Sparse Image Format (SIF)
Reference Manual and Format Specification

Software Version 1.0

Documentation Revision 1

by Damian Eads

November 30, 2007

Los Alamos National Laboratory
Los Alamos, New Mexico

The Doxygen and $\LaTeX$ tools were used to prepare this document.

LA-UR-07-8033

1 Reference Manual and Format Specification

Author:

Damian Eads

1.1 License

1.2 Installation and Compilation

The SIF library does not depend on any other libraries except for libc. The library has only been compiled on a few platforms, 32-bit machines running Visual Studio(R) on Windows 2000(R) as well as Windows XP(R) as well as GNU/Linux with large file support (64-bit) turned on. In theory, SIF should compile on platforms that do not support 64-bit files but this has not been tested. SIF has not been tested on Cygwin.

1.2.1 Building SIF on Windows

Open up the solution file sif.sol and click Build...Compile. This should build the sif.dll file, which you may put in a folder that is accessible by your PATH variable.

1.2.2 Installing SIF on Windows from Binaries

Download one of the binaries from the SIF website, `http://public.lanl.gov/eads/sif`.

1.2.3 Building SIF on Linux

We use the autotools, specifically automake and autoconf. First, your system needs to be inspected with autoconf by doing

```
./configure
```

Configuring without any options installs the library and include files to `/usr/local` by default. To change the prefix, do

```
./configure --prefix=/desired/prefix
```

After configuring your build environment, you are ready to build the library. The first command below builds the library while the second command installs it and the include files.

```
make
make install
```

1.2.4 Building against SIF in Linux

The compilation of your own code against SIF is not covered in this document since its a topic that others have certainly covered better than I could. However, there are two helpful points to note: SIF builds a shared library `libsif.so`, which it puts in `prefix/lib` and it puts a header file `sif-io.h` (p. 38) in `prefix/include`. You will need to include `sif-io.h` (p. 38) to have access to SIF functions and you will need to link against `libsif.so`. We provide a `pkg-config` file for your convenience called `sif-io.pc`, which is put in `prefix/lib/pkgconfig`. If you wish to use it, make sure your `PKG_CONFIG_PATH` includes that directory. Be sure your `C_INCLUDE_PATH` is set to include `prefix/include` and your `LD_LIBRARY_PATH` includes `prefix/lib`.

1.3 Introduction

The Sparse Image Format (SIF) is a file format for storing raster images with sparse pixel data. Images are broken down into a grid of tiles of fixed size. A tile is only stored in a file if any two pixels in it are different. This is particularly useful for images that are highly homogeneous in color. A few applications of SIF include using it to store:

- land-cover classifications
- training data for pixel classifiers
- sparse overlay rasters over large images

The SIF format is not intended for space-efficiently storing multispectral imagery or photos.

1.3.1 Basic Terminology

It is helpful to review some terminology used throughout this document since they are essential to understanding the basic operation of the library. Images are made up of **bands** and these bands are made up of **pixels** or **data units**. For example, an image of 32-bit floats with three bands, the data unit would be 32-bit float, and the **data unit size**, 4 bytes. A **tile** is a fixed-size cuboid of an **image** including all of its bands. A **slice** is a single band of a tile. The next figure illustrates all these primitive elements together.

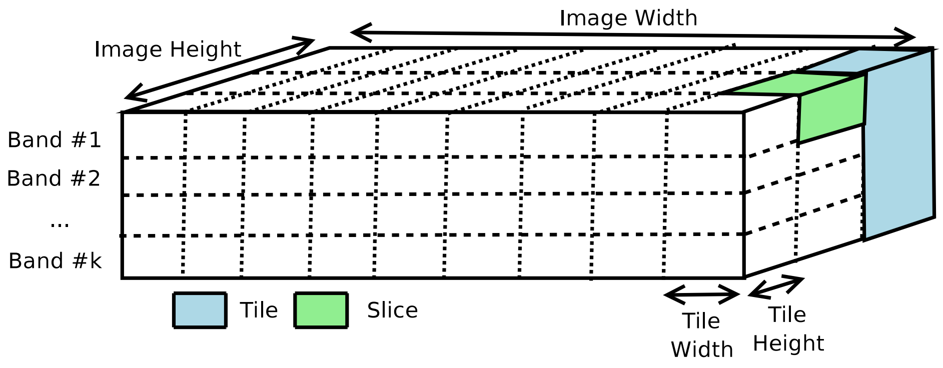

Figure 1: The abstract layout of a SIF image.

Every tile has a small **tile header** describing it: its uniformity and its byte offset location on disk (if applicable). A **block** refers to a unit of space on disk for storing a tile. The terms are different to easily differentiate between the physical entity of an image (a tile) with the storage space used to store it (block). A block need not contain a tile, in which case it is an **unused block**, and it can be reclaimed for later use. The next table shows how all of these image elements (tiles, blocks, slices) are related to one another and their relative sizes.

Relevant to	**Unit**	**Size**
Image	`image_width`	*User-defined*
Image	`image_height`	*User-defined*
Image/Tile/Block	`bands`	*User-defined*
Image	`image_wpixels`	`ceil(image_width / tile_width) * tile_width`
Image	`image_hpixels`	`ceil(image_height / tile_height) * tile_height`
Image	`image_data_units`	`image_wpixels * image_hpixels * bands`
Image	`image_bytes`	`image_data_units * data_unit_size`
Tile/Slice/Block	`tile_width`	*User-defined*
Tile/Slice/Block	`tile_height`	*User-defined*
Tile	`tile_data_units`	`tile_width * tile_height * bands`
Tile	`tile_bytes`	`tile_pixels * data_unit_size`
Block	`block_data_units`	`block_width * block_height * bands`
Block	`block_bytes`	`block_pixels * data_unit_size`
Slice	`slice_data_units`	`tile_width * tile_height * 1`
Slice	`slice_bytes`	`slice_data_units * data_unit_size`
Data Unit	`data_unit_size`	*User-defined*

1.3.2 Uniformity and Compression

Compression in the SIF format is quite simple. A slice is **uniform** if every data unit in the slice is the same and the common pixel value for the slice is called its **uniform pixel value**. A tile is uniform if all of its slices are uniform. Slices within the same tile need not have the same uniform pixel value.

The next figure shows an example of an image, tile headers for a few tiles, and the layout of various elements in the file. The red and white blocks correspond to used blocks and unused blocks, respectively. Unused blocks are occur when a previously used block is freed up. Let's examine five tiles (i, j, k, m, and n), their associated tile headers, and their placement on disk.

- **tile i** is clearly not uniform given the L shaped line in it. Its tile header is consistent with this. On disk, it is stored in the first block in the block region.
- **tiles j and k** are also not uniform. However, tile j's corresponding block on disk is stored before tile k's block region on disk however in the abstract image, tile j comes after tile i. Thus, there is a lack of contiguity or fragmentation. SIF contains routines for rearranging data blocks so that adjacent tiles in the image are contiguously stored on disk. Defragmentation occurs during a file's close if the **sif_header::defragment** (p. 32) flag is set in the tile's header. Otherwise, defragmentation only occurs when the **sif_defragment** (p. 45) function is called.
- **tile m** is uniform, the uniform pixel values for its bands are 0xFF, 0xCO, and 0x00, and it is not stored in the data block region of the file.
- **tile n** is also uniform. However, its tile header does not reflect this property, most likely because it was once non-uniform, but it was rewritten, and a uniformity check has yet to be made for it. SIF also provides routines for performing consolidation, checking non-uniform tiles for intrinsic uniformity and freeing up the blocks they use. If the **sif_header::consolidate** (p. 31) flag is set, consolidation is performed during the file's close.

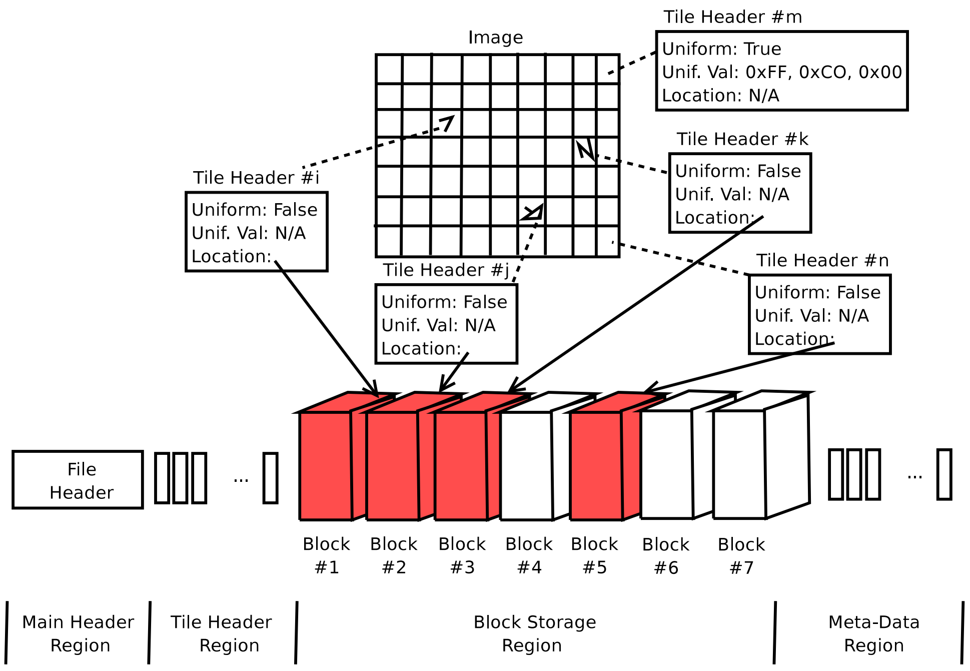

Figure 2: The physical layout of a SIF image on disk.

1.3.2.1 Types of uniformity

There are three different kinds of uniformity, which are described below.

- **shallow uniformity**: a tile has shallow uniformity only when its tile header indicates it is uniform. Tile m exhibits shallow uniformity but tile n does not.
- **hidden uniformity**: a tile has hidden uniformity only when its tile header indicates it is non-uniform but the block corresponding to it on disk is uniform. Tile n exhibits hidden uniformity but tile n does not.
- **intrinsic uniformity**: a tile has intrinsic uniformity if it has shallow uniformity or if the data block that corresponds to it is uniform. Both tiles m and n are intrinsically uniform.

Some functions only check for shallow uniformity when performing their operations while others consider intrinsic uniformity. The **sif_consolidate** (p. 43) function helps free up data blocks by checking for uniformity of underlying block rasters, labeling them as shallow uniform if it finds them to be intrinsically uniform, and freeing up the disk blocks used by them. The **sif_-consolidate** (p. 43) function also reduces external fragmentation by moving the used blocks to the front of the file and the unused blocks to the back. If the **sif_header::consolidate** (p. 31) flag is set, this consolidation process is performed on the file's closing.

1.3.2.2 Border tiles that overlap the image boundary

Sometimes the tile width does not divide the image width or the tile height does not divide the image height. The next Figure illustrates this. The border tile overlaps the border of the image. In cases such as these, only tile pixels within the image boundary are examined for uniformity. Border tiles cause some internal fragmentation but it is often negligible.

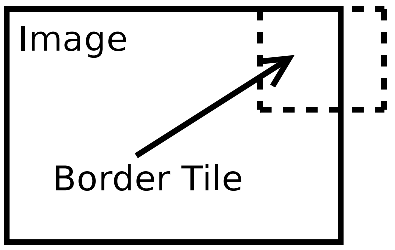

Figure 3: An image with the border tile overlapping the image border.

1.3.2.3 Choosing tile dimensions

The choice of tile width and tile height depends on several factors. The number of bytes needed to store the tile header relative to the bytes needed to store the tile raster. It is also useful to characterize the kind of uniformity expected in the image like the largest non-uniform region size and the number of these regions.

1.3.3 Meta-data

The SIF format facilitates the storage of (key, value)-paired meta-data. A (key, value) pair is called a meta-data item. A meta-data item is referred to by its **key** and the element data is its **value**. Keys are case-sensitive.

SIF has two types of meta-data values, strings and binary byte sequences. String values must be represented by null-terminated character arrays. Binary byte sequences permit the storage of arbitrary data in a SIF file. String values are stored as binary byte sequences. If an attempt is made to retrieve a meta-data item as a string but the value is not a null-terminated byte sequence, an error is returned.

1.3.3.1 Reserved meta-data Any meta-data key beginning with `_sif_` is reserved for special meta-data, as defined by the SIF file format specification. The following reserved meta-data keys are currently in usage:

- `_sif_proj`
- `_sif_agree`

The following example sets a meta-data field on the SIF file pointed to by the file `sif_file (p. 27)` pointer,

```
sif_set_meta_data(file, "model_file", "/afs/clue/gadm/833/hyper.model")
```

Now, let's retrieves it, and print it

```
printf("model file: %s\n", sif_get_meta_data(file, "model_file"));
```

Now let's try to store an array `V` of 32 doubles using native byte order,

```
sif_set_meta_binary(file, "my_32_doubles", V, sizeof(double) * 32);
```

Let's now retrieve it and print it out.

```
double *buf;
int nbytes, i;
buf = sif_get_meta_data_binary(file, "my_32_doubles", &nbytes);
if (nbytes != sizeof(double) * 32) {
  printf("Something bad happened.\n");
}
else {
  for (i = 0; i < 32; i++) {
    printf("my_double %d: %5.8f\n", i, buf[i]);
  }
}
```

1.3.3.2 Caveat Since a copy of all the meta-data in SIF file is always stored in memory, the meta-data feature is only intended for light use. In a future version, storage of a large meta-data footprint will be viable.

1.3.4 Pixel Data Types

The SIF file format does not establish a set of data types. The underlying pixel values are data-typeless to the SIF I/O library. The library permits the user to store a **sif_header::user_data_type** (p. 35) field but the field's value does not influence the behavior of SIF routines. This scheme of ignoring the underlying data type works if it can be guaranteed that two values pixel values p(x1,y1,b1) and p(x2,y2,b2) are the same if and only if the underlying byte sequences of these values are the same. Thus, SIF only needs to know the size of each underlying byte sequence that represents a single pixel value.

1.3.4.1 Agreement Meta-data String Agreeing to a data type convention provides a guarantee that the data type of pixels can easily be determined. We encourage users to set the `"_sif_agree"` meta-data value to a string indicating the data type convention used. If `"_sif_agree"` meta-data string is not set, all bets are off, and no guarantees can be made about the data type of the pixels. Alternatively, the **sif_get_agreement** (p. 47) and **sif_set_agreement** (p. 55) functions can be used to get and set the agreement string.

We define one data type convention, "simple". The type codes for the **sif_header::user_data_type** (p. 35) field are defined below.

Value of `user_data_type`	**Corresponding Data Type**
0 or `SIF_SIMPLE_UINT8`	`unsigned char` or `uint8_t` (little-endian)
1 or `SIF_SIMPLE_INT8`	`char` or `int8_t`
2 or `SIF_SIMPLE_UINT16`	`uint16_t`
3 or `SIF_SIMPLE_INT16`	`int16_t`
4 or `SIF_SIMPLE_UINT32`	`uint32_t`
5 or `SIF_SIMPLE_INT32`	`int32_t`
6 or `SIF_SIMPLE_UINT64`	`uint64_t`
7 or `SIF_SIMPLE_INT64`	`int64_t`
8 or `SIF_SIMPLE_FLOAT32`	`IEEE-754 32-bit float`
9 or `SIF_SIMPLE_FLOAT64`	`IEEE-754 64-bit float`

For example, suppose we've created a new SIF file with a data_unit_size of 4. Now let's write some code to indicate the "simple" convention: use unsigned 32-bit integers as the data type and use big-endian as the byte-order of the data units. Then, we'll print out these codes using functions that manipulate the compound type code (which we store in the **sif_header::user_data_-type** (p. 35) in the file's header) to give the base type code (i.e. the data type irrespective of the byte order) and the endian code.

```
int base_code = SIF_SIMPLE_UINT32;
int endian_code = SIF_SIMPLE_BIG_ENDIAN;
int compound_code = SIF_SIMPLE_TYPE_CODE(base_code, endian_code);
sif_set_agreement(file, SIF_AGREEMENT_SIMPLE);
sif_set_user_data_type(file, compound_code);
printf("Base Data Type: %d\n"
       "Compound Data Type: %d\n"
       "Endian: %d\n", SIF_SIMPLE_BASE_TYPE_CODE(compound_code),
       compound_code, SIF_SIMPLE_ENDIAN(compound_code));
```

Alternatively, you can use the **sif_simple_create** (p. 59) function to create a file using the "simple" data type convention.

1.4 SIF File Layout

The SIF file begins with a fixed-size header followed by **sif_header::n_tiles** (p. 33) fixed-sized tile headers, followed by a variable number of fixed-sized blocks. Finally, the meta-data is written after all the data blocks. The file header and tile headers are put in the beginning of the file since their size does not change, although their values may change. This means that the large data blocks need not be moved forward in the file. Meta-data is written after the data blocks since the number of meta-data items can change; thus, the approach eliminates the need to move up data blocks after inexpensive meta-data operations. Unfortunately, there is a danger that after a new data block is allocated, there may not be enough space on the partition for the meta-data, and the file's meta-data cannot be safely written as read. This kind of data loss is uncommon if efforts are made to ensure adequate disk space is available to the scientific programs that use SIF.

1.4.1 Overall Layout of a SIF File.

The overall layout of a SIF file from the first byte to the last is shown in the following table. The file header remains of constant size and most header fields are immutable. The tile headers are of constant size since the data unit size and number of bands are immutable quantities. The inclusion of routines for changing tile dimensions, image dimensions, endianess, and data types is under currently under consideration. The block region is of variable size and precedes the meta-data region. Consequently, the location of the meta-data region changes as the size of the block region changes. The size of the meta-data region changes as meta-data fields are modified, added, or removed. It was anticipated that meta-data would be modified more infrequently than the data blocks. If the meta-data were to precede the block region, a small increase in the size of the meta-data region would result in the need to move the entire block region, which is costly when the block region is large. This is a casual justification for our choice of storing the meta-data after the block region. Also under consideration is the ability to store the meta-data before the block region, employing preallocation strategies to minimize moves of the block region due to meta-data region resizing.

File Header
Tile Header 1 (starts at `header->header_bytes`)
Tile Header 2
...
Tile Header `n_tiles` (starts at `file->base_location`)
Block 1
Block 2
...
Block `n_blocks`
Meta-data Item 1 (starts after the last byte of the last block)
Meta-data Item 2
...
Meta-data Item `n_meta_data_items`

1.4.2 File Header Byte Layout

The absolute byte offset for each file header field is shown in the next table. The second column is the name of the field as stored in the **sif_header** (p. 30) struct stored when a SIF file is opened. Integers and doubles are signed and stored in big-endian (or network) byte order. Note that in SIF Format Version code 1, doubles were stored little-endian but we realized this was confusing so this has been changed to big endian in versions 2 and higher. The `header_-bytes` field enables the format to be changed without advancing the version code. Specifically, non-essential header fields can be added but they will be ignored by earlier versions of the I/O library.

The only fields that can change following the first write of a raster to an image are the defragmentation, consolidation, and intrinsic write flags as well as the georeferencing transform, and key count. If the caller wishes to change the image dimensions, data type, or tile dimensions after the first raster write, it must be done manually.

Absolute Offset	**Name**	**Description**	**Type**
0	`header_bytes`	The header size in bytes including the space needed for `header_bytes`.	32-bit int (b.e.)
4	`magic_number`	The magic number `"!**SIF**"`.	8 8-bit chars
12	`version`	The version of the SIF file format used for the target file. This field is not the version of the SIF I/O library used to write the file.	32-bit int
16	`width`	The width of the image in pixels.	32-bit int
20	`height`	The height of the image in pixels.	32-bit int
24	`bands`	The depth of the image in pixels.	32-bit int
28	`n_keys`	The number of meta data fields stored.	32-bit int
32	`n_tiles`	The number of tiles stored.	32-bit int
36	`tile_width`	The width of each tile and slice in pixels.	32-bit int
40	`tile_height`	The height of each tile and slice in pixels.	32-bit int
44	`tile_bytes`	The number of bytes to store a single tile with all bands.	32-bit int

1.4.3 Header Byte Layout (continued)

Absolute Offset	Name	Description	Type
48	`n_tiles_across`	The number of tiles for a single row of tiles on an image.	32-bit int
52	`data_unit_size`	The size of a single data unit.	32-bit int
56	`user_data_type`	A user-defined constant to represent the data type of the pixels, meaningful to the caller.	32-bit int
60	`defragment`	When set, defragments the file during close.	32-bit int
64	`consolidate`	When set, consolidates the file during close.	32-bit int
68	`intrinsic_-write`	When set, newly dirtied tiles are checked for intrinsic uniformity when written.	32-bit int

1.4.4 Header Byte Layout (continued)

Absolute Offset	Name	Description	Type
72	`tile_hd_bytes`	The number of bytes to store a single tile header on disk.	32-bit int
76	`n_unif_flags`	The number of bytes to store the uniform flags in the tile header.	32-bit int
80	`aff_geo_trans`	The affine geo-referencing transform.	Six 64-bit IEEE-754 doubles (b.e.)

1.4.5 Tile Header Byte Layout

The tile headers store information about the uniformity or non-uniformity of the block. If the tile is uniform, the `uniform_pixel_value` fields have meaning, and the i'th value is the uniform pixel value for the i'th. The `block_num` field is set to -1 if the tile header corresponds to a non-uniform tile. The first advancing index is the horizontal tile index and the second, the vertical tile index. This corresponds to how tiles are read and written from and to buffers in the image, i.e. the x coordinate of the pixels advances before the y.

Relative Offset (to the previous unit)	**Name**	**Description**	**Type**
0	uniform_-pixel_value[0]	The value to fill band 0 if the tile is uniform. Otherwise, the value is meaningless	*User defined* (of size data_-unit_size)
data_unit_size	uniform_-pixel_value[1]	The value to fill band 1 if the tile is uniform. Otherwise, the value is meaningless	*User defined* (of size data_-unit_size)
...	...	...	...
i*data_unit_-size	uniform_-pixel_value[i]	The value to fill band i if the tile is uniform. Otherwise, the value is meaningless	*User defined* (of size data_-unit_size)
...	...	...	...
(bands - 1) * data_unit_size	uniform_-pixel_-value[bands - 1]	The value to fill the last band if the tile is uniform. Otherwise, the value is meaningless	*User defined* (of size data_-unit_size)
r=bands * data_unit_size	uniform_flags	An array of bits, the i'th bit is TRUE if the i'th slice is uniform.	h=ceil(bands/8) 8-bit characters
r+h	block_num	The block number where this tile is stored if it is non-uniform. This value is -1 if uniform.	32-bit int

1.4.6 Meta-Data Item Byte Layout

The meta-data item byte layout is simple. Again, integer length fields are assumed to be big-endian.

