LizardTech

MrSID Decode SDK 9.5

for Raster

User Manual

Copyright © 2009–2015 Celartem, Inc., doing business as LizardTech. All rights reserved. Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of those agreements. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of LizardTech.

LizardTech, MrSID, GeoExpress and Express Server are registered trademarks in the United States and the LizardTech, GeoExpress, Express Server, ExpressView and GeoViewer logos are trademarks, and all are the property of Celartem, Inc., doing business as LizardTech. Unauthorized use is prohibited.

LizardTech acknowledges and thanks the many individuals and organizations whose efforts have made our products possible. A full list of copyright, trademark and credit information is available in the document "Copyrights, Trademarks and Credits" installed automatically with your product.

LizardTech
1008 Western Avenue, Suite 403
Seattle, Washington, USA 98104
206-652-5211
www.lizardtech.com

Table of Contents

Introduction

Welcome to the MrSID® Software Development Kit (SDK) for raster imagery. This is the documentation for the Decode version (DSDK).

Digital images have become important to every aspect of business, industry, and government. Because of the enormous amounts of data involved, the use of high-quality images has been hindered by storage and bandwidth constraints. LizardTech's technologies and products solve these problems and lay the foundation for truly dynamic image access.

Used as the foundation for LizardTech's other geospatial products including GeoExpress, ExpressServer, ExpressView Browser Plug-in and other applications, the MrSID SDK is a robust toolkit suitable for complex application development needs.

The SDK provides a framework for creating image pipelines that enable developers to efficiently read, write and manipulate data in a variety of formats, including MrSID, JPEG 2000, and other common geospatial raster formats.

NOTE: The Decode SDK does not support writing .sid and .jp2 images. For information about this kind of support, contact your regional LizardTech office.

The MrSID format supports LiDAR data as well as raster data, but a separate set of tools and libraries is used in supporting LiDAR data in the MrSID format. Separate documentation is available in your installation for integrating support for LiDAR-encoded MrSID files.

Your installation also includes a copy of the End User License Agreement (EULA) and a README file.

MrSID

MrSID's unique file format enhances the workflow of image data by compressing high-resolution images to a fraction of their original file size while maintaining image quality. Because MrSID images are scalable, you can reduce, enlarge, zoom, pan, or print without compromising integrity. And with the selective decoding capability of MrSID technology, you can view any region of an image at different resolutions. Separate encoding, optimizing, and decoding capabilities give you the flexibility you need to create and deliver imagery to users with different bandwidth or storage resources.

MrSID has been a mainstay of the geospatial community for many years, and support for MrSID imagery has been implemented in hundreds of applications using the MrSID SDK.

The latest version of MrSID, MrSID Generation 4 (MG4), supports the decoding of selected bands, which is critical if your application is being used to view multispectral imagery. Most users of multi- and hyperspectral data don't wish to view all bands from an image at once, so providing them with the means of selecting bands will save your application decoding time and provide users with a better user experience.

JPEG 2000

The JPEG 2000 image compression standard offers many of the same advantages as MrSID, plus the added benefits of being an international standard (ISO/IEC 15444). The MrSID SDK allows your

applications to use JPEG 2000 compression on geospatial images with the same level of efficiency, metadata, and large-image support already available with MrSID.

Features

The MrSID SDK is a toolkit for software developers. It is written in C++ to provide an object-oriented framework for working with images using MrSID and JPEG 2000 technology.

The major features of the MrSID SDK include:

Architecture

The SDK uses a classical image-pipeline design, providing a single, unified model for reading, writing, and transforming image data. Developers can easily use existing components or derive their own for additional functionality.

Classes

A set of basic classes for working with geospatial imagery are provided, including representation of decode scene extents, memory buffers for image data, pixel data, metadata, etc.

MrSID support

The SDK provides full support for MrSID Generation 2 (MG2), MrSID Generation 3 (MG3), and MrSID Generation 4 (MG4) images, including the ability to work with “composite” MG3 and MG4 images.

JPEG 2000 support

Decoding of JPEG 2000 images is provided using the same framework as the MrSID operations, including support for geospatial metadata.

NITF support

Decoding of NITF images is provided using the same framework as the MrSID operations, including support for geospatial metadata.

Support for WKTs

The MrSID SDK supports spatial reference system metadata for MrSID, JPEG 2000, and other georeferenced image formats.

Simple C API

For those developers who want rudimentary decoding of MrSID or JPEG 2000 imagery without the complexity of the C++ interface, a simple C API is provided. While limited in functionality, the C API enables access to basic image properties and scene decoding.

Image writers

The SDK provides support for writing a number of common file formats, including BIP/BIL/BSQ (raw), Windows BMP, JPEG, and TIFF/GeoTIFF.

Image transformers

The SDK also provides a number of common image filter or transform operations, including cropping, watermarking, mosaicking, scaling, dynamic range adjustment, datatype conversion, and colorspace conversion.

Documentation

The documentation includes both a full User Manual and a Reference Manual, plus a number of examples showing how to implement common tasks with the SDK.

Platforms

The SDK is available for many platforms, including Windows, Linux (x86), and Mac (OS X). The Windows version provides dynamic libraries (DLLs). Both 32- and 64-bit versions are available for all platforms.

Component interoperability

The MrSID SDK's public interfaces are coded to a clean subset of C++ language features to avoid the interoperability problems often encountered by developers: STL incompatibilities, exceptions and memory allocations crossing library boundaries, and advanced template compilation.

How to Read this Manual

This User Manual gives a high-level view of the design, features and classes that make up the MrSID SDK.

The chapter Architecture and Design describes the overall architecture of the image pipeline and strip-based decoding mechanism that this SDK implements.

Further chapters briefly motivate and describe the various classes that make up the SDK.

Note that the descriptions provided are designed only to introduce the classes; for detailed information on specific features, methods, etc, please refer to the Reference Manual at `doc/ReferenceManual/index.html`.

Included in this manual is a chapter describing the command line applications that are included with the SDK, as well as a glossary and an appendix of "technical notes" explaining minutiae about which you might be curious.

SDK Contents

The contents of the MrSID SDK include the following:

Documentation

Cover documentation

In the top-level directory, the `README.txt` and `CHANGES.txt` files contain information about late changes to the SDK

License

In the top-level directory, the file `LICENSE.txt` contains the complete licensing information for this SDK

User Manual

The User Manual (this document) can be found at `doc/UserManual/index.htm`

Reference Manual

The Reference Manual, containing detailed information about each class and method, can be found at `doc/ReferenceManual/index.html`

Headers and Libraries

Headers

The header files for the MrSID SDK are located in the `include` directory. (The Reference Manual provides full documentation for these headers.)

Libraries

The libraries for the SDK are located in the `lib` directory

Non-LizardTech libraries

Other non-LizardTech libraries supplied with this SDK are located in the `3rd-party` directory

Sample Applications

Command-line tools

A number of tools are provided in the `bin` directory to aid in development, debugging, and testing. (Documentation for these tools can be found in the chapter, Command Line Applications.)

Example Code

A number of example functions are included in the directory `examples/src`. The test images used by these examples are located in `examples/data`. (The Reference Manual provides additional information about these examples.)

Getting Started

This chapter provides some preliminary information to get you started using the MrSID SDK. At the least, we suggest you skim this User Manual and the accompanying Reference Manual at `doc/ReferenceManual/index.html`, then build and execute the provided example application. The example sources should give you enough information to determine what level of SDK support your own application will require. At that point you can go back and reread the User and Reference manuals more closely, focusing on the areas appropriate to your situation.

NOTE: The interfaces and libraries supplied with the MrSID SDK are not compatible or interoperable with the previous 7.x series of MrSID SDK releases, also called the GeoExpress SDK. Existing MrSID-enabled applications will require source code changes to make use of the SDK.

System Requirements

The MrSID SDK is a set of C++ libraries that must be used in conjunction with the specific development environment for your platform. The supported configurations are listed below.

For optimal performance, verify that your system meets the following minimum recommended hardware requirements:

- 2 GHz processor
- 2 GB of RAM

NOTE: Please contact LizardTech for additional distributions for other platforms.

Windows (64-bit)

Your development environment	Target platform	Library to use
Visual Studio 2017 on Windows Server 2008 or newer	64-bit Windows 7/ 8/ 10/ Server 2008/ Server 2012/ Server 2016	Visual C++ 14.0 (VC14.0 Update 3 or greater) / 64-bit
Visual Studio 2015 on Windows Server 2008 or newer	64-bit Windows 7/ 8/ 10/ Server 2008/ Server 2012/ Server 2016	Visual C++ 14.0 (VC14.0) / 64-bit
Visual Studio 2013 on Windows Server 2008 or newer	64-bit Windows Vista/ 7/ 8/ Server 2003/ Server 2008/ Server 2012/ Server 2016	Visual C++ 12.0 (VC12.0) / 64-bit

Windows (32-bit)

Your development environment	Target platform	Library to use
Visual Studio 2017 on Windows Server 2008 or newer	32-bit Windows 7/ 8/ 10/ Server 2008/ Server 2012/ Server 2016	Visual C++ 14.0 (VC14.0 Update 3 or greater) / 32-bit
Visual Studio 2015 on Windows Server 2008 or newer	32-bit Windows 7/ 8/ 10/ Server 2008/ Server 2012/ Server 2016	Visual C++ 14.0 (VC14.0) / 32-bit
Visual Studio 2013 on Windows Server 2008 or newer	32-bit Windows Vista/ 7/ 8/ Server 2003/ Server 2008/ Server 2012/ Server 2016	Visual C++ 12.0 (VC12.0) / 32-bit

Linux

Your development environment	Target platform	Library to use
GCC 5.3.1 on RHEL 6.8/64	64-bit RHEL 6.8/ RHEL 7.0/ CentOS 7.0	GCC 5.3.1 / 64-bit
GCC 4.8.2 on RHEL 6.8/64	64-bit RHEL 6.8/ RHEL 7.0	GCC 4.8.2 / 64-bit

NOTE: The MrSID libraries for Red Hat Linux are built using the Red Hat Developer Toolset on Red Hat Enterprise Linux 6.8. Execution of applications built using this SDK is only supported on RHEL 6.8 or later. (The GCC 5.3.1 compiler is included in version 4.1 of the Software Collections Developer Toolset.)

Macintosh

Your development environment	Target platform	Library to use
Clang 8.0 (part of Xcode 8.2) on macOS 10.12	macOS 10.12 or later	Mac OS X 10.12 / Universal / Darwin 16
Clang 7.0 (part of Xcode 7.3) on Mac OS X 10.11	macOS 10.11 or later	Mac OS X 10.11 / Universal / Darwin 15

iOS

Your development environment	Target platform	Library to use
Clang 8.0 (part of Xcode 8.2) on macOS 10.12	**OS:** iOS 8 and higher **Processor:** ARMv7/ ARMv7s/ ARM64/ x86 and x86-64 simulators	Xcode 8.2 iOS 8 / Universal
Clang 7.0 (part of Xcode 7.3) on Mac OS X 10.11	**OS:** iOS 8 and higher **Processor:** ARMv7/ ARMv7s/ ARM64/ x86 and x86-64 simulators	Xcode 7.3 iOS 8 / Universal

Android

Your development environment	Target platform	Library to use
Android NDK 13b on Ubuntu Desktop 12.04	**OS:** Android API Level 12 and higher **Processor:** armeabi/ armeabi-v7a/ x86/ arm64-v8a/ x86_64	GCC 4.9 / Universal

Installation

No specific installation is required to use the GeoExpress SDK beyond copying the SDK contents from the media provided (CD, ISO CD image, archive from FTP site, etc.) to your local computer.

See the file `README.txt` in your installation directory for complete instructions.

Example Code

The GeoExpress SDK includes a number of examples that demonstrate the major features of the SDK. The table below shows which files to consult for various types of operations.

Code example files to use as demonstrations of various common tasks

Opening image files	JP2	`DecodeJP2ToBBB.cpp`
		`DecodeJP2ToJPG.cpp`
		`DecodeJP2ToMemory.cpp`

	MrSID	`DecodeMrSIDBandSelection.cpp`
		`DecodeMrSIDToMemory.cpp`
		`DecodeMrSIDToRaw.cpp`
		`DecodeMrSIDToTIFF.cpp`

	NITF	`DecodeNITFToBBB.cpp`
Setting the number of threads for a decoding operation	MrSID	`DecodeMrSIDToMemory.cpp`
Getting image properties		`ImageInfo.cpp` `MetadataDump.cpp`

Writing images files	Raw	`DecodeJP2ToBBB.cpp`
		`DecodeMrSIDToRaw.cpp`
		`DecodeNITFToBBB.cpp`

	JPEG	`DecodeJP2ToJPG.cpp`
	TIFF	`DecodeMrSIDToTIFF.cpp`
Working with pixel data using `LTISceneBuffer`	Accessing the BSQ data	`DecodeMrSIDBandSelection.cpp` `DerivedImageFilter.cpp`
	Converting between BSQ and BIP	`DerivedImageReader.cpp` `DerivedImageWriter.cpp`
	Using your own memory	`DecodeJP2ToMemory.cpp` `DecodeMrSIDToMemory.cpp`
	Using sub-buffers	`SceneBuffer.cpp`
Working with multispectral data		`DecodeMrSIDBandSelection.cpp`
Working with Alpha		`DecodeMrSIDBandSelection.cpp`
Creating custom progress and interrupt delegates		`ProgressDelegate.cpp` `InterruptDelegate.cpp`

Converting a Geo region of interest to an `LTIScene`		`GeoScene.cpp`
Checking MrSID version		`DecodeMrSIDLidar.cpp`
Looking up error messages		`ErrorHandling.cpp`
Working with `LTIOStreamInf`		`DerivedStream.cpp` `UsingStreams.cpp`
Building an image pipeline		`Pipeline.cpp`
Creating sub-class of `LTIImageStage`	`LTIImageReader`	`DerivedImageReader.cpp`
	`LTIImageFilter`	`DerivedImageFilter.cpp`
	`LTIImageWriter`	`DerivedImageWriter.cpp`

Technical Support

Most technical issues can be resolved using the various resources you have available. In addition to the product documentation and the README file, LizardTech offers a knowledge base and product updates on the LizardTech website.

Knowledge Base

http://www.lizardtech.com/support/kb/

The LizardTech Knowledge Base contains articles about known technical and usage issues and is frequently updated.

Developer Website

http://developer.lizardtech.com

The LizardTech Developer Website provides you with the tools you need to support viewing MrSID and JPEG 2000 formats within your application: downloadable C++ SDKs, technical notes and documentation and a link to additional email support.

Community Forums

http://www.lizardtech.com/forums/

The fora are a place to engage in intelligent discourse with the geospatial community. Ask questions, provide answers, and share product usage tips with other Lizardtech customers around the world.

Product Updates

http://www.lizardtech.com/products

Updated versions of LizardTech viewer tools are available for download at no cost.

Support Plans

http://www.lizardtech.com/purchase/other.php

Protect your investment in LizardTech software by participating in a LizartTech support plan. For more details, please contact your regional LizardTech office.

Contacting Technical Support

http://www.lizardtech.com/support

To contact technical support, visit the website at the above URL and follow links to the LizardTech Knowledge Base or the Product Activation page. A Contact Form is also provided for issues that require further assistance.

In an emergency, call 206-902-2845 between the hours of 8 AM and 5 PM Pacific Time.

IMPORTANT: Please have the following information available to assist in resolving your problem:

- Which version of the MrSID SDK you are running
- Other LizardTech products you have installed
- Which operating system you use
- How much free hard drive space your computer has
- How much RAM your computer has
- Version of compiler
- Copy of source code demonstrating the problem, simplified as much as possible
- Relevant test data to allow us to reproduce the problem
- Copy of compiler error messages if appropriate

Architecture and Design

This section describes some of the design principles used by the MrSID SDK, including the image pipeline model, strip-based decoding and some of the C++ conventions used by the main classes.

Subsequent sections will describe the specific classes that make up the SDK.

Pipeline Design

An image processing pipeline is a system in which each image "stage" performs one specific operation on an image or a piece of an image, and then passes the resulting image data on to the next stage. The MrSID SDK uses this model to provide the ability to construct workflows that read, write, and manipulate images in a variety of ways.

Image Stages

There are three basic types of image stages:

- **Image reader:** an image reader will generate ("decode") pixel data from some external image source, such as a TIFF or MrSID file. A reader serves as the initial stage of a pipeline and passes pixel data to its successor stage(s).
- **Image filter:** an image filter or "transformer" will change the pixel data received from its predecessor image stage(s) in some way and pass the new data on to its successor image stage(s). Examples of filters include colorspace and datatype transformers, which change the properties of pixels; mosaickers, which combine multiple images into one image; and histogram stretchers, which dynamically scale the pixel values within some numeric range.
- **Image writer:** an image writer will produce ("encode") some external image object, such as an NITF or JPEG 2000 file. A writer serves as the final stage of a pipeline, receiving data from its predecessor stage; it can be viewed as the "opposite" of an image reader.

The simplest pipeline consists trivially of just a single image reader. The reader may or may not be connected to one or more filter stages. (As we will see later, a writer may not be required for all workflows.)

Some Examples

In Figure 1 an image pipeline is shown consisting of a MrSID image reader connected to a filter that changes the image's colorspace. Given a filename, the MrSID reader will pass pixel data to the filter, which will transform the (presumably RGB) pixel data to grayscale pixel data.

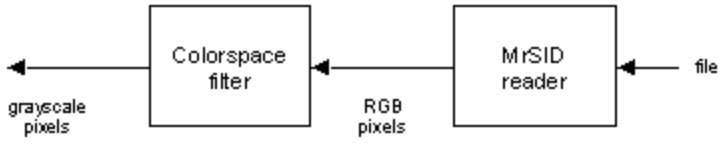


Figure 1: Simple pipeline with MrSID image reader and color space filter

Figure 2 shows a pipeline that reads raw image data (in BIP/BIL/BSQ form), performs some dynamic range adjustments on the data in a filter stage, and writes the image out again as a raw file. Such a pipeline might be used to massage 12-bit image data to more easily displayed 8-bit data.

Figure 2: Pipeline that reads and writes raw image data

Figure 3 shows a pipeline with two GeoTIFF image readers that are connected to a mosaicking stage, which is in turn connected to a JPEG 2000 writer. (The GeoTIFF images are presumed to be "compatible" in a geospatial sense.)

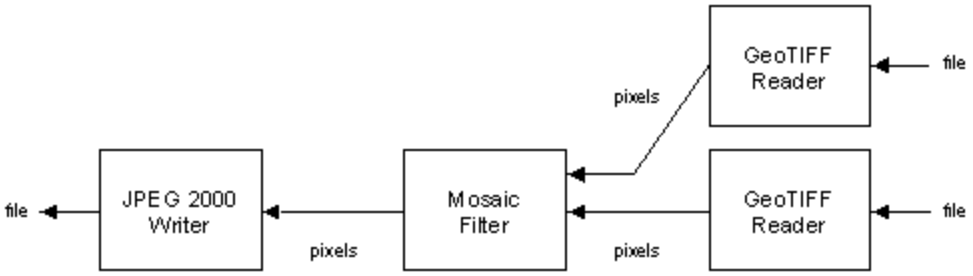


Figure 3: Pipeline with two image readers, a mosaicking filter and a JP2 writer

Finally, Figure 4 shows a watermarking pipeline: a JPEG watermark is inserted onto the base TIFF image, and the result is written to a MrSID file. Note how this pipeline diagram is structurally similar to Figure 3.

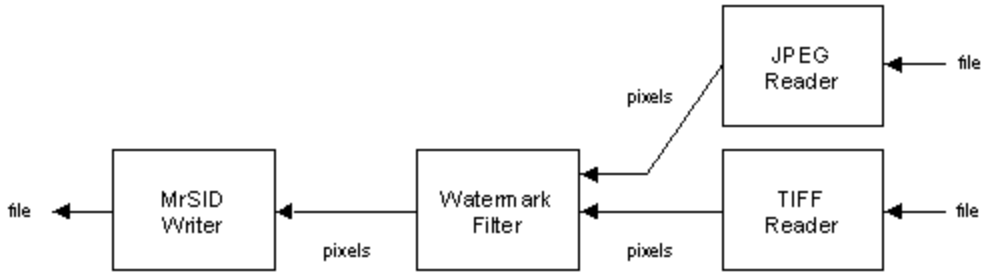


Figure 4: Watermarking pipeline

Implementation

Several abstract classes are used to construct the pipeline system.

- `LTIImage` provides the basic properties of an image, such as width and height, pixel type, etc.
- `LTIImageStage` extends the `LTIImage` class by adding functions for reading pixel data from an image.
- `LTIImageReader` extends the `LTIImageStage` class to serve as the base class for all readers.
- `LTIImageFilter` extends the `LTIImageStage` class to serve as the base class for all filters. The constructor for an `LTIImageFilter` takes an `LTIImageStage`.
- `LTIImageWriter` is the base class for all writers. It does not inherit from any of the above classes as it serves only as a "sink", i.e. it does not export any image properties as an `LTIImage` would. The constructor for an `LTIImageWriter` takes an `LTIImageStage`.

