calculations during processing. Call before a call to `CNCSRenderer::SetView`.

Parameters:

- `bEnable` - Whether or not to do histogram calculations.

Returns:

TRUE or FALSE.

CNCSRenderer::DrawImage

```
NCSError CNCSRenderer::DrawImage(
      HDC hDeviceContext,
      LPRECT pClipRect,
```

```
IEEE8 dWorldTLX,
IEEE8 dWorldTLY,
IEEE8 dWorldBRX,
IEEE8 dWorldBRY)
```

Remarks:

This method draws (blits) the internal buffer of the `CNCSRenderer` object to the screen. The imagery is drawn using the specified extents. The extents do not necessarily have to match the extents previously specified in a call to `CNCSRenderer::SetView`. If they do, the entire image is drawn. If they don't, only the intersection between the input extents and the amount of imagery in the input buffer is drawn.

Parameters:

- `hDeviceContext` - A Win32 device context.
- `pClipRect` - A point to a clip rectangle describing the width and height of the draw area.
- `dWorldTLX` - The top left X world coordinate of the device.
- `dWorldTLY` - The top left Y world coordinate of the device.
- `dWorldBRX` - The bottom right X world coordinate of the device.
- `dWorldBRY` - The bottom right Y world coordinate of the device.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::FreeJPEGBuffer

```
void CNCSRenderer::FreeJPEGBuffer(UINT8*pBuffer)
```

Remarks:

This static call is used to free the JPEG memory buffer returned by a call to `CNCSRenderer::WriteJPEG()`.

Parameters:

- `pBuffer` - The JPEG buffer previously returned that must now be freed.

Returns:

None.

CNCSRenderer::GetHistogram

```
BOOLEAN CNCSRenderer::GetHistogram(
      INT32 nBand,
      UINT32 Histogram[256])
```

Remarks:

Get the histogram calculated for a specific band in the image.

Parameters:

- `nBand` - The band for which to retrieve the associated histogram.
- `Histogram` - An array of size 256 to the histogram.

Returns:

`TRUE` or `FALSE`.

CNCSRenderer::GetTransparent

```
void CNCSRenderer::GetTransparent (
      BOOLEAN *pbTransparent)
```

Remarks:

Obtains the current transparency status from the renderer.

Parameters:

`BOOLEAN` buffer for the returned transparency status.

Returns:

None.

CNCSRenderer::ReadImage

```
NCSError CNCSRenderer::ReadImage(
      IEEE8 dWorldTLX,
      IEEE8 dWorldTLY,
      IEEE8 dWorldBRX,
      IEEE8 dWorldBRY,
      INT32 nDatasetTLX,
      INT32 nDatasetTLY,
      INT32 nDatasetBRX,
      INT32 nDatasetBRY,
      INT32 nWidth,
      INT32 nHeight)
```

Remarks:

Reads the current image into an internal buffer ready for blitting to a device context. In progressive mode, when a `RefreshUpdate` callback arrives from the network, the client should call `ReadImage` to transfer the pending imagery from the network into an internal buffer. Once this is done, the client can then call `DrawImage` at any time to draw from the internal buffer into the device. In non-progressive mode the client should call `ReadImage`, then immediately call `DrawImage` to draw to the device. This overloaded version of the function is called in progressive mode only.

Parameters:

- `dWorldTLX` - The view world top left X coordinate (must match the `SetView` top left X).
- `dWorldTLY` - The view world top left Y coordinate (must match the `SetView` top left Y).
- `dWorldBRX` - The view world bottom right X coordinate (must match the `SetView` bottom right X).
- `dWorldBRY` - The view world bottom right Y coordinate (must match the `SetView` bottom right Y).
- `nDatasetTLX` - The dataset top left X coordinate.
- `nDatasetTLY` - The dataset top left Y coordinate.
- `nDatasetBRX` - The dataset bottom right X coordinate.
- `nDatasetBRY` - The dataset bottom right Y coordinate.
- `nWidth` - The view width (must match the `SetView` width).
- `nHeight` - The view height (must match the `SetView` height).