Relative Offset (to the previous unit)	**Name**	**Description**	**Type**
`0`	`key_length`	The number of bytes to store the key including the null terminator.	32-bit int (b.e.)
`4`	`key`	The key as a string.	`key_length` bytes
`4+key_length`	`value_length`	The number of bytes to store the value including the null terminator (if the value is non-binary).	32-bit int (b.e.)
`8+key_length`	`value`	The value as a byte sequence.	`value_length` bytes

1.5 SIF Library and File Format Versions

Significant time was invested in designing the SIF file format. Yet, it is inevitable users will ask the author to make changes to it. This is a tricky road for several reasons. Changes that are only useful to a few users pose an issue where the rest of the user base may have compatibility issues when sharing their files since some users will choose to update their library while others will stick with older versions. Changes also add complexity to the unpacking logic in the I/O library, especially since an effort is made to effort ensure backwards compatibility of new versions of the library with older versions of the format. Therefore, my philosophy on changing and developing SIF is one that encourages improvements to the API over changes to the format.

The first release of the SIF I/O library (0.9) and SIF File Format (code 1) was internal while the second release (1.0 and code 2) was the first public release. Version 1 assumes integers in the header, tile headers, and meta-data headers are big-endian and doubles are little-endian. Realizing this was confusing, version 2 assumes doubles in the headers (namely **sif_header::affine_geo_transform** (p. 31) are also big-endian). Files can be written using older versions of the SIF File Format using the **sif_use_file_format_version** (p. 66) function.

The following table lists the file versions supported by each version of the SIF I/O library.

SIF Software Version	**Read**	**Write**
0.9	1	1
1.0	1-2	1-2

1.5.1 Image Pixel and Tile Header Index Computation

The number of pixels along the `x`-coordinate axis is given by the image width, and the number of pixels y-coordinate axis is given by the image height. Each tile is referenced by a tile coordinate, `(tx, ty)`. `tx` is the index of the tile with respect to the `x`-coordinate axis and ty is the index of the tile with respect to the y-coordinate axis. The `x` index is the fastest advancing index; the `y` index, the second fastest advancing; and the band index, the slowest advancing index. The absolute byte offset `q` is computed from a pixel coordinate (x,y,b) as follows:

```
q=(b * image_width * image_height) + (image_width * y + x)
```

The absolute tile index `r` is similarly computed,

```
r=(n_tiles_across*ty)+tx
```

1.6 Error Checking and Reporting

The SIF library performs extensive error checking for I/O errors, memory allocation errors, and errors in the parameters passed to SIF functions. The **sif_file::error** (p. 28) code is set to 0 or **SIF_ERROR_NONE** (p. 24) during normal operation. No **sif-io.h** (p. 38) function resets this flag so the caller must do so if the error is deemed as non-fatal and the caller wishes to perform further operations on the file. Most SIF functions return immediately the first time an error is encountered. Memory allocated during an operation resulting in an error is deallocated prior to returning. The value returned by non-void functions during an error depends on the expected range of values for that function. If a pointer is usually returned, 0 is returned; if a positive number is usually returned, a non-positive is returned; or if a non-positive is usually returned, a number greater than zero is returned. For ease of coding, callers should test the **sif_file::error** (p. 28) flag rather than checking the return value because of the lack of consistency of return values when returning due to an error. The **sif_get_error_description** (p. 47) function returns a string description of an error code, which callers may conveniently use when reporting errors.

1.7 Testing for a valid SIF file

The **sif_is_possibly_sif_file** (p. 52) function checks whether a file could possibly be a SIF file. The present version only checks whether the magic number is valid. Future versions of the library will ensure:

- given the header, there are enough tile headers;
- there are blocks stored in the file corresponding to the offsets specified in tile headers with non-negative block indices; and
- the meta-data is properly stored.

1.8 Simple Convention Interface

Bundled with the SIF library are functions for manipulating SIF files conforming to the "simple" data type convention. These functions begin with `sif_simple_`.

1.8.1 Creating SIF Simple Files

The **sif_simple_create** (p. 59) and **sif_simple_create_defaults** (p. 60) functions are both used to create a SIF file conforming to the "simple" data type convention. The latter function sets defaults related to consolidation, uniformity checking, defragmentation, and tile size. Native byte order is used to store the image rasters; however all header fields, tile header fields, and meta-data length fields are all stored in big-endian byte order, regardless of the endian of the image rasters. The **sif_simple_set_endian** (p. 64) function must be called after creating the file and prior to performing any image I/O if the file's image endian is changed. Undefined behavior occurs when the endian field is changed after performing image I/O.

1.8.2 Image I/O

When data blocks are written to or read from a file, the blocks are converted to the appropriate byte order prior to writing to the file or after reading from it. `sif_simple_` functions may not be used unless the file is opened with the **sif_simple_create** (p. 59), **sif_simple_create_defaults** (p. 60), or **sif_-simple_open** (p. 63) function.

1.8.3 Rectangular Region I/O

The **sif_simple_set_raster** (p. 64) and **sif_simple_get_raster** (p. 62) functions are used to write and read a rectangular region, respectively. Only one band can be read or written at a time. The offsets and dimensions of the region are in pixel units, not tile units. The **sif_simple_is_shallow_uniform** (p. 62) checks whether the tiles comprising a rectangular region are stored as shallow uniform.

1.8.4 Tile Block I/O

The **sif_simple_get_tile_slice** (p. 62) and **sif_simple_set_tile_slice** (p. 65) functions read and write a slice. The **sif_simple_fill_tile_slice** (p. 60) function fills a slice with a constant value. The **sif_is_slice_-shallow_uniform** (p. 53) function checks whether a slice is stored as shallow uniform in the file.

1.8.5 Checking for Conformity

The conformity of a file to the "simple" data type convention can be verified with the sif_is_simple_file or sif_is_simple_file_by_name functions. The first function assumes the file as already been opened with **sif_open** (p. 54) while the second accepts a filename.

1.9 Notes on Memory Preallocation

SIF allocates enough memory to hold two image blocks in memory for each open SIF file. When the **sif_simple_open** (p. 63) (for update), **sif_simple_-create** (p. 59), or **sif_simple_create_defaults** (p. 60) functions are used to open or create a SIF file conforming to the "simple" data type convention, a buffer is also allocated for converting the byte order of image rasters. The buffer is initially the size of a block. When a call is made to **sif_simple_set_raster** (p. 64) with a raster larger than the size of the buffer, the buffer is enlarged appropriately. Note that this buffer is not needed if the file is opened for read-only access, since the byte order conversion is performed on the caller's buffer. All of a file's memory buffers are deallocated during close.

1.10 Command Line Utilities

We provide the `sif-util` command for you to use to create, inspect and manipulate SIF files at a UNIX or DOS shell. The first argument is the name of the file to manipulate; the second argument, the name of the operation to perform; and the remaining arguments, the parameters of the operation.

```
  sif-util operation operation-args
```

1.10.1 sif-util Supported Operations

We now describe each of the operation supported by `sif-util`. Arguments are mandatory unless enclosed with square brackets. Indicated in parenthesis is whether the file must be writable to perform the operation.

- **consolidate** `on/off/now` *(writable)*: when set to `on`, a file opened for update is consolidated whenever it is closed. When set to `now` the `consolidate` flag in the file's header is unchanged and consolidation is performed immediately on the file.
- **create** `width height bands sdt [consolidate=on/off] [defragment=on/off] [tw=int] [th=int] [intrinsic=on/off] [endian=native]` *(writable)*: creates an image file conforming to the "simple" data type convention. The dimensions of the image are `width` by `height` with `band` bands. The simple data type code for the image is defined with `sdt`, which must be between 0 and 9 or a string type identifier (uint8, uint16, uint32, uint64, int8, int16, int32, int64, float, double). When the defragment flag (default=`on`) is set, the file is scheduled for defragmentation whenever it is closed. When the consolidate flag (default=`on`) is set, the file is scheduled for consolidation whenever it is closed. The `tw` (default=64) and `th` (default=64) parameters are the width and height of the tiles in the image. When the `intrinsic` write flag (default=`on`) is set, whenever a raster is written to a file, it is checked for intrinsic uniformity. Native byte order is used for storing the image raster unless the `endian` is set to `big` or `little`.
- **defragment** `on/off/now` *(writable)*: when set to `on`, a file opened for update is defragmented whenever it is closed. When set to `now` the `defragment` flag in the file's header is unchanged and defragmentation is performed immediately on the file.
- **get-md** `key`: returns the meta-data field referred to by `key`. If the meta-data is binary, each non-printable character is printed with `\xYY` where `YY` is the character's hexadecimal code. New lines are printed with `\n`.
- **has-key**: prints "yes" if a meta-data field with key `key` is stored in the file, and "no" otherwise.
- **header**: prints out the file's header in a human-readable format.
- **intrinsic** `on/off` *(writable)*: when set, whenever a raster is written to the file, it is checked for intrinsic uniformity.
- **is-sif**: writes "yes" if the file is a SIF file, and no, otherwise.
- **is-simple**: writes "yes" if the file is a SIF file and conforms to the "simple" data type convention, and no, otherwise.

- **list-keys** `filename`: lists the meta-data keys in the file. Each key is printed on a separate line.
- **list-md** `filename`: lists the meta-data in the file. Each `key=value` pair is printed on a separate line. Each non-printable character is printed with `\xYY` where `YY` is the character's hexadecimal code. New lines are printed with `\n`.
- **region-to-pnm** `[x y w h] [band=int/all]`: prints the file's raster in PNM format. This output can be redirected to a PNM file and then converted to another image format using your favorite image converter. When `x`, `y`, `w`, and `h` are set, the region is read starting at `(x,y)` with width `w` and height `h`. When the `band` parameter (default=all) is set to a non-negative integer value, a specific band is written. By the default, the entire image is printed.
- **set-md** `filename key value` *(writable)*: sets the meta-data field referred to by `key` with the string value `value`. If the meta-data is binary, each non-printable character must be written with `\xYY` where `YY` is the character's hexadecimal code. New lines must be written with `\n`.
- **statistics**`:` prints some file statistics, including the number of tiles that are either shallow uniform, hidden uniform, and intrinsically non-uniform. The number of blocks in the block region as well as the number of unused blocks is also printed. These statistics may be useful to decide whether to defragment or consolidate a file. Also printed is the percentage of space saved by the compression and the space that could be saved if consolidation was performed.
- **tile-to-pnm** `tx ty [band=int/all]`: prints out the tile in the file with tile coordinates `tx` and `ty` in PNM format. This output can be redirected to a PNM file and then converted to another image format using your favorite image converter. When the `band` parameter (default=all) is set to a non-negative integer value, a specific band is written.

1.10.1.1 A Note on PNM Output When the PNM output operations `region-to-pnm` and `tile-to-pnm` are used, the pixel values are assumed to be of unsigned type. If the number of bands to write is 3, the PPM subformat is used with each of the three bands representing a separate color (band[0]=R, band[1]=G, band[2]=B). When writing a single band, the PGM format is used. If the image contains any other number of bands, the PAM format is used. The data unit size must not exceed 2 bytes, i.e. only uint8 and uint16 are supported. It is assumed the image raster is stored in native byte order if the file does not conform to the "simple" data type convention. The image raster is translated into proper ASCII decimal form (PPM or PGM format) or big-endian byte order (PAM format) prior to being outputted.

2 SIF Module Index

2.1 SIF Modules

Here is a list of all modules:

3 SIF Data Structure Index

3.1 SIF Data Structures

Here are the data structures with brief descriptions:

4 SIF File Index

4.1 SIF File List

Here is a list of all files with brief descriptions:

5 SIF Module Documentation

5.1 SIF Error Codes

Defines

- #define **SIF_ERROR_NONE 0**
- **#define SIF_ERROR_MEM 1**
- **#define SIF_ERROR_NULL_FP 2**
- **#define SIF_ERROR_NULL_HDR 3**
- **#define SIF_ERROR_INVALID_BN 4**
- **#define SIF_ERROR_INVALID_TN 5**
- **#define SIF_ERROR_READ 6**
- **#define SIF_ERROR_WRITE 7**
- **#define SIF_ERROR_SEEK 8**
- **#define SIF_ERROR_TRUNCATE 9**
- **#define SIF_ERROR_INVALID_FILE_MODE 10**
- **#define SIF_ERROR_INCOMPATIBLE_VERSION 11**
- **#define SIF_ERROR_META_DATA_KEY 12**
- **#define SIF_ERROR_META_DATA_VALUE 13**
- **#define SIF_ERROR_CANNOT_WRITE_VERSION 14**
- **#define SIF_ERROR_INVALID_BAND 15**
- **#define SIF_ERROR_INVALID_COORD 16**
- **#define SIF_ERROR_INVALID_TILE_SIZE 17**
- **#define SIF_ERROR_INVALID_REGION_SIZE 18**
- **#define SIF_ERROR_INVALID_BUFFER 19**

5.1.1 Define Documentation

5.1.1.1 #define SIF_ERROR_CANNOT_WRITE_VERSION 14

Returned by `sif_use_file_version` when the library is not capable of writing the file in the requested version.

5.1.1.2 #define SIF_ERROR_INCOMPATIBLE_VERSION 11

A status code indicating that the currently loaded sif-io library is not capable of processing the version of a file. This is usually due to the fact that the file was written with a later version of the SIF format than the loaded library.

5.1.1.3 #define SIF_ERROR_INVALID_BAND 15

Returned if a band argument passed is invalid.

5.1.1.4 #define SIF_ERROR_INVALID_BN 4

A status code indicating a block number passed to a sif-io function was invalid (i.e. negative or out-of-bounds).

5.1.1.5 #define SIF_ERROR_INVALID_BUFFER 19

Returned if a tile size argument (e.g. `width` or `height`) is invalid.

5.1.1.6 #define SIF_ERROR_INVALID_COORD 16

Returned if a coordinate argument (e.g. `x` or `y`) is invalid.

5.1.1.7 #define SIF_ERROR_INVALID_FILE_MODE 10

A status code indicating that the file mode chosen is invalid. This usually occurs when a file is opened for update that is read-only or a opened when the permissions do not permit reading.

5.1.1.8 #define SIF_ERROR_INVALID_REGION_SIZE 18

Returned if a region size argument (e.g. `width` or `height`) is invalid.

5.1.1.9 #define SIF_ERROR_INVALID_TILE_SIZE 17

Returned if a tile size argument (e.g. `tile_width` or `tile_height`) is invalid.

5.1.1.10 #define SIF_ERROR_INVALID_TN 5

A status code indicating a tile number passed to a sif-io function was invalid (i.e. negative or out-of-bounds).

5.1.1.11 #define SIF_ERROR_MEM 1

A status code indicating an error occurred while allocating or freeing memory.

5.1.1.12 #define SIF_ERROR_META_DATA_KEY 12

Returned when a call is made that expects a key to be present when the key cannot be found.

5.1.1.13 #define SIF_ERROR_META_DATA_VALUE 13

Returned by `sif_get_meta_data` when the meta-data does not contain a null-terminated string.

5.1.1.14 #define SIF_ERROR_NONE 0

A status code indicating no error has been detected for the processing of the target file.

5.1.1.15 #define SIF_ERROR_NULL_FP 2

A status code indicating a file could not be processed because the file pointer is null. Admittedly, there is no way to store this in the **sif_file** (p. 27) struct passed since it is null. However, setting a static variable for the caller to check is under consideration for a future version.

5.1.1.16 #define SIF_ERROR_NULL_HDR 3

A status code indicating a file could not be processed because the header pointer is null.

5.1.1.17 #define SIF_ERROR_READ 6

A status code indicating an error occurred when reading from the file.

5.1.1.18 #define SIF_ERROR_SEEK 8

A status code indicating an error occurred when seeking in the file.

5.1.1.19 #define SIF_ERROR_TRUNCATE 9

A status code indicating an error occurred when truncating the file.

5.1.1.20 #define SIF_ERROR_WRITE 7

A status code indicating an error occurred when writing to the file.

5.2 Simple Data Type Convention Macro Definitions

Defines

- #define **SIF_SIMPLE_UINT8 0**
- **#define SIF_SIMPLE_INT8 1**
- **#define SIF_SIMPLE_UINT16 2**
- **#define SIF_SIMPLE_INT16 3**
- **#define SIF_SIMPLE_UINT32 4**
- **#define SIF_SIMPLE_INT32 5**
- **#define SIF_SIMPLE_UINT64 6**

- **#define SIF_SIMPLE_INT64 7**
- **#define SIF_SIMPLE_FLOAT32 8**
- **#define SIF_SIMPLE_FLOAT64 9**
- **#define SIF_SIMPLE_LITTLE_ENDIAN 0**
- **#define SIF_SIMPLE_BIG_ENDIAN 1**
- **#define SIF_SIMPLE_NATIVE_ENDIAN SIF_SIMPLE_-LITTLE_ENDIAN**
- **#define SIF_SIMPLE_ENDIAN(t) (((int)t)/10)**
- **#define SIF_SIMPLE_TYPE_CODE(bt, ec) ((bt)+(ec))**
- **#define SIF_SIMPLE_BASE_TYPE_-CODE(x) (((int)x)%10)**

5.2.1 Define Documentation

5.2.1.1 #define SIF_SIMPLE_BASE_TYPE_-CODE(x) (((int)x)%10)

A function macro that computes the base type code from the compound type code.

5.2.1.2 #define SIF_SIMPLE_BIG_ENDIAN 1

The endian code for little endian.

5.2.1.3 #define SIF_SIMPLE_ENDIAN(t) (((int)t)/10)

A function macro that returns the endian code for a simple type code *t*.

5.2.1.4 #define SIF_SIMPLE_FLOAT32 8

The base type code (i.e. `user_data_type mod 10`) for storing IEEE-754 standard 32-bit floats.

5.2.1.5 #define SIF_SIMPLE_FLOAT64 9

The base type code (i.e. `user_data_type mod 10`) for storing IEEE-754 standard 64-bit floats.

5.2.1.6 #define SIF_SIMPLE_INT16 3

The base type code (i.e. `user_data_type mod 10`) for storing signed 16-bit integers.

5.2.1.7 #define SIF_SIMPLE_INT32 5

The base type code (i.e. `user_data_type mod 10`) for storing signed 32-bit integers.

5.2.1.8 #define SIF_SIMPLE_INT64 7

The base type code (i.e. `user_data_type mod 10`) for storing signed 64-bit integers.

5.2.1.9 #define SIF_SIMPLE_INT8 1

The base type code (i.e. `user_data_type mod 10`) for storing signed 8-bit integers.

5.2.1.10 #define SIF_SIMPLE_LITTLE_ENDIAN 0

The endian code for little endian.

5.2.1.11 #define SIF_SIMPLE_NATIVE_ENDIAN SIF_-SIMPLE_LITTLE_ENDIAN

The endian code for the byte order of the native machine on which this library runs.

5.2.1.12 #define SIF_SIMPLE_TYPE_CODE(bt, ec) ((bt)+(ec))

A function macro that computes the compound type code from the base type code *bt* and endian code *ec*.

5.2.1.13 #define SIF_SIMPLE_UINT16 2

The base type code (i.e. `user_data_type mod 10`) for storing unsigned 16-bit integers.

5.2.1.14 #define SIF_SIMPLE_UINT32 4

The base type code (i.e. `user_data_type mod 10`) for storing unsigned 32-bit integers.

5.2.1.15 #define SIF_SIMPLE_UINT64 6

The base type code (i.e. `user_data_type mod 10`) for storing unsigned 64-bit integers.

5.2.1.16 #define SIF_SIMPLE_UINT8 0

The base type code (i.e. `user_data_type mod 10`) for storing unsigned 8-bit integers.

5.3 Simple Data-Type Convention Declarations

6 SIF Data Structure Documentation

6.1 sif_file Struct Reference

A struct for storing necessary data for the processing of an open file.

`#include <sif-io.h>`

Data Fields

- FILE ∗ **fp**
- **sif_header** ∗ **header**
- **sif_tile** ∗ **tiles**
- **sif_meta_data** ∗∗ **meta_data**
- **int read_only**
- **long** ∗ **blocks_to_tiles**
- **long** ∗ **dirty_tiles**
- **void** ∗ **buffer [2]**
- **LONGLONG base_location**
- **int error**
- **long sys_error_no**
- **long units_per_slice**
- **long units_per_tile**
- **long header_bytes**
- **u_char ubuf [8]**
- **long use_file_version**
- **void** ∗ **simple_region_buffer**
- **long simple_region_bytes**
- **int error_line_no**

6.1.1 Detailed Description

A struct for storing necessary data for the processing of an open file.

Warning:

Do not modify this data structure directly.

6.1.2 Field Documentation

6.1.2.1 LONGLONG sif_file::base_location

Stores the byte offset of the first block in the file.

6.1.2.2 long∗ sif_file::blocks_to_tiles

An array where the i'th value is the tile index of the tile stored in data block i.

If no tile is stored in data block i, the block is unused, and the corresponding value in this array is set to -1. Unused blocks can be reclaimed for use or truncated when the file is consolidated or defragmented.

6.1.2.3 void∗ sif_file::buffer[2]

Two buffers with enough memory to each store one block. The number of bytes for one block is computed by,.

```
   tile_width * tile_height * n_bands * data_unit_size .
```

6.1.2.4 long∗ sif_file::dirty_tiles

An array of booleans where the i'th value is one iff the i'th tile has been written and no uniformity check was made during the write. In this future, the type of the values contained in the array will be changed to char∗, pending confirmation that the change does not break regression tests.

6.1.2.5 int sif_file::error

An error code for the last error that occurred. The value is non-zero if an error occurred during the last sif-io call.

6.1.2.6 int sif_file::error_line_no

The line number of sif-io.c where the last SIF error occurred.

6.1.2.7 FILE∗ sif_file::fp

The handle to the internal file pointer.

6.1.2.8 sif_header∗ sif_file::header

The header corresponding to the target file.

6.1.2.9 long sif_file::header_bytes

The number of bytes to store the header.

6.1.2.10 sif_meta_data∗∗ sif_file::meta_data

The meta-data for the file. This structure is a linked list of (key, value) pairs. Meta-data in SIF can be null-terminated strings or binary data blocks.

6.1.2.11 int sif_file::read_only

A flag indicating whether the file is open in read-only mode.

6.1.2.12 void∗ sif_file::simple_region_buffer

A buffered only used by the SIF "simple" interface for byte swapping prior to writing to a file. It is initially holds the number of bytes needed to store a tile slice but grows as larger regions are written.

6.1.2.13 long sif_file::simple_region_bytes

The size of the simple region buffer (in bytes).

6.1.2.14 long sif_file::sys_error_no

When the C standary library is used, it represents the last errno encountered when executing a libc function in a sif-io function. Otherwise, it represents the WIN32 error code returned by the GetLastErr function.

6.1.2.15 sif_tile∗ sif_file::tiles

An array of tiles to store.

6.1.2.16 u_char sif_file::ubuf[8]

A character buffer with enough bytes to store a 64-bit integer.

6.1.2.17 long sif_file::units_per_slice

The number of pixels per band in a tile (slice). This value is simply tile_width ∗ tile_height. The term slice differs slightly from the term band, it is a band within a tile.

6.1.2.18 long sif_file::units_per_tile

The number of pixels per tile. This value is simply the number of units_per_-slice times the number of bands in the image. This number is also the number of units in a block.

6.1.2.19 long sif_file::use_file_version

A integer representing the SIF file version to use when writing the file. This is used to ensure that the file is written with an earlier version so that it can be read by previous versions of this library.

The documentation for this struct was generated from the following file:

- **sif-io.h**

6.2 sif_header Struct Reference

A struct for storing a SIF file header in memory.