Figure 5 shows the inheritance diagram for these classes. These classes, and classes derived from them, are described in subsequent sections.

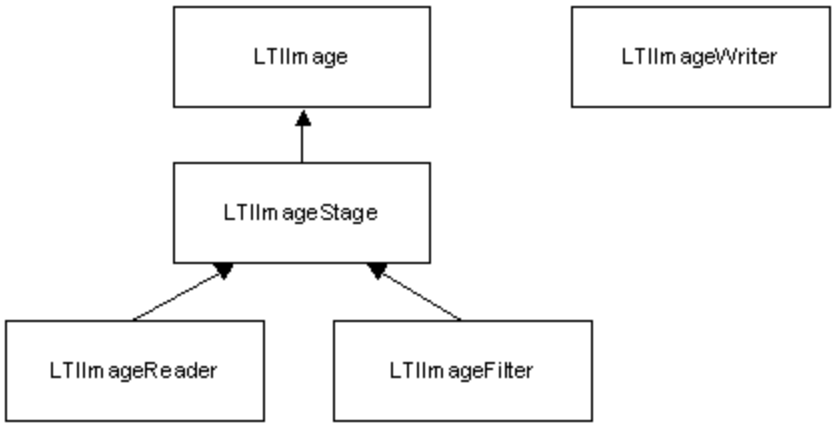


Figure 5: Inheritance

Another class, `LTIImageStageManager`, is a base class for managing a set of `LTIImageStage` objects – it can be thought of as an array of such objects. It used for passing a set of image stages to certain types of mosaic filters to reduce their resource usage. The `LTIImageStageManager` is also used for "wrapping" file formats that can hold multiple images.

Strip-Based Decoding

The MrSID SDK is designed for workflows involving large images. To accommodate large (gigabyte-sized) datasets, the image pipeline framework is designed to process images in advancing horizontal strips. This reduces the amount of memory required to process the image.

To read a scene from an image, the SDK uses the following general workflow:

```
begin read, for given scene
foreach strip in scene, do
     read strip
done
end read
```

These three phases are implemented in the `LTIImageStage` class using the methods `readBegin()`, `readStrip()`, and `readEnd()`. These methods handle all the logic for decomposing the full scene into sequential strips, and they internally call the corresponding pure virtual protected methods `decodeBegin()`, `decodeStrip()`, and `decodeEnd()`, which derived image stages are required to implement as appropriate.

The "connection" between stages is accomplished via an `LTISceneBuffer` object passed into the `read()` call. As each strip is processed in each stage, a subsection of the buffer is used and passed between stages; in general, no intermediate buffering is required as the same buffer is used through the entire pipeline.

The `LTISceneBuffer` class stores the image data in a BSQ format for a specified width and height. Typically the user will supply the allocated memory to be used for the image data; however, the class also supports creation of a new buffer with a "windowed" view into an original buffer. This enables the same region of memory to be used by multiple stages in the pipeline, without the need for excessive data copying.

For decodes of large images, calling `read()` with a single large buffer is clearly inappropriate. In this case, the client at the "end" of the pipeline can choose to explicitly implement the above decode begin/strip/end workflow in a "pull" model, and only need to have one strip worth of image data resident at a time. The `LTIImageWriter` class implements this functionality, so that derived writers can encode large images efficiently.

Scenes

As implied in the previous section, when a `read()` call is made to an image stage, the scene of interest must be specified. A scene, implemented by the `LTIScene` class, has three basic properties:

- upper-left *x*- and *y*-position
- width and height
- magnification (resolution)

The position is expressed in pixel space, relative to the given magnification, with (0,0) and (*w*-1,*h*-1) being the upper-left and lower-right corners of the image, where w and h are the width and height of the image (in pixels).

The scene width and height are also expressed in terms of pixels, but at the magnification of the scene. That is, the width and height values represent the pixel dimensions of the scene as decoded to the output buffer.

The magnification value is a floating point value representing the resolution or scale at which the image is to be read. A magnification of 1.0 corresponds to the full resolution of the image (one to one); a value of 0.5 represents the image at a downsampled view of half the full width and height, and a value of 2.0 represents the image at an upsampled view of twice the full width and height. **Note:** Only powers of two are supported for most image types (although filters for arbitrary resampling are available).

As an example, consider an image that is 625x625 pixels, as shown in Figure 1.

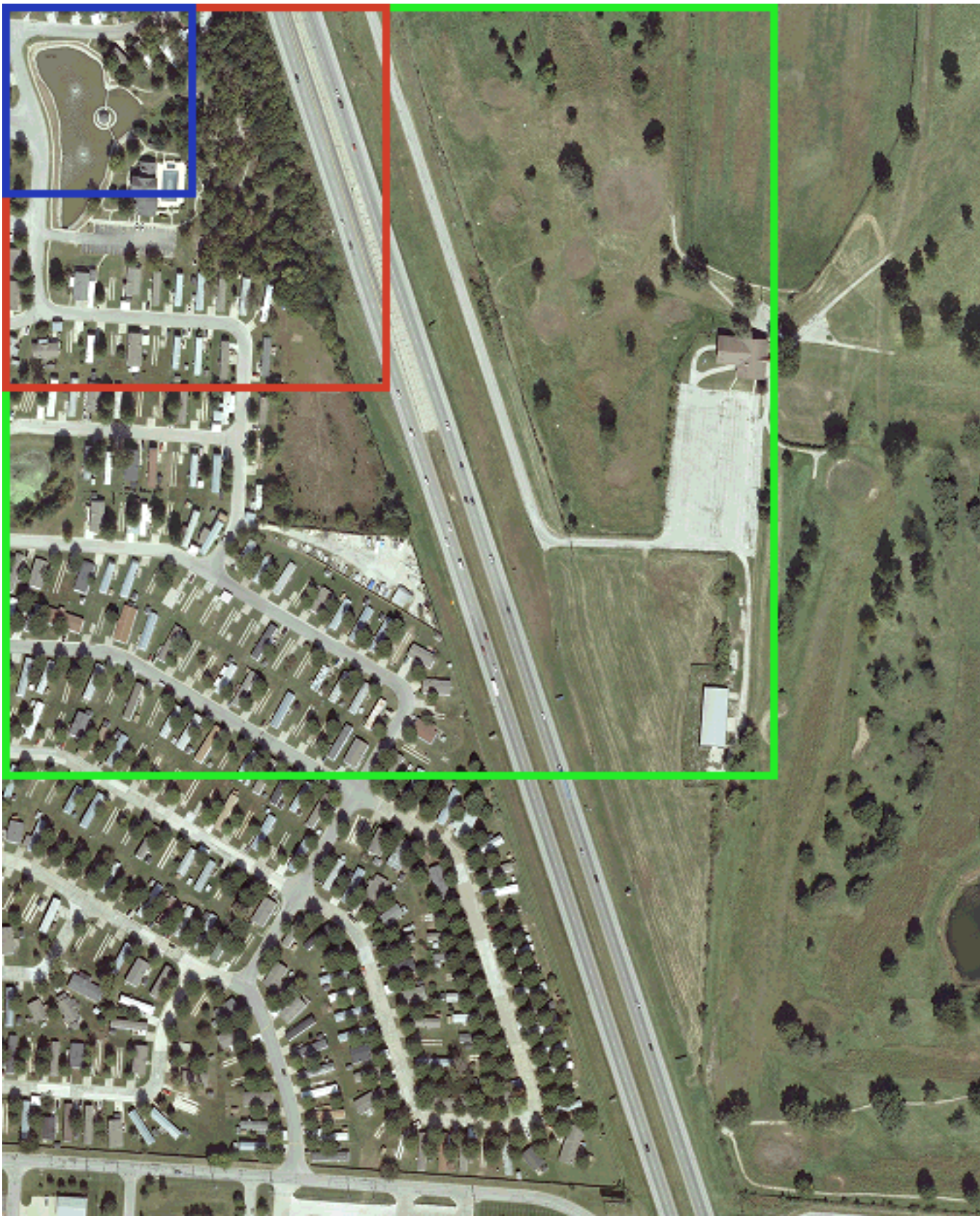


Figure 1: Input scene; (x,y)=(0,0) (w,h)=(625, 625) mag=1.0

The middle rectangle in red corresponds to a full-resolution scene that is 200x200 pixels taken from the upper-left corner of the image. This is shown in Figure 2.

Figure 2: Red Input scene: (x,y)=(0, 0) (w,h)=(200,200) mag=1.0

The inner rectangle in blue corresponds to a scene at twice the resolution (mag=2) that is also 200x200 pixels and also taken from the upper-left corner of the image. This scene is shown in Figure 3. Note that the scene is extracted into a buffer sized for 200x200 pixels, but the corresponding “footprint” in full-resolution space is only 100x100.

Figure 3: Blue Input scene: (x,y)=(0, 0) (w,h)=(200,200) mag=2.0

The outer rectangle in green corresponds to a scene at mag=0.5 that is again 200x200 pixels from the upper-left corner of the image; see Figure 4. Note that the scene is extracted into a buffer sized for 200x200 pixels, but the corresponding footprint in full-resolution space is 400x400.

Figure 4: Green Input scene: (x,y)=(0, 0) (w,h)=(200,200) mag=0.5

At mag=2.0 the scene is effectively sampling from an image of size 1250x1250. Likewise at mag=0.5 the scene is sampling from an image of size 313x313.

NOTE: Not all image formats natively support multiresolution decoding. MrSID and JPEG 2000, being wavelet-based systems, efficiently support decoding at powers-of-two scales. In order to achieve this effect with other formats a resampling filter must be applied.

Multi-Threading

If you run the SDK on a machine with a multi-core processor or on a machine with multiple processors, the SDK creates multiple threads to run decoding operations more quickly. By default, the SDK uses the maximum number of threads available for optimal performance. The maximum number of threads equals the number of cores in your processor.

IMPORTANT: If you do not want to use the maximum number of threads, you must set the number of worker threads that you want to use. Use the `setMaxWorkerThreads()` function defined in the `MrSIDImageReader` class to set the number of threads. For more information, see the Reference Manual.

Threaded Building Blocks

The MrSID SDK uses Intel Threaded Building Blocks (TBB) to manage threads. Programs that you create with the SDK must include the TBB library to take advantage of multi-threading. The TBB library is stored in the following location:

`<SDK Directory>\lib\tbb.dll`

To use additional TBB features in your program, download TBB header files and refer to the TBB documentation at the following URL:

https://www.threadingbuildingblocks.org/

Other Design Considerations

The C++ classes and functions that make up the MrSID SDK follow a few other general principles and conventions. They are explained here to motivate their usage. For other, less central principles, see "Coding Conventions" on page 59.

Status Codes

Many of the SDK member functions return status codes to report success or failure conditions, instead of other mechanisms such as throwing exceptions. The status codes are represented using the `LT_STATUS` datatype (typedef'd to an unsigned integer). Functions that return status codes must be checked for success or failure.

Initializations

Heavy-weight objects requiring nontrivial work in their constructors provide an initialize() member function that must be called immediately after invoking the object's constructor. While this requires extra code for the developer, it provides a means for returning status codes back to the caller without relying on constructor-thrown exceptions.

Creator Deletes Rule

The SDK classes generally follow the "creator deletes" rule. That is, the object that allocates a new object on the heap is responsible for deleting it as well. (The documentation notes specifically where this rule does not hold and ownership is to be passed across a call-site.) In some cases reference counting, via the `RC<>` class, is used to address this issue.

"No Magic" Rule

The functions in the SDK tend not to provide features or functionality that attempt to silently "guess" at default values for complex situations or otherwise "just do the right thing." While such guesses are often correct and occasionally easier for the developer, they are just as often incorrect or serve only to mask deeper errors or workflow inefficiencies. For example, the various decode methods can return pixels only in the colorspace and datatype of the input image stage: no mechanism for implicit conversion is available, e.g. via configuration of the output buffer. In this case, a datatype filter and a colorspace filter must be explicitly built into the image pipeline.

Reference Counting

The SDK uses reference counting to manage the lifespan of objects that comprise image pipelines. This is similar in spirit to "smart" or "auto" pointers. The static member function `create()` of a class should be used to create a new instance of that class. Similarly, the (non-static) member function `release()` should be used when you are done with object – for example, when the pointer to the object goes out of scope. The `retain()` function should be used to hold onto an object, i.e. increment the reference count; if you created the object, however, do not call `retain()`. When the last reference calls `release()`, the object will delete itself.

The Support Classes

We begin with the set of types, classes and functions that underlie the entire SDK. These do not provide any specific imaging functionality, but they are used as primitives for all the other classes that follow in subsequent chapters.

The names of the types, classes, etc. described in this chapter are generally prefixed with the letters “LT”. The corresponding header files can be found in the `include` directory.

Examples of the use of many of these classes can be found in the `examples` directory and in the Reference Manual at `doc/ReferenceManual/index.html`.

Preprocessor Constants and Basic Typedefs

The C header file `lt_base.h` should be included by all source files using the MrSID SDK. It includes four other C headers (`lt_define.h`, `lt_platform.h`, `lt_status.h`, and `lt_types.h`), which contain definitions for the following:

- miscellaneous preprocessor constants and macros used throughout the SDK
- preprocessor constants that define the compiler, OS, etc. used by the current target environment
- the `LT_STATUS` type used throughout the SDK for success/failure return values, and some common status codes
- the typedefs used by the SDK for the primitive/integral datatypes

Status Strings

The file `lt_utilStatusStrings.h` declares several functions useful for mapping `LT_STATUS` values into text strings that indicate the nature of the error condition encountered.

Most applications will want to use either the `getRawStatusString()` or `getLastStatusString()` function for retrieving error strings.

The textual error strings for the integer-values `LT_STATUS` codes can be found by searching in the files `./include/*status.h`. This may be particularly helpful during the development (debugging) process.

For details on using the error string system, consult the Reference Manual at `doc/ReferenceManual/index.html`.

The LTIFileSpec Class

For portability between Unix and Windows systems and ease of use in internationalized environments, the SDK uses the class `LTFileSpec` to represent file names and paths.

In all cases where the SDK requires use of a file name, for example as a function parameter, an `LTFileSpec` will be used. It is the application’s responsibility to translate filenames from the app-level representation, for example a `char*`, into an `LTFileSpec`.

The `LTFileSpec` class provides the following features:

- creation of an `LTFileSpec` from a directory path and file name, as represented in a variety of formats (multibyte, UTF-8, `wchar_t`, etc.)
- conversion from `LTFileSpec` to a variety of formats
- Unix-style `dirname()` and `basename()`
- file name extension extraction and replacement
- differentiation between relative and absolute paths

Streams

The MrSID SDK provides an abstract stream class, `LTIOStreamInf`, which provides a Unix stdio-like interface for working with data in files or file-like objects. Rather than relying on the application to provide a file name or a `FILE*`, many of the SDK functions are designed to operate on `LTIOStreamInf` objects instead. This enables more portable and extensible interfaces.

The `LTIOStreamInf` interface requires only a small set of primitives, including:

- `open()` and `close()`
- `read()` and `write()`, using byte arrays
- `seek()` and `tell()`, using 64-bit offsets
- `isOpen()` and `isEOF()`

(For details on the precise semantics of these primitives, see the Reference Manual at `doc/ReferenceManual/index.html` and "Notes on Streams" on page 65 of this documentation.)

A number of useful stream types, including files, in-memory buffers, sockets, and buffered streams, can be implemented using the stream interface. The SDK provides the following stream implementations:

- `LTIOFileStream` – OS-native file support (2GB or less only)
- `LTIOMemStream` – in-memory, read-only buffer of fixed size
- `LTIODynamicMemStream` – in-memory read/write buffer, which grows as required
- `LTIOBufferedStream` – nontrivial buffering for an underlying `LTIOStreamInf`
- `LTIOCallbackStream` – stream implemented via user-supplied callbacks for basic operations (read, write, open etc.)

Examples

The Reference Manual contains several examples of working with streams, including code showing how to perform simple reads and writes and how to derive your own simple stream from `LTIOStreamInf`.

The SDK Base Classes

The image pipeline reader, writer, and filter classes that support the MrSID SDK's image pipeline system all rely on a set of base or "core" classes. This chapter introduces these classes, which represent such constructs as pixels, geographic positions, scenes, buffers, and so on.

The names of the types, classes, etc., described in this chapter are generally prefixed with the letters "LTI". The corresponding header files can be found in the `include` directory.

For detailed information on these classes, please refer to the Reference Manual at `doc/ReferenceManual/index.html`. Examples of the use of many of these classes can be found in the Reference Manual and in the examples directory.

Subsequent chapters of this manual introduce the actual readers, writers, and filters provided with the SDK, including the MrSID and JPEG 2000 encode and decode classes.

Base Enums

The `lti_types.h` header contains a number of enums used throughout the SDK, including `LTIColorSpace` and `LTIColor`, for indicating pixel color spaces, e.g. RGB or multispectral, and `LTIDataTypes`, for indicating pixel datatypes, e.g. 8-bit unsigned integers or 32-bit floats.

Base Classes

A number of lightweight classes are used to represent primitive objects in support of the imaging framework.

The `LTISample` class represents a single sample of a pixel; the properties of a sample include datatype, color, and an optional value. The `LTIPixel` class represents a set of samples; the properties of a pixel include the samples themselves and a colorspace. The `LTIPixelLookupTable` class is used to represent the concept of a set of pixels in a color lookup table.

The `LTIGeoCoord` class represents simple geographic position information, analogous to the conventional AUX files or world files used in many GIS systems. The properties of this class include the projection system (WKT), the upper-left x- and y-position, the *x*- and *y*-resolution, and the two rotation terms. Every `LTIImage` has associated `LTIGeoCoord` information.

The SDK uses delegates (see delegates in Glossary) as a means of proving the functionality of callback functions in a more natural C++ style. The two most common delegate classes, `LTIInterruptDelegate` and `LTIProgressDelegate`, provide mechanisms for applications performing (potentially long-running) `read()` operations to make out-of-band requests to abort the operation and to receive out-of-band notifications of percent-complete. Examples of delegate usage can be found in the Reference Manual at `doc/ReferenceManual/index.html`.

Finally, the `LTIUtils` class contains a number of static methods generally useful for working with the SDK. These include conversion between dynamic range representations, conversion between scale/magnification representations, colorspace information, SDK version information, etc.

The Image Classes

As described in the previous chapter, a hierarchy of several classes is used to represent images, image stages, and the image pipeline. See Figure 1.

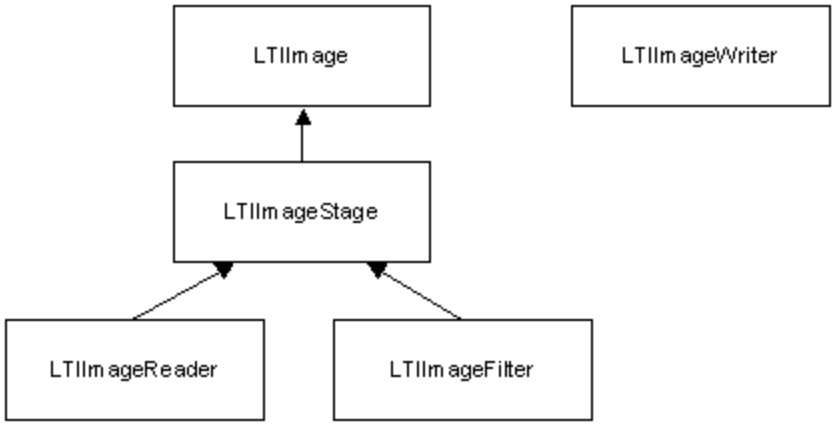


Figure 1: Inheritance

The `LTIImage` class represents the properties of an image, including:

- width and height
- pixel type (number of bands, colorspace, datatype)
- background and nodata pixel values
- dynamic range of the samples
- simple geographic coordinates
- supported magnification range
- metadata

The `LTIImageStage` class extends `LTIImage` by adding:

- strip-based read functions
- progress and interrupt delegates

The `LTIImageReader` and `LTIImageFilter` classes extend `LTIImageStage` in order to more closely reflect their positions and operations in the image pipeline. These two classes are discussed in subsequent chapters.

While the `LTIImageWriter` class does not inherit from any of the other `LTIImage` classes; it replicates some of the same functionality (such as strip-based operations and delegate control) so that it can be used as a client of an image pipeline. The `LTIGeoFileImageWriter` class is derived from `LTIImageWriter` to aid in implemented image writers. `LTIGeoFileImageWriter` extends `LTIImageWriter` to provide the ability to output to a specific form (stream or file) and optionally generate world files.

The `LTIImageStageManager` is a base class for managing a set of `LTIImageStage` objects – it can be thought of as an array of such objects. It used for passing a set of image stages to an

LTIImageMosaicFilter and enabling the mosaic filter to reduce its resource usage. The LTIImageStageManager is also used for “wrapping” file formats that can hold multiple images.

The Raw Readers and Writers

Because they are the foundation of many other readers and writers, as well as being useful for development and debugging, the SDK base classes include “raw” reader and writer classes, LTIRawImageReader and LTIRawImageWriter.