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::ReadImage

```
NCSError CNCSRenderer::ReadImage(
      INT32 nWidth,
      INT32 nHeight)
```

Remarks:

Reads the current image into an internal buffer ready for blitting to a device context. In progressive mode, when a `RefreshUpdate` callback arrives from the network, the client should call `ReadImage` to transfer the pending imagery from the network into an internal buffer. Once this is done, the client can then call `DrawImage` at any time to draw from the internal buffer into the device. In non-progressive mode the client should call `ReadImage`, then immediately call `DrawImage` to draw to the device. This overloaded version of the function is called in progressive mode only.

Parameters:

- `nWidth` - The view width (must match the `SetView` width).
- `nHeight` - The view height (must match the `SetView` height).

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::ReadImage

```
NCSError CNCSRenderer::ReadImage (
      NCSFileViewSetInfo *pViewSetInfo)
```

Remarks:

Reads the current image into an internal buffer ready for blitting to a device context. In progressive mode, when a `RefreshUpdate` callback arrives from the network, the client should call `ReadImage` to transfer the pending imagery from the network into an internal buffer. Once this is done, the client can then call `DrawImage` at any time to draw from the internal buffer into the device. In non-progressive mode the client should call `ReadImage`, then immediately call `DrawImage` to draw to the device.

Parameters:

- `pViewSetInfo` - A pointer to the `NCSFileViewSetInfo` structure passed to the `RefreshUpdate` function.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::ReadLineBGR

```
NCSEcwReadStatus CNCSRenderer::ReadLineBGR (
      UINT8 *pBGRTriplet) [virtual]
```

Remarks:

Read a line from the current view in BGR format. Optionally collect histograms based on the most recent call to `CNCSRenderer::CalcHistograms`.

Parameters:

- `pBGRTriplet` - A pointer to a buffer which receives a scanline of BGR packed image data.

Returns:

`NCSEcwReadStatus` read status code.

CNCSRenderer::ReadLineBIL

```
NCSEcwReadStatus CNCSRenderer::ReadLineBIL (
      UINT8 **ppOutputLine) [virtual]
```

Remarks:

Read a line from the current view in BIL (Band Interleaved by Line) format. Optionally collect histograms based on the most recent call to `CNCSRenderer::CalcHistograms`.

Parameters:

- `ppOutputLine` - A pointer to an array of band buffers which receive a scanline of BIL image data.

Returns:

`NCSEcwReadStatus`, read status code.

CNCSRenderer::ReadLineRGB

```
NCSEcwReadStatus CNCSRenderer::ReadLineRGB (
      UINT8 *pRGBTriplet) [virtual]
```

Remarks:

Read a line from the current view in RGB format. Optionally collect histograms based on the most recent call to `CNCSRenderer::CalcHistograms`.

Parameters:

- `pRGBTriplet` - A pointer to a buffer which receives a scanline of RGB packed image data.

Returns:

`NCSEcwReadStatus`, read status code.

CNCSRenderer::SetBackgroundColor

```
void CNCSRenderer::SetBackgroundColor (
      COLORREF nBackgroundColor)
```

Remarks:

Sets the background color of the display area, which is initialised as the system default background color. In non-transparent mode, this color will be drawn to the display area background before the rendered image.

Parameters:

- `nBackgroundColor` - Value specifying the desired color.

CNCSRenderer::SetTransparent

```
void CNCSRenderer::SetTransparent (
      BOOLEAN bTransparent)
```

Remarks:

Specifies whether the renderer is responsible for doing a background fill before drawing imagery. If the renderer is being used in an application that contains other image layers, the transparency mode should be set to FALSE and the application should do the work of managing the display. If the renderer is incorporated into a single-layered control then it is appropriate to set the transparency mode to TRUE to reduce the amount of work required from the renderer's container.

Parameters:

- `bTransparent BOOLEAN` - Value specifying whether or not to draw the image transparently

Returns:

None.