`#include <sif-io.h>`

Data Fields

- char **magic_number [SIF_MAGIC_NUMBER_SIZE]**
- **long version**
- **long width**
- **long height**
- **long bands**
- **long n_keys**
- **long n_tiles**
- **long tile_width**
- **long tile_height**
- **long tile_bytes**
- **long n_tiles_across**
- **long data_unit_size**
- **long user_data_type**
- **long defragment**
- **long consolidate**
- **long intrinsic_write**
- **long tile_header_bytes**
- **long n_uniform_flags**
- **double affine_geo_transform [6]**

6.2.1 Detailed Description

A struct for storing a SIF file header in memory.

Warning:

> Changing its fields does not result in an immediate change to the header stored in the file to which it points. The file must be flushed with **sif_flush** (p. 46) or closed with **sif_close** (p. 43). Integers are stored with a sign bit in big-endian form.

6.2.2 Field Documentation

6.2.2.1 double sif_header::affine_geo_transform[6]

Six doubles representing the affine georeferencing transform parameters.

The georeferenced coordinates of the pixel coordinate (Xpixel, Yline) are computed as follows (from GDAL documentation):

```
const double *GT = &(hd->affine_geo_transform);
Xgeo = GT[0] + Xpixel * GT[1] + Yline * GT[2];
Ygeo = GT[3] + Xpixel * GT[4] + Yline * GT[5];
```

The transform is set to {0.0, 1.0, 0.0, 0.0, 0.0, 1.0} by default by sif_create so that x and y are just mapped to themselves.

Warning:

> Do not edit this field directly. Instead use the **sif_set_affine_geo_-transform** (p. 54) function.

6.2.2.2 long sif_header::bands

The number of bands of the image.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.3 long sif_header::consolidate

A field, that when nonzero, indicates that the file should be consolidated when its closed.

This involves performing pixel uniformity checks on each dirty tile during close.

Warning:

> Do not edit this field directly. Instead use the **sif_set_defragment** (p. 56) or **sif_unset_defragment** (p. 66) functions.

6.2.2.4 long sif_header::data_unit_size

The number of bytes required to store each pixel.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.
> Non-square tiles have not been tested.

6.2.2.5 long sif_header::defragment

A field that, when nonzero, indicates the file should be defragmented when its closed.

Warning:

> Do not edit this field directly. Instead use the **sif_set_defragment** (p. 56) or **sif_unset_defragment** (p. 66) functions.

6.2.2.6 long sif_header::height

The height of the image in pixels.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.7 long sif_header::intrinsic_write

A field, that when nonzero, indicates that when each tile is written, a uniformity check should be performed.

Warning:

> Do not edit this field directly. Instead use the **sif_set_intrinsic_write** (p. 56) or **sif_unset_intrinsic_write** (p. 66) functions.

6.2.2.8 char sif_header::magic_number[SIF_MAGIC_-NUMBER_SIZE]

This field identifies whether the header read from a file is likely to correspond to a SIF file.

These bytes must equal the string "!**SIF**" or an error will occur when the header is processed by a SIF function. The byte offset of this field is 0.

Warning:

Do not edit this field directly.

6.2.2.9 long sif_header::n_keys

The number of keys stored in the meta-data.

Warning:

Do not edit this field directly. Instead use the **sif_get_meta_-data** (p. 47), **sif_get_meta_data_binary** (p. 48), **sif_set_meta_-data** (p. 56), and **sif_get_meta_data_binary** (p. 48) functions.

6.2.2.10 long sif_header::n_tiles

The number of tiles that comprise this image.

Warning:

Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.11 long sif_header::n_tiles_across

The number of tiles across the width of an image.

Warning:

Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.12 long sif_header::n_uniform_flags

The number of bytes to store the uniformity flags, i.e. Ceil(number_of_flags / 8).

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.13 long sif_header::tile_bytes

The number of bytes required to store a single tile raster. This is equal to tile_width ∗ tile_height ∗ n_bands ∗ data_unit_size.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.14 long sif_header::tile_header_bytes

The number of bytes needed to store the header for each tile.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

6.2.2.15 long sif_header::tile_height

The height of each tile in pixels.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.
> Non-square tiles have not been tested.

6.2.2.16 long sif_header::tile_width

The width of each tile in pixels.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.
> Non-square tiles have not been tested.

6.2.2.17 long sif_header::user_data_type

A number that is only read from and written to the SIF file header. It has no meaning to the sif-io functions since sif-io processes images without regard to the data type of the pixels. The caller to the library function can use the field to store an integer that represents the data type of the pixels in the image.

Warning:

> Do not edit this field directly. Instead use the **sif_set_user_data_type** (p. 59) function.

6.2.2.18 long sif_header::version

The minimum version of the SIF library needed to read this file.

Warning:

> Do not edit this field directly. Changing the value of this field without a corresponding change to the organization of the file may make it unreadable.

6.2.2.19 long sif_header::width

The width of the image in pixels.

Warning:

> Do not edit this field directly. Changing its value without changing the image layout on disk will make the file unreadable.

The documentation for this struct was generated from the following file:

- **sif-io.h**

6.3 sif_meta_data Struct Reference

A struct for storing meta-data in memory. It stores a node in a linked list of meta-data.

`#include <sif-io.h>`

Data Fields

- char * **key**
- **char * value**
- **unsigned long key_length**
- **unsigned long value_length**
- **sif_meta_data * next**

6.3.1 Detailed Description

A struct for storing meta-data in memory. It stores a node in a linked list of meta-data.

Warning:

> Do not modify this data structure directly. Instead use the **sif_set_-meta_data** (p. 56) and **sif_set_meta_data_binary** (p. 57) functions.

6.3.2 Field Documentation

6.3.2.1 char* sif_meta_data::key

The key identifier of this meta-data field.

6.3.2.2 unsigned long sif_meta_data::key_length

The number of bytes to store the key and its null terminator.

6.3.2.3 struct sif_meta_data* sif_meta_data::next

A pointer to the next meta-data field. The value is NULL if there is no next meta-data field.

6.3.2.4 char* sif_meta_data::value

The value of this meta-data field.

6.3.2.5 unsigned long sif_meta_data::value_length

The number of bytes to store the value. If the value is binary, the null terminator is included in this count.

The documentation for this struct was generated from the following file:

- **sif-io.h**

6.4 sif_tile Struct Reference

A struct for storing a SIF tile header in memory. It stores important information related to a tile, including which of its bands are uniform, and the uniform pixel values of the bands.

`#include <sif-io.h>`

Data Fields

- u_char ∗ **uniform_flags**
- **u_char ∗ uniform_pixel_values**
- **long block_num**

6.4.1 Detailed Description

A struct for storing a SIF tile header in memory. It stores important information related to a tile, including which of its bands are uniform, and the uniform pixel values of the bands.

6.4.2 Field Documentation

6.4.2.1 long sif_tile::block_num

The block location of the file where the tile is stored.

This number is -1 if the tile is completely uniform, i.e. each band in the tile is completely uniform. Note that the bands of a tile (i.e. slices) may have different uniform pixel values. A tile or block is uniform iff each of its slices is uniform.

6.4.2.2 u_char∗ sif_tile::uniform_flags

A byte sequence where the i'th bit in the sequence indicates whether the i'th band in the tile is uniform. The number of bytes is Ceil(n_bands / 8).

6.4.2.3 u_char∗ sif_tile::uniform_pixel_values

A sequence of pixel data units. The i'th data unit represents the uniform pixel value for the i'th band. The number of bytes is n_bands ∗ data_unit_size.

The documentation for this struct was generated from the following file:

- **sif-io.h**

7 SIF File Documentation

7.1 doc/index.dox File Reference

7.2 sif-io.h File Reference

The only header file to include for using sif-io library functions.

```
#include "SIFExport.h"
#include <sys/types.h>
#include <stdio.h>
```

Data Structures

- struct **sif_header**

  A struct for storing a SIF file header in memory.

- **struct sif_tile**

  A struct for storing a SIF tile header in memory. It stores important information related to a tile, including which of its bands are uniform, and the uniform pixel values of the bands.

- **struct sif_meta_data**

  A struct for storing meta-data in memory. It stores a node in a linked list of meta-data.

- **struct sif_file**

  A struct for storing necessary data for the processing of an open file.

Defines

- #define **LONGLONG long long**
- **#define SIF_ERROR_NONE 0**
- **#define SIF_ERROR_MEM 1**
- **#define SIF_ERROR_NULL_FP 2**
- **#define SIF_ERROR_NULL_HDR 3**
- **#define SIF_ERROR_INVALID_BN 4**
- **#define SIF_ERROR_INVALID_TN 5**
- **#define SIF_ERROR_READ 6**
- **#define SIF_ERROR_WRITE 7**

- **#define SIF_ERROR_SEEK 8**
- **#define SIF_ERROR_TRUNCATE 9**
- **#define SIF_ERROR_INVALID_FILE_MODE 10**
- **#define SIF_ERROR_INCOMPATIBLE_VERSION 11**
- **#define SIF_ERROR_META_DATA_KEY 12**
- **#define SIF_ERROR_META_DATA_VALUE 13**
- **#define SIF_ERROR_CANNOT_WRITE_VERSION 14**
- **#define SIF_ERROR_INVALID_BAND 15**
- **#define SIF_ERROR_INVALID_COORD 16**
- **#define SIF_ERROR_INVALID_TILE_SIZE 17**
- **#define SIF_ERROR_INVALID_REGION_SIZE 18**
- **#define SIF_ERROR_INVALID_BUFFER 19**
- **#define SIF_AGREEMENT_SIMPLE "simple"**
- **#define SIF_AGREEMENT_GDAL "gdal"**
- **#define SIF_MAGIC_NUMBER "!∗∗SIF∗∗"**
- **#define SIF_MAGIC_NUMBER_SIZE 8**
- **#define SIF_SIMPLE_ERROR_UNDEFINED_DT 100**
- **#define SIF_SIMPLE_ERROR_INCORRECT_DT 101**
- **#define SIF_SIMPLE_ERROR_UNDEFINED_ENDIAN 102**
- **#define SIF_SIMPLE_UINT8 0**
- **#define SIF_SIMPLE_INT8 1**
- **#define SIF_SIMPLE_UINT16 2**
- **#define SIF_SIMPLE_INT16 3**
- **#define SIF_SIMPLE_UINT32 4**
- **#define SIF_SIMPLE_INT32 5**
- **#define SIF_SIMPLE_UINT64 6**
- **#define SIF_SIMPLE_INT64 7**
- **#define SIF_SIMPLE_FLOAT32 8**
- **#define SIF_SIMPLE_FLOAT64 9**
- **#define SIF_SIMPLE_LITTLE_ENDIAN 0**
- **#define SIF_SIMPLE_BIG_ENDIAN 1**
- **#define SIF_SIMPLE_NATIVE_ENDIAN SIF_SIMPLE_-LITTLE_ENDIAN**
- **#define SIF_SIMPLE_ENDIAN(t) (((int)t)/10)**
- **#define SIF_SIMPLE_TYPE_CODE(bt, ec) ((bt)+(ec))**
- **#define SIF_SIMPLE_BASE_TYPE_-CODE(x) (((int)x)%10)**

Functions

- long **sif_get_version ()**
- **sif_file ∗ sif_open (const char ∗filename, int read_only)**
- **sif_file ∗ sif_create (const char ∗filename, long width, long height, long bands, int data_unit_size, int user_data_type, int consolidate_on_close, int defragment_on_close, long tile_-width, long tile_height, int intrinsic_write)**
- **sif_file ∗ sif_create_copy (sif_file ∗file, const char ∗filename)**
- **int sif_close (sif_file ∗file)**
- **void sif_consolidate (sif_file ∗file)**
- **void sif_defragment (sif_file ∗file)**
- **void sif_set_raster (sif_file ∗file, const void ∗data, long x, long y, long w, long h, long band)**
- **void sif_get_raster (sif_file ∗file, void ∗data, long x, long y, long w, long h, long band)**
- **void sif_fill_tiles (sif_file ∗file, long band, const void ∗value)**
- **void sif_get_tile_slice (sif_file ∗file, void ∗buffer, long tx, long ty, long band)**
- **void sif_set_tile_slice (sif_file ∗file, const void ∗buffer, long tx, long ty, long band)**
- **void sif_fill_tile_slice (sif_file ∗file, long tx, long ty, long band, const void ∗value)**
- **void sif_set_meta_data (sif_file ∗file, const char ∗key, const char ∗value)**
- **void sif_set_meta_data_binary (sif_file ∗file, const char ∗key, const void ∗buffer, int n_bytes)**
- **const char ∗ sif_get_meta_data (sif_file ∗file, const char ∗key)**
- **const void ∗ sif_get_meta_data_binary (sif_file ∗file, const char ∗key, int ∗n_bytes)**
- **int sif_is_shallow_uniform (sif_file ∗file, long x, long y, long w, long h, long band, void ∗uniform_value)**
- **int sif_is_slice_shallow_uniform (sif_file ∗file, long tx, long ty, long band, void ∗uniform_value)**
- **int sif_flush (sif_file ∗file)**
- **void sif_set_user_data_type (sif_file ∗file, long user_data_-type)**
- **long sif_get_user_data_type (sif_file ∗file)**
- **void sif_set_intrinsic_write (sif_file ∗file)**
- **int sif_is_intrinsic_write_set (sif_file ∗file)**
- **void sif_unset_intrinsic_write (sif_file ∗file)**
- **void sif_set_defragment (sif_file ∗file)**
- **int sif_is_defragment_set (sif_file ∗file)**
- **void sif_unset_defragment (sif_file ∗file)**

- void sif_set_consolidate (sif_file ∗file)
- int sif_is_consolidate_set (sif_file ∗file)
- void sif_unset_consolidate (sif_file ∗file)
- void sif_set_affine_geo_transform (sif_file ∗file, const double ∗trans)
- const double ∗ sif_get_affine_geo_transform (sif_file ∗file)
- const char ∗ sif_get_projection (sif_file ∗file)
- void sif_set_projection (sif_file ∗file, const char ∗proj)
- const char ∗ sif_get_agreement (sif_file ∗file)
- void sif_set_agreement (sif_file ∗file, const char ∗agree)
- int sif_get_meta_data_num_items (sif_file ∗file)
- void sif_get_meta_data_keys (sif_file ∗file, const char ∗∗∗key_strs, int ∗num_keys)
- void sif_remove_meta_data_item (sif_file ∗file, const char ∗key)
- void sif_use_file_format_version (sif_file ∗file, long version)
- int sif_is_possibly_sif_file (const char ∗filename)
- const char ∗ sif_get_error_description (int code)
- sif_file ∗ sif_simple_create (const char ∗filename, long width, long height, long bands, int simple_data_type, int consolidate_on_close, int defragment_on_close, long tile_-width, long tile_height, int intrinsic_write)
- sif_file ∗ sif_simple_create_defaults (const char ∗filename, long width, long height, long bands, int simple_data_type)
- void sif_simple_set_endian (sif_file ∗file, int endian)
- int sif_simple_get_endian (sif_file ∗file)
- void sif_simple_set_data_type (sif_file ∗file, int code)
- int sif_simple_get_data_type (sif_file ∗file)
- void sif_simple_set_raster (sif_file ∗file, const void ∗data, long x, long y, long w, long h, long band)
- void sif_simple_get_raster (sif_file ∗file, void ∗data, long x, long y, long w, long h, long band)
- void sif_simple_fill_tiles (sif_file ∗file, long band, const void ∗value)
- void sif_simple_get_tile_slice (sif_file ∗file, void ∗buffer, long tx, long ty, long band)
- void sif_simple_set_tile_slice (sif_file ∗file, const void ∗buffer, long tx, long ty, long band)
- void sif_simple_fill_tile_slice (sif_file ∗file, long tx, long ty, long band, const void ∗value)
- sif_file ∗ sif_simple_open (const char ∗filename, int read_only)
- int sif_simple_is_shallow_uniform (sif_file ∗file, long x, long y, long w, long h, long band, void ∗uniform_value)

- **int sif_simple_is_slice_shallow_uniform (sif_file *file, long tx, long ty, long band, void *uniform_value)**
- **int sif_is_simple (sif_file *file)**
- **int sif_is_sif_simple (const char *filename)**

7.2.1 Detailed Description

The only header file to include for using sif-io library functions.

7.2.2 Define Documentation

7.2.2.1 #define LONGLONG long long

7.2.2.2 #define SIF_AGREEMENT_GDAL "gdal"

A value to set the data-type convention agreement (i.e. "_sif_agree") meta-data field to indicate that the `gdal` data-type convention is used.

7.2.2.3 #define SIF_AGREEMENT_SIMPLE "simple"

A value to set the data-type convention agreement (i.e. "_sif_agree") meta-data field to indicate that the `simple` data-type convention is used.

7.2.2.4 #define SIF_MAGIC_NUMBER "!**SIF**"

A string representing the magic number. The obscure string used to easily identify a file as a file in SIF format.

7.2.2.5 #define SIF_MAGIC_NUMBER_SIZE 8

The number of bytes needed to store the magic number. The obscure string is used to easily identify that a file is likely to be in SIF format.

7.2.2.6 #define SIF_SIMPLE_ERROR_INCORRECT_DT 101

An error to indicate the data type of the image does not correspond to the data type requested.

7.2.2.7 #define SIF_SIMPLE_ERROR_UNDEFINED_DT 100

An error indicating that the data type is not recognized as a data type in the simple data-type convention.

7.2.2.8 #define SIF_SIMPLE_ERROR_UNDEFINED_-ENDIAN 102

An error to indicate an endian code is invalid.

7.2.3 Function Documentation

7.2.3.1 int sif_close (sif_file * *file*)

Close a SIF file.

If the file is open for reading and writing, defragmentation and consolidation occur only if the defragment and consolidate flags are set in the file's header. The file header, tile headers, and meta data are written upon close.

Parameters:

file The SIF file to close.

Returns:

The status of the close.

See also:

sif_header::consolidate (p. 31)
sif_header::defragment (p. 32)
sif_set_consolidate (p. 55)
sif_unset_consolidate (p. 65)
sif_is_consolidate_set (p. 51)

7.2.3.2 void sif_consolidate (sif_file * *file*)

Check all tiles in a file for intrinsic uniformity.

If a tile is found to be intrinsically uniform, its common pixel values for each slice is stored in its header and the physical storage block it is using is freed. If the consolidation flag in the file's header is turned off or the file is read only, this method does nothing.

Parameters:

file The file to mark for uniformity.

7.2.3.3 sif_file* sif_create (const char * *filename*, long *width*, long *height*, long *bands*, int *data_unit_size*, int *user_data_type*, int

***consolidate_ on_ close*, int *defragment_ on_ close*, long *tile_ width*, long *tile_ height*, int *intrinsic_ write*)**

Create a new Sparse Image Format (SIF) file with a given filename and attributes. The file's header and tile headers are written. No space is preallocated for data blocks.

Parameters:

filename The filename of the new file.

width The width of the image to store in the file to create.

height The height of the image to store in the file to create.

bands The number of bands of the image to store in the file to create.

data_ unit_ size The size of a single pixel in bytes, e.g. `sizeof(pixel_-data_type)`.

user_ data_ type A user-defined data type. The SIF I/O functions do not look at this value. This is strictly for the user's reference when opening a pre-existing file.

consolidate_ on_ close Defines whether an intrinsic uniformity check should be applied to dirty tiles during each close.

defragment_ on_ close Defines whether the file should be defragmented during each close.

tile_ width The width of a single tile.

tile_ height The height of a single tile.

intrinsic_ write Defines whether intrinsic uniformity checks should be performed when rasters are written to a file.

Returns:

A file structure containing the constructs needed to manipulate the file created by this function. This function returns NULL if an error occurs during creation.

7.2.3.4 sif_ file∗ sif_ create_ copy (sif_ file ∗ *file*, const char ∗ *filename*)

Create a copy of a SIF file.

Warning:

Note that this function has neither been tested nor ported for use with WIN32+MSVS.

Parameters:

file The file structure pointing to the file to copy. This file is flushed before its contents are read.

filename The filename of the file to store the copy.

Returns:

A file structure containing the constructs needed to manipulate the file copied by this function. This function returns NULL if an error occurs during file creation.

7.2.3.5 void sif_defragment (sif_file * *file*)

Defragment the file.

This results in a sort of the storage blocks so they are in the order of their corresponding tile indices. This enables faster reading/writing of continguous blocks. No unused storage blocks remain in the file (i.e. the used blocks are shifted so that they write over the unused blocks). The file is truncated at the position of the last used storage block byte. Meta-data and the file's header are rewritten.

Parameters:

file The file to defragment.

7.2.3.6 void sif_fill_tile_slice (sif_file * *file*, long *tx*, long *ty*, long *band*, const void * *value*)

Fill a tile slice with a constant value.

If all bands become uniform as a result of this fill, the block for this slice's corresponding cube will be freed.

Warning:

This function has not been tested.

Parameters:

file The file on which to perform the fill.

tx The horizontal index of the tile to fill (0..N-1 indexed).

ty The vertical index of the tile to fill (0..N-1 indexed).

band The band index of the tile to fill (0..N-1 indexed).

value The value to fill all values of the slice. It must be **sif_-header::data_unit_size** (p. 32) bytes in size.

See also:

sif_fill_tiles (p. 46)

7.2.3.7 void sif_fill_tiles (sif_file * *file*, long *band*, const void * *value*)

Fill all tiles of a particular band with a constant value.

If uniformity results as a result of this fill, the corresponding tiles are marked appropriately and the block space they use is freed.

Parameters:

file The file on which to perform the fill.

band The band index of the tile to retrieve (0..N-1 indexed).

value The value to fill all values of the slice. It must be data_unit_size in bytes.

7.2.3.8 int sif_flush (sif_file * *file*)

Flush all remaining unwritten data to the file.

This function immediately returns if the file passed is read-only.

Parameters:

file The SIF file to flush.

Returns:

A non-zero value if no error occurred during the flush.

7.2.3.9 const double* sif_get_affine_geo_transform (sif_file * *file*)

Get the affine georeferencing transform of an open file.

Parameters:

file The file to get the transform.

Returns:

An array of six doubles representing the transform.

See also:

sif_set_affine_geo_transform (p. 54)
sif_header::affine_geo_transform (p. 31)

7.2.3.10 const char* sif_get_agreement (sif_file * *file*)

Return a string indicating the data type convention used in this file. If the string is `"gdal"` then the GDT type codes in the GDAL library are used. If the string is `"simple"` then the convention presented earlier in this document is used.

Parameters:

file The file from which to get the projection string.

Returns:

The convention agreement string.

See also:

sif_set_agreement (p. 55)

7.2.3.11 const char* sif_get_error_description (int *code*)

Returns a description of a SIF error code.

Parameters:

code The code of the error.

Returns:

A description of the error as a string.

7.2.3.12 const char* sif_get_meta_data (sif_file * *file*, const char * *key*)

Get a string meta-data field with a given key. This function returns 0 and sets the error field in the file's header if the buffer stored for this meta-data is not a null-terminated string or if the field with the given key string could not be found.

Parameters:

file The file on which to set the meta-data field.

key The key of the field to set.

Returns:

The value of the field.