The two raw classes read and write image data directly to binary files (or streams) with no header information. Both support BIP/BIL/BSQ output formats and provide for big/little/host byte ordering (see Endianness in the "Glossary" on page 95), plus the usual LTIImage properties of background and nodata color, geographic position, metadata, etc.

The LTIBBBImageReader and LTIBBBImageWriter classes extend the raw reader and writer classes by providing the ability to read and write “BBB” headers to accompany the raw image data. For information on the supported header syntax see "Notes on BBB Files" on page 68.

Scene and Buffer Management

The two parameters to the LTIImageStage::read() function are an LTIScene object and an LTISceneBuffer object. These together specify the region of the image to be decoded and the buffer to put the data into.

The properties of an LTIScene are the upper-left point, dimensions, and magnification of the region to be decoded. (For more information see "Scenes" starting on page 14.)

As a scene is a largely read-only embodiment of its properties, the LTINavigator class extends LTIScene by adding functions to control the scene, including:

- move scene to a given location or by a given amount
- zoom to a given magnification or by a given amount
- determine the “best fit” scene of the image to a given size (within the power-of-2 resolution constraints)
- set the scene based on upper-left, lower-right, or center points
- set the scene using geospatial position, as opposed to pixel coordinates

By using the navigator class, issues of proper rounding control and keeping the scene within the image can be managed transparently to the user.

The LTISceneBuffer class is used to represent the in-memory buffer that will hold the data produced by the read() call. The buffer can be supplied directly by the user, allocated internally by the class, or inherited from another scene buffer. Depending on how it's configured, it can also represent only a subset of the bands in the image, so that only the bands that are asked for are decoded.

While the extents of the “visible” or “exposed” window of the buffer must be at least as large as the dimensions of the scene being used for the read operation, the actual buffer might be considerably larger; this allows for read requests to target specific regions of the buffer, as is required for strip-based decoding workflows.

As an example, consider Figure 1, which shows an `LTISceneBuffer` object that has a total size of 200x100 (20,000 pixels) and a window size of 160x60 (9,600 pixels). This buffer can be used to "insert" a scene onto the total buffer, using (40,20) as the upper-left position. (This upper-left position corresponds to byte 200*20+40 of the buffer, assuming an 8-bit, 1-banded image.)

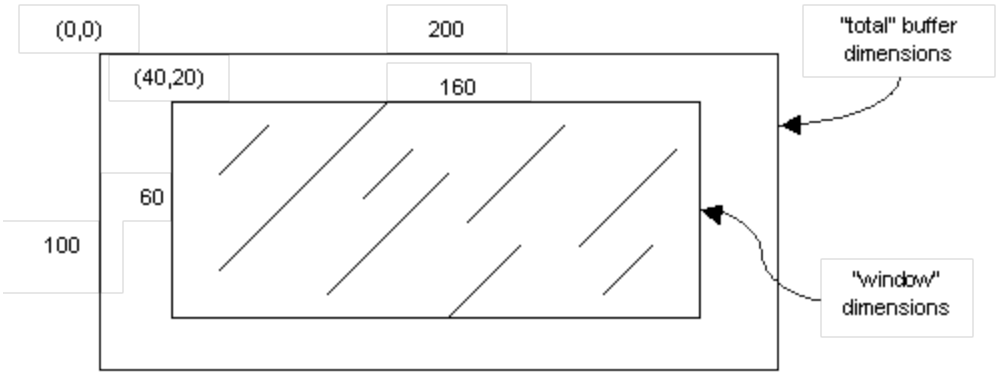


Figure 1: LTISceneBuffer

The data in the scene buffer is internally stored in a packed BSQ format. By supporting only one data layout, the complexity of the various image stages and the need for additional memory-to-memory copies is reduced. However, when constructing decoder pipelines the client often needs the data organized in some other fashion, so the scene buffer class supports these "import/export" features:

- copy data into the buffer from memory (in packed BSQ or BIP forms)
- copy data into the buffer from a stream (in packed BSQ or BIP form)
- copy data from the buffer to memory (in packed BSQ or BIP form)
- copy data from the buffer to a stream (in packed BSQ or BIP form)
- copy data from the buffer to memory, with full control over layout and padding (such as padding of rows to satisfy alignment constraints)

The copying is performed relative to the visible window of the buffer, not the total buffer. For the BSQ memory functions, the data may be held as one large BSQ buffer or as one individual buffer for each band.

See code samples for demonstrations of how to manage scenes and scene buffers.

Concrete Image Filters and Writers

The MrSID SDK uses two types of classes – image filters and image writers – for transforming and encoding images.

Image Filters

The `LTIImageFilter` class is derived from the `LTIImageStage` class and is used as the base class for implementing specific (and generally simple) image transformations. The following image readers are supplied with this SDK:

- `LTIAddAlphaFilter`: This class adds an alpha band to an image that does not have one, based on the image's transparency. This is useful for MG4 encoding pipelines.
- `LTIBandSelectFilter`: This class creates a single image which is a composition of selected bands from a set of source images. It can be used, among other things, to remap the color bands of a single image (formerly `LTISampleMapFilter`) or to merge several greyscale images into a single image (formerly `LTIColorCompositeFilter`).
- `LTIColorTransformer`: changes the colorspace of an image. The supported transforms include transforming to and from RGB, grayscale, and CMYK.
- `LTICropFilter`: crops the image to a smaller width and height. Note that this is a reduction in image size; this is a true crop, so some data will be lost.
- `LTIDynamicRangeFilter`: adjusts the sample values of the image to fit the given dynamic range by scaling the values by appropriate amounts. This is useful as a preprocessing step to displaying datasets that don't use the full precision of their datatypes, for example 7- or 12-bit data.
- `LTIEmbeddedImage`: creates a new image of the given size, containing the input image "embedded" within it. This class is used to make an image "larger" by increasing its width and height without actually stretching the image itself, for example to place an image on a larger "canvas" so as to provide it with a larger background. (This class is used by the `LTIMosaicFilter` class to simplify certain computations by making all the input images map to the same underlying grid shape and size.)
- `LTIMosaicFilter`: creates a single mosaicked image from a set of input images. The set of input images are all assumed to be in the same coordinate space.
- `LTIMultiresFilter`: extends the magnification range of an image to allow decodes at different resolutions than the image stage would normally allow. Note that this class is not the same as the `LTIStaticZoomFilter` class, which scales the magnification statically for the pipeline when initially constructed; this class allows for the zoom level to be extended for an individual decode operation.
- `LTIStaticZoomFilter`: magnifies the image by a fixed amount. In effect this simply changes the width and height of the image statically, i.e. for the life of the pipeline.
- `LTITranslationFilter`: translates (moves) the geo coordinates of the image.
- `LTIViewerImageFilter`: performs certain datatype and colorspace transforms on an image to make it more readily displayable. This class wraps the `LTIDataTypeTransformer`, `LTIColorTransformer`, and `LTIDynamicRangeFilter` classes in order to transform the input image into an 8-bit datatype with colorspace grayscale or RGB, as this is the format required by most display engines.

- `LTIWatermarkFilter`: inserts a watermark image onto the current image stage at a given position. The "watermark" is represented as an input image stage and must have the same pixel properties (colorspace, datatype, etc.) as the image it is to be inserted into.

Examples

The Reference Manual at `doc/ReferenceManual/index.html` contains several examples of working with image filters, including code showing how to insert a filter into an image pipeline and how to derive your own simple filter from `LTIImageFilter`.

Image Writers

The `LTIImageWriter` class is the base class for implementing specific file format writers. While not derived from any of the other `LTIImage` classes, it is nonetheless designed to act as an “end” of the image pipeline in order to support writing out an image in a stripwise fashion. The following image writers are supplied with this SDK:

- `BBBImageWriter` – writes the BIP/BIL/BSQ raw format
- `BMPImageWriter` – writes the Windows BMP format
- `GeoTIFFImageWriter` – writes the GeoTIFF format
- `JpegImageWriter` – writes the JPEG format (**Note:** "original" JPEG, not JPEG 2000)
- `TIFFImageWriter` – writes the TIFF format

Examples

The Reference Manual at `doc/ReferenceManual/index.html` contains several examples of working with image writers, including code showing how to write a scene with an `LTIImageWriter` and how to derive your own simple writer from `LTIImageWriter`.

MrSID Support

MrSID offers a solution to the challenges and a toolkit for the opportunities digital imaging presents. Because encoding to MrSID results in a lossless, single-source image, MrSID yields better storage economy, more efficient transfer of data, and improved workflow. A single, highly compressed, high-quality image serves as the source for any data requested from that image, optimized for instant viewing on any device.

This MrSID technology was acquired by LizardTech in its original form from Los Alamos National Laboratories (LANL), where it was developed under the aegis of the U.S. government for storing fingerprints for the FBI. LizardTech was formed to commercialize and further develop the technology, and MrSID quickly became the standard for viewing, distributing and storing raster imagery in geographic information system (GIS) environments. MrSID is the technology and image format that all major geospatial software applications support today.

MG2, MG3 and MG4

The most current version of the MrSID technology and file format is known as MrSID Generation 4 (MG4). Its predecessors were known as MG3 and MG2.

MG2 was the first commercial version of the compression technology and file format originally developed at Los Alamos National Laboratory. MG2 supports selective decoding (by scale or region), but unlike MG3 and MG4 it does not support optimization workflows, lossless encoding, or composite images (although an MG2 image can be a tile within an MG3 composite). Additionally, the quality of MG2 compression is in many cases somewhat inferior to that of MG3 and MG4.

MG3 introduced lossless encoding, optimization (by scale, quality or region) and selective decoding (by scale, quality or region). In addition, MG3 introduced the ability to store multiple images in a single file known as a “composite” MG3 file. This allows for large MrSID images to be “updated” when new data for certain areas becomes available. The “tiles” that make up an MG3 composite are complete and valid images unto themselves and may be in the MG3 or MG2 format (MG4 composites can contain only MG4 tiles). The MG3 reader (decoder) class allows the user to inspect the type and geographic extent of each tile in a composite image, and any subset of the tiles may be specified when opening the composite image for decoding. Internally, the SDK handles the logic of mosaicking the specified tiles together.

The key new features of MG4 are:

- multispectral (and hyperspectral) support
- alpha band support
- improved composite mosaicking (embedded overviews, multiresolution support)

Support for multispectral imagery in MG4 enables users to compress 4-band NAIP data, 8-band Landsat data or even 224-band AVIRIS data, losslessly or with LizardTech's usual high-quality visually lossless compression. The MG4 format also adds support for alpha bands, enabling users with shapefiles defining the boundaries of their image data to perform more complex mosaicking operations than ever before. Embedded overviews mean that decoding MG4 composites takes less time. Finally, MG4 composites can contain source tiles of differing resolutions.

For more information see "Key Features of MrSID" on page 28.

Differences Among the MG2, MG3, and MG4 Formats

As the MrSID technology has evolved over the years, the range of capabilities supported has evolved as well. Specifically:

- MG2 does not support lossless compression
- MG2 does not support optimization
- MG2 does not support composite images
- MG3 does not support 32-bit floating point data
- Only MG4 supports signed integer data
- Only MG4 supports alpha masking
- Only MG4 offers support for multispectral and hyperspectral imagery
- While MG3 composite mosaics can contain MG2 files, MG4 composite mosaics must be made up of only MG4 files.
- While both MG3 and MG4 allow creating flat mosaics using source tiles of different resolutions, only MG4 allows different resolutions in source tiles for composite mosaics.

While some applications may only write newer versions of the MrSID format, all applications that read MrSID files will always continue to support all versions of the format. These considerations are important to keep in mind, since there are so many older MG2 and MG3 files kept in long-term archives.

Key Features of MrSID

In this section we describe in more detail some of the features and capabilities of the MrSID technology for raster image data.

NOTE: The MrSID format also supports LiDAR data, but a separate set of tools and libraries is used in supporting LiDAR data in the MrSID format, and separate documentation is available in your installation for integrating support for LiDAR-encoded MrSID files.

Datatypes and Formats

The MrSID technology is agnostic with respect to the input file format, as long as the input pixel data meets certain datatype requirements. This means that MrSID files can be generated from a variety of data sources including GeoTIFF, Imagine, and ECW.

The MrSID technology supports most data types used in geospatial raster imagery today: up to 16 bits per sample (signed or unsigned). MG2 and MG4 also support floating point data. Raster image data is almost always represented using unsigned integers. Digital elevation models and file formats like DTED, however, often use a signed integer representation, and so to support situations where our users want to compress these sorts of datasets, or perhaps use terrain models as base layers for their visualizations, MrSID supports signed integer data of up to 16 bits.

The MrSID technology also supports 1-band grayscale, 3-band RGB, and 1- to 255-band multispectral or hyperspectral imagery.

Image Quality

As discussed above, MrSID technology offers excellent image quality for a given file size target.

- Numerically lossless: This level of compression typically yields a 2:1 compression ratio, for a 50% reduction in storage space. Lossless compression should be used when it is critical that all bits of the original image be preserved. This is the case for archival storage, as well as for uncommon workflows where no possible loss of precision is ever acceptable. You may also wish to use lossless compression when you are generating a “master” image from which other derivative images will be made, as through the MrSID optimization process described below.
- Visually lossless: This level of compression is typically 20:1 for RGB and 10:1 for grayscale imagery. This is the most common level of compression quality used, as it preserves the appearance of the imagery for most workflows, including use of your imagery as a background layer and for many forms of visual analysis and exploitation.
- Lossy: Beyond 20:1, image degradation and artifacts can appear, although often not too significantly until ratios of 40:1 or 50:1. Such lossy quality may be acceptable when the imagery is used only as a background layer for appearance or when the image quality is less important than the storage size or speed, such as for informal visual inspections.

Performance

When considering performance, we usually consider the cost of running some process, such as compression or decompression, in terms of memory usage, CPU usage, and I/O bandwidth. The MrSID technology is designed with these concerns in mind.

Compression

When dealing with very large images, many image processing algorithms first partition the image into tiles and then process each tile independently. This allows the computation to proceed without slowing down due to excessive paging of memory to disk. However, especially in the case of compression algorithms, such tiling can introduce artifacts in the resulting image because the algorithms cannot efficiently process cross-tile regions. MrSID technology is specifically designed to process imagery whose size is larger than the amount of RAM available on the machine without resorting to tiling schemes and therefore without introducing any tiling artifacts.

Decompression

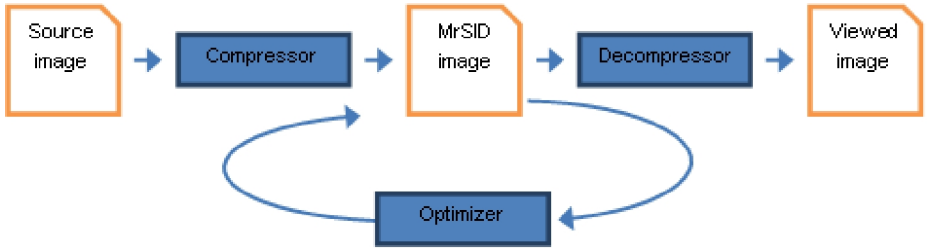
When decompressing imagery, the most common use case is for viewing, which means extracting out scenes – only some subsets or regions of the image are needed at any one time. With the multiresolution support inherent in the MrSID format, the viewing application may first decide the resolution level needed to display the scene at some physical screen resolution and then extract only the resolution levels needed; this significantly improves disk I/O time and lowers the amount of imagery the CPU must process. Additionally, the viewer need only request those portions of the file that correspond to the region of interest; the entire image (at the given level) need not be processed, again saving I/O bandwidth and processing time.

When decompressing the entire image is required, the performance of the decompression step is roughly comparable to that of the earlier compression step: again, MrSID technology is designed to run within reasonable amounts of RAM, even for large datasets. If lossy compression was used, the decompression will be somewhat faster since there is correspondingly less data being read in and processed.

Optimization

For most users, the typical image compression workflow consists of a compression followed by one or more decompressions, either for viewing (small decodes) or for bringing the image back into some other format for some other tool or purpose (large decodes), as shown in the top line of Figure 1. In many cases, however, the need for the large decode step can be reduced.

Once an image is in the MrSID format, a new MrSID file can be generated from it without resorting to a decode followed by a re-encode – this means you can generate derivative products from a single source, as shown in the bottom of Figure 1. This is referred to as “optimizing” the image.



For example, a data provider might create and archive a lossless MrSID file to use as a “master”, and then as customer requests come in, that master copy can be used to quickly generate new MrSID files that fit a variety of needs:

- a MrSID file with a lower baseline resolution – for example, resolution levels can be removed if only one foot per pixel resolution is needed from a six-inch resolution image
- a MrSID file requiring less storage space – for example, 20:1 compression can be used to fit the image onto a CD or DVD
- a MrSID file containing less area – for example, a scene containing only a certain neighborhood can be extracted from an image covering a whole city

Again, to meet these three different requirements (or perhaps some combination of them) only one fast step is required to generate a new MrSID file from the original MrSID file. There is no need to decode the entire image first.

Metadata

Because MrSID is a geospatial data format, MrSID files also include geospatial referencing information such as the coordinate reference system (CRS), the geographic extents (corner points) of the image, and the pixel resolution.

This metadata is an inherent part of the MrSID file format and is based on the well-known GeoTIFF tag scheme. When performing a reprojection operation or one of the optimization steps described above, the metadata is updated to reflect the properties of the derived image: when performing scale reduction, for example, the resolution metadata is updated accordingly.

MrSID metadata also is used to record what operations may have been performed on your dataset. For example, you can determine if the file you have still corresponds to the lossless original data or if it has been modified in some way.

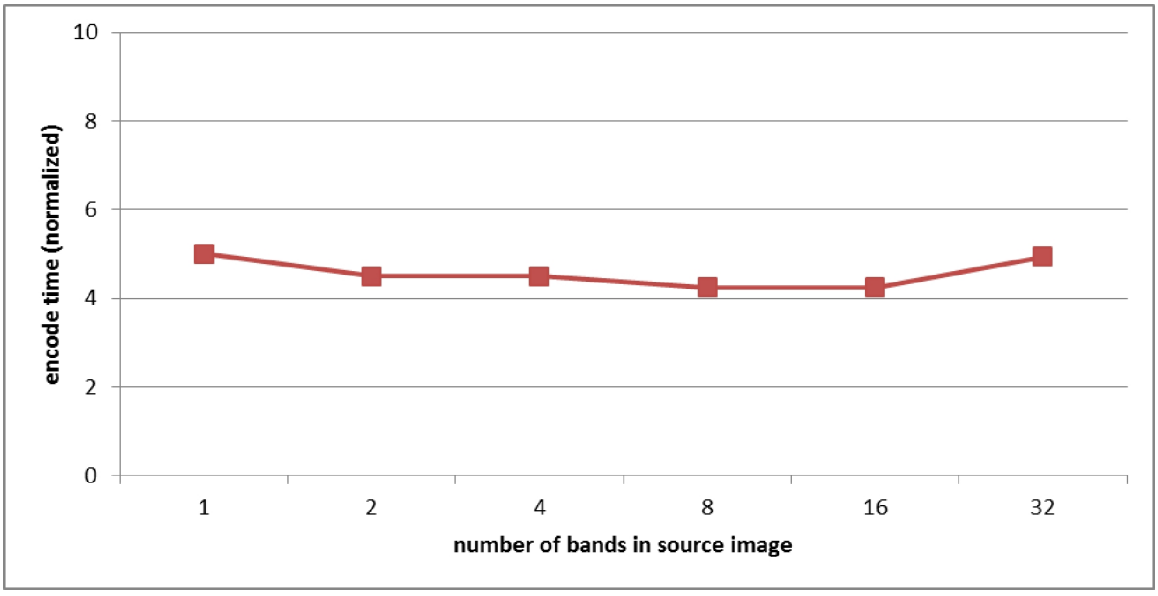
This native geographic metadata support allows you use a third-party application to import your MrSID imagery for use as a base map with other georeferenced datasets you might have.