CNCSRenderer::WriteJPEG

```
NCSError CNCSRenderer::WriteJPEG (
      UINT8 **ppBuffer,
      UINT32 *pBufferLength,
      INT32 nQuality)
```

Remarks:

Writes a JPEG file based on the current view, and stores it in a buffer that can be output to file later, or used for some other purpose. This function can only be called successfully if the current view has been opened on an ECW or JPEG 2000 file in non-progressive mode.

Parameters:

- `ppBuffer` - Pointer to a buffer in which to store the JPEG output.
- `pBufferLength` - Pointer to an integer buffer that receives the length of the created JPEG buffer.
- `nQuality` - Integer specified of JPEG output quality.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::WriteJPEG

```
NCSError CNCSRenderer::WriteJPEG (
      char *pFilename,
      INT32 nQuality)
```

Remarks:

Writes the current view to a JPEG file with the specified filename. This function can only be called successfully if the current view has been opened on an ECW or JPEG 2000 file in non-progressive mode.

Parameters:

- `pFilename` - ASCII string specifying the output filename.
- `nQuality` - Desired quality of the output JPEG file.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

CNCSRenderer::WriteWorldFile

```
NCSError CNCSRenderer::WriteWorldFile(char *pFilename)
```

Remarks:

This call is used to write a world file containing the georeferencing information for the current view. The world file written is given the same name as the input filename, excepting that its extension is constructed from the first and third letters of the extension of the input and the character “w”. For example, .jpg becomes **.jgw** and **.tif** becomes **.tfw**.

Parameters:

- `pFilename` - The filename on which to base the output world filename.

Returns:

An `NCSError` value to use for error checking. The return value is `NCS_SUCCESS` if the operation succeeds.

Class Reference: CNCSError

The `CNCSError` class is used for the purpose of detailed error reporting. The class wraps an `NCSError` enum value, and allows detailed control of error logging level and error messaging. Many of the methods of the `CNCSFile` and `CNCSRenderer` classes that you will be using to build your SDK application with the C++ API return a `CNCSError` value which you will find helpful in handling problems and debugging your work.

Constructor

```
CNCSError::CNCSError(
      const NCSError eError = NCS_SUCCESS,
      char *pFile = __FILE__,
      int nLine = __LINE__,
CNCSLog::NCSLogLevel eLevel = CNCSLog::
      LOG_LEVEL1, const char *pText = (char *)NULL)
```

Remarks

This overloaded constructor has a large number of initialization parameters, all with sensible default values. In the main you will probably be interested in using the `eError` and `pText` parameters within your own classes.

```
CNCSError::CNCSError(const CNCSError &Error)
```

This copy constructor initializes a new CNCSError object from an existing object.

Destructor

```
CNCSError::~CNCSError()
```

Remarks

Releases all resources associated with a `CNCSError` object.

CNCSError::GetErrorMessage

```
char *CNCSError::GetErrorMessage (
      char *pFormat = NULL, ...)
```

Remarks:

Obtain a descriptive error message from this `CNCSError` object with optional formatting.

Parameters:

- `pFormat` - Optional printf style format string
- `...` - Optional additional arguments for `printf` style format string.

Returns:

Formatted ASCII string describing the error that has occurred.

CNCSError::GetErrorNumber

```
NCSError CNCSError::GetErrorNumber() [inline]
```

Remarks:

Returns the `NCSError` enum value associated with this `CNCSError` object.

Parameters:

None.

Returns:

Associated `NCSError` enum value.

CNCSError::Log

```
void Log(CNCSLog::NCSLogLevel eLevel)
```

Remarks:

Log the error to the log file, if the logging level is greater than or equal to the level specified by `eLevel`.

Parameters:

- `eLevel` - Log level required before the error should be logged. This can have the values:
  - `LOG_LOW = 0,`
  - `LOG_MED = 1,`
  - `LOG_HIGH = 2,` or
  - `LOG_HIGHER = 3.`

Returns:

None.