See also:

> **sif_get_meta_data_binary** (p. 48)
> **sif_set_meta_data** (p. 56)
> **sif_set_meta_data_binary** (p. 57)

7.2.3.13 const void∗ sif_get_meta_data_binary (sif_file ∗ *file*, const char ∗ *key*, int ∗ *n_bytes*)

Get a string meta-data field with a given key. This function returns 0 and sets the error field in the file's header.

Parameters:

- ***file*** The file to set the meta-data.
- ***key*** The key of the field to set.
- ***n_bytes*** A pointer to an integer value. This value is set to the size of the buffer returned.

Returns:

The value of the field.

7.2.3.14 void sif_get_meta_data_keys (sif_file ∗ *file*, const char ∗∗∗ *key_strs*, int ∗ *num_keys*)

Retrieve the keys of the meta data stored in the file. It is the responsibility of the caller to free the memory to which ∗key_strs points but not (∗key_strs)[i] for any i, non-negative i < **sif_header::n_keys** (p. 33).

Parameters:

- ***file*** The file from which to retrieve the meta-data keys.
- ***key_strs*** A pointer pointing to the pointer to set to the location of the array of strings. The last value of the array of strings is set to 0 (sentinel).
- ***num_keys*** A pointer to the int to store the number of keys retrieved by this function.

7.2.3.15 int sif_get_meta_data_num_items (sif_file ∗ *file*)

Get the number of meta data (key, value) pairs in this file.

Parameters:

- ***file*** The file from which to get the number of meta-data items.

7.2.3.16 const char* sif_get_projection (sif_file * *file*)

Return the projection string of an open file. It is expected to be in OpenGIS WKT format. The string is stored in the "_sif_proj" field in the meta-data region of the file. If the projection string cannot be found, the empty string is returned.

Parameters:

file The file from which to get the projection string.

Returns:

The projection string.

See also:

sif_set_projection (p. 57)

7.2.3.17 void sif_get_raster (sif_file * *file*, void * *data*, long *x*, long *y*, long *w*, long *h*, long *band*)

Reads a rectangular raster region from a file. It may overlap multiple tiles in the file.

Warning:

This function has not been tested.

Parameters:

file The file on which to read the raster plane out.

data The buffer to store the raster plane.

x The starting horizontal pixel offset (0..N-1 indexed) to read.

y The starting vertical pixel offset (0..N-1 indexed) to read.

w The width of the region.

h The height of the region.

band The band offset (0..N-1 indexed).

See also:

sif_set_raster (p. 57)
sif_get_tile_slice (p. 50)
sif_set_tile_slice (p. 58)

7.2.3.18 void sif_get_tile_slice (sif_file * *file*, void * *buffer*, long *tx*, long *ty*, long *band*)

Retrieve a tile slice.

If the tile is uniform, no access to the disk is made; instead, the uniform pixel value for the band in the tile's header is used to fill the buffer. The buffer must contain enough bytes to hold a slice.

Parameters:

file The file on which to perform the fill.

tx The horizontal index of the slice to retrieve (0..N-1 indexed).

ty The vertical index of the slice to retrieve (0..N-1 indexed).

band The band index of the slice to retrieve (0..N-1 indexed).

buffer The buffer to store the tile slice. It must be data_unit_size in bytes.

See also:

sif_fill_tile_slice (p. 45)

7.2.3.19 long sif_get_user_data_type (sif_file * *file*)

Get the user data type integer for the file.

This value does not change the behavior of any **sif-io.h** (p. 38) functions. The user may use it to store an integer representing the data type of the pixel values in the image.

Parameters:

file The file on which to get the user-defined data type flag.

Returns:

The user-defined data type of the data units in the file.

7.2.3.20 long sif_get_version ()

Return the latest version of the SIF file format that the currently loaded SIF library can process.

Returns:

The latest version number of a SIF file this library can process.

7.2.3.21 int sif_is_consolidate_set (sif_file * *file*)

Return whether the file will be scheduled for consolidation on its close. Used blocks are moved toward the begining of the file, taking up the space of unused blocks before them. If there are no unused blocks or all of the unused blocks are at the end of the file, this file is simply truncated at the location of the first byte of the unused block and the meta-data is rewritten.

Parameters:

file The file to check.

Returns:

The value of the consolidation flag.

See also:

sif_unset_consolidate (p. 65)
sif_set_consolidate (p. 55)
sif_consolidate (p. 43)
sif_close (p. 43)
sif_header::consolidate (p. 31)

7.2.3.22 int sif_is_defragment_set (sif_file * *file*)

Unsets the defragmentation flag.

This cancels defragmentation when the file is closed.

Parameters:

file The file to change.

Returns:

The value of the defragmentation flag.

See also:

sif_set_defragment (p. 56)
sif_unset_defragment (p. 66)
sif_defragment (p. 45)
sif_close (p. 43)
sif_header::defragment (p. 32)

7.2.3.23 int sif_is_intrinsic_write_set (sif_file ∗ *file*)

Return the value of the uniformity flag. When true, a uniformity check is perfomed on all dirty tiles during close.

Parameters:

file The file to check.

See also:

sif_unset_intrinsic_write (p. 66)
sif_is_intrinsic_write_set (p. 52)
sif_is_shallow_uniform (p. 52)
sif_is_slice_shallow_uniform (p. 53)

Returns:

The value of the flag.

7.2.3.24 int sif_is_possibly_sif_file (const char ∗ *filename*)

Returns a positive value if the file referred to by `filename` could possibly be a SIF file.

Parameters:

filename The filename of the file to check.

Returns:

A boolean indicating the result of the check.

7.2.3.25 int sif_is_shallow_uniform (sif_file ∗ *file*, long *x*, long *y*, long *w*, long *h*, long *band*, void ∗ *uniform_value*)

Determine if the tiles comprising a region are shallow uniform.

Parameters:

file The file to perform the check.

x The starting horizontal pixel offset (0..N-1 indexed) of the region to check.

y The starting vertical pixel offset (0..N-1 indexed) of the region to check.

w The width of the region.to check.

h The height of the region to check.

band The band offset (0..N-1 indexed).

uniform_value This value is only meaningful when this function returns true (non-zero). It is expected that the pointer passed point to at least data_unit_size bytes. When the region is completely uniform, the uniform pixel value is stored here.

Returns:

0 if the tiles are not shallow uniform or a memory allocation error occurred, otherwise a non-zero value.

7.2.3.26 int sif_is_sif_simple (const char * *filename*)

Return if the file referred to by the filename conforms to the "simple" data type convention.

Parameters:

filename The filename of the file on which to perform the operation.

Returns:

A non-zero value if the file referred to by the filename conforms to the "simple" data type convention. If the file could not be opened, -1 is returned.

7.2.3.27 int sif_is_simple (sif_file * *file*)

Return if the file conforms to the "simple" data type convention.

Parameters:

file The file on which to perform the operation.

Returns:

A non-zero value if the file conforms to the "simple" data type convention.

7.2.3.28 int sif_is_slice_shallow_uniform (sif_file * *file*, long *tx*, long *ty*, long *band*, void * *uniform_value*)

Determine if a tile has shallow uniformity.

Parameters:

file The file to perform the check.

tx The horizontal tile index (0..N-1 indexed) of the region to check.

ty The vertical tile index (0..N-1 indexed) of the region to check.

band The band offset (0..N-1 indexed).

uniform_ value This value is only meaningful when this function returns true (non-zero). It is expected that the pointer passed point to at least data_unit_size bytes. When the region is completely uniform, the uniform pixel value is stored here.

Returns:

0 if the tiles are non-uniform or a memory allocation error occurred, otherwise a non-zero value.

7.2.3.29 sif_file∗ sif_open (const char ∗ *filename*, int *read_ only*)

Open a Sparse Image File (SIF) format file for reading or update.

Parameters:

filename The filename of the SIF file to open.

read_ only A flag indicating whether to open as read-only (1) or update (0).

Returns:

A file structure containing all the constructs needed to manipulate the opened SIF file is returned. NULL is returned if an error occured during open.

7.2.3.30 void sif_remove_meta_data_item (sif_file ∗ *file*, const char ∗ *key*)

Removes a meta-data item by its key string.

Parameters:

file The file from which to remove a meta-data item.

key The key of the meta-data item to remove.

7.2.3.31 void sif_set_affine_geo_transform (sif_file ∗ *file*, const double ∗ *trans*)

Set the affine georeferencing transform of an open file.

Parameters:

file The file to set the affine georeferencing transform.

trans A double array of size 6 with the new value of the georeferencing transform.

See also:

sif_get_affine_geo_transform (p. 46)
sif_header::affine_geo_transform (p. 31)

7.2.3.32 void sif_set_agreement (sif_file * *file*, const char * *agree*)

Set a string indicating the data type convention used in this file. If the string is "gdal" then the GDT type codes in the GDAL library are used. If the string is "simple" then the convention presented earlier in this document is used.

Parameters:

file The file to set the projection string.

agree The new projection string value.

See also:

sif_get_agreement (p. 47)

7.2.3.33 void sif_set_consolidate (sif_file * *file*)

Set the consolidation flag.

Consolidation is then performed during the files close.

Parameters:

file The file to change.

See also:

sif_unset_consolidate (p. 65)
sif_is_consolidate_set (p. 51)
sif_consolidate (p. 43)
sif_close (p. 43)
sif_header::consolidate (p. 31)

7.2.3.34 void sif_set_defragment (sif_file * *file*)

Set the defragmentation flag.

Defragmentation is then performed during the file's close. Data blocks are rearranged in the order they appear in the image.

Parameters:

file The file to change.

See also:

sif_unset_defragment (p. 66)
sif_is_defragment_set (p. 51)

7.2.3.35 void sif_set_intrinsic_write (sif_file * *file*)

Set the uniformity flag. This results in a pixel uniformity check on all dirty tiles.

Parameters:

file The file to change.

See also:

sif_is_intrinsic_write_set (p. 52)
sif_unset_intrinsic_write (p. 66)
sif_is_shallow_uniform (p. 52)
sif_is_slice_shallow_uniform (p. 53)

7.2.3.36 void sif_set_meta_data (sif_file * *file*, const char * *key*, const char * *value*)

Set a meta-data field with a given key to a value defined by a null-terminated character string.

Parameters:

file The file to set the meta-data.

key The key of the field to set.

value The value to set the field.

See also:

sif_set_meta_data_binary (p. 57)
sif_get_meta_data (p. 47)
sif_get_meta_data_binary (p. 48)

7.2.3.37 void sif_set_meta_data_binary (sif_file * *file*, const char * *key*, const void * *buffer*, int *n_bytes*)

Set a meta-data field with a given key to a given sequence of bytes.

The length of the value is passed here, thereby allowing for binary, non-null-terminated, meta-data.

Parameters:

file The file on which to set the meta-data field.

key The key of the field to set.

buffer The value to set the field.

n_bytes The length of the value (in bytes).

Warning:

The meta-data is not written to the file until the file is closed or flushed.

See also:

sif_set_meta_data (p. 56)
sif_get_meta_data (p. 47)
sif_get_meta_data_binary (p. 48)

7.2.3.38 void sif_set_projection (sif_file * *file*, const char * *proj*)

Set the projection string of an open file. This is expected to be empty ("") or in OpenGIS WKT format. The string is stored in the "_sif_proj" field in the meta-data region of the file.

Parameters:

file The file to set the projection string.

proj The new projection string value.

See also:

sif_get_projection (p. 49)

7.2.3.39 void sif_set_raster (sif_file * *file*, const void * *data*, long *x*, long *y*, long *w*, long *h*, long *band*)

Writes a rectangular image region to a file.

The tiles changed by this write are not checked for pixel uniformity. This results in the dirty flags in their respective tile headers being set to true. This results in a uniformity check during the file's close unless the uniformity check flag is set to false in the file's header. Also, any fragmentation caused by this function is not resolved until the file is closed.

Warning:

This function has not been tested.

Parameters:

file The file on which to write the raster plane.

data The buffer containing the raster to write.

x The starting horizontal pixel offset (0..N-1 indexed) to write.

y The starting vertical pixel offset (0..N-1 indexed) to write.

w The width of the region.

h The height of the region.

band The band offset (0..N-1 indexed).

See also:

sif_get_raster (p. 49)
sif_get_tile_slice (p. 50)
sif_set_tile_slice (p. 58)

7.2.3.40 void sif_set_tile_slice (sif_file * *file*, const void * *buffer*, long *tx*, long *ty*, long *band*)

Store a tile slice.

A check is not made to determine pixel uniformity. The tile's dirty flag is set to true. This results in a uniformity check during the file's close, unless uniformity check flag is disabled in the file's header. Also, any fragmentation caused by this function is not resolved until the file is closed.

Parameters:

file The file on which to perform the write.

tx The horizontal index (0..N-1 indexed) of the slice to write.

ty The vertical index (0..N-1 indexed) of the slice to write.

band The band index (0..N-1 indexed) of the slice to write.

buffer The buffer to write. It must have enough bytes for an entire tile slice, e.g. **sif_header::tile_width** (p. 34) * **sif_header::tile_-height** (p. 34) * **sif_header::data_unit_size** (p. 32).

7.2.3.41 void sif_set_user_data_type (sif_file * *file*, long *user_-data_type*)

Set the user data type for the file.

This value does not change the behavior of any sif-io functions. The user may use it to store an integer representing the data type of the pixel values in the image.

Parameters:

- ***file*** The file to change the data type flag.
- ***user_data_type*** The value of the new user-defined data type flag.

7.2.3.42 sif_file* sif_simple_create (const char * *filename*, long *width*, long *height*, long *bands*, int *simple_data_type*, int *consolidate_on_close*, int *defragment_on_close*, long *tile_width*, long *tile_height*, int *intrinsic_write*)

Create a new Sparse Image Format (SIF) file with a given filename and attributes. The file's header and tile headers are written. No space is preallocated for data blocks. The simple data-type convention is used. When reading, writing, or filling the image file created by this function, the `sif_simple_*` functions must be used to ensure the image data elements are written with the proper byte order.

Unless **sif_simple_set_endian** (p. 64) is called prior to reading or writing any image raster, the pixels will be stored in native byte order.

Parameters:

- ***filename*** The filename of the new file.
- ***width*** The width of the image to store in the file to create.
- ***height*** The height of the image to store in the file to create.
- ***bands*** The number of bands of the image to store in the file to create.
- ***simple_data_type*** The data type code.
- ***tile_width*** The width of a single tile.
- ***tile_height*** The height of a single tile.
- ***consolidate_on_close*** Defines whether a pixel uniformity check should be applied to dirty tiles during each close.
- ***defragment_on_close*** Defines whether the file should be defragmented during each close.
- ***intrinsic_write*** Defines whether intrinsic uniformity checks should be performed when rasters are written to a file.

Returns:

A file structure containing the constructs needed to manipulate the file created by this function. This function returns NULL if an error occurs during creation.

7.2.3.43 sif_file* sif_simple_create_defaults (const char * *filename*, long *width*, long *height*, long *bands*, int *simple_data_type*)

Create a new Sparse Image Format (SIF) file with a given filename and attributes. The file's header and tile headers are written. No space is preallocated for data blocks. The simple data-type convention is used.

When reading, writing, or filling the image file created by this function, the `sif_simple_*` functions must be used to ensure the image data elements are written with the proper byte order.

Unless **sif_simple_set_endian** (p. 64) is called prior to reading or writing any image raster, the pixels will be stored in native byte order.

The **sif_header::consolidate** (p. 31), **sif_header::defragment** (p. 32) and the sif_header::intrisic_write flags are all set to true. The **sif_header::tile_-width** (p. 34) and **sif_header::tile_height** (p. 34) are both set to 64.

Parameters:

filename The filename of the new file.

width The width of the image to store in the file to create.

height The height of the image to store in the file to create.

bands The number of bands of the image to store in the file to create.

simple_data_type The data type code.

Returns:

A file structure containing the constructs needed to manipulate the file created by this function. This function returns NULL if an error occurs during creation.

7.2.3.44 void sif_simple_fill_tile_slice (sif_file * *file*, long *tx*, long *ty*, long *band*, const void * *value*)

Fill a tile slice with a constant value. The value is converted from host order to the byte order in the file.

Parameters:

file The file on which to perform the operation.

tx The horizontal index of the tile slice.

ty The vertical index of the tile slice.

band The band of the tile to which the slice corresponds.

value The pointer to the value to fill the tile slice.

7.2.3.45 void sif_simple_fill_tiles (sif_file * *file*, long *band*, const void * *value*)

Fill a band with a constant value. The byte order of the value is converted to the byte order of the file's image.

Parameters:

file The file on which to perform the operation.

band The band to fill.

value The pointer to the value with which to fill the band.

7.2.3.46 int sif_simple_get_data_type (sif_file * *file*)

Get the simple data type code of the pixel values in the image of this file.

Parameters:

file The file on which to perform the operation.

Returns:

The simple data type code of the pixel values.

See also:

Simple Data Type Convention Macro Definitions (p. 24)

7.2.3.47 int sif_simple_get_endian (sif_file * *file*)

Get the network byte order of the pixel values in the image of this file.

Parameters:

file The file on which to perform the operation.

Returns:

The endian code of this file.

7.2.3.48 void sif_simple_get_raster (sif_file * *file*, void * *data*, long *x*, long *y*, long *w*, long *h*, long *band*)

Read a rectangular region from a file. The byte order of the data values in the buffer is automatically converted to the byte order of the file.

Parameters:

file The file on which to perform the operation.

data The buffer into which the data will be read.

x The horizontal starting index of the file.

y The vertical starting index of the file.

w The width of the region.

h The height of the region.

band The band of the region.

7.2.3.49 void sif_simple_get_tile_slice (sif_file * *file*, void * *buffer*, long *tx*, long *ty*, long *band*)

Retrieve a tile slice. The byte order of the data values in the buffer are in host byte order.

Parameters:

file The file on which to perform the operation.

buffer The buffer to store the slice.

tx The horizontal index of the tile slice.

ty The vertical index of the tile slice.

band The band of the tile to which the slice corresponds.

7.2.3.50 int sif_simple_is_shallow_uniform (sif_file * *file*, long *x*, long *y*, long *w*, long *h*, long *band*, void * *uniform_value*)

Determine if the tiles comprising a region are shallow uniform. The "simple" data-type convention is assumed.

Parameters:

file The file to perform the check.

x The starting horizontal pixel offset (0..N-1 indexed) of the region to check.

y The starting vertical pixel offset (0..N-1 indexed) of the region to check.

w The width of the region.to check.

h The height of the region to check.

band The band offset (0..N-1 indexed).

uniform_value This value is only meaningful when this function returns true (non-zero). It is expected that the pointer passed point to at least data_unit_size bytes. When the region is completely uniform, the uniform pixel value is stored here.

Returns:

0 if the tiles are not shallow uniform or a memory allocation error occurred, otherwise a non-zero value.

7.2.3.51 int sif_simple_is_slice_shallow_uniform (sif_file * *file*, long *tx*, long *ty*, long *band*, void * *uniform_value*)

Determine if a tile slice has shallow uniformity. The simple user data type convention is assumed.

Parameters:

file The file to perform the check.

tx The horizontal tile index (0..N-1 indexed) of the region to check.

ty The vertical tile index (0..N-1 indexed) of the region to check.

band The band offset (0..N-1 indexed).

uniform_value This value is only meaningful when this function returns true (non-zero). It is expected that the pointer passed point to at least data_unit_size bytes. When the region is completely uniform, the uniform pixel value is stored here.

Returns:

0 if the tiles are non-uniform or a memory allocation error occurred, otherwise a non-zero value.

7.2.3.52 sif_file* sif_simple_open (const char * *filename*, int *read_-only*)

Open a Sparse Image File (SIF) format file for reading or update. The file is expected to use the Simple data type convention.

When reading, writing, or filling the image file opened by this function, the `sif_simple_*` functions must be used to ensure the image data elements are written with the proper byte order.

If the file does not use the simple data-type convention, 0 is returned.

Parameters:

filename The filename of the SIF file to open.

read_only A flag indicating whether to open as read-only (1) or update (0).

Returns:

A file structure containing all the constructs needed to manipulate the opened SIF file is returned. NULL is returned if an error occured during open.

7.2.3.53 void sif_simple_set_data_type (sif_file ∗ *file*, int *code*)

Set the simple data type code for the pixel. Note that this field should never be set once a raster is written to a file.

Parameters:

file The file on which to perform the operation.

code The simple data type code of the pixel values.

See also:

Simple Data Type Convention Macro Definitions (p. 24)

7.2.3.54 void sif_simple_set_endian (sif_file ∗ *file*, int *endian*)

Set the network byte order for the pixel. Note that this field should never be set once a raster is written to a file. This field must be set to one of **SIF_SIMPLE_LITTLE_ENDIAN** (p. 26) or **SIF_SIMPLE_BIG_-ENDIAN** (p. 25).

Parameters:

file The file on which to perform the operation.

endian The endian code to set the file.

7.2.3.55 void sif_simple_set_raster (sif_file ∗ *file*, const void ∗ *data*, long *x*, long *y*, long *w*, long *h*, long *band*)

Write a rectangular region to a file. The byte order of the data values is automatically converted from host order to the the byte order of the file.

Parameters:

file The file on which to perform the operation.

data The buffer containing the data to write.

x The horizontal starting index of the file.

y The vertical starting index of the file.

w The width of the region.

h The height of the region.

band The band of the region.

7.2.3.56 void sif_simple_set_tile_slice (sif_file ∗ *file*, const void ∗ *buffer*, long *tx*, long *ty*, long *band*)

Store a tile slice. The byte order of the data values in the buffer are converted to the byte order of the file.

Parameters:

file The file on which to perform the operation.

buffer The buffer to store the slice.

tx The horizontal index of the tile slice.

ty The vertical index of the tile slice.

band The band of the tile to which the slice corresponds.

7.2.3.57 void sif_unset_consolidate (sif_file ∗ *file*)

Unset the consolidation flag.

Consolidation is then performed during the files close.

Parameters:

file The file to change.

See also:

sif_set_consolidate (p. 55)
sif_is_consolidate_set (p. 51)
sif_consolidate (p. 43)
sif_close (p. 43)
sif_header::consolidate (p. 31)

7.2.3.58 void sif_unset_defragment (sif_file * *file*)

Set the defragmentation flag.

Defragmentation on the file's close is cancelled.

Parameters:

file The file to change.

See also:

sif_set_defragment (p. 56)
sif_is_defragment_set (p. 51)
sif_defragment (p. 45)
sif_close (p. 43)
sif_header::defragment (p. 32)

7.2.3.59 void sif_unset_intrinsic_write (sif_file * *file*)

Unset the uniformity flag.

This cancels pixel uniformity checks during close.

Parameters:

file The file to change.

See also:

sif_unset_intrinsic_write (p. 66)
sif_set_intrinsic_write (p. 56)
sif_is_shallow_uniform (p. 52)
sif_is_slice_shallow_uniform (p. 53)

7.2.3.60 void sif_use_file_format_version (sif_file * *file*, long *version*)

Specifies that when the file is written, the file should be written with using the SIF file format version specified. If the version is not supported for write, a SIF_ERROR_CANNOT_WRITE_VERSION error is set in the header's error code field.