Multispectral Support

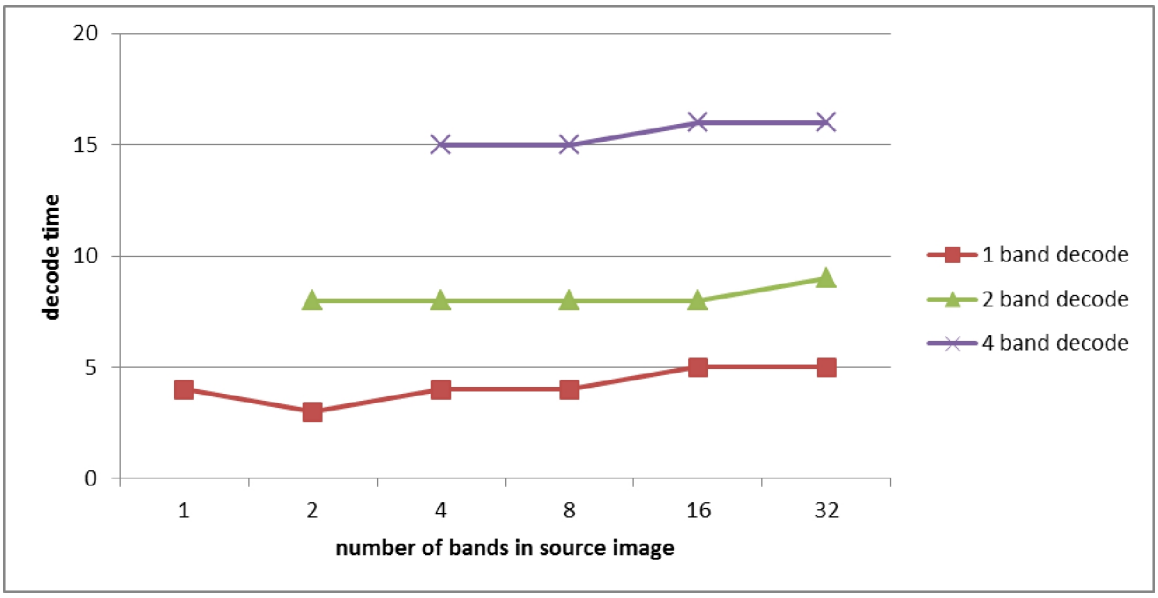
For many years, some types of geospatial data have included more than just the usual three color (RGB) bands. Only recently, however, have these kinds of multispectral datasets started to be widely available to GIS users. For example, in 2011, USDA's NAIP program plans to collect data for 15 states which will contain the red, green, and blue (RGB) bands plus a fourth infrared (IR) band. DigitalGlobe's recently launched WorldView 2 satellite records RGB plus five additional bands: a yellow band, two IR bands, and two "coastal" bands. NASA's MODIS now collects 36 bands. Other remote sensing platforms are now collecting hyperspectral datasets, typically one hundred or more narrow bands. All these additional bands are chosen for their abilities to improve feature classification and extraction by providing more discriminating information in areas such as vegetation cover, shallow-water bathymetry, and man-made features.

To support these new, richer datasets, the MG4 format can compress images with up to 255 bands. The same key features are still available: lossless and lossy encoding, multiple resolution levels, and selective decoding.

As more data is being encoded and decoded, of course, more time will be required. The figure below shows the relative performance of encoding 5Kx5K pixel images with 1, 2, 4, 8, 16, and 32 bands of data: the time required scales linearly, when normalized to the number of bands. That is, if it takes 1 minute to encode a 1-banded image, it will take 10 minutes to encode an 10-banded image of the same width and height.



The time required to decode imagery with varying numbers of bands scales similarly. However, many users of multispectral imagery only view one or perhaps three of the bands at a time, mapping the bands into the familiar grayscale or RGB space. In the same way that the MrSID algorithms will perform selective decompression for viewing only the scene of interest, they will also decode only the bands of interest. The figure below shows the relative time it takes to decode 1-, 2-, and 4-band scenes from images with 1, 2, 4, 8, 16, and 32 bands of data: the time required does not depend on the number of bands. More concretely, if it takes 1 minute to extract a single band from 1-banded image, it will take only 1 minute to extract a single band from a 10-banded image of the same width/height.



Alpha Bands

In previous versions of the MrSID format, nodata regions were indicated by a sentinel pixel value, typically black. When mosaicking tiles together, nodata regions would be used to indicate how to "combine" one image on top of another. Users who have worked with MrSID images in the past, however, may have noticed a problem with this. A black nodata pixel, represented by (0,0,0) might be slightly changed when subjected to lossy compression. The value (0,0,0) might change to (1,0,2) or (0,2,0) – by itself visually indistinguishable from black, but in a mosaicking context it is no longer the nodata sentinel value and so in the worst case this might have caused "speckling" artifacts to appear.

The MG4 format uses an alpha band instead of a single nodata pixel value to indicate which areas of the image do not have valid data. When encoding existing imagery, users indicate which pixel value corresponds to nodata and a mask is created corresponding to those values. Subsequent mosaicking operations then use that mask to determine how to combine tiles. Lossy compression no longer affects this process, because while the putative nodata pixels might get slightly changed, the alpha mask is always kept lossless and is always honored by the decoders.

The alpha band is treated just like the other bands in the image, such as the RGB bands, except that it is never subjected to any lossy compression. Because the alpha band contains relatively simple sequences of data – very long runs of ones or of zeros – it compresses losslessly extremely well and little or no overhead will be noticeable in your MrSID files.

Tiling and Composites

Many of our customers have a single MrSID file which covers a large geographic region. With the ability of the MrSID technology to composite multiple MrSID files together, you can have one MrSID file that is made up internally of dozens of MrSID files serving as image tiles.

As new MrSID tiles are acquired – such as from a more recent flight, perhaps with higher accuracy data – these tiles can be easily added to the existing MrSID composite image. Because only MrSID files are involved, this process does not require any decompression or compression steps and so can

be done very quickly. When displaying the data, the new tiles' data will correctly layer on top of the older data. Additionally, the overview tile is automatically updated to account for the new tiles.

There are several important differences between MG3 composites and MG4 composites.

- In MG3 format you can also combine MG2 files in your composite image, whereas MG4 composites restrict input to MG4 files.
- MG4 composite images are created with a special overview tile, so even files consisting of hundreds of tiles can be quickly viewed at lower resolutions which span multiple tiles.
- MG4 composite images can be composed of files of multiple resolutions.

MrSID Readers

The MrSID SDK provides a class for reading MrSID imagery. The `MrSIDImageReader` class supports reading from any MrSID image, MG2, MG3 or MG4. (If the image is a composite, all tiles in the image will be used.) This class provides an interface for querying the tile contents of composite images.

This class supports both stream inputs as well as filenames. The ability to control whether world files are honored is also provided.

The class has initialization parameters to control resource usage by the object. The "memory usage" parameter allows the user to choose to use more memory to hold the MrSID data in memory, for improved performance at the cost of higher memory usage. By limiting the memory usage, certain decode operations may be slower but more images may be opened simultaneously.

Similarly, the "stream usage" parameter allows the user to choose to keep the underlying file handle (or stream) always open or open only when required by the decoder. The always-open mode may reduce file I/O overhead, but at the cost of potentially critical resources (file handles) on some systems.

By default, this class uses multiple threads to perform decoding operations. The class includes a `setMaxWorkerThreads()` function that you can use to modify the number of threads created. For more information, see "Multi-Threading" on page 17.

JPEG 2000 Support

JPEG 2000 is the state-of-the-art successor to the popular JPEG compression standard. The original JPEG was based on the discrete cosine transform (DCT) and Huffman encoding and is not suitable for current workflows relying on high quality imaging, lossless encoding, or very large images.

Algorithmically JPEG 2000 and MrSID (MG3 and MG4) are very similar. Both are wavelet-based systems which use arithmetic encoding to support lossless compression, image optimization workflows, and selective decoding by quality, resolution, and region. Details of the algorithms and the underlying file formats are, however, quite different; MrSID and JPEG 2000 data and files are not interchangeable.

As JPEG 2000 is an international standard (ISO/IEC 15444), an increasing variety of products can create, read, and work with JPEG 2000 imagery. By using the MrSID SDK to encode and decode your imagery, your applications will be more valuable in open standards environments.

The MrSID SDK supports reading Geography Markup Language (GML) in JPEG 2000 files. When you decode a JPEG 2000 image that contains GML, you can extract the coordinate reference system stored in GML.

NOTE: JPEG 2000 is an extremely complex framework for managing compressed imagery, with a correspondingly complex set of encoding options. The choice of encoding parameters you should use will depend greatly on your performance requirements and anticipated workflows. See LizardTech's developer website (http://developer.lizardtech.com/) for current information.

The JPEG 2000 Reader

The `J2KImageReader` class is used to read JPEG 2000 files or streams.

This class supports a number of query functions to find information about how the image was encoded with JPEG 2000, as well as some decode-time functions to control how the image will be decoded. Please consult the Reference Manual at `doc/ReferenceManual/index.html` for details.

NOTE: The `J2KimageReader` class cannot be used to read "old-style" JPEG files.

NITF Support

The National Imagery Transmission Format (NITF) standard was created by the Department of Defense as a means of formatting digital imagery and imagery-related products and exchanging them among members of the intelligence community, the Department of Defense (DOD), and other government departments and agencies. NITF was created partly because government agencies needed a single common image representation that supported certain metadata features and workflows. While it is called an image format, NITF is more precisely described as a file format that wraps one or more image files and their corresponding metadata.

The DSDK supports reading NITF files and includes the following features:

1. support for image segments with JPEG and JPEG 2000 compression
2. some support for files containing multiple image segments
3. some support for TRE data

NOTE: Text, label, and symbol segment types are not supported.

Important Notes On NITF Compliance in the MrSID SDK

This version of the MrSID SDK follows the following NITF standards:

1. versions 2.1 (MIL-STD-2500C, Draft April 2004) and 2.0 of the NITF file format for reading
2. the BIIF Profile (BPJ2K01.00, Draft January 2004) including the J2KLRA TRE

NITF v1.1. files are not supported.

Some of these NITF standards are not yet ratified; future SDK releases will track the evolution of these standards to ensure interoperability.

The architecture of the NITF classes may change in a future release, so as to provide simpler and more efficient access to NITF features such as multiple image segments and TRE data.

The motivated reader may wish to learn more in "Georeferencing of NITF Imagery" on page 81.

The NITF Reader

The `NITFImageManager` and `NITFImageReader` classes are used to read NITF files.

Unlike the other image readers, the `NITFImageManager` class must be used to construct an `NITFImageReader` object for a given image segment via the `createReader` method. The `NITFImageReader` object behaves like any other image reader in all other respects. The Reference Manual at `doc/ReferenceManual/index.html` contains several examples of how to read an NITF image.

The following restrictions apply to the reading of NITF imagery:

- Only uncompressed (raw), simple JPEG, and JPEG 2000 segments are supported. (**Note:** JPEG files with explicit quantization tables are not supported.)
- All modes (blocked, masked, etc) are supported.

- Images must be of datatypes, colorspaces, etc, that the SDK normally supports (typically unsigned 8- and 16-bit data, of 1, 3, or *n* bands).
- Color lookup tables (LUTs) are not supported.
- All `CLEVEL`s are supported.
- Label, text, graphic/symbol, DES, and RES segments are recognized and parsed, but access to the data within them is not supported.
- When reading a NITF image, all file header and image segment fields are stored in the metadata for the image (see "NITF Input Metadata" on page 38).
- Each TRE is read and stored as a single binary metadata tag/value pair (again, see "NITF Input Metadata" on page 38). Some TREs are recognized, however, and have their constituent fields extracted into tag/value pairs as well, including: `J2KLRA`, `USE00A`, `STDIDC`, `PIAIMC`, and `RPC00A/B`.

NITF Input Metadata

NOTE: The following discussion assumes familiarity with the NITF specification.

When reading a NITF file, the SDK stores many NITF fields as metadata, for example as tag/value pairs in MrSID imagery (see "Metadata Tags" on page 84).

In general, the tag name is of the form

```
NITF::xxnnn::field
```

where `xx` is a two letter code representing the NITF segment (`IM` for image, `FH` for file header) and `nnn` is the NITF segment number. TRE fields contain the prefix "`TRE_`". Specifically, the SDK stores input fields in metadata tags as described in the following tables.

File Header Fields

The SDK stores file header fields in the following tags:

File Header Fields

NITF Field	GeoExpress Tag
FHDR	`NITF::FH000::FHDR`
FVER	`NITF::FH000::FVER`
STYPE	`NITF::FH000::STYPE`
OSTAID	`NITF::FH000::OSTAID`
FDT	`NITF::FH000::FDT`
FTITLE	`NITF::FH000::FTITLE`
ONAME	`NITF::FH000::ONAME`
OPHONE	`NITF::FH000::OPHONE`

NITF Field	GeoExpress Tag
NUMI	NITF::FH000::NUMI
NUMS	NITF::FH000::NUMS
NUML	NITF::FH000::NUML
NUMT	NITF::FH000::NUMT
NUMDES	NITF::FH000::NUMDES
NUMRES	NITF::FH000::NUMRES

Of the above, the SDK allows the user to set the values for OSTAID, FDT, FTITLE, ONAME, and OPHONE when encoding NITF images.

Security-Related Fields

The SDK stores security-related fields from the file header (unless they are blanks) in the following tags:

Security-Related Fields

NITF Field	GeoExpress Tag
SCLAS	NITF::FH000::SCLAS
SCLSY	NITF::FH000::SCLSY
SCODE	NITF::FH000::SCODE
SCTLH	NITF::FH000::SCTLH
SREL	NITF::FH000::SREL
SDCTP	NITF::FH000::SDCTP
SDCDT	NITF::FH000::SDCDT
SDCXM	NITF::FH000::SDCXM
SDG	NITF::FH000::SDG
SDGDT	NITF::FH000::SDGDT
SCLTX	NITF::FH000::SCLTX
SCATP	NITF::FH000::SCATP
SCAUT	NITF::FH000::SCAUT
SCRSN	NITF::FH000::SCRSN

NITF Field	GeoExpress Tag
SSRDT	NITF::FH000::SSRDT
SCTLN	NITF::FH000::SCTLN

The SDK allows the user to custom set all of the above.

Image Segment Fields

The SDK stores image segment fields in the following tags:

Image Segment Fields

NITF Field	GeoExpress Tag
IID1	NITF::IM001::IID1
IDATIM	NITF::IM001::IDATIM
TGTID	NITF::IM001::TGTID
IID2	NITF::IM001::IID2
ISORCE	NITF::IM001::ISORCE
ICORDS	NITF::IM001::ICORDS
IGEOLO	NITF::IM001::IGEOLO
NICOM	NITF::IM001::NICOM
ICOM1	NITF::IM001::ICOM1
...	...
ICOM9	NITF::IM001::ICOM9

The SDK allows the user to custom set these image segment fields: `IID1`, `IDATIM`, `TGTID`, `IID2`, `ISORCE`, `NICOM`, `ICOM1...ICOM9`.

Additional Notes

In addition, the SDK does the following:

- handles the security fields in the Image Segment as they are handled in the file header
- stores all detected TREs in metadata as binary data. For example, USE00A data would be stored as an array of bytes using the following tag:

  ```
  NITF::IM001::TRE_USE00A
  ```

- explicitly recognizes the following TREs when reading in a NITF file:

  ```
  J2KLRA, USE00A, STDIDC, PIAIMC, RPC00A/B
  ```

For these TREs, additional metadata entries are created corresponding to each of their constituent fields. For example, PIAIMC data is represented as follows:

```
NITF::IM001::TRE_PIAIMC_CLOUDCVR
NITF::IM001::TRE_PIAIMC_SRP
NITF::IM001::TRE_PIAIMC_SENSMODE
NITF::IM001::TRE_PIAIMC_SENSNAME
...
```

Metadata Support

Metadata is data about an image, as distinct from the actual image data (pixels). This includes everything from the basic width/height and datatype information up to geospatial information, background colors, creation dates, etc.

The MrSID SDK uses a key/value pair system for recording tags and their values, similar to the system used by TIFF and GeoTIFF. A “metadata database” containing a set of records, one for each key/value pair, is associated with each image. This chapter describes the basic features and classes of this metadata system.

The metadata classes can all be found in the headers in the `include` directory. Full details and examples may be found in the Reference Manual at `doc/ReferenceManual/index.html`.

The Metadata Record

A single key/value pair is represented by the `LTIMetadataRecord` class. This class has several properties:

- tag name
- datatype
- dimension information (number of dimensions and dimension array)
- data pointer

The tag name is represented as an ASCII string. The well-known strings (those for common metadata tags) have corresponding values in the `LTIMetadataTag` enum. Either form may be used for most metadata operations.

The dimension information describes the “shape” of the data element(s), which is pointed to with a `void*` pointer.

- If the data is a single value (scalar), the number of dimensions is 1 and the dimension array is a one-element vector whose entry is one, i.e. `int dims[1] = { 1 }`.
- If the data is a vector of values, the number of dimensions is 1 and the dimension array is a one-element vector whose entry is the number of the values, i.e. `int dims[1] = { 10 }` for a 10-element data record.
- If the data is an arbitrary vector of values, the number of dimensions is *n* and the dimension array is an N-element vector whose entries are the lengths of each dimension, i.e. `int dims[2] = { 4,5 }` for a 20-element data record formatted as a 4x5 array of values.

In all cases, the data values will all have the same data type. The `void*` pointer is cast to the appropriate type and dereferenced accordingly to access the true value.

The `LTIMetadataRecord` class contains a number of helper functions to get and set the data values.

The class `LTIMetadataDumper` can be used to pretty print a metadata record as a development and debugging aid.

The Metadata Database

Each image stage has an associated `LTIMetadataDatabase` object. This database supports several types of operations:

- query for a record
- add a record
- retrieve a record
- remove a record

Records may be accessed via tag name or (positional) record number.

Each image stage is responsible for maintaining the consistency of its own database. For example, if an image filter changes the width of its image, the corresponding metadata tag must be updated in the local copy of the database. In most situations the metadata database for an image pipeline should be considered to be the database presented by the last stage.

The class `LTIMetadataDumper` can be used to pretty print a metadata database as a development and debugging aid.

The Tags

Certain metadata tags are set by almost all image stages and are known as the “classical” metadata tags:

- width and height
- datatype
- colorspace
- dynamic range
- background and nodata pixel values
- geospatial (*x*,*y*) origin, *x*- and *y*-resolution, and rotation
- original file size
- original filename

The corresponding image tag names can be found in the `LTIMetadataTag` enum. See "Metadata Tags" on page 84 for further details.

The C API

The MrSID SDK is a large set of C++ classes containing many features and designed for many different needs and workflows. There is a corresponding cost in learning and development time to working with the full SDK.

Recognizing this cost, the SDK also provides a small, C-based API for developers who wish to quickly implement or prototype simple decode support for MrSID, JPEG 2000 and NITF images.

The C API provides the following features:

- decode support for MrSID, JPEG 2000 and NITF
- basic metadata access
- file-based and stream-based decodes
- BSQ format output

Major features not available through the C API include:

- access to the image pipeline architecture (readers, filters, writers)
- support for other file formats
- encode support
- progress and interrupt delegates

NOTE: The C API is built on top of the C++ API. Although the linkage is done with C names, use of C++ runtime libraries is still required.

This chapter describes the basics of the C API, as implemented in the two files `ltic_api.h` and `lt_ioCStream.h`, in the `include` directory. The Reference Manual at `doc/ReferenceManual/index.html` contains complete details and examples.

Image Support

The functions for opening and closing existing MrSID, JPEG 2000 and NITF images are:

- `ltic_openMrSIDImageFile()`
- `ltic_openMrSIDImageStream()`
- `ltic_openJP2ImageFile()`
- `ltic_openJP2ImageStream()`
- `ltic_openNITFImageFile()`
- `ltic_closeImage()`

These functions all take either a filename or a stream, denoting the input, and a pointer to an “image handle” of type `LTICImageH`. This handle is used to provide the context for all other functions in the C API. Returned status codes are used to indicate success or failure.

To close an image, the `ltic_closeImage()` function is used.

Due to rounding and file format differences, you should not manually calculate the image dimensions at magnification. Instead, you should use the `ltic_getDimsAtMag()` function to ask the SDK for the dimensions of the image at a given magnification.

Basic image properties such as width, height, colorspace, geospatial position, etc, are accessed using simple query functions. For example, the width of an image is determined by calling this function:

```
lt_int32 ltic_getWidth(const LTICImageH image)
```

Decode Support

To decode a scene from an image only one function call is required:

```
LT_STATUS ltic_decode(LTICImageH image,
                      double xUpperLeft,
                      double yUpperLeft,
                      double width,
                      double height,
                      double magnification,
                      void** buffers)
```

Note that this function closely approximates the C++ API for read operations: the five double arguments correspond exactly to an `LTIScene` object and the buffers argument is a pointer to an array of band buffers, corresponding to the constructor of an `LTISceneBuffer` object.

Only “packed BSQ” output is available using the C API.

Metadata Support

The metadata of an image is accessed via two functions. The `ltic_getNumMetadataRecords()` function returns the number of metadata records in the image. The `ltic_metadataRecord()` function is used to extract the actual data values of a specific record.

As with the `ltic_decode()` call, the parameters to the `ltic_getMetadataRecord()` function closely mimic their C++ SDK counterparts in the `LTIMetadataDatabase` class.

Streams

An image may be opened either with a filename (const char*) or a stream. The MrSID SDK provides a class derived from `LTIOStreamInf`, called `LTIOCallbackStream`, which is designed for third-party streams and stream-like data structures.

The file `lt_ioCStream.h` contains a set of C interfaces for constructing an `LTIOCallbackStream` object. The `lt_ioCallbackStreamCreate()` takes as parameters a set of function pointers corresponding the operations for opening, closing, reading, writing, etc, as is required by the `LTIOCallbackStream` class. The functions `lt_ioCStreamOpen()`, `lt_ioCStreamRead()` etc., all invoke the underlying `LTIOStreamInf` methods for the `LTIOCallbackStream` object.