CNCSError::operator=

```
CNCSError &CNCSError::operator= (
```

```
        const CNCSError &Error)
```

Remarks:

Overloaded assignment operator.

Parameters:

- `Error` - Reference to the error to be assigned.

Returns:

Reference to the newly altered error object.

CNCSError::operator==

```
bool CNCSError::operator== (
      const CNCSError &Error) [inline]
```

Remarks:

Compare two `CNCSError` objects.

Parameters:

- `Error` - Object to compare this object with.

Returns:

`TRUE` if the two objects have the same error number (only the error number is compared in checking equality).

CNCSError::operator!=

```
bool CNCSError::operator!= (
      const NCSError eError) [inline]
```

Remarks:

Overloaded inequality operator checking whether a `CNCSError` object and an `NCSError` enum value have different types.

Parameters:

- `eError` - Enum value for comparison.

Returns:

`TRUE` if this object and the `NCSError` enum value are not of the same type.

CNCSError::operator!=

```
bool CNCSError::operator!= (
      const CNCSError &Error) [inline]
```

Remarks:

Overloaded inequality operator checking whether two `CNCSError` objects are not equal.

Parameters:

- `Error` - Object to compare to this object.

Returns:

TRUE if the two objects do not share the same error number (only the error number is compared in checking inequality).

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.

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

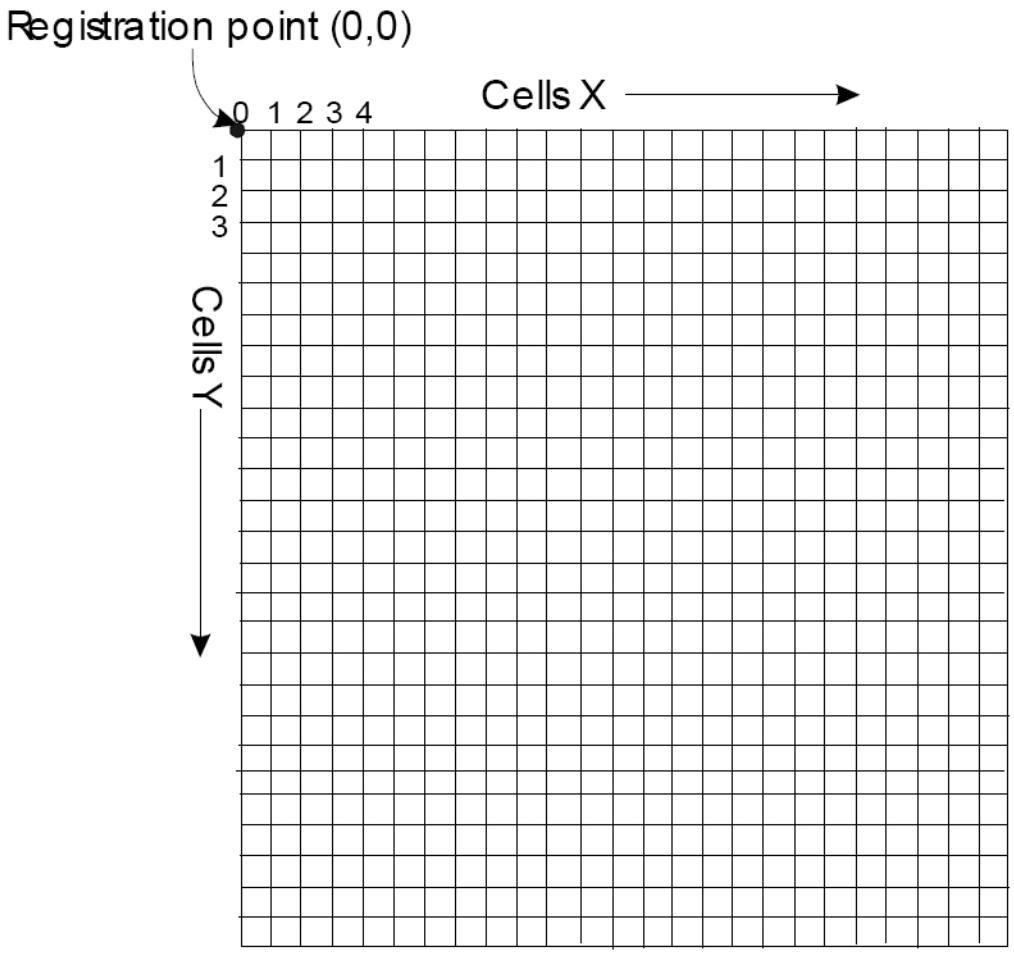
The measurement units are usually set for the specific projection. They can be:

- 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).