Index

```
/**
 * Copyright (C) 2004-2006 The Regents of the University of California.
 * Copyright (C) 2006-2008 Los Alamos National Security, LLC.
 *
 * This material was produced under U.S. Government contract
 * DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is
 * operated by Los Alamos National Security, LLC for the U.S.
 * Department of Energy. The U.S. Government has rights to use,
 * reproduce, and distribute this software.  NEITHER THE
 * GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY,
 * EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS
 * SOFTWARE.  If software is modified to produce derivative works, such
 * modified software should be clearly marked, so as not to confuse it
 * with the version available from LANL.
 *
 * Additionally, this library is free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either version 2.1
 * of the License, or (at your option) any later version. Accordingly, this
 * library is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * Los Alamos Computer Code LA-CC-06-105
 */

#ifndef SIFExport_h_
#define SIFExport_h_
#if defined(_MSC_VER)
#if defined(SIF_DLL)
#define SIF_EXPORT __declspec(dllexport)
#else
#define SIF_EXPORT __declspec(dllimport)
#endif
#pragma warning(disable : 4503)
#else
#define SIF_EXPORT
#endif
#endif
```

```
/**
 * File:             sif-io.c
 * Date:             December 17, 2004
 * Author:           Damian Eads
 * Description:      A C library for manipulating Sparse Image Format (SIF) files.
 *
 * Copyright (C) 2004-2006 The Regents of the University of California.
 * Copyright (C) 2006-2008 Los Alamos National Security, LLC.
 *
 * This material was produced under U.S. Government contract
 * DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is
 * operated by Los Alamos National Security, LLC for the U.S.
 * Department of Energy. The U.S. Government has rights to use,
 * reproduce, and distribute this software.  NEITHER THE
 * GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY,
 * EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS
 * SOFTWARE.  If software is modified to produce derivative works, such
 * modified software should be clearly marked, so as not to confuse it
 * with the version available from LANL.
 *
 * Additionally, this library is free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either version 2.1
 * of the License, or (at your option) any later version. Accordingly, this
 * library is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * Los Alamos Computer Code LA-CC-06-105
 */

#include "sif-io.h"

#ifndef WIN32
#include <errno.h>
#include <strings.h>
#include <unistd.h>
#endif

/**#define SIF_ASSERT assert(0)**/  /** used for debugging.**/
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <math.h>

/** By defining the macro definition below, as opposed to the one above,
    SIF does not return when an error occurs when executing a function
    in the SIF API. */

#define SIF_ASSERT

#ifdef WIN32
#define SIF_RECORD file->sys_error_no = GetLastError();
#else
#define SIF_RECORD file->sys_error_no = errno;
#endif

#define SIF_ERROR_CHECK(cond, code) if (cond) { file->error = code; file->error_line_no = __LI
NE__; SIF_RECORD; SIF_ASSERT; }
#define SIF_ERROR_CHECK_RETURN(cond, code, retcode) if (cond) { file->error = code; file->erro
r_line_no = __LINE__; SIF_RECORD; SIF_ASSERT; return retcode; }
#define SIF_ERROR_CHECK_RETURN_V(cond, code) if (cond) { file->error = code; file->error_line_
no = __LINE__; SIF_RECORD; SIF_ASSERT; return; }
#define SIF_ERROR_RETURN(code) if (file->error != 0) { SIF_RECORD; SIF_ASSERT; return code; }
#define SIF_ERROR_RETURN_V() if (file->error != 0) { SIF_RECORD; SIF_ASSERT; return; }

#define SIF_SIZE_FLAG_ARRAY(num_bits) (CEIL_DIV(num_bits, 8))
#define SIF_GET_BIT(uca, i) ((uca[i / 8] >> (7-(i % 8))) & 0x1)
#define SIF_SET_BIT(uca, i) (uca[i / 8] |= ((0x1) << (7-(i % 8))))
#define SIF_CLEAR_BIT(uca, i) (uca[i / 8] &= ~((0x1) << (7-(i % 8))))

#define SIF_HASH_TABLE_SIZE 128

/** The three below are function versions of the three macros above. **/

/**
 * Return a non-zero integer if i'th bit in the character array is set.
 *
 * @param v    The character array to retrieve the bit.
 * @param i    The bit number of the bit to retrieve.
 *
 * @return A non-zero integer if the bit is set.
 */
u_int _sif_get_bit(const u_char *v, u_int i) {
  return SIF_GET_BIT(v, i);
}

/**
 * Sets the i'th bit in the character array.
 *
 * @param v    The character array to retrieve the bit.
 * @param i    The bit number of the bit to retrieve.
 *
 * @return The first character in the array.
 */
int _sif_set_bit(u_char *v, u_int i) {
  SIF_SET_BIT(v, i);
  return (int)*v;
}

/**
 * Clears the i'th bit in the character array.
 *
 * @param v    The character array to retrieve the bit.
 * @param i    The bit number of the bit to retrieve.
 *
 * @return The first character in the array.
 */
int _sif_clear_bit(u_char *v, u_int i) {
  SIF_CLEAR_BIT(v, i);
  return (int)*v;
}

/** Dan Bernstein's hash. */

unsigned long _sif_hash(const unsigned char *str) {
  unsigned long hash = 5381;
  int c = *str;
  while (c != 0) {
    hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    str++;
    c = *str;
  }
  return hash;
}
```

```
/**
 * Swaps the data elements in a buffer to change the byte order.
 *
 * @param n_bytes    The number of bytes in the buffer on which to change the byte order.
 * @param elem_size  The number of bytes per element, assumed to be even.
 */

void _sif_swap_bytes(unsigned char *buffer,
                              int n_bytes, int elem_size) {
  int i, j, k, m;
  const int esz_h = elem_size / 2;   /** Half the number of bytes per element. */
  unsigned char tmp;
  for (i = 0; i < esz_h; i++) {
    m = esz_h - i - 1;  /* The byte index to byte swapped. */
    /** For each element, the i'th byte of the element is going to be swapped with
        the m'th.

        We loop over the bytes in the buffer with two looper variables, j and k.

          j: the byte index (in the buffer!) of the byte of the element to swap
          k: the byte index (in the buffer!) of the byte of the element to swap

    **/
    for (j = i, k = m; k < n_bytes; j += elem_size, k += elem_size) {
      tmp = buffer[k];
      buffer[k] = buffer[i];
      buffer[i] = tmp;
    }
  }
}


void _sif_buffer_host_to_big(unsigned char *buffer, int n_bytes, int elem_size) {
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  _sif_swap_bytes(buffer, n_bytes, elem_size);
#elif __BYTE_ORDER == __BIG_ENDIAN
  return;
#endif
}

void _sif_buffer_host_to_little(unsigned char *buffer, int n_bytes, int elem_size) {
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  return;
#elif __BYTE_ORDER == __BIG_ENDIAN
  _sif_swap_bytes(buffer, n_bytes, elem_size);
#endif
}

void _sif_buffer_host_to_code(unsigned char *buffer, int n_bytes, int elem_size, int simple_en
dian_code) {
  if (simple_endian_code == SIF_SIMPLE_BIG_ENDIAN) {
    _sif_buffer_host_to_big(buffer, n_bytes, elem_size);
  }
  if (simple_endian_code == SIF_SIMPLE_LITTLE_ENDIAN) {
    _sif_buffer_host_to_little(buffer, n_bytes, elem_size);
  }
}

void _sif_buffer_little_to_host(unsigned char *buffer, int n_bytes, int elem_size) {
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  return;
#elif __BYTE_ORDER == __BIG_ENDIAN
```

```
  _sif_swap_bytes(buffer, n_bytes, elem_size);
#endif
}

void _sif_buffer_big_to_host(unsigned char *buffer, int n_bytes, int elem_size) {
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  _sif_swap_bytes(buffer, n_bytes, elem_size);
#elif __BYTE_ORDER == __BIG_ENDIAN
  return;
#endif
}

void _sif_buffer_code_to_host(unsigned char *buffer, int n_bytes, int elem_size, int simple_en
dian_code) {
  if (simple_endian_code == SIF_SIMPLE_BIG_ENDIAN) {
    _sif_buffer_big_to_host(buffer, n_bytes, elem_size);
  }
  if (simple_endian_code == SIF_SIMPLE_LITTLE_ENDIAN) {
    _sif_buffer_little_to_host(buffer, n_bytes, elem_size);
  }
}

void _sif_fwrite64_with_swap(sif_file *file, size_t size, size_t nmemb, const void *buffer) {
  SIF_ERROR_CHECK_RETURN_V(_sif_simple_alloc_region_buffer(file, size * nmemb) == 0, SIF_ERROR
_MEM);
  memcpy(file->simple_region_buffer, buffer, size * nmemb);
  _sif_swap_bytes(file->simple_region_buffer, size * nmemb, size);
  FWRITE64V(file->simple_region_buffer, size, nmemb, file->fp);
}

void _sif_fwrite64_without_swap(sif_file *file, size_t size, size_t nmemb, const void *buffer)
 {
  FWRITE64V(buffer, size, nmemb, file->fp);
}

void _sif_fread64_with_swap(sif_file *file, size_t size, size_t nmemb, void *buffer) {
  FREAD64V(buffer, size, nmemb, file->fp);
  _sif_swap_bytes(buffer, size * nmemb, size);
}

void _sif_fread64_without_swap(sif_file *file, size_t size, size_t nmemb, void *buffer) {
  FREAD64V(buffer, size, nmemb, file->fp);
}

typedef void (*sif_buffer_preprocessor) (sif_file *file, size_t size, size_t nmemb, void *buff
er);

/** These two macros check if the file pointer is null, the header is not null, and if the fil
e version
    is compatible with this library version. */

#define SIF_CHECK_FILE(fp) if (fp == 0) { \
                                       return 0; \
                                   } else if (fp->header == 0) { \
                                       fp->error = SIF_ERROR_NULL_HDR;    \
                                       return 0; \
                                   } else if (fp->header->version > SIF_VERSION) { \
                                       fp->error = SIF_ERROR_INCOMPATIBLE_VERSION; \
                                       return 0; \
                                   }

#define SIF_CHECK_FILE_V(fp) if (fp == 0) { \
                                       return; \
                                    } else if (fp->header == 0) { \
                                      fp->error = SIF_ERROR_NULL_HDR; \
                                      return; \
                                    } else if (fp->header->version > SIF_VERSION) { \
                                      fp->error = SIF_ERROR_INCOMPATIBLE_VERSION; \
                                      return; \
                                    }

#ifdef WIN32
#define FILE_IS_OKAY(fp) (fp != INVALID_HANDLE_VALUE)
#define bzero(buf, n) memset(buf, 0, n);
#define FCLOSE64(fp) _sif_close_win(fp);
#define REWIND64(fp) SIF_ERROR_CHECK_RETURN(_sif_fseek_win(fp, 0, FILE_BEGIN) == -1, SIF_ERROR
_SEEK, 0)
#define REWIND64V(fp) SIF_ERROR_CHECK_RETURN_V(_sif_fseek_win(fp, 0, FILE_BEGIN) == -1, SIF_ER
ROR_SEEK)
#define REWIND64NEC(fp) _sif_fseek_win(fp, 0, FILE_BEGIN)
#define FSEEK64(fp, loc, typ) SIF_ERROR_CHECK_RETURN(_sif_fseek_win(fp, loc, _sif_linux_to_win
32_seek_type(typ)) == -1, SIF_ERROR_SEEK, 0)
#define FSEEK64V(fp, loc, typ) SIF_ERROR_CHECK_RETURN_V(_sif_fseek_win(fp, loc, _sif_linux_to_
win32_seek_type(typ)) == -1, SIF_ERROR_SEEK)
#define FSEEK64NEC(fp, loc, typ) _sif_fseek_win(fp, loc, _sif_linux_to_win32_seek_type(typ))
#define FWRITE64(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN(_sif_fwrite_win(fp, buf, els * nel)
 == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64V(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN_V(_sif_fwrite_win(fp, buf, els * n
el) == 0, SIF_ERROR_WRITE)
#define FWRITE64CNT(buf, cnt, fp) cnt += sizeof(buf); SIF_ERROR_CHECK_RETURN(_sif_fwrite_win(f
p, &(buf), sizeof(buf)) == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64INT32CNT(vv, cnt, fp) cnt += 4; SIF_ERROR_CHECK_RETURN(_sif_write_int32(fp, vv
) == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64INT32(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_write_int32(fp, vv) == 0, SIF_ERROR_
WRITE, 0)
#define FWRITE64DOUBLE64CNT(vv, cnt, fp) cnt += 8; SIF_ERROR_CHECK_RETURN(_sif_write_double64(
fp, vv) == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64DOUBLE64(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_write_double64(fp, vv) == 0, SIF_
ERROR_WRITE, 0)
#define FWRITE64NEC(buf, els, nel, fp) (_sif_fwrite_win(fp, buf, els * nel) / els)
#define FREAD64(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN(_sif_fread_win(fp, buf, els * nel) =
= 0, SIF_ERROR_READ, 0)
#define FREAD64V(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN_V(_sif_fread_win(fp, buf, els * nel
) == 0, SIF_ERROR_READ)
#define FREAD64CNT(buf, cnt, fp) cnt += sizeof(buf); SIF_ERROR_CHECK_RETURN(_sif_fread_win(fp,
 &(buf), sizeof(buf)) == 0, SIF_ERROR_READ, 0)
#define FREAD64INT32CNT(vv, cnt, fp) cnt += 4; SIF_ERROR_CHECK_RETURN(_sif_read_int32(fp, &(vv
)) == 0, SIF_ERROR_READ, 0)
#define FREAD64INT32(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_read_int32(fp, &(vv)) == 0, SIF_ERROR
_READ, 0)
#define FREAD64INT32NEC(vv, fp) _sif_read_int32(fp, &(vv))
/** NEC -- No Error Checking. */
#define FREAD64NEC(buf, els, nel, fp) (_sif_fread_win(fp, buf, els * nel) / els)
#else
#define FILE_IS_OKAY(fp) (fp)
#define FCLOSE64(fp) fclose(fp);
#define REWIND64(fp) SIF_ERROR_CHECK_RETURN(fseek(fp, ((off64_t)0), SEEK_SET) != 0,  SIF_ERROR
_SEEK, 0)
#define REWIND64V(fp) SIF_ERROR_CHECK_RETURN_V(fseek(fp, ((off64_t)0), SEEK_SET) != 0, SIF_ERR
OR_SEEK)
#define REWIND64NEC(fp) fseek(fp, ((off64_t)0), SEEK_SET)
#define FSEEK64(fp, loc, typ) SIF_ERROR_CHECK_RETURN(fseek(fp, ((off64_t)loc), (typ)) != 0, SI
F_ERROR_SEEK, 0)
#define FSEEK64V(fp, loc, typ) SIF_ERROR_CHECK_RETURN_V(fseek(fp, ((off64_t)loc), (typ)) != 0,
 SIF_ERROR_SEEK)
#define FSEEK64NEC(fp, loc, typ) fseek(fp, ((off64_t)loc), (typ))
```

```
#define FWRITE64(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN(fwrite(buf, els, nel, fp) != nel, S
IF_ERROR_WRITE, 0)
#define FWRITE64V(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN_V(fwrite(buf, els, nel, fp) != nel
, SIF_ERROR_WRITE)
#define FWRITE64CNT(buf, cnt, fp) cnt += sizeof(buf); SIF_ERROR_CHECK_RETURN(fwrite(&(buf), si
zeof(buf), 1, fp) != 1, SIF_ERROR_WRITE, 0)
#define FWRITE64INT32(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_write_int32(fp, vv) == 0, SIF_ERROR_
WRITE, 0)
#define FWRITE64INT32CNT(vv, cnt, fp) cnt += 4; SIF_ERROR_CHECK_RETURN(_sif_write_int32(fp, vv
) == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64DOUBLE64(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_write_double64(fp, vv) == 0, SIF_
ERROR_WRITE, 0)
#define FWRITE64DOUBLE64CNT(vv, cnt, fp) cnt += 8; SIF_ERROR_CHECK_RETURN(_sif_write_double64(
fp, vv) == 0, SIF_ERROR_WRITE, 0)
#define FWRITE64NEC(buf, els, nel, fp) fwrite(buf, els, nel, fp)
#define FREAD64(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN(fread(buf, els, nel, fp) != nel, SIF
_ERROR_READ, 0)
#define FREAD64V(buf, els, nel, fp) SIF_ERROR_CHECK_RETURN_V(fread(buf, els, nel, fp) != nel,
SIF_ERROR_READ)
#define FREAD64CNT(buf, cnt, fp) cnt += sizeof(buf); SIF_ERROR_CHECK_RETURN(fread(&(buf), size
of(buf), 1, fp) != 1, SIF_ERROR_READ, 0)
#define FREAD64INT32CNT(vv, cnt, fp) cnt += 4; SIF_ERROR_CHECK_RETURN(_sif_read_int32(fp, &(vv
)) == 0, SIF_ERROR_READ, 0)
#define FREAD64INT32(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_read_int32(fp, &(vv)) == 0, SIF_ERROR
_READ, 0)
#define FREAD64DOUBLE64CNT(vv, cnt, fp) cnt += 8; SIF_ERROR_CHECK_RETURN(_sif_read_double64(fp
, &(vv)) == 0, SIF_ERROR_READ, 0)
#define FREAD64DOUBLE64(vv, fp) SIF_ERROR_CHECK_RETURN(_sif_read_double64(fp, &(vv)) == 0, SIF
_ERROR_READ, 0)
#define FREAD64INT32NEC(vv, fp) _sif_read_int32(fp, &(vv))
/** NEC -- No Error Checking. */
#define FREAD64NEC(buf, els, nel, fp) fread(buf, els, nel, fp)
#endif

/**
 * Computes the minimum of two primitive values.
 */

#ifndef MAX
#define MAX(x, y) ((x > y) ? (x) : (y))
#endif

/**
 * Computes the minimum of two primitive values.
 */

#ifndef MIN
#define MIN(x, y) ((x < y) ? (x) : (y))
#endif

/**
 * Divides two integers (int or long) and takes the ceiling of the result as
 * if they were divided as real numbers.
 */

#ifndef CEIL_DIV
#define CEIL_DIV(x, y) ((((double)x)/(double)y) == ((double)((x)/(y))) ? ((x)/(y)) : ((x)/(y)
+ 1))
#endif

#define SIF_VERSION 2

/**
```

```
 * Returns the latest version of the SIF file format that the
 * currently loaded SIF library can process.
 *
 * @return The version number.
 */

long sif_get_version() {
  return SIF_VERSION;
}

/** Function prototype. The function definition has more specific
documentation.*/

static int _sif_is_uniform(sif_file *file, const void *data, int extentX, int extentY);
#ifdef WIN32

/**
 * A POSIX-like wrapper function around the Windows API for closing
 * files. The FCLOSE64 macro uses this function instead of fclose when
 * Visual Studio is used as the compiler. This ensures consistency,
 * i.e., FCLOSE64 always evaluates to an integer.
 *
 * @param hFile The windows handle of the file to close.
 *
 * @return Always returns 0.
 */

static int _sif_close_win(HANDLE hFile) {
  /** Flushing should be unnecessary, but I’ve noticed strange behavior
      for files accessed over the network without it. */
  FlushFileBuffers(hFile);

  /** Finally, close the handle. */
  CloseHandle(hFile);
  return 0;
}

/**
 * A POSIX-like wrapper function for seeking using the Windows API for
 * seeking within potentially large (i.e. 64-bit) byte-addressable
 * files.
 *
 * @param hf         The file to seek.
 * @param distance   The distance from the starting point to seek.
 * @param MoveMethod Defines how the starting point is
 *                   calculated, e.g. FILE_BEGIN means the starting
 *                   point is the begining of the file. Pass
 *                   a POSIX-whence field to
 *                   _sif_linux_to_win32_seek_type, and it will
 *                   return the POSIX equivalent.
 *
 * @return 0 if successful, -1 if unsuccessful.
 * @see _sif_linux_to_win32_seek_type
 */

static long long _sif_fseek_win(HANDLE hf, long long distance, DWORD MoveMethod) {
         unsigned long dwMoveHigh, nMoveLow;
   LARGE_INTEGER li;

   int rc = 0;

   /** Calculate the windows high and low seek offsets. */
   li.QuadPart = distance;
   nMoveLow = li.LowPart;
   dwMoveHigh = li.HighPart;
   /**rc = SetFilePointerEx(hf, m, &p, MoveMethod);**/

   /** Set the error status to no error.. */
   SetLastError(0);

   /** Set the location in the file accordingly. */
   rc = SetFilePointer(hf, (LONG)nMoveLow, (PLONG)&dwMoveHigh, MoveMethod);

   /** See if windows gets an error... */
   if (GetLastError() != NO_ERROR) {
     /** ... if so, return a negative status code. */
     return -1;
   }

   /** if not, return zero. */
   return 0;
}

/**
 * A POSIX-like wrapper to the Windows API function for writing to
 * large files. Writing starts at the current position in the
 * file.
 *
 * @param hFile     The file to write.
 * @param buffer    The buffer containing the bytes to be written to
 *                  the file.
 * @param num_bytes The number of bytes to write to the file.
 *
 * @return The number of bytes written.
 */

static DWORD _sif_fwrite_win(HANDLE hFile, void *buffer, DWORD num_bytes) {
   DWORD bytes_written;
   int rc;
   rc = WriteFile(hFile, buffer, num_bytes, &bytes_written, NULL);
   /** If an error occurred, return 0, giving the caller the
       indication that no bytes were written.*/
   if (rc == 0) {
      return 0;
   }
   /** Otherwise, return the number of bytes written by the Windows
       API. */
   return bytes_written;
}

/**
 * A wrapper to the Windows API function for reading from large
 * files. Read starts at the current position in the file.
 *
 * @param hFile     The file to read.
 * @param buffer    The buffer to store the read bytes.
 * @param num_bytes The number of bytes to read from the file.
 *
 * @return The number of bytes read.
 */

static DWORD _sif_fread_win(HANDLE hFile, void *buffer, DWORD num_bytes) {
   DWORD bytes_read;
   int rc;
   rc = ReadFile(hFile, buffer, num_bytes, &bytes_read, NULL);
   if (rc == 0) {
      return 0;
   }
   return bytes_read;
}

/**
 * A POSIX-like wrapper to the Windows API function for reading from
 * large files. Read starts at the current position in the
 * file.
 *
 * @param hFile     The file to read.
 * @param size      The size of each member.
 * @param nmemb     The number of members to write.
 * @param stream    The buffer to store the bytes read.
 *
 * @return The number of bytes read. If an error occurs or the end of
 * file is reached, this value may not necessarily equal size * nmemb.
 */

static DWORD _sif_fread(HANDLE ptr, size_t size, size_t nmemb, void *stream) {
        return (_sif_fread_win(ptr, stream, size * nmemb) == size * nmemb) ? nmemb : 0;
}

/**
 * Translates a POSIX fseek/fseek64 code into a WIN32 API seek code. More
 * specifically, SEEK_SET returns FILE_BEGIN, SEEK_CUR returns FILE_CURRENT,
 * and SEEK_END returns FILE_END.<p>
 *
 * This function is useful because it allows SIF I/O library functions
 * to use the macros FXXX macros and pass the POSIX seek-type
 * equivalents. When Visual Studio is used as the compiler, the macro
 * converts the POSIX equivalents to windows equivalents using this function.
 *
 * @param whence The POSIX fseek code.
 *
 * @return The Windows API equivalent code.
 */

static DWORD _sif_linux_to_win32_seek_type(int whence) {
  DWORD retval = FILE_BEGIN;
  switch (whence) {
  case SEEK_SET:
    retval = FILE_BEGIN;
    break;
  case SEEK_CUR:
    retval = FILE_CURRENT;
    break;
  case SEEK_END:
    retval = FILE_END;
    break;
  }
  return retval;
}

#else

/** If Visual Studio is not the compiler, wrapping fread is easy. We don’t
 even need a function. */
#define _sif_fread(ptr, size, nmemb, stream) fread((FILE*)ptr, size, nmemb, stream)
#endif

/**
 * To ensure portability across platforms, the byte ordering must be
 * consistent for all 32-bit integers written to a file. This function
 * takes in a non-portable long and writes the 32 most-significant
 * bits to the buffer in big-endian network byte order.
 *
 * @param val   The value to write to the buffer.
 * @param ptr   A buffer containing at least 4 bytes (32-bits). This
 *              function guarantees that only the first 4 bytes will
 *              be written.
 */

static void _sif_int32_to_packed_bytes(long val, u_char *ptr) {
  ptr[0] = (val >> 24) & 0xFF;
  ptr[1] = (val >> 16) & 0xFF;
  ptr[2] = (val >> 8) & 0xFF;
  ptr[3] = val & 0xFF;
}

/**
 * To ensure portability across platforms, the byte ordering must be
 * consistent for all 64-bit integers written to a file. This function
 * takes in a non-portable long and writes the 64 most-significant
 * bits to the buffer in big-endian network byte order.
 *
 * @param val   The value to write to the buffer.
 * @param ptr   A buffer containing at least 8 bytes (64-bits). This
 *              function guarantees that only the first 8 bytes will
 *              be written.
 */

static void _sif_int64_to_packed_bytes(long long val, u_char *ptr) {
  ptr[0] = (val >> 56) & 0xFF;
  ptr[1] = (val >> 48) & 0xFF;
  ptr[2] = (val >> 40) & 0xFF;
  ptr[2] = (val >> 32) & 0xFF;
  ptr[4] = (val >> 24) & 0xFF;
  ptr[5] = (val >> 16) & 0xFF;
  ptr[6] = (val >> 8) & 0xFF;
  ptr[7] = val & 0xFF;
}

/**
 * This function takes in a buffer containing at least 4 bytes and
 * reads a 32-bit integer value from it. It assumes the integer value
 * is stored in big-endian network byte order.
 *
 * @param ptr   A buffer containing at least 4 bytes (32-bits) from
 *              which to read the 32-bit integer value stored as a
 *              big-endian.
 *
 * @return The long value.
 */

static long _sif_packed_bytes_to_int32(const u_char* ptr) {
 // MSB first
 return ((long)ptr[0] << 24) | ((long)ptr[1] << 16)
   | ((long)ptr[2] << 8) | (long)ptr[3];
}


/**
 * This function takes in a buffer containing at least 4 bytes and
 * reads a 64-bit integer value from it. It assumes the integer value
 * is stored in big-endian network byte order.
 *
 * @param ptr   A buffer containing at least 8 bytes (64-bits) from
 *              which to read the 64-bit integer value stored as a
 *              big-endian.
 *
 * @return The long value.
 */

static long long sif_packed_bytes_to_int64(const u_char* ptr) {
 // MSB first
 return ((long long)ptr[0] << 56) | ((long long)ptr[1] << 48)
   | ((long long)ptr[2] << 40) | ((long long)ptr[3] << 32)
   | ((long long)ptr[4] << 24) | ((long long)ptr[5] << 16)
   | ((long long)ptr[6] << 8) | (long long)ptr[7];
}

static double _sif_swap_double64(const double val) {
  double retval;
  unsigned char *out = (unsigned char *)&retval;
  const unsigned char *in = (unsigned char *)&val;
  for (int i = 0; i < 8; i++) {
    out[i] = in[7-i];
  }
  return retval;
}

static double _sif_swap_float32(const float val) {
  float retval;
  unsigned char *out = (unsigned char *)&retval;
  const unsigned char *in = (unsigned char *)&val;
  for (int i = 0; i < 4; i++) {
    out[i] = in[4-i];
  }
  return retval;
}

static double _sif_hton_double64(const double val) {
  double retval;
  /** If we're on a windows machine, assume 'val' is little endian (even
      Windows on DEC-Alpha requires little-endian.) Otherwise, assume
      big-endian.*/
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  retval = _sif_swap_double64(val);
#elif __BYTE_ORDER == __BIG_ENDIAN
  retval = val;
#endif
  return retval;
}

static double  _sif_ntoh_double64(const double val) {
  double retval;
#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
  retval = _sif_swap_double64(val);
#elif __BYTE_ORDER == __BIG_ENDIAN
  retval = val;
#endif
  return retval;
}

static int _sif_write_double64(sif_file *file, const double val) {
  double net = _sif_hton_double64(val);
  if (FWRITE64NEC(&net, sizeof(u_char), 8, file->fp) != 8) {
    return 0;
  }
  return 1;
}

static int _sif_read_double64(sif_file *file, double *val) {
  double *net = (double*)file->ubuf;
  if (FREAD64NEC(file->ubuf, sizeof(u_char), 8, file->fp) != 8) {
    return 0;
  }
  *val = _sif_ntoh_double64(*net);
  return 1;
}

/**
 * Write an integer to a file in big endian network byte order. Only
 * the first 32 significant bits of an integer are considered.
 *
 * @param file   The file on which to write an integer.
 * @param long   The integer to write.
 *
 * @return 1 if successful, 0 if an error occurred.
 */

static int _sif_write_int32(sif_file *file, long val) {
  _sif_int32_to_packed_bytes(val, (u_char*)&(file->ubuf));
  if (FWRITE64NEC(&(file->ubuf), sizeof(u_char), 4, file->fp) != 4) {
    return 0;
  }
  return 1;
}


/**
 * Read an integer from a file in big endian network byte order. Four
 * bytes are read, and the integer is assumed to be stored as a big
 * endian.
 *
 * @param file   The file on which to write an integer.
 * @param long   The integer to write.
 *
 * @return 1 if successful, 0 if an error occurred.
 */

static int _sif_read_int32(sif_file *file, long *val) {
  if (FREAD64NEC(&(file->ubuf), sizeof(u_char), 4, file->fp) != 4) {
    return 0;
  }
  *val = _sif_packed_bytes_to_int32((const u_char *)&(file->ubuf));
  return 1;
}

/**
 * A shallow check for complete uniformity. Each flag in the tiles
 * header is examined however the raster is not scanned for
 * uniformity.
 *
 * @param tile_no The tile to check for complete uniformity.
 *
 * @return Returns true iff all bands in a tile are uniform.
 */

static int _sif_completely_uniform_shallow(sif_file *file, long i) {
  sif_tile *tile = file->tiles + i;
  sif_header *hd = file->header;
  int j, retval = 0xFF;
  u_char *last_flag_byte = tile->uniform_flags + (hd->n_uniform_flags - 1);
  *last_flag_byte = (u_char)(0xFF >> (8 - (hd->bands % 8))) | *last_flag_byte;
  for (j = 0; j < hd->n_uniform_flags && retval == 0xFF; j++) {
    retval = retval & tile->uniform_flags[j];
  }
  return retval == 0xFF;
}

/**
 * Returns true if a band in a tile is uniform.
 */

static int _sif_band_of_tile_is_uniform_shallow(sif_file *file, long i, long b) {
  sif_tile *tile = file->tiles + i;
  return SIF_GET_BIT(tile->uniform_flags, b);
}

/**
 * Computes the starting offset for a specific data block in the file. This
 * offset is computed by multiplying the block size in bytes by the block
 * index passed and adding the location where the block data starts.
 *
 * @param file      A pointer to a sif file of interest.
 * @param block_num The index of the block in the file passed.
 *
 * @return          The offset where the data starts.
 */

static LONGLONG          _sif_get_block_location(const sif_file *file, long block_num) {
  assert(block_num >= 0LL);
  return file->base_location
    + ((LONGLONG)file->header->tile_bytes * (LONGLONG)block_num);
}

/**
 * Allocates enough space for the meta-data table.
 *
 * @return The newly allocated meta-data table.
 */

static sif_meta_data   **_sif_alloc_meta_data_table() {
  sif_meta_data **result = (sif_meta_data**)malloc(sizeof(sif_meta_data*) * SIF_HASH_TABLE_SIZ
E);
  if (result != 0) {
    bzero(result, sizeof(sif_meta_data*) * SIF_HASH_TABLE_SIZE);
  }
  return result;
}

/**
 * Frees the space for the meta-data table and the items inside of it.
 */

static void            _sif_free_meta_data(sif_file *file) {
  int j;
  sif_meta_data *cur, *next = 0;
  for (j = 0; j < SIF_HASH_TABLE_SIZE; j++) {
    for (cur = file->meta_data[j]; cur != 0; cur = next) {
      next = cur->next;
      free(cur->value);
      free(cur->key);
    }
  }
  free(file->meta_data);
}

/**
 * Allocates enough space for a header struct. Sets all header values to their
 * defaults (usually zero).
 *
 * @return          A pointer to a newly allocated header struct.
 *                  Returns zero if an error occurred during allocation.
 */

static sif_header       *_sif_alloc_header() {
  sif_header *retval = 0;
  retval = (sif_header*)malloc(sizeof(sif_header));
  if (retval != 0) {
    bzero(retval, sizeof(sif_header));
  }
  return retval;
}

/**
 * Allocates enough space for a SIF file pointer struct. Sets all values to
 * their defaults (usually zero).
 *
 * @return          A pointer to a newly allocated file pointer struct.
 *                  Returns zero if an error occurred during allocation.
 */

static sif_file         *_sif_alloc_fp() {
  sif_file *retval = 0;
  retval = (sif_file*)malloc(sizeof(sif_file));
  if (retval != 0) {
    bzero(retval, sizeof(sif_file));
  }
  return retval;
}

/**
 * Allocates enough space for all the SIF tile header structures.
 *
 * @param           n_tiles The number of tiles to allocate.
 *
 * @return          A pointer to a newly allocated file pointer struct.
 *                  Returns zero if an error occurred during allocation.
 */

static sif_tile         *_sif_alloc_tile_headers(sif_file *file) {
  sif_tile *retval = 0, *tile;
  sif_header *hd = file->header;
  int i, s = SIF_SIZE_FLAG_ARRAY(hd->bands);

  /** Allocate enough space (in bytes) to hold the "band" number of
      flags for all tiles. */
  u_char *master_uf = (u_char*)malloc(hd->n_tiles * s);

  /** Allocate enough space to hold the uniform pixel values for each
      tile and for each band.*/
  u_char *master_upv = (u_char*)malloc(hd->n_tiles * hd->bands * hd->data_unit_size);
  retval = (sif_tile*)malloc(sizeof(sif_tile) * hd->n_tiles);

  /** If there was an error allocating any block, free all allocated blocks
```

```
      and exit. */
  if (retval == 0 || master_uf == 0 || master_upv == 0) {
    free(master_uf);
    free(master_upv);
    free(retval);
    return 0;
  }

  /** Set everything to be initially uniform. */
  bzero(master_upv, hd->n_tiles * hd->bands * hd->data_unit_size);
  memset(master_uf, 0xFF, s * hd->n_tiles);
  if (retval != 0) {
    for (i = 0; i < hd->n_tiles; i++) {
      tile = retval + i;
      /** The flags for ALL the tiles are stored in one big array. Do
       some point arithmetic so the tile’s header points to its
       flags.*/
      tile->uniform_flags = master_uf + (i * s);

      /** Do the same kind of thing for the uniform pixel values. */
      tile->uniform_pixel_values = master_upv + (i * hd->bands * hd->data_unit_size);

      /** Initially, no raster block is allocated for the tiles. */
      tile->block_num = -1;
    }
  }

  /** We need enough space to hold uniform pixel values,
      the uniformity flags, and the block number (32-bits).*/
  hd->tile_header_bytes = hd->bands * hd->data_unit_size + s + 4;

  /**
   * The number of bytes to store the uniformity flags. Thus,
   * s = Ceil(number_of_flags / 8).
   */
  hd->n_uniform_flags = s;
  return retval;
}

/**
 * Free the tile headers for a file.
 *
 * @param file  The file to free the headers.
 */

static void            _sif_free_tile_headers(sif_file *file) {
  /** Since the fields uniform flags and uniform_pixel_values
      are both located in a continguous blocks across all
      tile headers, we can free the fields of the first tile,
      thereby free all the data for the other tiles. */
  free(file->tiles->uniform_flags);
  free(file->tiles->uniform_pixel_values);
  free(file->tiles);
  file->tiles = 0;
}

/**
 * Writes the header for the file passed.
 *
 * @param           file The file pointer corresponding to the file
 *                       to write the header.
 */
```

```
static int              _sif_write_header(sif_file *file) {
  sif_header *hd = file->header;
  int cnt = 0, dummy = 0, i;
  REWIND64(file->fp);
  FWRITE64INT32CNT(dummy, cnt, file);
  FWRITE64CNT(hd->magic_number, cnt, file->fp);
  FWRITE64INT32CNT(file->use_file_version, cnt, file);
  hd->version = file->use_file_version;
  FWRITE64INT32CNT(hd->width, cnt, file);
  FWRITE64INT32CNT(hd->height, cnt, file);
  FWRITE64INT32CNT(hd->bands, cnt, file);
  FWRITE64INT32CNT(hd->n_keys, cnt, file);
  FWRITE64INT32CNT(hd->n_tiles, cnt, file);
  FWRITE64INT32CNT(hd->tile_width, cnt, file);
  FWRITE64INT32CNT(hd->tile_height, cnt, file);
  FWRITE64INT32CNT(hd->tile_bytes, cnt, file);
  FWRITE64INT32CNT(hd->n_tiles_across, cnt, file);
  FWRITE64INT32CNT(hd->data_unit_size, cnt, file);
  FWRITE64INT32CNT(hd->user_data_type, cnt, file);
  FWRITE64INT32CNT(hd->defragment, cnt, file);
  FWRITE64INT32CNT(hd->consolidate, cnt, file);
  FWRITE64INT32CNT(hd->intrinsic_write, cnt, file);
  FWRITE64INT32CNT(hd->tile_header_bytes, cnt, file);
  FWRITE64INT32CNT(hd->n_uniform_flags, cnt, file);
  /** For SIF file versions 2 and higher, floats/doubles are big-endian.
      SIF file version 1 has an anomaly where the float is written as
      little-endian while the ints are written as big-endian. */

  /** Version 1 logic. **/
  if (hd->version < 2 || file->use_file_version < 2) {
    for (i = 0; i < 6; i++) {
      FWRITE64CNT(hd->affine_geo_transform[i], cnt, file->fp);
    }
  }
  /** Version 2 and higher logic. **/
  else {
    for (i = 0; i < 6; i++) {
      FWRITE64DOUBLE64CNT(hd->affine_geo_transform[i], cnt, file);
    }
  }
  REWIND64(file->fp);
  FWRITE64INT32(cnt, file);
  file->header_bytes = cnt;
  return 1;
}

/**
 * Reads the header from the file passed.
 *
 * @param           file The file pointer corresponding to the file
 *                       to write the header.
 */

static int              _sif_read_header(sif_file *file) {
  sif_header *hd = file->header;
  int cnt = 0, i;
  REWIND64(file->fp);
  FREAD64INT32CNT(file->header_bytes, cnt, file);
  FREAD64CNT(hd->magic_number, cnt, file->fp);
  FREAD64INT32CNT(hd->version, cnt, file);
  file->use_file_version = hd->version;
  FREAD64INT32CNT(hd->width, cnt, file);
  FREAD64INT32CNT(hd->height, cnt, file);
  FREAD64INT32CNT(hd->bands, cnt, file);
  FREAD64INT32CNT(hd->n_keys, cnt, file);
  FREAD64INT32CNT(hd->n_tiles, cnt, file);
  FREAD64INT32CNT(hd->tile_width, cnt, file);
  FREAD64INT32CNT(hd->tile_height, cnt, file);
  FREAD64INT32CNT(hd->tile_bytes, cnt, file);
  FREAD64INT32CNT(hd->n_tiles_across, cnt, file);
  FREAD64INT32CNT(hd->data_unit_size, cnt, file);
  FREAD64INT32CNT(hd->user_data_type, cnt, file);
  FREAD64INT32CNT(hd->defragment, cnt, file);
  FREAD64INT32CNT(hd->consolidate, cnt, file);
  FREAD64INT32CNT(hd->intrinsic_write, cnt, file);
  FREAD64INT32CNT(hd->tile_header_bytes, cnt, file);
  FREAD64INT32CNT(hd->n_uniform_flags, cnt, file);

  if (hd->version < 2) {
    for (i = 0; i < 6; i++) {
      FREAD64CNT(hd->affine_geo_transform[i], cnt, file->fp);
    }
  }
  else if (hd->version >= 2) {
    for (i = 0; i < 6; i++) {
      FREAD64DOUBLE64CNT(hd->affine_geo_transform[i], cnt, file);
    }
  }
  return 1;
}

/**
 * Writes tile headers for the file passed.
 *
 * @param           file The file pointer corresponding to the file
 *                       to write the header.
 */

static int             _sif_write_tile_headers(sif_file *file) {
  long i = 0;
  long long base = file->header_bytes;
  sif_tile *tile = 0;
  sif_header *hd = file->header;
  FSEEK64(file->fp, base, SEEK_SET);
  for (; i < hd->n_tiles; i++, base += hd->tile_header_bytes) {
    tile = file->tiles + i;
    FWRITE64(tile->uniform_pixel_values, hd->data_unit_size, hd->bands, file->fp);
    FWRITE64(tile->uniform_flags, 1, hd->n_uniform_flags, file->fp);
    FWRITE64INT32(tile->block_num, file);
  }
  return 0;
}

/**
 * Reads tile headers for the file passed.
 *
 * @param           file The file pointer corresponding to the file
 *                       to write the header.
 */

static int             _sif_read_tile_headers(sif_file *file) {
  long i = 0, j = 1;
  long long base = file->header_bytes;
  sif_header *hd = file->header;
  sif_tile *tile = 0;
  FSEEK64(file->fp, base, SEEK_SET);
  for (; i < hd->n_tiles; i++, base += hd->tile_header_bytes) {
    tile = file->tiles + i;
    FREAD64(tile->uniform_pixel_values, hd->data_unit_size, hd->bands, file->fp);
    FREAD64(tile->uniform_flags, 1, hd->n_uniform_flags, file->fp);
    FREAD64INT32(tile->block_num, file);
  }
  return j;
}

/**
 * Writes a specific tile header for the file passed.
 *
 * @param           file     The file pointer corresponding to the file
 *                           to write the header.
 * @param           tile     The tile to write.
 * @param           tile_num The number of the tile to write.
 */

static int               _sif_write_tile_header(sif_file *file, sif_tile *tile, long tile_num)
 {
  LONGLONG loc;
  sif_header *hd = file->header;
  assert(file);
  assert(tile_num >= 0L);
  assert(tile_num < file->header->n_tiles);
  /** In theory, our tile header block would never be longer than the size of a long long.*/
  loc = (LONGLONG)(file->header_bytes + tile_num * file->header->tile_header_bytes);
  /** Set the location.*/
  FSEEK64(file->fp, loc, SEEK_SET);
  /** Write to the file. */
  FWRITE64(tile->uniform_pixel_values, hd->data_unit_size, hd->bands, file->fp);
  FWRITE64(tile->uniform_flags, 1, hd->n_uniform_flags, file->fp);
  FWRITE64INT32(tile->block_num, file);
  return 1;
}

/**
 * Retrieves a meta-data pair by its key so that its value may be inspected
 * or modified. Returns 0 if the meta-data pair corresponding to the key
 * could not be found.
 *
 * @param file     The file on which to perform the operation.
 * @param key      The key of the meta-data pair to retrieve.
 *
 * @return A pointer to the meta-data pair, 0 if not found.
 */

sif_meta_data*   _sif_get_meta_data_pair(sif_file *file, const char *key) {
  sif_header *hd = 0;
  sif_meta_data *q = 0, *r = 0, *result = 0;
  unsigned int hash;
  SIF_CHECK_FILE(file);
  hash = _sif_hash(key) % SIF_HASH_TABLE_SIZE;
  hd = file->header;
  q = file->meta_data[hash];
  for (r = q; r != 0; r = r->next) {
    if (strcmp(key, r->key) == 0) {
      result = r;
      break;
    }
  }
  result = r;
  return result;
```

```
}

/**
 * Unlink a meta-data pair by its key. The function simply unlinks it from
 * the collision chain but does not delocate the pair or its key and value
 * fields. Returns the pair unlinked so it may be deallocated by the
 * caller.
 *
 * This function is guaranteed to modify only the next field of any
 * meta-data pair in the file. The next field of the meta-data pair unlinked
 * is set to NULL for safety.
 *
 * @param     file       The file on which to perform the operation.
 * @param     key        The key of the meta-data pair to remove.
 *
 * @return The unlinked meta-data pair.
 */
sif_meta_data           *_sif_unlink_meta_data_pair(sif_file *file, const char *key) {
  sif_meta_data *q = 0, *r = 0, *result = 0, *prev = 0;
  unsigned int hash;
  SIF_CHECK_FILE(file);
  hash = _sif_hash(key) % SIF_HASH_TABLE_SIZE;
  q = file->meta_data[hash];
  for (r = q; r != 0; r = r->next) {
    if (strcmp(key, r->key) == 0) {
      result = r;
      /** If the previous is NULL, then the pair we’re unlinking is certainly
          the first element. Thus, the new first element becomes the result’s
          next.*/
      if (prev == 0) {
        file->meta_data[hash] = result->next;
      }
      else {
        prev->next = result->next;
      }
      result->next = 0;
      break;
    }
    prev = r;
  }
  return r;
}

void           _sif_insert_meta_data_pair(sif_file *file, const char *key, sif_meta_data *to_i
nsert) {
  sif_meta_data *q = 0, *result = 0;
  unsigned int hash;
  SIF_CHECK_FILE_V(file);
  hash = _sif_hash(key) % SIF_HASH_TABLE_SIZE;
  to_insert->next = file->meta_data[hash];
  file->meta_data[hash] = to_insert;
  (file->header->n_keys)++;
}

const int        _sif_null_terminator_check(const char *v, int n) {
  int i;
  for (i = 0; i < n; i++) {
    if (v[0] == 0x00) { return 1; }
  }
  return 0;
}
```

```
/* See sif-io.h for detailed documentation of public functions. */
const void*      sif_get_meta_data_binary(sif_file *file, const char *key, int *n_bytes) {
  const void *retval = 0; /** By default, null is returned (not found).*/
  sif_meta_data *q = 0;
  SIF_CHECK_FILE(file);
  q = _sif_get_meta_data_pair(file, key);
  if (q == 0) {
    file->error = SIF_ERROR_META_DATA_KEY;
    *n_bytes = 0;
  }
  else {
    *n_bytes = q->value_length;
    retval = q->value;
  }
  return retval;
}


/* See sif-io.h for detailed documentation of public functions. */
const char       *sif_get_meta_data(sif_file *file, const char *key) {
  const char *retval = 0; /** By default, null is returned (not found).*/
  sif_meta_data *q = 0;
  SIF_CHECK_FILE(file);
  q = _sif_get_meta_data_pair(file, key);
  if (q == 0) {
    file->error = SIF_ERROR_META_DATA_KEY;
    retval = 0;
  }
  else {
    retval = (char*)q->value;
    if (!_sif_null_terminator_check(retval, q->value_length)) {
      file->error = SIF_ERROR_META_DATA_VALUE;
    }
  }
  return retval;
}

/**
 * Sets a meta-data field with a given key to a value. The
 * length of the value is specified, thereby allowing for
 * binary, non-null-terminated, meta-data.