Full details and examples may be found in the Reference Manual at `doc/ReferenceManual/index.html`.

Command Line Applications

Several simple command line applications are included in this SDK to aid in development and testing.

Switches Common to All Tools

The following switches are common to mrsidinfo, mrsiddecode, mrsidencode, mrsidoptimize and mrsidtile.

`-h` or `-?`

Show short usage message

`-help`

Show detailed usage message

`-v` or `-version`

Show version information

`-credits`

Show credits and copyrights

`-quiet`

Don't show informational messages. This does not affect the `logfile` output.

`-log STRING`

Write output to log file. This does not affect the `stdout` output.

`-progress STRING`

Progress meter style: `none | default | timer`

If the environment variable `LT_NITF_IMAGENUM` is set to a number, the NITF decode applications (mrsiddecode, mrsidinfo, mrsidviewer) will attempt to open only the image with the segment number given by the environment variable. Note that image segments are numbered starting at 1, not zero. The default is to attempt to open all segments as a mosaic of images.

mrsidinfo

The mrsidinfo tool displays basic information about an image, such as width, height, etc. Various image formats are supported.

File Control

`-if` or `-inputFormat STRING`

Input image type: `tif | ras | jpg | nitf | doq | doq_nc | bbb | lan | bmp | img | jp2 | sid`

`-mos` or `-mosaic`

Image format is mosaic (aux file)

Main Switches

`-cralpha` or `-crWithAlpha`

include alpha band in compression ratio calculation.

`-meta` or `-metadata`

Show metadata tags and values

`-tile` or `-tiles`

Show tile information (composite MG3 or MG4 only)

`-wf` or `-worldFile`

Generate world file

`-dims` or `-projectDims`

Show image dimensions at various resolution levels

`-input` or `-inputFile STRING`

Name of input file (required)

`-gml` or `-gmljp2`

Show embedded GML information including schemas and custom coordinate reference system information

`-prof` or `-genProfile STRING`

Generate JP2 profile

`-ignorewf` or `-ignoreWorldFile`

Iignore georeferencing from world files

`-wkt`

Dump the Well Known Text

`-aoi`

Dump all AOI info

Examples

In the following example, the command displays the basic image information for a MrSID image named `image.sid`:

```
mrsidinfo image.sid
```

In the following example, the command line displays the basic image information and the metadata for a TIFF image named `foo.tif`:

```
mrsidinfo –meta foo.tif
```

In the following example, the command line displays metadata information for an AUX file containing JPEG format images named `mosaic.txt`:

```
mrsidinfo –mos –if jpg mosaic.txt
```

mrsiddecode

The mrsiddecode tool extracts (decodes) all or a portion of a MrSID or JPEG 2000 image to one of several image formats.

Switches

File Control

`-i` or `-inputFile STRING`

Input file (required)

`-o` or `-outputFile STRING`

Output file (required)

`-of` or `-outputFormat STRING`

Output image format: `tif | tifg | jpg | bip | bil | bsq | bmp`

`-if` or `-inputFormat STRING`

Input image format

`-mos` or `-mosaic`

Image format is mosaic (aux file)

Main Switches

`-j` or `-numThreads`

Sets the number of available threads used while processing MG3 and MG4 files. If no value is specified, the number of threads is set to the maximum. For more information, see "Multi-Threading" on page 17.

`-watermarkFile STRING`

Watermark file

`-watermarkPosition STRING`

Position of watermark: `center | center_left | center_right | upper_left | upper_center | upper_right | lower_left | lower_center | lower_right (default)`

`-drmin` or `-dynRangeMin FLOAT`

Minimum dynamic range

`-drmax` or `-dynRangeMax FLOAT`

Maximum dynamic range

`-drauto` or `-autoDynRange`

Automatic dynamic range adjustment

`-wf` or `-worldFile`

Generate world file

`-sh` or `-stripHeight UNSIGNED-INT`

Strip height

`-endian STRING`

Endianness (BBB output only): `host | little | big`

`-bsq`

Output is BSQ (BBB output only)

`-bgc` or `-backgroundColor STRING`
Set background color

`-tpc` or `-transparencyColor STRING`
Set transparency color

MrSID Options

`-pwd` or `-password STRING`
password to decode image (MG2 and MG3 only)

`-b` or `-bandList STRING`
a comma separated list of band to decode (MG4 only)

JPEG 2000 Options

`-layers` or `-qualityLayers UNSIGNED-INT`
Number of quality layers

`-prec` or `-precision UNSIGNED-INT`
bits of precision in output

Scene Selection

`-ulxy FLOAT0 FLOAT1`
Upper-left of scene (x,y)

`-lrxy FLOAT0 FLOAT1`
Lower-right of scene (x,y)

`-cxy FLOAT0 FLOAT1`
Center of scene (x,y)

`-wh FLOAT0 FLOAT1`

Size of scene (width,height)

`-s` or `-scale SIGNED-INT`
Scale of scene

`-coord` or `-coordspace STRING`
Coordinate space of scene: `geo | image`

Examples

To decode a MrSID file to a JPG file:

```
mrsiddecode -i input.sid -o output.jpg
```

To decode a MrSID file to a GeoTIFF file:

```
mrsiddecode -i input.sid -o geotiff.tif -of tifg
```

To decode the upper-left 50x50 pixel scene from a JPEG 2000 image:

```
mrsiddecode -i input.jp2 -o output.tif -ulxy 0 0 -wh 50 50
```

To decode an image at scale 2, i.e. one-quarter resolution:

```
mrsiddecode -i input.jp2 -o output.tif -s 2
```

Some Definitions

Let us define "input scene" (or just "scene") to be the portion of the MrSID image to be decoded. The scene may be the whole image or some subset. The scene may extend "outside" of the image proper.

Let us also define "output window" (or just "window") to be the region occupied by the image produced by the decoder, e.g. the generated TIFF. Note that the output window may not be the same size as the input scene.

Finally, let us define "coordinate space" to refer to one of two possible ways of specifying regions of the image:

- image space: what we normally think of as "pixel" space, at full resolution
- geo space: the image's geographic space

Basic Scene Selection

The command line decoder allows you to specify the input scene explicitly in one of three ways:

- upper-left (x,y) corner of scene, and (width,height) of scene

  ```
  -ulxy X Y -wh W H
  ```

- upper-left (x,y) corner of scene, and lower-right (x,y) corner of scene

```
-ulxy X Y -lrxy X Y
```

- center (x,y) of scene, and (width,height) of scene

```
-cxy X Y -wh W H
```

For example, consider a 100x100 image. To select the upper-left quarter of the image, the following three ways are equivalent:

```
-ulxy 0 0 -wh 50 50
-ulxy 0 0 -lrxy 49 49
-cxy 25 25 -wh 50 50
```

(The reader is strongly encouraged to work these examples through manually, to ensure full understanding of the material being presented.)

Note that if no scene is specified, the decoder defaults to the entire image.

Scaled Scene Selection

Consider again the 100x100 image. At scale 1 (half-resolution), this is a 50x50 image. To decode the same upper-right quarter at half-res -- a 25x25 image -- these three are equivalent (modulo round-off errors):

```
-s 1 -ulxy 0 0 -wh 25 25
-s 1 -ulxy 0 0 -lrxy 24 24
-s 1 -cxy 12.5 12.5 -wh 25 25
```

The first form is straight-forward: it just says make a scale 1 image, starting at the upper-left corner, of size 25x25. The second form is also clear: make a scale 1 image, starting at the upper-left corner and extending down to (24,24).

And the third form is really just a variation of the first: make a scale 1 image, centered at (12.5,12.5), of size 25x25...

Input Scenes and Output Windows

Recall that in the definitions above, we noted the output window may be different than the input scene. Here's how that can happen.

Consider the following decode on our 100x100 image:

```
-s 0 -ulxy 0 0 -wh 125 125.
```

This is a normal scale 0 decode, but the output window is 125x125 -- considerably larger than our input scene. The result is an image that really is 125x125, with the right and bottom edges left as black (technically, the background color).

One could reasonably argue that allowing the user to specify a decode "outside the image" is a bug, but for an end-user app it's actually a nice feature to have: it lets the user extract out some arbitrarily-sized region of her image, but turn it into a more well-formed size. The classic example of this feature is a decode along these lines:

```
-s 4 –ulxy 0 0 -wh 32 32
```

which extracts the scaled image into a stock 32x32 icon (we assume here that the scale 4 image is of some size less than or equal to 32x32, otherwise they'd have to use a smaller scale).

The input scene is always positioned in the upper-left corner of the output window -- unless you are using the `-cxy` scene selection mechanism, in which case the input scene is always positioned in the center of the output window.

Geo Coordinate Spaces

The above examples are all done in the familiar "pixel space", where (0,0) is the upper-left corner and the lower-right corner is (width-1,height-1) and each pixel is a 1x1 object. We call this "image" space and it is the default.

For geo images, however, we often prefer to express points on the image and dimensions in terms of geo coordinates. That is, the upper-left might be (32929.00,28292.25) and the size of the pixel might be (0.0005,0.0005).

To accomodate this, we allow the above scene selections to be done in "geo" space. If our 100x100 image had an upper-left coordinate of (100.0,50.0) and an (*x,y*) resolution of (10.0,5.0), we would use the following to decode the whole image:

```
-coord geo -ulxy 100 50 -wh 1000 500
```

That is, the input scene starts at the upper-left position of (100.0,50.0) and the output window is the full size of the input image (100*10=1000 and 100*5=500).

More Examples

To conclude, the examples within each of the following groups of scene selections using our 100x100 image with UL=(100,50) and res=(10,5) are all equivalent.

Full scene, full res (100x100 image, all picture at full res)

```
-ulxy 0 0 -wh 100 100 -s 0 -coord image
-ulxy 0 0 -lrxy 99 99 -s 0 -coord image
-cxy 50 50 -wh 100 100 -s 0 -coord image
-ulxy 100 50 -wh 1000 500 -s 0 -coord geo
-ulxy 100 50 -lrxy 1090 -445 -s 0 -coord geo
-cxy 600 -200 -wh 1000 500 -s 0 -coord geo
```

Part scene, full res (50x50 image, UL quarter of image at full res)

```
-ulxy 0 0 -wh 50 50 -s 0 -coord image
-ulxy 0 0 -lrxy 49 49 -s 0 -coord image
-cxy 25 25 -wh 50 50 -s 0 -coord image
-ulxy 100 50 -wh 500 250 -s 0 -coord geo
-ulxy 100 50 -lrxy 590 -195 -s 0 -coord geo
-cxy 350 -75 -wh 500 250 -s 0 -coord geo
```

Full scene, half-res (whole image at half res, in UL quadrant of 100x100 image)

```
-ulxy 0 0 -wh 100 100 -s 1 -coord image
-ulxy 0 0 -lrxy 99 99 -s 1 -coord image
-ulxy 100 50 -wh 1000 500 -s 1 -coord geo
-ulxy 100 50 -lrxy 1090 -445 -s 1 -coord geo
```

Full scene, half-res (centered) (whole image at half res, in center quadrant of 100x100 image)

```
-cxy 50 50 -wh 100 100 -s 1 -coord image
```

Full scene, half-res (centered) (whole image at half res, in center of a 50x50 image)

```
-cxy 600 -200 -wh 1000 500 -s 1 -coord geo
```

mrsidviewer

The mrsidviewer application is a very simple Windows application for viewing images in all of the formats supported by the SDK, including MrSID and JPEG 2000.

NOTE: The mrsidviewer application is available on Windows SDKs only.

Operations

- zoom in: left-click
- zoom out: shift + left-click
- pan: right-click and drag

The viewer may be invoked from the command line:

```
C:\>mrsidviewer image.sid
```

Normal drag and drop operations are also supported.

The (classical) metadata for the image can be displayed via the **Options | Metadata** menu item.

Appendix A - Technical Notes

This chapter contains several articles providing background information on specific issues and questions frequently asked by some SDK developers. Please check LizardTech's Developer website (http://developer.lizardtech.com) for additional and updated technical notes.

Zoom Levels

The following describes the MrSID SDK's handling of zoom levels for MG2, MG3, MG4 and JP2. In the code below, assume the following:

`W` = image width (constant)

`H` = image height (constant)

`n` = number of levels

MG2 Behavior

The default number of zoom levels is computed by initially assuming the maximum possible number of levels. The size of the image that would be needed to support that number of levels is then computed. If the actual image is larger than the computed size, the number is determined and the SDK inquires no further. Otherwise, the SDK recomputes the image size with a reduced number of levels, and continues to do so until the actual image size is equal to or greater than the required size.

```
n = 9
loop
{
     min_length = (IPOW(2, n+1) - 2) * 4 + 1
     if (W > min_length) and (H > min_length) break;
     --n
}
```

The `min_length` values for `n=9, 8, …, 3` are `4089, 2041, …, 57`.

The maximum number of levels for the image is computed the same way.

These zoom level calculations are in the `MG2WriterParams` class. However, if you request less than 3 levels (or more than 9), the SDK will clamp your request to 3 (or 9). These are hard limits for MG2.

MG3 and MG4 Behavior

MG3 and MG4 use different equations than MG2 and do not have the hard limits of 3 and 9 levels.

The default number of zoom levels is determined by initially assuming zero levels, then reducing the minimum image dimension by a factor of two until it falls below 32.

```
n = 0
minWH = MIN(W,H)
while (minWH >= 32)
{
     minWH = minWH / 2
```

```
        ++n
    }
```

The maximum number of zoom levels is computed by a different method: the number of levels is increased as long as the dimension at that number of levels is not greater than the minimum actual image dimension.

```
n = 1
minWH = MIN(W,H)
while (minWH > (1 << (n+1)) )
{
     ++n
}
--n
```

These zoom level calculations are in the `MG3WriterParams` and `MG4WriterParams` classes.

JP2 Behavior

JP2 behaves identically to MG3 and MG4. These zoom level calculations are in the `JP2WriterParams` class.

Example

Consider a 512x512 image. Using the SDK encoder and info tools, the following results are obtained:

mg2: image is encoded to 6 levels

0: 512x512

1: 256x256

..

5: 16x16

6: 8x8

mg3: image is encoded to 5 levels

0: 512x512

1: 256x256

..

5: 16x16

jp2: image is encoded to 5 levels

0: 512x512

1: 256x256

..

5: 16x16

Coding Conventions

The developers of the MrSID SDK follow a general set of coding conventions and guidelines. While we certainly don't expect third-party developers to adhere to these conventions, they are described here as an aid to understanding and working with the SDK, both at the syntactic and semantic level.

Note that these are only conventions, not hard rules; the SDK does not follow all of these guidelines all the time, for reasons of practicality, historical practice, compatibility, etc. Where important for developers, such deviations are noted in the Reference Manual at `doc/ReferenceManual/index.html`.

Syntactic Conventions

Line length

Lines should be no longer than 78 columns in order to avoid line- and wordwrapping.

Tabbing

Tabs are implemented as 3 spaces.

Namespaces

Header files should surround declarations with `LT_BEGIN_NAMESPACE (LizardTech)` and `LT_END_NAMESPACE(LizardTech)`, and source files should declare `LT_USE_NAMESPACE(LizardTech)` at the top.

Warning levels

Source files should be compiled with high warning levels on Win32 compilers:

```
#if defined(LT_COMPILER_MS)
     #pragma warning(push,4)
#endif
...
#if defined(LT_COMPILER_MS)
     #pragma warning(ppop)
#endif
```

Variable names, type names, etc

- variables are spelled using camelcase: `upperLeftPos`
- function names are also spelled using camelcase: `getUpperLeft()`
- class member variables use an "`m_`" prefix: `m_xPosition`
- static variables (whether class members or not) use an "`s_`" prefix: `s_twoPi`
- enum and type names are capitalized: `LTIImageStage`, `LTIDataType`
- however, the integral primitive types are an exception: `lt_uint8`
- enum values are uppercase and generally are named to reflect their datatype: `LTI_DATATYPE_UINT8`
- macros are uppercase: `LT_USE_NAMESPACE`

Integral primitives

The use of the typedef'd primitive datatypes, e.g. `lt_uint8`, is preferred when

possible and practical.

File names

For the "LT" and "LTI" classes, the source filenames are spelled with leading prefixes – for example, class `LTIGeoCoord` is defined in header `lti_geoCoord.h`. For all other classes, the filename matches the class name – for example, class `MG4ImageWriter` and header `MG4ImageWriter.h`.

Semantic Conventions

Initialization

As a precaution, all variables should be initialized at the point of declaration. A value of 0 or -1, `INVALID` (available for most enums), or `LT_STS_Uninit` (for status codes) is usually appropriate.

Status codes

The public SDK does not use exceptions, and the internal implementation uses them only sparingly. In general, functions should return status codes rather than relying on `throw`. While putting an additional burden on the user, our experience has shown that status codes are both more portable and less prone to error than the alternatives. If a function returns a status code, always check it. The standard idiom for checking status codes is:

```
LT_STATUS
Class::foo()
   {
      ...
      LT_STATUS sts = bar();
      if (!LT_SUCCESS(sts)) return sts;
      ...
      return LT_STS_Success;
   }
```

When writing a new function, consider returning an `LT_STATUS` status code hardwired to `LT_STS_Success` instead of just returning `void`. If future development of the function requires the ability to return failure codes, downstream uses will not have to be changed.

Constructors

"Heavy" objects which require nontrivial work in their constructors should use an `initialize()` function. This requires extra code for the user, but provides a means for returning status codes back to the user without relying on exceptions. In particular, objects should not (explicitly or implicitly) call new or any other nontrivial function from within their constructors.

The "creator-deletes" rule

The object that creates a new object is considered to be the "owner" of that new object and as such has responsibility for deleting it. Passing the allocated object by address to a function doesn't pass ownership to that function (unless documented otherwise).

Reference counting

The SDK uses reference counting to manage the lifespan of objects that comprise image pipelines. This is similar in spirit to “smart” or “auto” pointers. The static member function `create()` of a class should be used to create a new instance of that class. Similarly, the (non-static) member function `release()` should be used when you are done with object – for example, when the pointer to the object goes out of scope. The `retain()` function should be used to hold onto an object, i.e. increment the reference count; if you created the object, however, do not call `retain()`. When the last reference calls `release()`, the object will delete itself. The two most common reference counted classes are `LTIImageStage` and `LTIImageStageManager`.

Overrides

The SDK uses a system of “mixins” to simplify the overriding of `LTIImageStage` properties. Specifically, `LTIImageStage` defines an abstract interface that needs to be implemented by derived class of `LTIImageFilter`; `LTIImageFilter` implements the `LTIImageStage` interface by forwarding the method call to the next `LTIImageStage` in the pipeline. Derived classes of `LTIImageFilter` that change image properties will need to override the accessor functions.

Pass-by-reference

Data should be passed by reference whenever possible, for example `LTIImageStage&` rather than `LTIImageStage*`. Pass-by-address is preferred when reassigning ownership or when `NULL` is being used for some sentinel value.

Const

Use `const` whenever possible. In function declarations, it is used to indicate "in/out" semantics for parameters (excluding by-value parameters). In variable declarations, it is used to clarify intent of the variable's usage.

Disabled standard class member functions

Unless they are required by the class, it is preferable to explicitly disable the assignment operator, copy constructor, and default constructor.

```
private:

            LTINavigator();

            LTINavigator(LTINavigator&);

            LTINavigator& operator=(const LTINavigator&);
```

One class per header file

A header file should contain the declaration of only one class. Implementations should not be in the header file unless truly warranted.

Inlining

Implementations should not be defined inline in the header file unless truly warranted.

Do not use compiler-specific inlining pragmas.

Templates

Use templates sparingly and only when truly warranted, as they lead to code bloat and/or incompatibilities among various compilers. Consider using a templated function private to a source module instead of making a public templated class. If you must use templates, avoid the more complex parts of the language.

Template headers

A header file containing only a templated class should have a corresponding source file which includes it, even though the source file contains no implementation code.

Global data

The SDK contains no global variables. No non-const module statics or singleton classes are used.

Threadsafety

The SDK may be safely used in multithreaded applications. Locking of SDK objects is NOT provided, however; the application must guarantee each object is accessed serially within a particular thread context.

Assertions

Assertion macros, specifically `LT_ASSERT()`, should be used liberally.

Delegates

Where appropriate, for "callback" or "handler" mechanisms the use of simple abstract classes ("delegates") is preferred to using a function pointer typedef.

Set/get

Use `set()/get()` style member variable access instead of making data members public.

Other Conventions

STL

Do not use STL in public interfaces, as this may cause linkage problems. While the SDK uses STL internally in some areas, its use is in general discouraged.

Optimization

Do not write "hand-optimized" code. Write first for correctness and maintainability; optimize it later only if profiling justifies it.

Overrides

The SDK uses a system of “mixins” to simplify the overriding of `LTIImageStage` properties. Specifically, `LTIImageStage` defines an abstract interface that needs to be implemented by derived class of `LTIImageFilter`; `LTIImageFilter` implements the `LTIImageStage` interface by

forwarding the method call to the next `LTIImageStage` in the pipeline. Derived classes of `LTIImageFilter` that change image properties will need to override the accessor functions.