Geodetic Transform Database

The Geodetic Transform Database package (known as GDT), can be used to perform all coordinate transformations. This includes calculations of easting/northing from latitude/longitude, and the reverse, for a point with a given map projection and datum. GDT provides projection parameters and all mathematical software to perform the transform calculations.

GDT File Formats

The GDT database stores all associated files in the runtime/GDT_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 GDT_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 x 180) / PI = 39.6666666 degrees = 39 degrees 40 minutes East.

How the ECW JPEG2000 SDK Reads Geocoding Information

The SDK represents registration, projection and datum information internally using fields in the `NCSFileViewFileInfoEx` 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 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 OpenGIS Consortium 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: http://www.opengis.org/

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 `NCSFileViewFileInfoEx` data structure which can be queried from the open file using `NCScbmGetViewFileInfoEx` (via the C API) or `CNCSFile::GetFileInfo` (via the C++ API). The GML data itself is not exposed to the application programmer.

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 GeoTIFF file. The information is processed into ER Mapper style projection, datum, linear units and registration information. The table below indicates which GeoTIFF tags and `GeoKeys` are supported by the SDK.

Supported GeoTIFF tags:

- `ModelTiePoint,`
- `ModelPixelScale,`
- `ModelTransformation,`
- `m GeoKeyDirectory,`
- `Geo-ASCIIParams,`
- `GeoDoubleParams.`

Supported GeoTIFF GeoKeys:

```
GTRasterType, GTModelType, GeographicType,
ProjectedCSType, GeogLinearUnits, ProjLinearUnits
```

After the SDK opens a JPEG 2000 file which contains an embedded GeoTIFF file, the processed georeferencing information is not available to the SDK user in GeoTIFF format, but can instead be queried from an `NCSFileViewFileInfoEx` structure obtained from a call to `NCScbmGetViewFileInfoEx` in the C API or to `CNCSFile::GetFileInfo` in the C++ API.

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 `NCSFileViewFileInfoEx` 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
`JP2_GEODATA_USE_NONE`	No processing of metadata.
`JP2_GEODATA_USE_WLD_ONLY`	World file only.
`JP2_GEODATA_USE_GML_ONLY`	GML header box only.
`JP2_GEODATA_USE_PCS_ONLY`	GeoTIFF UUID box only.
`JP2_GEODATA_USE_WLD_GML`	World file, then GML box.
`JP2_GEODATA_USE_WLD_PCS`	World file, then GeoTIFF box.
`JP2_GEODATA_USE_GML_WLD`	GML box, then world file.
`JP2_GEODATA_USE_GML_PCS`	GML box, then GeoTIFF box.

JP2_GEODATA_USE_PCS_WLD	GeoTIFF box, then world file.
JP2_GEODATA_USE_PCS_GML	GeoTIFF box, then GML box.
JP2_GEODATA_USE_WLD_GML_PCS	World file, then GML, then GeoTIFF.
JP2_GEODATA_USE_WLD_PCS_GML	World file, then GeoTIFF, then GML.
JP2_GEODATA_USE_GML_WLD_PCS	GML, then world file, then GeoTIFF.
JP2_GEODATA_USE_GML_PCS_WLD	GML, then GeoTIFF, then world file.
JP2_GEODATA_USE_PCS_WLD_GML	GeoTIFF, world file, then GML.
JP2_GEODATA_USE_PCS_GML_WLD	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 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 ER Mapper projection and datum strings, the SDK converts these strings to a corresponding EPSG code which is embedded in the GML or GeoJP2 header boxes, and can subsequently be re-read by ER Mapper and third party software.