 *
 * @param file      The file to set the meta-data.
 * @param key       The key of the field to set.
 * @param value     The value for which to set the field.
 * @param value_len The length of the value.
 */

static void             _sif_set_meta_data_len(sif_file *file, const char *key, const char *va
lue, int value_len) {
  int success = 0, key_len;       /** We need room for the null terminator. */
  sif_meta_data *i = 0;

  assert(file);
  i = _sif_get_meta_data_pair(file, key);
  if (i == 0) {
    key_len = strlen(key) + 1;
    SIF_ERROR_CHECK_RETURN_V((i = (sif_meta_data*)malloc(sizeof(sif_meta_data))) == 0, SIF_ERR
OR_MEM);
    SIF_ERROR_CHECK_RETURN_V((i->value = (char*)malloc(value_len * sizeof(char))) == 0, SIF_ER
ROR_MEM);
    SIF_ERROR_CHECK_RETURN_V((i->key = (char*)malloc(key_len * sizeof(char))) == 0, SIF_ERROR_
MEM);
```

```
    i->key_length = key_len;
    i->value_length = value_len;
    memcpy(i->key, key, sizeof(char) * key_len);
    memcpy(i->value, value, sizeof(char) * value_len);
    _sif_insert_meta_data_pair(file, key, i);
  }
  else {
    if (value_len != i->value_length) {
      SIF_ERROR_CHECK_RETURN_V((i->value = (char*)realloc(i->value, value_len * sizeof(char)))
 == 0, SIF_ERROR_MEM);
      i->value_length = value_len;
      memcpy(i->value, value, sizeof(char) * value_len);
    }
  }

}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_set_meta_data(sif_file *file, const char *key, const char *value) {
  SIF_CHECK_FILE_V(file);
  _sif_set_meta_data_len(file, key, value, strlen(value) + 1);
}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_set_meta_data_binary(sif_file *file, const char *key, const void *buffer,
 int n_bytes) {
  SIF_CHECK_FILE_V(file);
  _sif_set_meta_data_len(file, key, (const char*)buffer, n_bytes);
}

/**
 * Return the last used block index in the file.
 *
 * @param file The file to compute the last used block index.
 *
 * @return The last used block index. -1 is returned
 *         if no blocks are in use.
 */

static long         _sif_get_last_used_block_index(sif_file *file) {
  long i = 0;
  long last_known_used_block = -1;
  for (; i < file->header->n_tiles; i++) {
    if (file->blocks_to_tiles[i] != -1) {
      last_known_used_block = i;
    }
  }
  return last_known_used_block;
}

/**
 * Truncate the file at a particular position.
 *
 * @param file   The file to truncate.
 * @param pos    The position of truncation.
 */

static void             _sif_truncate(sif_file *file, LONGLONG pos) {
#ifdef WIN32
  FSEEK64V(file->fp, pos, SEEK_SET);
  SIF_ERROR_CHECK_RETURN_V(SetEndOfFile(file->fp) == 0, SIF_ERROR_TRUNCATE)
#else
  SIF_ERROR_CHECK_RETURN_V(ftruncate(fileno(file->fp), pos) != 0, SIF_ERROR_TRUNCATE);
```

```
#endif
}

/**
 * Read the meta-data from the disk, storing the contents in
 * the file structure for easy access.
 *
 * @param file   The file to read the meta data.
 */
static void             _sif_read_meta_data(sif_file *file) {
  sif_header *header = 0;
  LONGLONG loc = 0;
  unsigned int i = 0;
  long val = 0;
  int n_keys;

  /**  sif_meta_data *md = 0;**/
  sif_meta_data md;
  header = file->header;

  n_keys = header->n_keys;
  header->n_keys = 0;
  /** move the file pointer to one after the last used block. **/
  loc = _sif_get_block_location(file, _sif_get_last_used_block_index(file) + 1);
  FSEEK64V(file->fp, loc, SEEK_SET);

  /**
   * This code is usually executed during an open. As such, the file structure
   * has not been completely initialized. FREAD64NEC must be used instead of FREAD64
   * because we must free the memory before returning upon an error.
   */
  for (i = 0; i < n_keys; i++) {
    if (FREAD64INT32NEC(val, file) == 0) {
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_READ);
    }
    md.key_length = (unsigned long)val;
    if ((md.key = malloc(sizeof(char) * md.key_length)) == 0) {
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_MEM);
    }
    if (md.key_length != 0) {
      (md.key)[md.key_length - 1] = 0;
    }
    if (FREAD64NEC(md.key, 1, md.key_length, file->fp) != md.key_length) {
      free(md.key);
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_READ);
    }
    if (FREAD64INT32NEC(val, file) == 0) {
      free(md.key);
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_READ);
    }
    md.value_length = val;
    if ((md.value = malloc(sizeof(char) * md.value_length)) == 0) {
      free(md.key);
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_MEM);
    }
    /**    (md.value)[md.value_length] = 0;**/
    if (FREAD64NEC(md.value, 1, md.value_length, file->fp) == 0) {
      free(md.value);
      free(md.key);
      _sif_free_meta_data(file);
      SIF_ERROR_CHECK_RETURN_V(1, SIF_ERROR_READ);
    }
    sif_set_meta_data_binary(file, md.key, md.value, md.value_length);
    free(md.value);
    free(md.key);
    if (file->error) {
      return;
    }
  }
}

/**
 * Write the meta data from the file structure to the disk.
 *
 * @param file        The file containing the meta-data structure to write.
 */

static int             _sif_write_meta_data(sif_file *file) {
  sif_header *header = 0;
  LONGLONG loc = 0, eofpos = 0;
  sif_meta_data *i = 0;
  int j = 0;
  header = file->header;
  loc = _sif_get_block_location(file, _sif_get_last_used_block_index(file) + 1);
  eofpos = loc;
  FSEEK64(file->fp, loc, SEEK_SET);
  for (j = 0; j < SIF_HASH_TABLE_SIZE; j++) {
    for (i = file->meta_data[j]; i != 0; i = i->next) {
      FWRITE64INT32(i->key_length, file); eofpos += 4;
      FWRITE64(i->key, i->key_length, 1, file->fp); eofpos += i->key_length;
      FWRITE64INT32(i->value_length, file); eofpos += 4;
      FWRITE64(i->value, i->value_length, 1, file->fp); eofpos += i->value_length;
    }
  }
  eofpos = eofpos + 1;
  _sif_truncate(file, eofpos);
  return 1;
}

/* See sif-io.h for detailed documentation of public functions. */
void            sif_get_tile_slice(sif_file *file, void *buffer, long tx, long ty, long band)
 {
  sif_tile *tile = 0;
  sif_header *hd = 0;
  long i = 0, tile_num = 0;
  LONGLONG pos = 0;
  u_char *data = buffer, *upv;
  //  printf("get x: %d y: %d b: %d\n", tx, ty, band);
  SIF_CHECK_FILE_V(file);
  hd = file->header;
  if (tx < 0 || ty < 0 || tx >= hd->n_tiles_across) {
    file->error = SIF_ERROR_INVALID_TN;
    return;
  }
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  if (buffer == 0) {
    file->error = SIF_ERROR_INVALID_BUFFER;
    return;
  }
  tile_num = (hd->n_tiles_across * ty) + tx;
  tile = file->tiles + tile_num;
  bzero(buffer, hd->data_unit_size * file->units_per_slice);
  if (_sif_band_of_tile_is_uniform_shallow(file, tile_num, band)) {
          upv = tile->uniform_pixel_values + (hd->data_unit_size * band);
          if (hd->data_unit_size > 1) {
                for (i = 0, data = buffer; i < file->units_per_slice; i++, data += hd->data_un
it_size) {
                        memcpy(data, upv, hd->data_unit_size);
                }
          }
          else {
                memset(data, upv[0], file->units_per_slice);
          }
  }
  else {
    pos = _sif_get_block_location(file, tile->block_num) + (hd->data_unit_size * file->units_p
er_slice) * band;
    FSEEK64V(file->fp, pos, SEEK_SET);
    FREAD64V(buffer, hd->data_unit_size, file->units_per_slice, file->fp);
  }
  return;
}

/* See sif-io.h for detailed documentation of public functions. */
void            sif_fill_tile_slice(sif_file *file, long tx, long ty, long band, const void *v
alue) {
  sif_tile *tile;
  sif_header *hd = 0;
  long tile_num;
  SIF_CHECK_FILE_V(file);
  hd = file->header;
  if (tx < 0 || ty < 0 || tx >= hd->n_tiles_across) {
    file->error = SIF_ERROR_INVALID_TN;
    return;
  }
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  if (value == 0) {
    file->error = SIF_ERROR_INVALID_BUFFER;
    return;
  }
  /** Compute the tile number using the stride stored in the header. */
  tile_num = (hd->n_tiles_across * ty) + tx;
  tile = file->tiles + tile_num;
  //  printf("set x: %d y: %d b: %d\n", tx, ty, band);

  /** We should not be changing tiles for read-only files. Return an error. */
  if (file->read_only) {
    file->error = SIF_ERROR_INVALID_FILE_MODE;
    return;
  }

  memcpy(tile->uniform_pixel_values + (hd->data_unit_size * band), value, hd->data_unit_size);
  SIF_SET_BIT(tile->uniform_flags, band);
  if (_sif_completely_uniform_shallow(file, tile_num) && tile->block_num != -1) {
    file->blocks_to_tiles[file->tiles[tile_num].block_num] = -1;
    file->tiles[tile_num].block_num = -1;
  }
  _sif_write_tile_header(file, tile, tile_num);
  return;
}

/* See sif-io.h for detailed documentation of public functions. */
void            sif_fill_tiles(sif_file *file, long band, const void *value) {
  sif_tile *tile;
  sif_header *hd = 0;
  long tile_num;

  SIF_CHECK_FILE_V(file);
  hd = file->header;
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  if (value == 0) {
    file->error = SIF_ERROR_INVALID_BUFFER;
    return;
  }
  /** We should not be changing tiles for read-only files. Return an error. */
  if (file->read_only) {
     file->error = SIF_ERROR_INVALID_FILE_MODE;
     return;
  }

  for (tile_num = 0; tile_num < hd->n_tiles; tile_num++) {
     /** Compute the tile number using the stride stored in the header. */
     tile = file->tiles + tile_num;
     /**  printf("set x: %d y: %d b: %d\n", tx, ty, band);**/

     memcpy(tile->uniform_pixel_values + (hd->data_unit_size * band), value, hd->data_unit_siz
e);
     SIF_SET_BIT(tile->uniform_flags, band);
     if (_sif_completely_uniform_shallow(file, tile_num) && tile->block_num != -1) {
        file->blocks_to_tiles[file->tiles[tile_num].block_num] = -1;
        file->tiles[tile_num].block_num = -1;
     }
  }
  _sif_write_tile_headers(file);
}

/* See sif-io.h for detailed documentation of public functions. */
void            sif_set_tile_slice(sif_file *file, const void *buffer, long tx, long ty, long
band) {
  sif_tile *tile = 0;
  sif_header *hd = 0;
  long i = 0, free_b = 0, extentX = 0, extentY = 0;             ;
  LONGLONG loc, tile_num;
  SIF_CHECK_FILE_V(file);
  hd = file->header;

  if (tx < 0 || ty < 0 || tx >= hd->n_tiles_across) {
    file->error = SIF_ERROR_INVALID_TN;
    return;
  }
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  if (buffer == 0) {
    file->error = SIF_ERROR_INVALID_BUFFER;
    return;
  }
  extentX = hd->tile_width;
  extentY = hd->tile_height;
  //  printf("set x: %d y: %d b: %d\n", tx, ty, band);
  /** We should not be changing tiles for read-only files. Return an error. */
  if (file->read_only) {
    file->error = SIF_ERROR_INVALID_FILE_MODE;
    return;
  }

  extentX = MIN(hd->tile_width, hd->width - tx * hd->tile_width);
  extentY = MIN(hd->tile_height, hd->height - ty * hd->tile_height);

  /** Compute the tile number using the stride stored in the header. */
  tile_num = (hd->n_tiles_across * ty) + tx;
  tile = file->tiles + tile_num;
  /** If the flag intrinsic_write is set, that means we check for pixel
      uniformity on a write. */
  if (hd->intrinsic_write && _sif_is_uniform(file, buffer, extentX, extentY)) {
    memcpy(tile->uniform_pixel_values + (hd->data_unit_size * band), buffer, hd->data_unit_siz
e);
    SIF_SET_BIT(tile->uniform_flags, band);
    if (_sif_completely_uniform_shallow(file, tile_num) && tile->block_num != -1) {
      file->blocks_to_tiles[file->tiles[tile_num].block_num] = -1;
      file->tiles[tile_num].block_num = -1;
    }
    _sif_write_tile_header(file, tile, tile_num);
    return;
  }
  /** If we’ve gotten here then the tile is non-uniform or we’re presuming that
      it is. If each slice of the tile cube was uniform before, we need to find
      a free spot on disk to put the tile cube. */
  if (tile->block_num == -1) {
    /** Look for the first free tile block. */
    for (i = 0; i < hd->n_tiles; i++) {
      if (file->blocks_to_tiles[i] == -1) {
        free_b = i;
        break;
      }
    }
    tile->block_num = free_b;
    file->blocks_to_tiles[free_b] = tile_num;
    loc = _sif_get_block_location(file, tile->block_num);
    FSEEK64V(file->fp, loc, SEEK_SET);
    /** Write the buffer out n times where n is the number of bands. Raster
        data stored for uniform bands will be ignored.*/
    for (i = 0; i < hd->bands; i++) {
      FWRITE64V((u_char*)buffer, hd->data_unit_size, file->units_per_slice, file->fp);
    }

  }
  /** If we already checked for pixel uniformity, we don’t need to do
      it again. */
  if (hd->intrinsic_write == 0) {
    file->dirty_tiles[tile_num] = 1;
  }
  /** Compute the location for the non-uniform slice and go there. */
  loc = _sif_get_block_location(file, tile->block_num) + hd->data_unit_size * file->units_per_
slice * band;
  FSEEK64V(file->fp, loc, SEEK_SET);
  /** Write the non-uniform slice to disk. */
  FWRITE64V((u_char*)buffer, hd->data_unit_size, file->units_per_slice, file->fp);

  /** Set the uniformity flag for this band to false. */
  SIF_CLEAR_BIT(tile->uniform_flags, band);

  /** Write the tile header out to disk. */
  _sif_write_tile_header(file, tile, tile_num);
}

/* See sif-io.h for detailed documentation of public functions. */
static void             _sif_swap_blocks(sif_file *file, long a, long b, void *da, void *db, i
nt assign) {
  LONGLONG pos_a = 0, pos_b = 0, tb = file->header->tile_bytes;
  if (a == b) {
    return;
  }
  pos_a = _sif_get_block_location(file, a);
  pos_b = _sif_get_block_location(file, b);
  if (a != -1 && !assign) {
    FSEEK64V(file->fp, pos_a, SEEK_SET);
    FREAD64V((unsigned char*)da, 1, tb, file->fp);
  }
  if (b != -1) {
    FSEEK64V(file->fp, pos_b, SEEK_SET);
    FREAD64V((unsigned char*)db, 1, tb, file->fp);
  }
  if (a != -1 && !assign) {
    FSEEK64V(file->fp, pos_b, SEEK_SET);
    FWRITE64V((unsigned char*)da, 1, tb, file->fp);
  }
  if (b != -1) {
    FSEEK64V(file->fp, pos_a, SEEK_SET);
    FWRITE64V((unsigned char*)db, 1, tb, file->fp);
  }
}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_set_raster(sif_file* file, const void *data,
                                long x, long y, long w, long h, long band) {
  const unsigned char *datav = data; /** makes VC++ happy. can't do
                                         pointer arithmetic on void*'s! */
  long tnx1, tny1, tnx2, tny2; /** the starting and ending tile indices. */
  long sxt, syt, ext, eyt;     /** the starting and ending coordinates on the tile raster. */
  long sxd, syd;               /** the starting coordinates on the data raster. */
  long cyd, cyt;               /** the current ordinates for the data and tile rasters. */
  long tx, ty;                 /** the current working tile indices. */
  long tw, th, trs, dus, wdus; /** the tile width, height, data unit size, scan line byte size
. */
  sif_header *hd;                  /** header */
  unsigned char *buffer;                     /** buffer */
  SIF_CHECK_FILE_V(file);
  if (file->read_only) {
    return;
  }
  hd = file->header;
  if (x < 0 || y < 0) {
    file->error = SIF_ERROR_INVALID_COORD;
    return;
  }
  if (w < 1 || h < 1 || x + w > hd->width || y + h > hd->height) {
    file->error = SIF_ERROR_INVALID_REGION_SIZE;
    return;
  }
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  buffer = file->buffer[0];
  tw = hd->tile_width;
  th = hd->tile_height;
  trs = hd->n_tiles_across;
  dus = hd->data_unit_size;/** size of a single pixel in bytes. */
  wdus = dus * w;          /** width of a single scan line. */
  tnx1 = x / tw;           /** the starting tile horizontal index. */
  tny1 = y / th;           /** the starting tile vertical index. */
  tnx2 = (x + w - 1) / tw; /** the end tile horizontal index. */
  tny2 = (y + h - 1) / th; /** the end tile vertical index. */
  for (ty = tny1; ty <= tny2; ty++) {
    for (tx = tnx1; tx <= tnx2; tx++) {
      /** grab the tile. */
      sif_get_tile_slice(file, buffer, tx, ty, band);
      sxt = MAX(0, x - tx * tw);                  /** starting x pixel on tile raster. */
      syt = MAX(0, y - ty * th);                  /** starting y pixel on tile raster. */
      ext = MIN(tw - 1, x + w - 1 - (tx * tw));   /** ending x pixel on tile raster. */
      eyt = MIN(th - 1, y + h - 1 - (ty * th));   /** ending y pixel on tile raster. */
      sxd = (tx * tw + sxt) - x;                  /** starting x pixel on data raster. */
      syd = (ty * th + syt) - y;                  /** starting y pixel on data raster. */

      /** copy the window from the raster to the tile.*/
      for (cyd = syd, cyt = syt; cyt <= eyt; cyd++, cyt++) {
        memcpy(buffer + (cyt * trs) + (sxt * dus), datav + (cyd * wdus) + (sxd * dus), (ext -
sxt + 1) * dus);
      }
      /** put the tile back with modifications. */
      sif_set_tile_slice(file, buffer, tx, ty, band);
      if (file->error) {
        return;
      }
    }
  }
}

/**
 * Retrieves an entire tile (all bands).
 *
 * @param file    The file where the tile is stored.
 * @param tile_no The index of the tile.
 * @param data    The buffer to store the result of the read.
 */

void              _sif_get_tile(sif_file *file, LONGLONG tile_no, unsigned char *data) {
  sif_tile *tile = file->tiles + tile_no;
  sif_header *hd = file->header;
  u_char *buffer = 0;
  u_char *upv = 0;
  long i = 0, j = 0;
  LONGLONG pos = 0;
  bzero(data, hd->tile_bytes);
  buffer = data;
  for (i = 0; i < hd->bands; i++) {
    buffer = ((u_char*)data) + (file->units_per_slice * hd->data_unit_size * i);
        if (_sif_completely_uniform_shallow(file, tile_no)) {
                upv = tile->uniform_pixel_values + i * hd->data_unit_size;
                if (hd->data_unit_size == 1) {
                        for (j = 0; j < file->units_per_slice; j++, buffer += hd->data_unit_si
ze) {
                                memcpy(buffer, upv, hd->data_unit_size);
                        }
                }
                else {
                        memset(buffer, upv[0], file->units_per_slice);
                }
        }
        else {
                pos = _sif_get_block_location(file, tile->block_num) + file->units_per_slice *
 hd->data_unit_size * i;
                FSEEK64V(file->fp, pos, SEEK_SET);
                FREAD64V(buffer, 1, hd->tile_bytes, file->fp);
        }
  }
  return;
}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_get_raster(sif_file* file, void *data,
                                long x, long y, long w, long h, long band) {
  unsigned char *datav = data;
  long tnx1, tny1, tnx2, tny2; /** the starting and ending tile indices. */
  long sxt, syt, ext, eyt;     /** the starting and ending coordinates on the tile raster. */
  long sxd, syd;               /** the starting coordinates on the data raster. */
  long cyd, cyt;               /** the current ordinates for the data and tile rasters. */
  long tx, ty;                 /** the current working tile indices. */
  long tw, th, trs, dus, wdus; /** the tile width, height, data unit size, scan line byte size
. */
  sif_header *hd;                  /** header */
  unsigned char *buffer;                    /** buffer */
  SIF_CHECK_FILE_V(file);
  hd = file->header;
  if (x < 0 || y < 0) {
    file->error = SIF_ERROR_INVALID_COORD;
    return;
  }
  if (w < 1 || h < 1 || x + w > hd->width || y + h > hd->height) {
    file->error = SIF_ERROR_INVALID_REGION_SIZE;
    return;
  }
  if (band < 0 || band >= hd->bands) {
    file->error = SIF_ERROR_INVALID_BAND;
    return;
  }
  buffer = file->buffer[0];
  tw = hd->tile_width;
  th = hd->tile_height;
  trs = hd->n_tiles_across;
  dus = hd->data_unit_size;/** size of a single pixel in bytes. */
  wdus = dus * w;          /** width of a single scan line. */
  tnx1 = x / tw;           /** the starting tile horizontal index. */
  tny1 = y / th;           /** the starting tile vertical index. */
  tnx2 = (x + w - 1) / tw; /** the end tile horizontal index. */
  tny2 = (y + h - 1) / th; /** the end tile vertical index. */
  for (ty = tny1; ty <= tny2; ty++) {
    for (tx = tnx1; tx <= tnx2; tx++) {

      sxt = MAX(0, x - tx * tw);                  /** starting x pixel on tile raster. */
      syt = MAX(0, y - ty * th);                  /** starting y pixel on tile raster. */
      ext = MIN(tw - 1, x + w - 1 - (tx * tw));   /** ending x pixel on tile raster. */
      eyt = MIN(th - 1, y + h - 1 - (ty * th));   /** ending y pixel on tile raster. */
      sxd = (tx * tw + sxt) - x;                  /** starting x pixel on data raster. */
      syd = (ty * th + syt) - y;                  /** starting y pixel on data raster. */

      /** grab the tile. */
      sif_get_tile_slice(file, buffer, tx, ty, band);
      /** copy the window from the tile to the input raster.*/
      for (cyd = syd, cyt = syt; cyt <= eyt; cyd++, cyt++) {
        memcpy(datav + (cyd * wdus) + (sxd * dus), buffer + (cyt * trs) + (sxt * dus), (ext -
sxt + 1) * dus);
      }
    }
  }
}

/* See sif-io.h for detailed documentation of public functions. */
int              sif_is_shallow_uniform(sif_file *file, long x, long y, long w, long h, long b
and, void *uniform_value) {
  sif_header *hd = file->header;
  long sx = 0;            /** starting tile x index **/
  long sy = 0;            /** starting tile y index **/
  long ex = 0;            /** ending tile x index **/