A cropping filter would change the width and height of the image at that stage in the pipeline, and so would need to override `getWidth()` and `getHeight()`.

For example:

```
#include "lti_imageFilter.h"
#include "lti_imageStageOverrides.h"

class MyCropFilter : public LTIOverrideDimensions< LTIImageFilter >

{
   ...
   LT_STATUS initialize(... lt_uint32 newWidth, lt_uint32
   newHeight, ...)
   {
      ...
      sts = setDimensions(newWidth, newHeight);
      ...
   }
   ...
};
```

`LTIOverrideDimensions<>` adds `getWidth()` and `getHeight()`, a protected `setDimensions()` function, and the needed data members.

The SDK uses templates to implement the mixins over virtual inheritance. To override many sets of properties you use the following code:

```
class MyFilter : public LTIOverrideXXX < LTIOverrideYYY <
LTIOverideZZZ < LTIImageFilter > > >
{ ... };
```

The list of override mixin templates and the `LTIImageStage` functions they override is as follows:

- `LTIOverrideMetadata`
  - `getMetadata()`
- `LTIOverrideDimensions`
  - `getWidth(), getHeight()`
- `LTIOverridePixelProps`
  - `getPixelProps()`
  - `getMinDynamicRange()`
  - `getMaxDynamicRange()`
- `LTIOverrideNoDataPixels`
  - `getNoDataPixel()`
  - `getBackgroundPixel()`
- `LTIOverrideGeoCoord`
  - `getGeoCoord()`
  - `isGeoCoordImplicit()`
- `LTIOverrideMagnification`
  - `getMinMagnification()`
  - `getMaxMagnification()`

- LTIOverrideIsSelectiveData
  - isSelective()
- LTIOverrideStripHeight
  - getStripHeight()
  - setStripHeight()
- LTIOverrideDelegates
  - setProgressDelegate()
  - getProgressDelegate()
  - setInterruptDelegate()
  - getInterruptDelegate()
- LTIOverridePixelLookupTables:
  - getPixelLookupTable()

Reference Counting

LTIImageStage objects are now reference counted.

- Use ClassName::create() to create a new object of type ClassName.
- Use object->retain() when you want to keep an object around (but not if you created it)
- Use object->release() when you are done using the object.

As a rule of thumb, if you *create* or *retain* an object then you must issue a corresponding *release* for it. ClassName::create() and LTIImageStageManager::createImageStage() are the two must common ways to create an LTIImageStage object. When you pass a reader to a filter, the filter calls retain() on the reader which "keeps the object alive". Before the pointer to the reader goes out of scope or is reassigned you must call release() as shown in the following example:

```
LTIImageStage *pipeline = NULL;
{

   MrSIDImageReader *reader = MrSIDImageReader::create();
   sts = reader->initialize(filename);
   pipeline = reader;
}

{
   MyImageFilter *filter = MyImageFilter::create();
   // 'pipeline' will be retained by 'filter'
   sts = filter->initialize(pipeline);

   // release 'pipeline' because we are reusing the variable
   pipeline->release();
   pipeline = filter;
}

...
... do something with the pipeline
...
// done with the pipeline: releasing 'pipeline' will release the
// filter, which will in turn release the reader
pipeline ->release();
pipeline = NULL;
```

Notes on Streams

The `LTIOStreamInf` class, and streams derived from it, provide an abstraction for performing I/O in a variety of ways, including "large file" I/O, buffered I/O, memory-based I/O, etc. As it is a well-known model, the semantics of the stream operations are very similar to those of the Unix stdio operations.

This technical note provides some technical details on the `LTIOStreamInf` operations.

Offsets

The term "offset *n*" refers to byte (*n*+1) in the file; that is, offset 0 is the first byte and offset 10 is the eleventh byte. When a stream is "positioned at offset *n*", we mean than the next byte read in will be byte (*n*+1).

Initialization

Each derived class has constructor which has no parameters and an `initialize()` function which zero or more parameters which will vary according to each derived class. (**Note:** this initialization process is different from that of most of the other SDK functions, which put all constructor parameters in the constructor and not the initialization function.)

The `initialize()` function must be called prior to any other member functions.

open()

- `open()` must be called before any other stream functions can be called (excluding `initialize()` and `close()`)
- beyond that, the semantics of `open()` are undefined; typically, it will allocate resources on behalf of the stream, e.g. a `FILE` handle, and/or make them available to the user
- after `open()`, the stream will be positioned at offset 0 and the `EOF` flag will be false
- calling `open()` on an already opened file will return an error

`close()`

- `close()` will deallocate the resources, but in such a way that a subsequent call to `open()` will restore them for use
- calling `close()` on a closed stream will have no effect (and is not an error)
- strictly speaking, `close()` need not be called as the destructor is expected to call `close()`; relying on this is considered bad form, however
- a closed stream must be opened again before any other functions may be called

`read(lt_uint8 *buffer, lt_uint32 len)`

- `read()` will return the number of bytes successfully read; only that many bytes are valid within the read buffer
- if the number of bytes read is not equal to the number of bytes asked for, then exactly one of the following is true:
- `EOF` was encountered
    - the stream uses "socket semantics", and one or more additional reads will be required to get the remaining desired bytes
    - an error occurred

- the `getLastError()` function is used to determine the precise error condition, if the number of bytes read is not equal to the number of bytes requested
- the position of the stream after the read is equal to the position of the stream prior to the read plus the number of bytes successfully read
- if `EOF` is true when the read is requested, read will return 0 bytes read and keep `EOF` set to `true`

`write(const lt_uint8 *buffer, lt_uint32 len)`

- `write()` will return the number of bytes successfully written
- if the number of bytes written is not equal to the number of bytes requested, then exactly one of the following is true:
  - the stream uses "socket semantics", and one or more additional writes will be required to output the remaining bytes
  - an error occurred
- the `getLastError()` function is used to determine the precise error condition, if bytes read != bytes given
- the position of the stream after the write is equal to the position of the stream prior to the write plus the number of bytes successfully written
- a call to `write()` will always clear the `EOF` flag; `write()` never sets the `EOF` flag

`tell()`

- `tell()` returns the current offset as a 64-bit value

`seek()`

- `seek()` positions the stream to the given offset using a 64-bit value
- the `EOF` flag is reset

`EOF`

- when an attempt is made to read past the last byte of the file, the `EOF` flag becomes true
- in particular, note that merely reading the last byte will not set `EOF` to true
- for example, consider a file of 4 bytes, with the stream positioned at offset 0:
  - a read request of 4 bytes will return 4 bytes read, position is offset 4, `EOF` is not set
  - a read request of 6 bytes will return 4 bytes read, position is offset 4, `EOF` is set
- a write operation has no effect on the `EOF` flag
- a seek operation always clears the `EOF` flag

`duplicate()`

- `duplicate()` creates a new stream of the same type as the original stream and calls `initialize()` on it with same parameters as original stream
- `isOpen()` should initially return false; it is up to the caller to call `open()` on the newly created stream

`getLastError()`

- The `getLastError()` function is used to get the status code when one of the following I/O functions failed:
  - `read()`
  - `write()`
  - `tell()`
  - `duplicate()`

- The `getLastError()` function is required because, like the other I/O functions, these functions do not return status codes.
- The value returned by `getLastError()` is undefined unless called immediately after a failed call to one of the above functions. A call to any other I/O function will invalidate the state of `getLastError()`.
- The minimal implementation of this function is to return `LT_STS_Failure`.

Modes

Any "modes" that a stream supports ("`w`", "`wb`", "`r+`", etc) are defined by the derived class; there is no notion of mode at the base class level.

For example, it is entirely possible one would want to make a "read-only file stream" class. Such a class would be implemented with the `write()` function always returning 0 bytes read.

Notes on World Files

A "world file" is a simple text file containing auxilliary georeferencing information for an image. It can be used to georeference an image that has no georeferencing information within it, or to override existing georeferencing information.

By convention, the filename for a world file is the same as the image it pertains to, with a different extension. The three-letter extension is made up of the first and last characters of the image filename extension, followed by a 'w'. For example, the world file for a TIFF image named `bainbridge.tif` would be `bainbridge.tfw`; the world file for a MrSID image named `madison.sid` would be named `madison.sdw`.

The `LTIGeoCoord` class stores the same information as a world file. It has member functions for reading and writing world files. Many of the image reader and writer classes support the use of world file georeferencing information.

Format

The world file format is six lines, each line containing a double precision value (represented in text). No additional lines may be present. Leading and trailing whitespace are allowed. The meanings of the six values are:

- dimension of a pixel in map units in *x* direction
- first rotation term
- second rotation term
- dimension of a pixel in map units in *y* direction
- *x*-coordinate of the center of the upper-left pixel
- *y*-coordinate of the center of the upper-left pixel

The *y*-dimension is, by convention, a negative value.

The MrSID SDK currently ignores the rotation term in most cases.

Because the floating point values are represented textually, when using world files be wary of errors due to roundoff, imprecise scanf/printf conversion, etc.

Example

This is an example of a world file:

```
  0.20000000
  0.00000000
  0.00000000
 -0.20000000
780.10000000
219.90000000
```

This world file indicates the image resolution is (0.2, -0.2) and the upper left is at (780.1, 219.9). The rotation terms are zero, meaning no rotation is required.

Notes on BBB Files

A BBB image consists of two files, a binary file containing only the raw sample values of the image and a text file describing the image properties. The raw data may be organized in one of three layouts: "band-interleaved by pixel" (BIP), "band-interleaved by line" (BIL), and "band sequential" (BIL). The three formats are collectively referred to as the BBB file format.

Because BBB files only contain raw data with an easily editable header format, they are often used as a "least common denominator" interchange format. However, there is no set standard for the keywords that may be contained in the header. This document describes the header format that the MrSID SDK supports, via the `LTIBBBImageReader` and `LTIBBBImageWriter` classes.

Filenames

The `LTIBBBImageReader` class supports four filename extensions: `.bip`, `.bil`, `.bsq`, and `.bbb`. The first three imply the layout is BIP, BIL, or BIP respectively; the `.bbb` extension implies the default layout, which is BIP.

The header file for a BBB image has the same name as the image, but with a `.hdr` extension.

Header Syntax

The header file is a simple text file containing keywords and their associated value, one keyword/value(s) set per line.

All keywords and values are case-insensitive.

Blank lines are ignored. Leading and trailing whitespace is ignored.

A line that begins with a '#' character, possibly preceded by whitespace, indicates a comment line. Comment lines are ignored.

Supported Keywords (Reader)

The keywords and their allowed values, as supported by the `LTIBBBImageReader`, are as follows:

`BANDGAPBYTES`

Not currently supported – value is ignored

BANDROWBYTES

Not currently supported – value is ignored)

BANDS

Same as NBANDS

BYTE_ORDER

Endianness interpretation of data

Allowed values:

- MOTOROLA, M, BIG, BIGENDIAN
- INTEL, I, LITTLE, LITTLEENDIAN
- NA

Default: host endianness

The value NA (not applicable) may only be used if the number of bands is 1

BYTEORDER

Same as BYTE_ORDER

COLORSPACE

The colorspace of the image

Allowed values:

- GREY, GRAY, GREYSCALE, GRAYSCALE
- RGB
- CMYK
- MULTISPECTRAL

Default: GRAY for 1-banded images, RGB for 3-banded images, otherwise MULTISPECTRAL

COLS

Same as NCOLS

DATATYPE

The data type of the samples

Allowed values: U8, U16, F32

Default: U8

DYNAMICRANGELEVEL

The midpoint of the range of the data

Allowed values: a single floating-point value (applies to all bands)
Default: (none – value is determined by `LTIImage`)

`DYNAMICRANGEMAX`
The maximum dynamic range
Allowed values: a single floating-point value (applies to all bands)
Default: (none – value is determined by `LTIImage`)

`DYNAMICRANGEMIN`
The minimum dynamic range
Allowed values: a single floating-point value (applies to all bands)
Default: (none – value is determined by `LTIImage`)

`DYNAMICRANGEWINDOW`
The size of the range of the data
Allowed values: a single floating-point value (applies to all bands)
Default: (none – value is determined by `LTIImage`)

`E_SQUARED`
Sphere eccentricity squared, for georeferencing
Allowed values: (floating-point)
Default: (none)

`INTERLEAVING`
Same as `LAYOUT`

`LAYOUT`
The data layout; use of this keyword overrides the layout implied by the filename extension
Allowed values: `BIP`, `BIL`, `BSQ`, or `NA`
Default: `BIP`
The value `NA` (not applicable) may only be used if the number of bands is 1

`MAP_UNITS`
Measurement unit for georeferencing
Allowed values: (string)
Default: (none)

`NBANDS`

The number of bands in the image
Allowed values: 1-65535
Default: (none – this keyword is required)

NBITS
Number of bits used per sample
Allowed values: 1 to (total number of bits per sample)
Default: the total number of bits per sample

NCOLS
Width of image, in pixels
Allowed values: 1 to 2^31
Default: (none – this keyword is required)

NROWS
Height of image, in pixels
Allowed values: 1 to 2^31
Default: (none – this keyword is required)

PIXEL_HEIGHT
Same as YDIM

PIXEL_WIDTH
Same as XDIM

PROJECTION_NAME
Name of projection system, for georeferencing
Allowed values: (string)
Default: (none)

PROJECTION_PARAMETERS
Numeric projection parameters, for georeferencing
Allowed values: (1 to 15 floating point values)
Default: (none)

PROJECTION_ZONE
Projection zone number, for georeferencing
Allowed values: (int32)

Default: (none)

RADIUS

Sphere radius, for georeferencing

Allowed values: (floating point)

Default: (none)

ROWS

Same as NROWS

SEMI_MAJOR_AXIS

Semimajor axis, for georeferencing

Allowed values: (floating point)

Default: (none)

SEMI_MINOR_AXIS

Semiminor axis, for georeferencing

Allowed values: (floating point)

Default: (none)

SKIPBYTES

Number of bytes at top of image file to skip

Allowed values: 0 to (image size in bytes)

Default: 0

This can be used for raw formats which contain a fixed number of "header" bytes at the top of the data file

SPHEROID_NAME

Name of projection system, for georeferencing

Allowed values: (string)

Default: (none)

TOTALROWBYTES

Not currently supported – value is ignored

UL_X_COORDINATE

Same as ULXMAP

UL_Y_COORDINATE

Same as ULYMAP

ULXMAP

Upperleft *x*-position, for georeferencing

Allowed values: (any floating point value)

Default: (none – value is determined by LTIImage)

ULYMAP

Upperleft *y*-position, for georeferencing

Allowed values: (any floating point value)

Default: (none – value is determined by LTIImage)

WORDLENGTH

Number of bytes per sample

Allowed values: 1 or 2

Default: 1, unless overridden by DATATYPE

XDIM

Size of pixel in *x*-direction, for georeferencing

Allowed values: (any floating point value)

Default: (none – value is determined by LTIImage)

YDIM

Size of pixel in *y*-direction, for georeferencing

Allowed values: (any floating point value)

Default: (none – value is determined by LTIImage)

This is expected to be a positive value

Additional Notes

- These keywords are required: NBANDS, NCOLS, NROWS. All other keywords have default values.
- If dynamic range is used, either both DYNAMICRANGEMIN and DYNAMICRANGEMAX must be set or both DYNAMICRANGEWINDOW and DYNAMICRANGELEVEL must be set.

Supported Keywords (Writer)

The `LTIBBBImageWriter` class only writes a subset of the above keywords to the header file. The keywords used are:

`BYTEORDER`

Determined by constructor

`COLORSPACELAYOUT`

Only set if image colorspace is

`CMYK`

Set to `BIP`, `BIL`, or `BSQ` as per constructor argument

`NROWS`

Height of scene being written

`NCOLS`

Width of scene being written

`NBANDS`

Number of bands in image

`NBITS`

Bits of precision of image samples

`ULXMAP`

Determined by scene/image

`ULYMAP`

Determined by scene/image

`XDIM`

Determined by scene/image

`YDIM`

Determined by scene/image

Example

This BBB header files describes a 640x480 color image, using 16 bits per sample.

```
NROWS 480

NCOLS 640

NBANDS 3

DATATYPE U16
```

Extensions

Note that some of the header syntax supported by the MrSID SDK may not be supported by other vendors' BIP/BIL/BSQ implementations. In particular, the following features and keywords may be somewhat specific to LizardTech:

- interpretation of .bbb extension as meaning layout of BIP
- support for comment lines
- the `COLORSPACE` keyword
- the `DYNAMICRANGEMIN`, `DYNAMICRANGEMAX`, `DYNAMICRANGEWINDOW`, and `DYNAMICRANGELEVEL` keywords

GeoTIFF Metadata for JPEG 2000

Following is a copy of the first draft of *The "GeoTIFF Box" Specification for JPEG 2000 Metadata*.

*** DRAFT ***

The "GeoTIFF Box" Specification for JPEG 2000 Metadata

Version 0.0

30 April 2004

*** DRAFT ***

Michael P. Gerlek, editor

mpg(AT)lizardtech(DOT)com

LizardTech, Inc.

1008 Western Ave Suite 403

Seattle, WA 98104 USA

0 Editorial Notes

This is a DRAFT document. Comments welcome.

Sections in [brackets] are editorial asides, calling out specific questions or details to be resolved.

0.1 Disclaimers and Copyrights

This document is copyright © 2004 Celartem, Inc., doing business as LizardTech. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct or commercial advantage and this copyright notice appears.

LizardTech assumes no liability for any special, incidental, indirect or consequences of any kind, or any damages whatsoever resulting from loss of use, data or profits, whether or not advised of the possibility of damage, and on any theory of liability, arising out of or in connection with the use of this specification.

1.0 Introduction

This specification describes a GeoTIFF-based method for adding geospatial metadata to a JPEG 2000 file. While the actual specification is not at all complex or hard to implement, there has been some confusion about what it actually is, what restrictions apply to its use, etc. Enough people have asked about it that we considered it worthwhile to put something on paper and have it reviewed by some independent developers.

Note the intent of this document is only to codify existing practice as of this writing; no modifications or extensions to this specification are planned or expected.

1.1 History, Background

Mapping Science Inc. (MSI) provided the first implementation of this specification in their GeoJP2(tm) encoder product in 2003. At that time, the definition of the specification was available only under certain licensing restrictions from MSI.

LizardTech, Inc. acquired the assets of Mapping Science in 2004. It is LizardTech's position that this specification should be publicly available for anyone to implement. Neither JPEG 2000 nor GeoTIFF are proprietary standards; the combination should not be either.

Note that "GeoJP2" is a trademark that refers to the original MSI encoder (now owned by LizardTech). Please don't use the term "GeoJP2" to refer to this metadata specification -- we don't want this specification to be encumbered by trademark issues.

1.2 Box Structure

Two UUID boxes are defined.

The first, called the GeoTIFF box, contains a degenerate GeoTIFF file as described in section 2.

The second, called the world file box, contains the usual six doubles as in an external world (.wld) file, plus some additional version information. This is described in section 3. Presence of the world file box is optional.

This specification assumes a compliant JP2 file with only one codestream box.

2.0 The GeoTIFF Box

The GeoTIFF box provides a simple mechanism for a JP2 file to have the same level of geospatial metadata as is provided by the widely supported GeoTIFF standard, using the normal GeoTIFF implementations.

2.1 UUID

The UUID for this box is

```
static unsigned char geotiff_box[16] =

  {

    0xb1, 0x4b, 0xf8, 0xbd,

              0x08, 0x3d, 0x4b, 0x43,

              0xa5, 0xae, 0x8c, 0xd7,

              0xd5, 0xa6, 0xce, 0x03

  };
```

2.2 Box Contents

This box contains a valid GeoTIFF image. The image is "degenerate", in that it represents a very simple image with specific constraints:

- the image height and width are both 1
- the datatype is 8-bit
- the colorspace is grayscale
- the (single) pixel must have a value of 0 for its (single) sample

The TIFF image is to be encoded in little endian format. [Note that an early and possibly unreleased MSI encoder seems to have used big endian form, but the GeoTIFF data appears corrupt.]

The intent is that any compliant GeoTIFF reader/writer will be able to read/write this image.

Note that the TIFF image properties -- width, bitdepth, etc -- do NOT reflect the image properties of the JP2 image. These image properties are not to be used in the interpretation of the geospatial metadata.

Other TIFF image properties maybe present; if so, they should be similarly ignored.

[If the TIFF image properties do not meet the constraints above, the geospatial information represented by this box should be considered to be undefined.]

The GeoTIFF image may contain TIFF metadata tags. These should be ignored; they do not apply to the JP2 image.

The GeoTIFF image may contain any number of GeoTIFF keys, as allowed by the GeoTIFF standard. These keys define the geospatial metadata of this box and of the JP2 image itself.

3.0 The World File Box

[This description is based on my reading of the MSI source code; I will have to flesh this out as I become more confident of it. Alternatively, at some point if I can get the code suitably cleaned up I may just publish the implementation itself... If anyone needs this information now, feel free to contact me.]