The mapping between 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:

- by 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";
  ```

- by 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 `CNCSFile::SetGDTPath` calls, if your application uses the C or C++ APIs respectively.

FAQ

This section serves as a quick reference guide to some of the terminologies you may encounter when using this software.

What is Lossless Compression

Lossless compression provides a compressed image that can uncompress to an identical copy of the original image. This perfect reconstruction is the advantage of lossless compression. The disadvantage of lossless compression is a ratio limit of approximately 2:1 compressed file size.

What is Lossy Compression

Lossy compression provides a compressed image that can uncompress to an approximate copy of the original image.

Lossy compression sacrifices some data fidelity, in order to achieve much higher compression rates than those available through lossless compression. These higher compression ratios are the advantages of lossy compression.

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) 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.

SDK Compresssion Rates

The SDK can achieve compression rates of up to 95%. Your actual results will vary depending upon the type of compression you use, your settings, and the original file.

Maximum Output Bit Depth Per Image Component Supported by the

SDK

The SDK's 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, so if image quality is a high priority you can still compress to (say) 26 bits of depth per component and later extract the image data into 32- bit pixel buffers.

Any files (lossy or lossless) compressed to greater 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 compressions with more than 32 bits of depth due to some rather obscure restrictions in the format of the JPEG 2000 codestream. A 64 bit pipeline should allow 33-38bit lossy compressions.

Is the SDK 64 Bit Enabled

The SDK source is ready for the next generation of 64-bit processors and operating systems.

Which File Formats are Encoded by the ECW JPEG2000 SDK

The SDK will compress to the ECW and JPEG 2000 file formats. Currently the SDK supports image compressions of up to 10,000 gigabytes for ECW and over 1,000 gigabytes for JPEG 2000. Compression of .ecw files operates as in previous versions of the SDK. The .jp2 files compressed by the SDK will actually be backwards-compatible jpx files from Part 2 of the ISO JPEG 2000 Standard, allowing ER Mapper to embed georeferencing 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.

How the SDK Manages Decompression Functions on the Opacity Channel

Various BGRA, RGBA etc decompression functions will either return a value of 0 or 255 for the opacity channel. It should also be noted that multiband format is the correct format in which to compress actual RGBA information, which can then be retrieved using the BIL reading functions.

How the SDK Manages 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 (i.e. compression input buffer) is determined by the `NCSEcwCellType` 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 no other types.

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

What Happens if the Bit Depth Specified is Greater than the Maximum Bit Depth for the Cell Type

If for example you inserted a value 8 in `nBits` but the cell type was `NCST_UINT8` it will most likely work. However by specifying more bits than there actually, you are just confusing downstream applications decompressing the image - i.e. most will assume 16 bit will be 16 bits of data, and rescale for display as appropriate (kdu_show does this - ER Mapper however calculates a histogram and creates a default transform based on that).

How does the SDK Handle Optimal Block Sizes

The SDK now 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 backwards compatibility with old SDK applications.

This section outlines the functionality present in the current version of the SDK.