The world file box contains one or more "chunks" of metadata data of various types. The most common chunk type encodes the normal six-doubles style of geopositioning information found in the conventional external world files often used with some image types.

[Other chunk types were used to indicate the operating system the JP2 image created on, the MSI command line used, and arbitrary user-defined bytes. It is not clear if these other chunk types were ever widely used or not. I will attempt to define these other chunks, but they use should be considered to be OBSOLETE and not used in future implementations.]

3.1 UUID

The UUID for this box is

```
static unsigned char world_box[16] =

  {

    0x96, 0xa9, 0xf1, 0xf1,

              0xdc, 0x98, 0x40, 0x2d,

              0xa7, 0xae, 0xd6, 0x8e,

              0x34, 0x45, 0x18, 0x09

  };
```

3.2 Box Contents

The first bytes in the box, which we will call the "header", give some versioning information and the number of chunks in the box. The "chunks" themselves then follow, laid out as contiguous bytes. The box ends with a small of amount of data in what we will call the "footer".

3.2.1 Header Format

Bytes 0-3: 'M', 'S', 'I', 'G'.

Bytes 4-5: major and minor version numbers (shifted and packed together) - the actual values of these numbers may not be used for anything

Bytes 6-13: feature set flags - current values are {1, 0, 0, 0, 0, 0, 0, 0} - first flag controls interpretation of the world file values, see section 3.2.2.1 - second flag indicates windows or linux build of encoder; not used for anything? - remaining flags undefined (leave as zero)

Byte 14: number of chunks in the box?

Byte 15: next box?; apparently always 0

The next bytes in the file correspond to the serialization of each chunk. There may be zero or more chunks present; each chunk type may appear at most once.

3.2.2 Chunk Format

The chunk format appears to be a simple header of six bytes, followed by the chunk-specific data.

Byte 0: chunk index, used to indicate type of chunk

Byte 1: chunk properties [not used?]

Bytes 2-5: chunk length (including these six bytes) - stored as little-endian unsigned int

3.2.2.1 World Chunk Format

Byte 0: chunk index (equal to 0)

Byte 1: chunk properties

Bytes 2-5: chunk length (equal to 2 + 4 + 6*8) - stored as little-endian unsigned int

Bytes 6-13: x scale (resolution)

Bytes 14-21: x rotation

Bytes 22-29: y rotation

Bytes 30-37: y scale (resolution)

Bytes 38-45: x upper-left

Bytes 46-53: y upper-left

The six geo values are stored as little-endian doubles.

The first feature flag (defined in section 3.2.1) control the interpretation of these values. According to the comments in the source code, if set to 1 then the following applies:

> *"This was instituted with version 1.03.11 (May 15, 2003) to signify that we clarified the definition of the georeferencing data and found out that that data represents the upper left corner of the upper left pixel, not the center as we had thought, so the [world chunk values are] not equal to the geotiff data, but is shifted by 0.5*scale to the center of the pixel."*

If the world chunk is present, these values should override the corresponding values in the GeoTIFF box.

3.2.2.2 User Data Chunk Format

Byte 0: chunk index (equal to 1)

Byte 1: chunk properties

Bytes 2-5: chunk length - stored as little-endian unsigned int

Bytes 6..n: user-defined data (chunk length minus 6 bytes)

3.2.2.2 Command-Line Chunk Format

Byte 0: chunk index (equal to 2)

Byte 1: chunk properties

Bytes 2-5: chunk length - stored as little-endian unsigned int

Bytes 6..n: command-line string (chunk length minus 6 bytes)

3.2.2.2 OS Data Chunk Format

Byte 0: chunk index (equal to 3)

Byte 1: chunk properties

Bytes 2-5: chunk length - stored as little-endian unsigned int

Bytes 6..n: unknown data (chunk length minus 6 bytes)

3.2.3 Footer Format

The footer, coming after the chunk data, is six bytes long.

Byte 0: set to 0xff

Byte 1: set to 0x00

Bytes 2-5: file offset of next world file box?

Georeferencing of NITF Imagery

Typically, TREs are used to provide the necessary georeferencing. As the current SDK release does NOT provide TRE support, however, some additional documentation may be helpful to some developers.

This technical note provides commentary on how the SDK reads and writes “positioning information” in our NITF support – including basic world file information (upper left points, resolution), coordinate system information (WKT data), and positioning of image segments relative to each other.

Geo Support Features in NITF

Independent of any of our implementation issues, the file format itself can provide georeferencing support in the following ways.

IGEOLO

Each image segment may contain an ICORDS and an IGEOLO field. Together these fields give low-precision positioning data for the four corner points of the image. The standard makes it very clear that this data is only for cataloging purposes and should not be used for accurate georeference positioning.

The ICORDS field indicates whether the IGEOLO points are in UTM, UTM/MGRS, or WGS84. The four pairs of numbers are expressed in either lat/long (decimal degrees) or UTM form, as appropriate.

NOTE: The standard does not address the coordinate system of the image data itself. IGEOLO data only indicates what the boundary points of the image would be, if the image were to be projected into WGS84.

Use of the IGEOLO data is optional. For more information see MIL-STD-2500C, table A-3 (page 86).

CCS

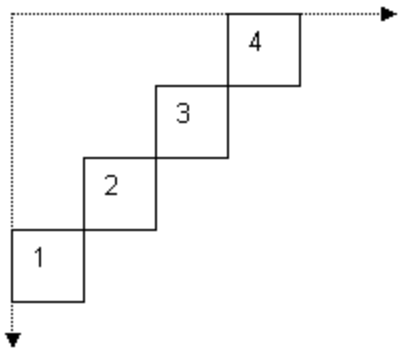
Each image segment in the file must be positioned according to the Common Coordinate System, a virtual, underlying grid which is used to collocate all the segments in the file. The grid is expressed using integer values with (0,0) at the upper-left. The CCS info for a segment has four parts:

ILOC: the upper-left (*x*,*y*) offset of the segment on the CCS grid. This may be an “absolute offset” (which is relative to (0,0)) or it may be a “relative offset” (which is relative to another segment’s ILOC), depending on the IALVL value.

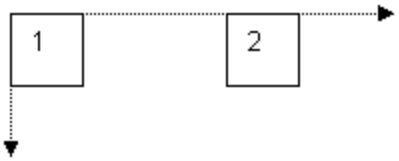
IDLVL: display level, or Z-ordering. Each segment must have a unique IDLVL value. The “lowest” segment has the lowest number. (**Note:** segment numbering starts at 1, not 0.)

IALVL: attachment level. A segment’s ILOC position on the CCS grid may be expressed relative to another segment, by setting its IALVL value to the IDLVL level of the that other segment. If IALVL is zero, the ILOC position is relative to (0,0).

The ILOC values must be in the range -9,999 to 99,999 (5-digit field). This means that certain mosaics cannot be represented. For example, consider four 40Kx40K images laid out as follows:



The ILOC for images 1 and 4 cannot be expressed relative to (0,0) as they would require offsets of 120,000, which is out of range. They cannot be expressed relative to the origin of images 2 and 3 either, as they would require offsets of -40,000, which is also out of range. A more obvious – but less likely – mosaic which cannot be represented is simply two images spread far apart:



where the images are both 1Kx1K and image 2 is 150,000 units from the origin.

Use of the CCS data is required. For more information see MIL-STD-2500C, table A-3 (page 97).

BLOCKA TRE

This metadata extension contains four fields which repeat the IGEOLO fields for the segment but provide higher precision. They are always expressed in WGS84 lat/long, as degrees/minutes/seconds or decimal degrees.

NOTE: As with IGEOLO, the BLOCKA data does not say anything about the coordinate system of the image data itself.

Use of the BLOCKA TRE is optional. Within BLOCKA, the LOC fields are themselves optional. For more information see STDI-0002, section 8.3.4 (page 83).

World Files

Although not part of any NITF standard, the normal LizardTech world file conventions nonetheless can be used to specify the upper-left coordinates and resolution of the image.

Image Segment CRSes

A careful consideration of the above will reveal that, given the data available, *the SDK has no way to determine the coordinate system of any image segment* absent any TRE support. It may know where the image is, for cataloging or indexing purposes, but it knows nothing about the data itself.

This could be addressed by adding our own “proprietary” TRE or Text Segment which would contain our WKT string. Such extensions would go against the intent of the standard, however. Absent the

TRE information, in the near term we suggest that image segments should generally be projected into the common WGS84 space, where possible and practical.

Implementation Support

This section describes the rules governing how the above fields are interpreted and represented within the SDK for NITF files. We consider the four cases of reading and writing both single and multiple segment files.

In the following discussion, recall that within the SDK framework, an “image” (class `LTIImage`) contains a “geo coord” object (class `LTIGeoCoord`) which contains three main pieces of data:

- the (*x*,*y*) upper-left position
- the (*x*,*y*) resolution
- the WKT string

Single Segment Reading

The SDK supports the ability to read any one single segment from a NITF file.

As discussed above, the SDK can never know the projection system of an image segment. Nonetheless, the geo position is determined as follows:

1. First, we fall back to the normal default of (0,h) for the upper-left position. The WKT string is left empty.
2. If set, we interpret the IGEOLO/ICORDS field to determine the *x*,*y* corner position.
3. If present, a world file will then override the default positions.
4. We then offset the geo position of the image based on the ILOC data for the segment. (**Note:** The ILOC data is multiplied by the resolution, before adding it to the position)

NOTE: The world file serves to describe the position of CCS (0,0), not any individual segment in particular.

Multiple Segment Reading

The SDK only supports the ability to read a single segment. To read multiple segments, to be displayed as a mosaic of tiles, the SDK's mosaic filter stage should be used.

Single Segment Writing

When writing an image, the SDK will always have good georeferencing information, but we may or may not have good WKT data.

- set ILOC to (0,0), IDLVL to 1, and IALVL to 0
- if we have WKT data, and if the points can be reprojected into WGS84, then
    - write the IGEOLO data
    - write the BLOCKA TRE
- write a world file (at user’s discretion)

Multiple Segment Writing

When writing multiple segments, the SDK is given both the constructed mosaic and each of the individual segments.

- ILOC data:
  - the ILOC (x,y) of each segment is set based on the segment's georeferencing information, relative to the georeferencing information of the origin of the overall mosaic (and scaled appropriately by resolution)
  - the IDLVL is set to the segment number
  - the IALVL is set to 0
- as in the single segment case, if we have WKT data, and if the points can be reprojected into WGS84, then
  - write the IGEOLO data
  - write the BLOCKA TRE
- write a world file (at user's discretion)
  - using the georeferencing information of the overall mosaic

NOTE: Because we keep IALVL at 0, we're limited to having any one segment at most 100K units from the CCS origin. That is, we do not yet support the ability to position one segment relative to another.

Writing JPEG 2000 Segments

When writing JPEG 2000 segments, you may use the "boxed" format. This means the classical metadata box is typically present; if so, this is used to recover the georeferencing information for the segment per the normal SDK processes.

Metadata Tags

Encoded within each MrSID (and JPEG 2000) file is a set of metadata tags, used to convey additional information about the image. These tags are similar to TIFF tags, although the actual encoding mechanism is different. The mrsidinfo tool can be used to dump the metadata for an image.

The standard metadata tags used by all MrSID images, referred to as the "classical" tags, are listed below. Other than the "start" and "end" tags, the tags may appear in any order.

For mosaicked images, the metadata encoded with the image generally corresponds to the first image in the mosaic. The filename metadata tag contains a comma-separated list of the input files. The metadata refers to the image as presented to the encoder; this means that any filter or transforms operations will be correctly accounted for.

The metadata is supplemental information only. Some of this metadata is also contained in the `LTIImage` class; in general, querying the `LTIImage` class is preferred to evaluating the metadata, for example for information such as geographic position, background color, etc.

General Tags

The tags in the table below apply to MrSID encoding.

General Metadata Tags

Tag	Description	Notes
IMAGE::BITS_PER_SAMPLE	Number of bits per sample (uint16)	
IMAGE::COLOR_SCHEME	Colorspace of image (uint32)	Values: 0 for RGB, 3 for grayscale, 2 for CMYK, 10 for multispectral
IMAGE::DATA_TYPE	Datatype of samples in image (uint32)	Values: 0 for unsigned 8-bit int, 1 for 32-bit float, 2 for unsigned 16-bit int
IMAGE::DEFAULT_DATA_VALUE	Sample values for background pixel	Stored as an array of values, one for each band, in order. The values are stored in the datatype of the samples **NOTE:** With older images containing uint8 data, the tag `IMAGE::NO_DATA_VALUE` may be used
IMAGE::DYNAMIC_RANGE_WINDOW	Custom contrast setting (double)	Represents the size of the range of the data. This tag can have per-band values for MG4 images
IMAGE::DYNAMIC_RANGE_LEVEL	Custom brightness setting (double)	Represents the midpoint of the range of the data. This tag can have per-band values for MG4 images
IMAGE::ENCODING_APPLICATION	Name and version of encoding application	Required tag
IMAGE::ENCODING_COMMENT	Context of encoding operation	
IMAGE::EOM	End of metadata	
IMAGE::FORMAT	Text representation of the format	Examples: "MrSID/MG4", "JPEG 2000"
IMAGE::HEIGHT	Height of the image, in pixels (uint32)	
IMAGE::INPUT_FILE_SIZE	Size of the input image or mosaic in bytes (double)	
IMAGE::INPUT_FORMAT	Name of input image type (string)	
IMAGE::INPUT_LUT	Color lookup table	Stored as an array of 256*3 values, one value for each band (R,G,B) for each of the 256 entries in the table
IMAGE::INPUT_NAME	Filename of the input image (string)	
IMAGE::LTI_ESDK_VERSION	The LizardTech SDK version number	
IMAGE::MODIFICATIONS	Changes to input pixel data	See list following this table for values.
IMAGE::NO_DATA_VALUE	Sample values for background pixel	Used only by older MrSID images; see `IMAGE::DEFAULT_DATA_VALUE`
IMAGE::QUANTIZATION_SCALE	The quantization scale is a measure of the precision of the image data. The pixel values of the compressed output image are accurate to within half of the scale value.	Only floating point MG4 images use the quantization compression type.
IMAGE::SOM	Start of metadata	
IMAGE::TRANSPARENT_	Sample values for the "no data" pixel	Stored as an array of values, one for each

Tag	Description	Notes
DATA_VALUE		band, in order. Values are stored in the datatype of the samples
IMAGE::WIDTH	Width of the image, in pixels (uint32)	
IMAGE::X_RESOLUTION	Georeferencing pixel resolution in x-direction (double)	
IMAGE::XY_ORIGIN	Georeferencing (x,y) location for the center of the upper left corner pixel	Stored as an array of two doubles
IMAGE::Y_RESOLUTION	Georeferencing pixel resolution in x-direction (double)	

Values for IMAGE::MODIFICATIONS Tag

The following are acceptable as values for the `IMAGE::MODIFICATIONS tag:`

- LOSSLESS – this value will only ever appear by itself, and indicates that the pixels have been losslessly compressed and can be recovered in their original form.
- COMPRESSED – the image has been compressed at a rate that did not preserve the pixels losslessly.
- CROPPED – some of the original extent of the image has been cropped (those pixels have been discarded).
- EMBEDDED – the image has been embedded in a larger extent (new pixels have been added to the image.
- SCALED – the image has been made larger or smaller by stretching, using interpolation or wavelet decomposition.
- MASKED – some pixels from the image have been masked out (made transparent).
- INTERPRETED-ALPHA – the original image had an alpha band which was dropped or reconstructed from transparency information.
- REORDERED-BANDS – the bands represent a reordering and/or subset of the bands as they were in the original image.
- TRANSFORMED-COLORSPACE – the original pixel colors were transformed into a different colorspace.
- CHANGED-DATATYPE – the image datatype is not the same as that of the original.
- ALTERED-COLOR – the image has been color-balanced.
- MOSAICKED – the image was created from more than one source image. In this case, MODIFICATIONS flags from all component tiles have been incorporated into the resulting image.
- REPROJECTED – the image was warped to a different projection system.
- WATERMARKED – a watermark has been added to the image.
- OVERLAID – the image has had other information (text or vector data) superimposed on it.
- QUANTIZED – the image was created with the quantization compression method. Quantization is a lossy compression method that reduces the precision of pixel values in the image. Quantization is required to compress floating point MG4 images.

Area of Interest (AOI) Tags

The tags in the table below apply only to images encoded with areas of interest:

Area of Interest (AOI)Tags

Tag	Description	Notes
IMAGE::AOI::n::REGION::VECTOROVERLAY	Name of vector overlay file, if any – (string)	
IMAGE::AOI::n::REGION::VECTOROVERLAY_LAYER	Layer number from vector overlay file (if one is used) – (integer)	
IMAGE::AOI::n::REGION::X	Upper left X pos of region – (integer)	
IMAGE::AOI::n::REGION::Y	Upper left Y pos of region – (integer)	
IMAGE::AOI::n::METHOD	The AOI method used – (string)	Values: • "shift inner" • "shift outer" • "weight"
IMAGE::AOI::n::WEIGHT	Weight value used – (double)	
IMAGE::AOI::n::MAGNIFICATION	Magnification at which AOI was applied – (double)	
IMAGE::AOI::n::NAME	Optional name of AOI region – (string)	
IMAGE::AOI::n::COMMENT	Optional comment for AOI region – (string)	
IMAGE::AOI::n::URL	Optional URL referring to AOI region – (string)	

MG2-Only Tags

The classical tags in the table below apply to MG2 images only:

MG2-Only Tags

Tag	Description	Notes
IMAGE::COMPRESSION_BLOCK_SIZE	Block size used in MrSID encoding (uint32)	
IMAGE::COMPRESSION_GAMMA	G-weight value used in MrSID encoding (float)	
IMAGE::COMPRESSION_VERSION	Version of encoder used (array of 3 sint32 values)	
IMAGE::COMPRESSION_WEIGHT	Weight value used in MRSID encoding (float)	
IMAGE::CREATION_DATE	Date and time of image encoding (string)	
IMAGE::COMPRESSION_NLEV	Number of zoom (resolution) levels in the image (uint32)	
IMAGE::STATISTICS:MAXIMUM	Maximum sample values for each band in the input image (array of values)	The number and datatype of the values correspond to the number of bands and sample type of the image
IMAGE::STATISTICS:MINIMUM	Maximum sample values for each band in the input image (array of values)	The number and datatype of the values correspond to the number of bands and sample type of the image
IMAGE::STATISTICS:MEAN	Average value of all samples for each band (array of doubles)	
IMAGE::STATISTICS:STANDARD_DEVIATION	Standard deviation of	

Tag	Description	Notes
	all samples for each band	
IMAGE::TARGET_ COMPRESSION_RATIO	Compression ratio used for encoding (float)	For MG2, this only approximates the actual compression ratio achieved

Other Metadata Tags

When using GeoTIFF input images, the GeoTIFF metadata tags are copied directly into the MrSID or JPEG 2000 file. When using ERDAS IMAGINE and USGS DOQ metadata, certain other custom metadata tags are inserted as well.

Negative *y*-Resolutions

The *y*-resolution ("YRES" or "YDIM") of an image can be either positive or negative, depending on what type of image is being used and which interface is being used to query the resolution. This note provides some background on this issue.

First, some definitions:

By "negative YDIM" (hereafter, "-YDIM"), we mean that (0,0) is in the LOWER LEFT and extends up and to the right to (*w*,*h*). This is the normal Cartesian representation you learned in high school algebra.

By “positive YDIM” ("+YDIM"), we mean that (0,0) is in the UPPER LEFT and extends down and to the right to (*w*,*h*). This is a common representation in computer graphics.

Figure 1 shows the default geo coordinates for a 640x480 image with no internal georeferencing, using -YDIM conventions. Note that while the rows of the image will proceed visually down the page, the *y*-value of the rows decreases, from 479 down to 0.

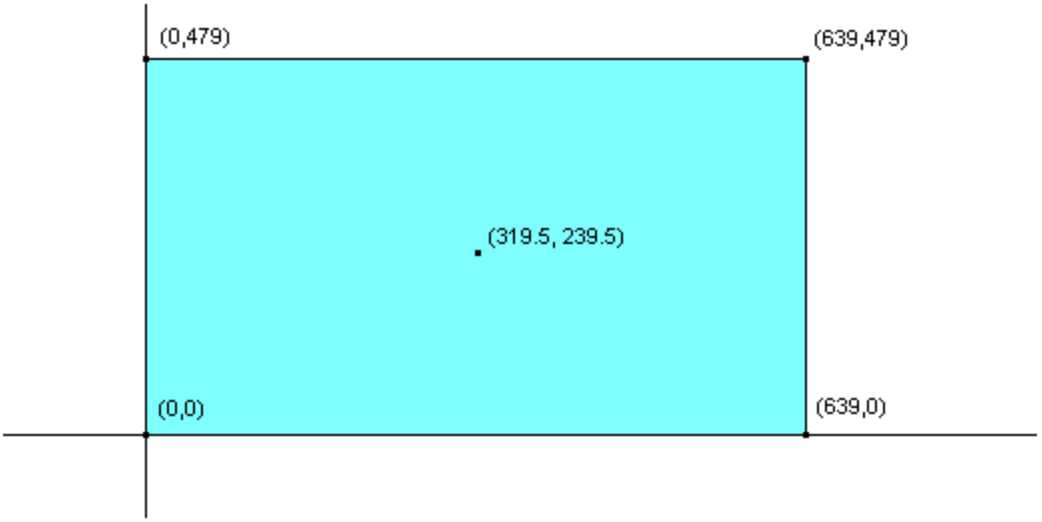


Figure 1: 640w x 480h

The MrSID SDK follows the -YDIM convention. This means that the function `LTIGeoCoord::getYRes()` will generally return a value less than zero. Image reader and writer classes must respect this: when importing or exporting geospatial resolutions, care must be taken to adjust the sign if needed in order to match the SDK requirements and the external file format requirements.