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 technology to achieve scalable compression ranging from lossless to arbitrarily lossy, while retaining unprecedented decompressed image quality.

What is the Extent of SDK Support for JPEG 2000

ERDAS is the only vendor in the geospatial industry to commit to developing its own JPEG 2000 implementation in order to provide superior solutions for multi-terabyte JPEG 2000 images. This includes compliance class 2 JPEG 2000 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.

GeoJP2 Support in the SDK

The GeoJP2 standard for embedding geospatial information in .jp2 files, started by the now defunct Mapping Science Inc., 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 georeferenced 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 SDK.

ERDAS also supports the inclusion of georeferencing 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.

Inconsistent Decompressiong Speeds with JPEG 2000 Files

JPEG 2000 files are instances of a large and highly customizable specification. The JPEG 2000 standard supports many different compression formats, some of which are more optimized towards quick loading than others. As a consequence, the speed performance of the SDK can be somewhat variable across the range of all input files. The SDK will decompress JPEG 2000 files at rates comparable to or better than those achieved by other decoder implementations.

What is GML

The Geography Markup Language is an XML grammar and schema for recording and transferring geographic data. GML has been developed by the OpenGIS Consortium 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).

Support for Bi-level Imagery in the SDK

Bi-level images are an important subset of the images that can conform to the JPEG 2000 specification. The standard supports bit-depths from 1-31 allowing a maximum level of flexibility. The 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.

This section of the FAQ discusses the compression format ECW.

What is ECW

ECW is an acronym for Enhanced Compressed Wavelet, a popular standard for compressing and using very large images.

What is ECWP

ECWP is an acronym for Enhanced Compression Wavelet Protocol. It is the protocol used to transmit images compressed with ECW over networks such as the Internet. ECWP offers the fastest possible access to large ECW and JPEG 2000 images and is fully supported by ERDAS Image Web Server.

What is ECWPS

ECWPS is the version of ECWP that includes security. ECWPS enables private and secure encrypted streaming of image data over public networks such as the Internet. ECWPS is a feature included with Image Web Server. It is also available for ArcGIS through the ECW JPEG2000 Plug-in.

What is GeoTIFF

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

What is NITF

The 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, the NITF has been accepted through Standardization Agreements by NATO and other international defense organizations.

NITF Support in the SDK

It is possible to enable the SDK to encode and decode to NITF/NSIF BIIF NPJE, EPJE compliant codestreams.

What is a Projection

A map projection is a mathematical function used to plot a point from an ellipsoid, on a plane such as a sheet of paper. Projections attempt to replicate characteristics of the surface geometry at the given point. Dozens of different projections are available.

How the SDK Handles Partially Georeferenced Datasets

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, so that multiple such datasets can be accurately overlaid (similar to the use of world files for this purpose).

An ECW file that is listed as in WGS84/LOCAL is processed with vertical values inverted in 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 a partially georeferenced dataset may be compressed using the SDK, 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 georeferenced dataset is compressed without a stored EPSG code, and when reloaded 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.