Additional notes

The classical MrSID metadata tag `IMAGE::Y_RESOLUTION` is stored with a positive sign, for historical reasons. If you access this metadata value directly, you must multiply the value by -1.0 before using it.

As a debugging aid, the mrsidinfo tool can be used to show the georeferencing of the image, including all four corner points.

World files expect the use of negative YDIMs (fourth line).

BBB files use positive YDIMs in their headers. (The `LTIBBBImageReader` class will internally negate the value to satisfy the SDK requirements.)

YDIM and XDIM should never be 0.0.

Nodata and Background Pixels

This note describes how the MrSID SDK implements the concepts of "nodata" and "background" pixelsPixels.

NOTE: The following is not applicable to the MG4 format, in which transparency is implemented through the use of alpha bands.

Definitions

Nodata

The nodata pixel of an image is used to indicate which pixels within the bounds of the image are to be ignored when performing an operation on the image, for example mosaicking, rendering, etc. The default is to have no nodata pixel associated with an image; in this case, all pixels in the image represent valid data.

Background Data

The background pixel of an image is used to provide valid pixel values outside of the bounds of the image. In rendering the image, if the specified scene exceeds the boundaries of the image, the background pixel can be used by an application to fill in the portion of the output buffer beyond the image boundary.

The default is to have no background pixel associated with an image. In this case, when pixels "outside of" the image are required, the black pixel consisting of a zero in each band will be used. For CMYK images, however, the pixel consisting of the maximum sample values in each band will be used.

While the `LTIImageStage::read()` function does not allow for decoding outside the bounds of the image, the background pixel and the nodata pixel are used when mosaicking multiple images together.

Transparency

The MrSID SDK does not support transparency operations more generally than the above simple nodata support, for example via bitmasks, clipping paths, or alpha blending.

MrSID Metadata Tags

For compatibility reasons, the classical metadata tag names used in MrSID do not correlate well with the definitions given above.

The tags used are:

`IMAGE::NO_DATA_VALUE`

Represents the background pixel (used by older MrSID images)

`IMAGE::DEFAULT_DATA_VALUE`

Represents the background pixel (used by newer MrSID images)

`IMAGE::TRANSPARENT_DATA_VALUE`

Represents the nodata pixel

Figure 1 shows two images, A and B, that are to be mosaicked together into image C, such that B lies precisely on top of A. Both images have a background color of red and a nodata color of blue. In the mosaic C, the upper rectangle is yellow because the data comes from the nodata region of B, which allows the A image to “show through”. The lower rectangle is red, however, because we again "see

through" B and onto the corresponding region of A; that region is set to the nodata color, therefore the red background of A is used.

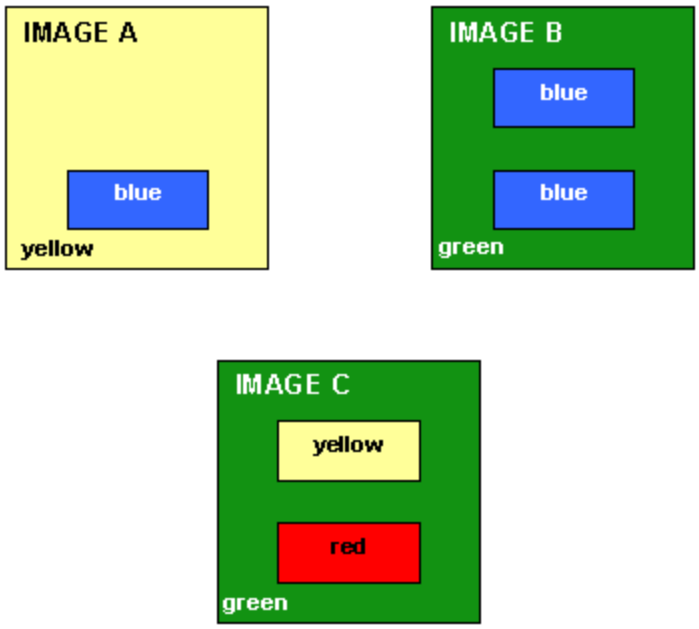


Figure 1: Nodata and background color in overlapping images

Notes on the Mosaicking Process

When mosaicking two images together, the nodata pixel of the image "on top" is honored. Many images do not have a nodata pixel explicitly set, however, so the following rules are applied:

1. If the image has a nodata pixel set, use that.
2. Otherwise, if the image has a background pixel set, use that.
3. Otherwise, use the "black" pixel.

The motivation for this treatment of nodata is to give the best looking image for the case when two images are being mosaicked together such that only their corners overlap, resulting in a large image with a lot of "black" background. If this resulting mosaic is then laid on top of some other even bigger image, we want to be able to treat that black background as nodata. (**Note:** this solution does allow for speckling in the image, as the original two squares might have legitimate black pixels within their data.)

If either background or nodata is to be set for an image, set both properties to the same value. When mosaicking images together, all images should have the same background and nodata color.

Appendix B - Company and Product Information

This chapter contains information about LizardTech and its products as well as copyrights, trademarks and other information pertaining to this LizardTech software.

About LizardTech

Since 1992, LizardTech has delivered state-of-the-art software products for managing and distributing massive, high-resolution geospatial data such as aerial and satellite imagery and LiDAR data. LizardTech pioneered the MrSID® technology, a powerful wavelet-based image encoder, viewer, and file format. LizardTech has offices in Seattle, Denver, London and Tokyo and is a division of Celartem Technology Inc. For more information about LizardTech, visit www.lizardtech.com.

Other LizardTech Products

Thank you for using LizardTech® software. We at LizardTech are glad to have you as a customer. While you're "in the shop," explore LizardTech's other products for managing high-quality geospatial images and LiDAR data.

GeoViewer

Efficient Viewing and Exporting of MrSID and JPEG 2000 Layers

GeoViewer is LizardTech's free, standalone application for viewing geospatial imagery, vector overlays and LiDAR data. GeoViewer enables you to combine, view and export visual layers from varied sources, such as local repositories, Express Server catalogs, and WMS and JPIP servers. GeoViewer supports a wide range of input formats and exports to GeoTIFF, PNG and JPEG. It's the most efficient means of viewing MrSID and JPEG 2000 images.

For more information about GeoViewer visit http://www.lizardtech.com/downloads/category/#viewers.

ExpressView Browser Plug-in

Fast and Easy Viewing of Large Images

ExpressView™ Browser Plug-in enables you to view, navigate and print MrSID and JPEG 2000 imagery in Internet Explorer or Firefox. Like GeoViewer, ExpressView enables you to save a portion of an image in a number of other image formats. ExpressView Browser Plug-in is quickly downloaded, easily installed, and free for individual use. It's the most convenient way to view MrSID and JPEG 2000 imagery over networks!

For more information about ExpressView Browser Plug-in visit http://www.lizardtech.com/downloads/category/#viewers.

GeoExpress

The Industry's Best Image Manipulation and Compression Software

With powerful tools for reprojecting, color balancing, and mosaicking, GeoExpress® software is the industry's choice for manipulating and compressing geospatial imagery to industry standard formats. You can configure Express Server and Spatial Express® software directly from GeoExpress, which makes it the ideal command center for your storage and distribution workflows.

For more information about GeoExpress visit www.lizardtech.com/products/geo/.

LiDAR Compressor

LiDAR Data Meets the MrSID Format

LizardTech LiDAR Compressor™ software enables you to turn giant point cloud datasets into efficient MrSID files that retain 100 percent of the raw data at just 25 percent or less of the original file size (lossless compression). If storage requirements are critical, you can reduce your LiDAR file sizes by 90 percent or more by choosing a higher compression ratio and letting LiDAR Compressor select the best way to reach a desired file size (lossy compression). Unlike raw LAS or ASCII data, LiDAR files compressed to MrSID are easily managed resources you can extract derivatives from again and again.

For more information about LiDAR Compressor visit www.lizardtech.com/products/lidar/.

Express Server

Image Delivery Software for Geospatial Workflows

LizardTech Express Server software is the best solution for distributing imagery in MrSID or JPEG 2000 format. With Express Server, users on any device access imagery faster, even over low-bandwidth connections. Express Server is faster, more stable and easier to use than any other solution for delivering high-resolution raster imagery.

For more information about Express Server visit http://www.lizardtech.com/products/exp/.

Glossary

Following are descriptions of some terms, phrases and acronyms used in this documentation that you may not be familiar with.

Background color

The pixel value that defines the color of the image outside the pixel extents of the image. This property is most often used in mosaicking. *See also "Nodata color" and "Transparency color".*

Band

A set of samples corresponding to one spectral component of an image e.g. the “red” band of an RGB image. Also known as *component*. There is a weak correlation between the colorspace and the number of bands: grayscale is 1, RGB is 3, CMYK is 4. (Multispectral is the exception, as it has one or more bands.)

BBB

Name of a file format for raw images, short for "BIP/BIL/BSQ."

BIL

One of the three “BBB” formats; abbreviation for "band-interleaved by line." In this data layout, the samples of the image are laid out per line, one sample at a time. Figure 1 shows a 3-banded image of width 3 and height 2 using a BIL layout.

R_{00}	R_{01}	R_{02}
G_{00}	G_{01}	G_{02}
B_{00}	B_{01}	B_{02}
R_{10}	R_{11}	R_{12}
G_{10}	G_{11}	G_{12}
B_{10}	B_{11}	B_{12}

Figure 1: BIL Layout

BIP

One of the three “BBB” formats; abbreviation for "band-interleaved by pixel." In this data layout, the samples of the image are laid out per pixel, one sample at a time. Figure 2 shows a 3-banded image of width 3 and height 2 using a BIP layout.

R_{00}	G_{00}	B_{00}	R_{01}	G_{01}	B_{01}	R_{02}	G_{02}	B_{02}
R_{10}	G_{10}	B_{10}	R_{11}	G_{11}	B_{11}	R_{12}	G_{12}	B_{12}

Figure 2: BIP Layout

BSQ

One of the three “BBB” formats; abbreviation for "band-sequential." In this data layout, the samples of the image are laid out per band, one sample at a time. Figure 3 shows a 3-banded image of width 3 and height 2 using a BSQ layout.

R_{00}	R_{01}	R_{02}
R_{10}	R_{11}	R_{12}
G_{00}	G_{01}	G_{02}
G_{10}	G_{11}	G_{12}
B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}

Figure 3: BSQ Layout

Byte order

See "Endianness".

Component

See "Band".

Composite image

With respect to MG3 and MG4, an image made up of several internal images (known as *tiles*). *See also "MG3" and "MG4".*

Compression

An operation that creates a new image file from an original image file such that the file size of the new image is smaller. The reduction in file size may be at the expense of some image quality. (Note that the file size is what is reduced, not the dimensions of the image itself.) *See also "Lossless" and "Lossy".*

Compression ratio

The amount or degree of reduction in file size, expressed as the ratio of the nominal file size to the target size. Because nominal size is used, compression ratios from compressed formats, e.g. JPEG, can be misleading. For example, consider a 50MB raw file compressed to a 10MB JPEG file: this is 5:1 compression ratio. Performing a further 10:1 MrSID compression on that 10MB JPEG file will produce a 5MB MrSID file, not a 1MB file, because the 10:1 is relative to the image's nominal size and not it's current physical file size. *See also "Nominal image size".*

Delegate

An abstract class used to implement callbacks. The MrSID SDK uses delegates for situations such as detecting interrupts or reporting progress during a decode operation.

Dynamic range

The range values of the samples of an image.

Dynamic range can be expressed in two forms, "min/max" and "window/level". In the first form, the range is expressed using minimum and maximum values. In the second form, the range is expressed with a "window" value equal to the size of the range, for example (max-min+1), and a "level" value equal to the midpoint of the range, for example (min+max)/2. Window and level correspond roughly to the concepts of "contrast" and "brightness", respectively.

Some images, notably those with 16-bit data, often display as "very dark" because the full 16-bit range of the sample size is not used by the actual sample values; dynamic range information can be used to scale or stretch the data for better presentation. The figure below illustrates the dynamic range of a 10-bit image (log scale).

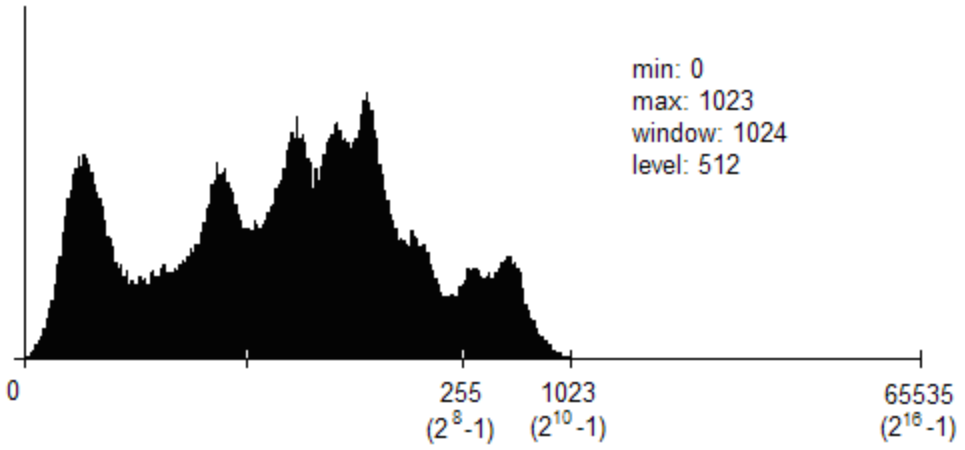


Figure 4: Dynamic range of a 10-bit image (log scale)

Endianness

the order in which bytes in a multibyte word are stored. In a *big endian* machine the most significant byte of the word is stored at the lowest address of the word; in a *little endian* machine, the most significant byte is stored at the highest address. Big endian machines include SPARC and PowerPC. Little endian processors include the Intel x86.

The table below shows the effects of endianness where:

```
unsigned int value = 0x01020304;
unsigned char* p = (unsigned char*)&value;
```

Address	Little Endian Value	Big Endian Value
*p	0x04	0x01
*(p+1)	0x03	0x02
*(p+2)	0x02	0x03
*(p+3)	0x01	0x04

Figure 5: Endianness and Byte Order

Frequency balance

A parameter used in MrSID to determine the emphasis given to edges and "flat" color areas when performing compression. In MG2 encoding this parameter is called *gamma*. A lower value creates more defined edges, while a higher setting creates softer edges. *See also "Weight" and "Sharpness".*

Gamma

See "Frequency balance".

GML

An acronym for *geography markup language*, an XML-based language for representing many types of geographic content. Using GML primitives one can describe coordinate reference systems, units of measure, features, geometries and topologies, coverages, annotations and more.

JPEG

A widely used image format that supports image compression. JPEG is an ISO standard. *See also JPEG 2000.*

JPEG 2000

A new, wavelet-based image format designed for high-quality imagery and advanced imaging workflows. JPEG 2000 is an ISO standard. *See also JPEG, MrSID.*

K-weight

A parameter used in MrSID to determine the emphasis given to the K (black) band of a CMYK image when performing compression.

Level

See "Zoom level".

Lossless

An image that contains a representation of all of the original pixel values; when decoded, a lossless image is mathematically identical to the original. Note that lossless refers only to the image data and implies nothing about the image metadata. *See also "Visually lossless" and "Lossy".*

Lossy

An image that contains an approximation of all of the original pixel values; when decoded, depending on the quality of the compression, a lossy image may appear to be a poor representation of the original, visually indistinguishable from the original, or anywhere in between. The perceived image quality is affected by changes in the sharpness of edges, color balance, reduced resolution, and so forth. *See also "Lossless" and "Visually lossless".*

Magnification

The scale at which an image is represented. Magnification is expressed as a positive floating point value: 1.0 represents the full image (at “one to one”), 0.5 represents a half-scale (lower resolution) version, and 2.0 represents a double-scale (expanded) resolution. The magnification value must be a power of two. *See also "Resolution", "Scale" and "Zoom level".*

Metadata

Information about the image, as opposed to the actual image data (pixels). Metadata can refer to basic image properties such as width, height, colorspace, etc., but generally refers to less fundamental properties such as geospatial position, name of image creator, image date, etc.

MG2

MrSID Generation 2. The earliest released version of the MrSID image format. MG2 is limited to lossy encoding and does not support optimization.

See also "MrSID", "MG3" and "MG4".

MG3

Mrsid Generation 3. The second released version of the MrSID image format. MG3 supports lossless encoding, image optimization, and composite images. *See also "MrSID", MG2 and "MG4".*

MG4

Mrsid Generation 4. The current version of the MrSID image format. MG4 supports most of the features that MG2 and MG3 support, but also supports alpha channels, multispectral and hyperspectral imagery and improved composite mosaics. *See also "MrSID", MG2 and "MG3".*

Mosaic

A composition of two or more images to form a new, larger image. Positioning of the images is generally based on geospatial coordinates.

MrSID

Acronym abbreviating Multiresolution Seamless Image Database. MrSID is a wavelet-based image format designed for large, high-quality geospatial imagery. There are two versions of MrSID, known as MG2, MG3 and MG4. *See also "MG2", "MG3" and "MG4".*

Nodata color

Same as "Transparency color". The pixel value that defines the color of the image that corresponds to a region of the image that has no valid data. Note that a transparency region of the image is contained with the image extents, whereas the background color is used for regions outside of the image. This property is most often used in mosaicking. *See also "Background color".*

Nominal image size

The size of an image, in bytes, defined by the product of

```
image width x image height x number of bands x number of
bytes per sample
```

See also "Physical image size".

Optimization

The process of creating an MG3 or MG4 image from a source MG3 or MG4 image, such that the new image is better suited for some purpose or workflow. The most common optimization is compression; other optimization operations include cropping and removal of resolution levels.

Physical image size

The size of an image, in bytes, as defined by the file size required to represent the image on disk. *See also "Nominal image size".*

Pixel

A set of samples, together making up a single (x,y) point in the image; for example, a 100 x 200 RGB image will contain 20,000 pixels. *See also "Sample".*

Resolution

Definition #1) The scale at which an image is represented as expressed in a wavelet level. The pixel resolution at level 0 (zero) is equal to a magnification of 1.0. *See also "Magnification", "Zoom level" and "Scale".*

Definition #2) Ground units per pixel as used for *georeferencing* (a value stored in the metadata of an image).

Sample

A value representing a magnitude or intensity, for example a red sample in an RGB pixel. A 100 x 200 RGB image will contain 60,000 samples. *See also "Pixel".*

Scale

The resolution or magnification at which an image is represented. Scale is represented as a signed integer, corresponding to the negative of the log of the magnification. That is, magnifications of 1.0, 0.5, and 2.0 are equivalent to scales of 0, 1, and -1, respectively. *See also "Magnification", "Zoom level" and "Resolution".*

Scene

The region of an image to be decoded, as defined by an upper-left position, width and

height dimensions, and magnification.

Sharpness

A parameter used in MrSID to determine the sharpness of boundaries between different areas of an image when performing compression. *See also "Frequency balance" and "Weight".*

Stream

An abstract interface to an arbitrary chunk of data, represented as an array of bytes; examples of data that can be represented as streams include files, memory buffers, and sockets.

Strip height

The number of rows of an image to be processed in each step of an image read operation. Use of smaller strip heights may reduce memory requirements, but at a possible performance loss.

Transparency color

Same as "Nodata color". *See also "Background color".*

Visually lossless

An image that contains a close approximation of all of the original pixel values; when decoded, a visually lossless image appears to a typical viewer as being indistinguishable from the original. A visually lossless image is a lossy image. *See also "Lossless" and "Lossy".*

Wavelet

A mathematical representation of a pixel value that varies by frequency and duration; in wavelet space, the importance of a pixel depends on the values of its neighboring pixels. Wavelet-based image formats are able to present images at multiple levels of resolution without the overhead of pyramidal formats.

Weight

A parameter used in MrSID to determine the emphasis given to the grayscale portion of a color image when performing compression. *See also "Frequency balance", "Sharpness" and "K-weight".*

World file

A text file that contains geospatial positioning information to augment or replace the geospatial information in an image file.

Zoom level

The scale at which an image is represented. Levels are generally expressed with signed integer values. An image “at scale 1” has half the width and height of the original. *See also "Scale", "Magnification" and "Resolution".*

Index