Index

A

Applications 11

B

bin Directory 17
Blocking Reads 23, 24

C

C API 70
Caching 27
CNCSError
    Constructor 124
    Destructor 124
    GetErrorMessage 124
    GetErrorNumber 125
    Log 125
    operator!= 126, 127
    operator= 125
    operator== 126
CNCSFile
    AddBox 85
    BreakdownURL 85
    Close 86
    ConvertDatasetToWorld 86
    ConvertWorldToDataset 87
    DetectGDTPath 87
    FormatErrorText 88
    GetBox 88
    GetClientData 89
    GetEPSGCode 89
    GetFile 90
    GetFileInfo 91
    GetFileMimeType 91
    GetFileType 91
    GetFileViewSetInfo 92
    GetGDTPath 92
    GetNCSFileView 93
    GetPercentComplete 95
    GetPercentCompleteTotalBlocksInView 95
    GetProjectionAndDatum 95
    GetStream 96
    GetUUIDBox 96
    GetXMLBox 97
    Open 97, 98
    ReadLineABGR 98
    ReadLineARGB 99
    ReadLineBGR 99
    ReadLineBGRA 100
    ReadLineBIL 100, 101
    ReadLineRGB 104
    RefreshUpdate 105
    RefreshUpdateEx 105
    SetClientData 106
    SetCompressClient 106
    SetFileInfo 106
    SetGDTPath 107
    SetParameter 107, 108
    SetRefreshCallback 110
    SetView 110, 111, 112
    Write 113
    WriteCancel 113
    WriteLineBIL 114
    WriteReadLine 114
    WriteStatus 115
CNCSRenderer
    ApplyLUTs 116
    CalcHistograms 116
    DrawImage 116
    FreeJPEGBuffer 117
    GetHistogram 118
    GetTransparent 118
    ReadImage 118, 119, 120
    ReadLineBGR 120
    ReadLineBIL 121
    ReadLineRGB 121
    SetBackgroundColor 122
    SetTransparent 122
    WriteJPEG 122, 123
    WriteWorldFile 123
Compression Examples 35
Compression Ratio, Optimizing 31
Contacting Us 4
Coordinate Information 28

D

Datum 129
Decompression Examples 43

E

ECW 145
ECW Compression 6
ECWP 145

ECWPS 145

edistributable 17
Embedded Geography Markup Language (GML) 132
EPSG codes 138
examples Directory 17

G

Geodetic Transform Database 131
GeoJP2 144
GeoTIFF 145
GML 144
GML Examples 133

H

Hyperspectral Imagery 32

I

include Directory 17
Installation 12
Introduction 5

J

JP2_COMPRESS_INCLUDE_EPH 109
JP2_COMPRESS_INCLUDE_SOP 109
JP2_COMPRESS_LAYERS 109
JP2_COMPRESS_LEVELS 109
JP2_COMPRESS_PRECINCT_HEIGHT 109
JP2_COMPRESS_PRECINCT_WIDTH 109
JP2_COMPRESS_PROFILE_BASELINE_0 108
JP2_COMPRESS_PROFILE_NITF_BIIF_NP JE 108
JP2_COMPRESS_PROGRESSION_LRCP 109
JP2_COMPRESS_PROGRESSION_RLCP 109
JP2_COMPRESS_TILE_HEIGHT 109
JP2_COMPRESS_TILE_WIDTH 109
JP2_DECOMPRESS_LAYERS 109
JP2_DECOMPRESS_RECONSTRUCTION_ PARAMETER 109
JP2_GEODATA_USAGE 109
JPEG 2000 Compression 8

L

lib Directory 17
License and Documentation Information 17
Lossless Compression 141
Lossy Compression 141

M

Memory Management 26

N

NCScbmCloseFileView 53
NCScbmCloseFileViewEx 54
NCScbmGetFileMimeType 79
NCScbmGetFileType 79
NCScbmGetViewFileInfo 54
NCScbmGetViewFileInfoEx 55
NCScbmGetViewInfo 56
NCScbmOpenFileView 56
NCScbmReadViewLineBGR 57
NCScbmReadViewLineBGRA 58
NCScbmReadViewLineBIL 58
NCScbmReadViewLineBILEx 59
NCScbmReadViewLineRGB 60
NCScbmReadViewLineRGBA 61
NCScbmSetFileView 61
NCScbmSetFileViewEx 62
NCSCopyFileInfoEx 80
NCSDetectGDTPath 80
NCSEcwCompress 72
NCSEcwCompressAllocClient 71
NCSEcwCompressClient 76
NCSEcwCompressClose 72
NCSEcwCompressFreeClient 73
NCSEcwCompressOpen 71
NCSecwSetConfig 64
NCSecwSetIOCallbacks 65
NCSFileBandInfo 69
NCSFileViewFileInfo 66
NCSFileViewFileInfoEx 67
NCSFileViewSetInfo 69
NCSFreeFileInfoEx 80
NCSGetEPSGCode 81
NCSGetGDTPath 82
NCSGetProjectionAndDatum 82
NCSInitFileInfoEx 83
NCSSetGDTPath 83
NCSSetJP2GeodataUsage 83
newlink contact 4
NITF 145
NITF Standard Support 9

O

Operating system 11

P

pCancelCallback 75
Platforms 11
pReadCallback 74
Projection 129, 146
pStatusCallback 75

R

Read a View, 20
Reads, canceling 26
redistributable 17
Refresh Callbacks 25
Registration Point 130

S

SetFileView 20
Start Menu Items 16
Subdirectories and Files 16
System Requirements 11

T

testdata Directory 18
Third Party Packages 11
Transparent Proxying 29

U

Units 130

V

Viewing and Printing this Document 1

W

Wavelet Based Encoding 6
Wavelet Compression 141
What's New in Version 3.3 3