  long ey = 0;            /** ending tile y index **/
  long ix, iy;            /** current tile index. */
  int uniform = 1;
  sif_tile *first_tile = 0;
  u_char *upv = 0;
  SIF_CHECK_FILE(file);
  hd = file->header;
  sx = x / hd->tile_width;
  sy = y / hd->tile_height;
  ex = (x + w - 1) / hd->tile_width;
  ey = (y + h - 1) / hd->tile_height;
  first_tile = file->tiles + ((hd->n_tiles_across * sy) + sx);
  upv = ((u_char*)first_tile->uniform_pixel_values) + hd->data_unit_size * band;

  /** Scan through each tile in the region. If we reach a tile that is
      uncompressed, stop, return false.  If we reach a tile that is
      compressed but whose data differs from the first tile, stop,
      return false.  If we scan through every tile, and each one is
      compressed and has a uniform pixel value that is identical to
      the first tile, return true. Note that we incure the cost,
      albeit minimal, by comparing the first tile with itself. This
      approach has the advantage of avoiding an extra if in the for
      statement.*/

  for (ix = sx; (ix <= ex) && uniform; ix++) {
    for (iy = sy; (iy <= ey) && uniform; iy++) {
      uniform =
        uniform
        && sif_is_slice_shallow_uniform(file, ix, iy, band, uniform_value)
        && !memcmp(upv, uniform_value, hd->data_unit_size);
    }
  }
  return uniform;
}

/* See sif-io.h for detailed documentation of public functions. */
int              sif_is_slice_shallow_uniform(sif_file *file, long tx, long ty, long band,
                                              void *uniform_value) {
  sif_header *hd = 0;
  long tile_num = 0;
  sif_tile *tile = 0;
  u_char *upv = 0;
  SIF_CHECK_FILE(file);

  hd = file->header;
  tile_num = (hd->n_tiles_across * ty) + tx;
  tile = file->tiles + tile_num;
  upv = ((u_char*)tile->uniform_pixel_values) + band * hd->data_unit_size;

  /** Is the bit for the band set? If so, the tile is compressed. Copy the
      uniform pixel value and return true. */
  if (SIF_GET_BIT(tile->uniform_flags, band)) {
    memcpy(uniform_value, upv, hd->data_unit_size);
    return 1;
  }
  /** The band is not compressed. Return false. */
  return 0;
}

/**
 * Check whether a tile (all bands) is uniform. If the tile is found to be
 * uniform (i.e., each data unit in the tile is represented by an identical
 * sequence of bytes), this common data unit sequence is stored in the
 * tile’s header, and the physical block is freed. If the tile is already
 * uniform, no change is made to the file or tile header. The tile’s dirty flag
 * is set to FALSE (not dirty) upon completion.
 *
 * @param file    The file containing the tile to check.
 * @param tile_no The index of the tile to check.
 * @param data    A buffer which contains enough bytes to store the tile.
 */

static int            _sif_uniform_check(sif_file *file, long tile_no, void *data) {
  long i = 0;
  int uniform = 1;
  sif_tile *tiles = file->tiles;
  sif_tile *tile = tiles + tile_no;
  sif_header *hd = file->header;
  u_char *datau = data, *upv = 0;
  long row = tile_no / hd->n_tiles_across, col = tile_no % hd->n_tiles_across, extentX = hd->t
ile_width, extentY = hd->tile_height;

  if (tile->block_num == -1) {
    return 1;
  }
  _sif_get_tile(file, tile_no, data);
  if (file->error != 0) {
    return -1;
  }

  extentX = MIN(hd->tile_width, hd->width - col * hd->tile_width);
  extentY = MIN(hd->tile_height, hd->height - row * hd->tile_height);

  for (i = 0; i < hd->bands; i++) {
    datau = ((u_char*)data) + (i * file->units_per_slice * hd->data_unit_size);
    if (!SIF_GET_BIT(tile->uniform_flags, i)
            && _sif_is_uniform(file, datau, extentX, extentY)
            && tile->block_num != -1) {
      upv = tile->uniform_pixel_values + (i * hd->data_unit_size);
      memcpy(upv, datau, hd->data_unit_size);
      SIF_SET_BIT(tile->uniform_flags, i);
    }
  }
  if (_sif_completely_uniform_shallow(file, tile_no) && tile->block_num != -1) {
    file->blocks_to_tiles[tiles[tile_no].block_num] = -1;
    tile->block_num = -1;
  }
  _sif_write_tile_header(file, tiles + tile_no, tile_no);
  return uniform;
```

```
}

int             _sif_is_uniform(sif_file *file, const void *data, int extentX, int extentY) {
  int uniform = 1;
  long i = 0, j = 0, k = 0;
  const unsigned char *datav = data;
  unsigned char first, second;
  sif_header *hd = file->header;
  long data_unit_size = file->header->data_unit_size;
  long num_units = 0;
  num_units = extentY * hd->tile_width;
  if (data_unit_size == 1) {
    first = *datav;

        /** In the case of border tiles (right-most and bottom-most), we don’t want to evaluat
e pixels that
            are outside the image, thus, we have two for loops. */
        for (i = 0; i < num_units && uniform; i += hd->tile_width) {
                for (j = i, k = 0; k < extentX && uniform; j++, k++) {
                   uniform = uniform && (first == datav[j]);
                }
        }
  }
  /** If our data units are words, things are a bit more complicated. We
      still want to avoid using memcmp. */
  else if (data_unit_size == 2) {
    first = datav[0];
    second = datav[1];
    for (i = 0; i < num_units && uniform; i += hd->tile_width) {
          for (j = i, k = 0; k < extentX && uniform; j++, k++) {
        uniform = uniform && (first == datav[j * 2]) && (second == datav[j * 2 + 1]);
          }
    }
  }
  else {
        for (i = 0; i < num_units && uniform; i += hd->tile_width) {
          for (j = i, k = 0; k < extentX && uniform; j++, k++) {
        uniform = uniform && (memcmp(data,
                                     datav + (data_unit_size * j),
                                                                    data_unit_size) == 0)
;
          }
    }
  }
  return uniform;
}

/**
 * Check all tiles in a file for pixel uniformity. If any tiles are found
 * to be uniform (i.e., each data unit in a tile is represented by an
 * identical sequence of bytes), the common data units are stored
 * in the tile headers, and the physical storage blocks are freed. If
 * the consolidation flag in the file’s header is turned off or
 * the file is read only, this method does nothing.
 *
 * @param file   The file to mark for uniformity.
 */

static void             _sif_mark_uniform_tiles(sif_file *file, void *buffer) {
  long i = 0;
  sif_header *hd = file->header;
  if (file->read_only || !file->header->consolidate) {
    return;
  }
  for (i = 0; i < hd->n_tiles; i++) {
    if (file->tiles[i].block_num != -1 && file->dirty_tiles[i]) {
      _sif_uniform_check(file, i, buffer);
      if (file->error != 0) {
        return;
      }
      file->dirty_tiles[i] = 0;
    }
  }
}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_defragment(sif_file *file) {
  void *buf1, *buf2;
  sif_header *hd;
  LONGLONG i = 0, tn1 = 0, tn2 = 0, bn1 = 0, bn2 = 0;
  SIF_CHECK_FILE_V(file);
  if (file->read_only || !file->header->defragment) {
    return;
  }
  hd = file->header;
  buf1 = file->buffer[0];
  buf2 = file->buffer[1];
  for (i = 0, bn1 = 0; i < hd->n_tiles; i++) {
    if (file->tiles[i].block_num != -1) {
      bn2 = file->tiles[i].block_num;
      tn1 = file->blocks_to_tiles[bn1];
      tn2 = i;

      /** Swap the block nums for bookkeeping purposes. **/
      file->tiles[tn2].block_num = bn1;
      file->blocks_to_tiles[bn1] = tn2;

      /** It’s possible that the other tile is uniform... */
      if (tn1 != -1) {
        file->tiles[tn1].block_num = bn2;
        file->blocks_to_tiles[bn2] = tn1;
        _sif_write_tile_header(file, file->tiles + tn1, tn1);
        if (file->error != 0) {
          return;
        }
      }
      else {
        file->blocks_to_tiles[bn2] = -1;
      }

      /** Swap the disk blocks. **/
      _sif_swap_blocks(file, bn1, bn2, buf1, buf2, tn1 == -1);
      if (file->error != 0) {
        return;
      }

      _sif_write_tile_header(file, file->tiles + tn2, tn2);
      if (file->error != 0) {
        return;
      }
      bn1++;
    }
  }

  /** Truncate the file. */
  /**_sif_truncate(file, _sif_get_block_location(file, _sif_get_last_used_block_index(file) +
1));**/
  if (file->error != 0) {
    return;
  }

  /** We lost the meta data, write it out again. */
  _sif_write_meta_data(file);
  if (file->error != 0) {
    return;
  }
}

/* See sif-io.h for detailed documentation of public functions. */
sif_file*        sif_open(const char *filename, int read_only) {
  sif_file *retval = 0;
#ifdef WIN32
  HANDLE fp = 0;
#else
  FILE *fp = 0;
#endif
  sif_header *header = 0;
  int i = 0;
#ifdef WIN32
  if (read_only) {
    fp = CreateFile(TEXT(filename), GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OP
EN_EXISTING, FILE_ATTRIBUTE_READONLY, NULL);
  }
  else {
    fp = CreateFile(TEXT(filename), GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE
_WRITE, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  }

#else
  if (read_only) {
    fp = fopen64(filename, "rb");
  }
  else {
    fp = fopen64(filename, "r+b");
  }
#endif
  if (FILE_IS_OKAY(fp)) {
    retval = _sif_alloc_fp();
    if (retval == 0) {
      return 0;
    }
    retval->fp = fp;
    retval->error = 0;
    header = (retval->header = _sif_alloc_header());
    if (header == 0) {
      free(retval);
      FCLOSE64(fp);
      return 0;
    }
    retval->tiles = 0;

    retval->meta_data = _sif_alloc_meta_data_table();
    retval->read_only = read_only;
    REWIND64NEC(fp);
    if (_sif_read_header(retval) != 1 ||
        header->version > SIF_VERSION ||
        strncmp(header->magic_number, SIF_MAGIC_NUMBER, SIF_MAGIC_NUMBER_SIZE) != 0 ||
        (retval->tiles = _sif_alloc_tile_headers(retval)) == 0) {
      free(header);
      free(retval);
      FCLOSE64(fp);
      return 0;
    }
    retval->base_location =  retval->header_bytes + (header->tile_header_bytes * header->n_til
es);
    retval->units_per_tile = header->tile_width * header->tile_height * header->bands;
    retval->units_per_slice = header->tile_width * header->tile_height;

    if (_sif_read_tile_headers(retval) != 1 ||
        (retval->blocks_to_tiles = (long*)malloc(header->n_tiles * sizeof(long))) == 0 ||
        (retval->dirty_tiles = (long*)malloc(header->n_tiles * sizeof(long))) == 0 ||
        (retval->buffer[0] = malloc(header->tile_bytes)) == 0 ||
        (retval->buffer[1] = malloc(header->tile_bytes)) == 0) {
      free(header);
      free(retval->tiles);
      free(retval->blocks_to_tiles);
      free(retval->dirty_tiles);
      free(retval->buffer[0]);
      free(retval->buffer[1]);
      free(retval);
      FCLOSE64(fp);
      return 0;
    }
    bzero(retval->dirty_tiles, sizeof(long) * header->n_tiles);
    for (i = 0; i < header->n_tiles; i++) {
      retval->blocks_to_tiles[i] = -1;
    }
    for (i = 0; i < header->n_tiles; i++) {
      if (retval->tiles[i].block_num != -1) {
        retval->blocks_to_tiles[retval->tiles[i].block_num] = i;
      }
    }
    _sif_read_meta_data(retval);
    if (retval->error != 0) {
      free(header);
      free(retval->tiles);
      free(retval->blocks_to_tiles);
      free(retval->dirty_tiles);
      free(retval->buffer[0]);
      free(retval->buffer[1]);
      free(retval);
      FCLOSE64(fp);
    }
  }
  return retval;
}

/* See sif-io.h for detailed documentation of public functions. */
int              sif_close(sif_file* file) {
  int status = 0;
  /** Flush whatever data has not been written to disk. */
  sif_flush(file);
  _sif_free_tile_headers(file);
  _sif_free_meta_data(file);
  free(file->header);
  free(file->blocks_to_tiles);
  free(file->dirty_tiles);
  free(file->buffer[0]);
  free(file->buffer[1]);
  free(file->simple_region_buffer);
  status = FCLOSE64(file->fp);
  if (file->error) { free(file); return -1; }
  free(file);
  return status;
}

/* See sif-io.h for detailed documentation of public functions. */
int             sif_flush(sif_file* file) {
  if (!file->read_only) {
    _sif_write_header(file);
    _sif_write_tile_headers(file);
    _sif_write_meta_data(file);
    /** Detect pixel uniformity in blocks. Any block that has pixel uniformity will be compres
sed. */
    if (file->header->consolidate) {
        sif_consolidate(file);
    }
    /** Defragment the block space. */
    if (file->header->defragment) {
        sif_defragment(file);
    }
#ifdef WIN32
    FlushFileBuffers(file->fp);
#else
    fflush(file->fp);
#endif
  }
  return 0;
}

/* See sif-io.h for detailed documentation of public functions. */
void             sif_consolidate(sif_file *file) {
  if (file->read_only || !file->header->consolidate) {
    return;
  }
  _sif_mark_uniform_tiles(file, file->buffer[0]);
  /**_sif_truncate(file, _sif_get_block_location(file, _sif_get_last_used_block_index(file) +
1));**/
  _sif_write_meta_data(file);
}

/* See sif-io.h for detailed documentation of public functions. */
sif_file*        sif_create(const char *filename, long width, long height,
                            long bands, int data_unit_size,
                            int user_data_type, int consolidate_on_close,
                            int defragment_on_close,
                            long tile_width, long tile_height,
                            int intrinsic_write) {
  sif_file *retval = 0;
  sif_header *hd = 0;
  long i = 0, tb = tile_width * tile_height * bands * data_unit_size;

  /** Check for basic sanity of the arguments. */
  if (bands < 1 || width < 1 || height < 1 || tile_width < 1 || tile_height < 1 || data_unit_s
ize < 1
      || filename == 0) {
    return 0;
  }
#ifdef WIN32
  HANDLE fp = 0;
  fp = CreateFile(filename, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
#else
  FILE *fp = 0;
  fp = fopen64(filename, "wb+");
```

```
#endif
  if (!FILE_IS_OKAY(fp)) {
    return 0;
  }
  if ((retval = _sif_alloc_fp()) == 0
      || (hd = _sif_alloc_header()) == 0
      || (retval->buffer[0] = malloc(tb)) == 0
      || (retval->buffer[1] = malloc(tb)) == 0) {
    free(hd);
    free(retval->buffer[0]);
    free(retval);
    return 0;
  }
  retval->fp = fp;
  retval->header = hd;
  retval->read_only = 0;
  retval->simple_region_buffer = 0;
  retval->simple_region_bytes = 0;
  hd->consolidate = consolidate_on_close;
  hd->intrinsic_write = intrinsic_write;
  hd->defragment = defragment_on_close;
  hd->intrinsic_write = 1;
  hd->width = width;
  hd->height = height;
  hd->bands = bands;
  hd->tile_width = tile_width;
  hd->tile_height = tile_height;
  hd->version = SIF_VERSION;
  retval->units_per_tile = tile_width * tile_height * bands;
  retval->units_per_slice = tile_width * tile_height;
  hd->tile_bytes = tile_width * tile_height * bands * data_unit_size;
  hd->data_unit_size = data_unit_size;
  hd->user_data_type = user_data_type;
  hd->n_tiles_across = CEIL_DIV(hd->width, hd->tile_width);
  hd->n_tiles = hd->n_tiles_across * CEIL_DIV(hd->height, hd->tile_height);
  hd->n_keys = 0;
  retval->meta_data = _sif_alloc_meta_data_table();
  if ((retval->tiles = _sif_alloc_tile_headers(retval)) == 0 ||
      (retval->blocks_to_tiles = (long*)malloc(hd->n_tiles * sizeof(long))) == 0 ||
      (retval->dirty_tiles = (long*)malloc(hd->n_tiles * sizeof(long))) == 0) {
    _sif_free_tile_headers(retval);
    free(hd);
    free(retval->meta_data);
    free(retval->blocks_to_tiles);
    free(retval->dirty_tiles);
    free(retval->buffer[0]);
    free(retval->buffer[1]);
    free(retval);
    return 0;
  }
  bzero(retval->dirty_tiles, hd->n_tiles * sizeof(long));
  for (i = 0; i < hd->n_tiles; i++) {
    retval->blocks_to_tiles[i] = -1;
  }
  memcpy(&(hd->magic_number), SIF_MAGIC_NUMBER, SIF_MAGIC_NUMBER_SIZE);
  _sif_write_header(retval);
  retval->base_location = retval->header_bytes + (hd->tile_header_bytes * hd->n_tiles);
  if (retval->error != 0) {
    _sif_free_tile_headers(retval);
    free(hd);
    free(retval->blocks_to_tiles);
    free(retval->dirty_tiles);
    free(retval->buffer[0]);
    free(retval->buffer[1]);
    _sif_truncate(retval, 0);
    FCLOSE64(fp);
    free(retval);
    return 0;
  }
  _sif_write_tile_headers(retval);
  if (retval->error != 0) {
    _sif_free_tile_headers(retval);
    free(hd);
    free(retval->blocks_to_tiles);
    free(retval->dirty_tiles);
    free(retval->buffer[0]);
    free(retval->buffer[1]);
    _sif_truncate(retval, 0);
    FCLOSE64(fp);
    free(retval);
  }
  sif_use_file_format_version(retval, sif_get_version());
  return retval;
}

/* See sif-io.h for detailed documentation of public functions. */
sif_file         *sif_create_copy(sif_file *file, const char *filename) {
  sif_header *hd = file->header;
  long i;
  /**  retval = sif_create(filename, hd->width, hd->height, hd->bands, hd->data_unit_size,
                      hd->user_data_type, hd->consolidate, hd->defragment,
                      hd->tile_width, hd->tile_height);**/
#ifdef WIN32
  HANDLE fp = 0;
  LARGE_INTEGER lsz;
  LONGLONG j, k, bufsize, sz;
  fp = CreateFile(filename, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
  if (!FILE_IS_OKAY(fp)) {
    return 0;
  }
  sif_flush(file);
  GetFileSizeEx(file->fp, &lsz);
  sz = lsz.QuadPart;
  REWIND64NEC(file->fp);
  bufsize = file->header->data_unit_size * file->units_per_tile;
  for (j = 0; j < sz; j += bufsize) {
    k = MIN(sz - j, bufsize);
    if (FREAD64NEC(file->buffer[0], 1, k, file->fp) == 0 ||
        FWRITE64NEC(file->buffer[0], 1, k, fp) == 0) {
      FCLOSE64(fp);
      return 0;
    }
  }
#else
  FILE *fp = 0;
  sif_flush(file);
  REWIND64NEC(file->fp);
  fp = fopen64(filename, "wb+");
  if (!FILE_IS_OKAY(fp)) {
    return 0;
  }
  while (!feof(file->fp)) {
    i = FREAD64NEC(file->buffer[0], 1, hd->data_unit_size * file->units_per_tile, file->fp);
    if (ferror(file->fp)
        || FWRITE64NEC(file->buffer[0], 1, i, fp) != i) {
```

```
      fclose(fp);
      return 0;
    }
  }
#endif
  FCLOSE64(fp);
  return sif_open(filename, 0);
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_set_user_data_type(sif_file *file, long user_data_type) {
  SIF_CHECK_FILE_V(file);
  file->header->user_data_type = user_data_type;
}

/* See sif-io.h for detailed documentation of public functions. */
long sif_get_user_data_type(sif_file *file) {
  SIF_CHECK_FILE(file);
  return file->header->user_data_type;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_set_intrinsic_write(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->intrinsic_write = 1;
}

/* See sif-io.h for detailed documentation of public functions. */
int sif_is_intrinsic_write_set(sif_file *file) {
  SIF_CHECK_FILE(file);
  return file->header->intrinsic_write;
}


/* See sif-io.h for detailed documentation of public functions. */
void sif_unset_intrinsic_write(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->intrinsic_write = 0;
}


/* See sif-io.h for detailed documentation of public functions. */
void sif_set_defragment(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->defragment = 1;
}


/* See sif-io.h for detailed documentation of public functions. */
int sif_is_defragment_set(sif_file *file) {
  SIF_CHECK_FILE(file);
  return file->header->defragment;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_unset_defragment(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->defragment = 0;
}


/* See sif-io.h for detailed documentation of public functions. */
void sif_set_consolidate(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->consolidate = 1;
}

/* See sif-io.h for detailed documentation of public functions. */
int sif_is_consolidate_set(sif_file *file) {
  SIF_CHECK_FILE(file);
  return file->header->consolidate;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_unset_consolidate(sif_file *file) {
  SIF_CHECK_FILE_V(file);
  file->header->consolidate = 0;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_set_affine_geo_transform(sif_file *file, const double *trans) {
  int i;
  SIF_CHECK_FILE_V(file);
  for (i = 0; i < 6; i++) {
    file->header->affine_geo_transform[i] = trans[i];
  }
}

/* See sif-io.h for detailed documentation of public functions. */
const double *sif_get_affine_geo_transform(sif_file *file) {
  SIF_CHECK_FILE(file);
  return file->header->affine_geo_transform;
}

/* See sif-io.h for detailed documentation of public functions. */
const char *sif_get_projection(sif_file *file) {
  const char *result;
  SIF_CHECK_FILE(file);
  result = sif_get_meta_data(file, "_sif_proj");
  if (result == 0 && file->error == SIF_ERROR_META_DATA_KEY) {
    result = "";
    file->error = SIF_ERROR_NONE;
  }
  return result;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_set_projection(sif_file *file, const char *proj) {
  SIF_CHECK_FILE_V(file);
  sif_set_meta_data(file, "_sif_proj", proj);
}

/* See sif-io.h for detailed documentation of public functions. */
const char *sif_get_agreement(sif_file *file) {
  const char *result;
  SIF_CHECK_FILE(file);
  result = sif_get_meta_data(file, "_sif_agree");
  if (result == 0 && file->error == SIF_ERROR_META_DATA_KEY) {
    result = "";
    file->error = SIF_ERROR_NONE;
  }
  return result;
}

/* See sif-io.h for detailed documentation of public functions. */
void sif_set_agreement(sif_file *file, const char *proj) {
  SIF_CHECK_FILE_V(file);
  sif_set_meta_data(file, "_sif_agree", proj);
}

/* See sif-io.h for detailed documentation of public functions. */
int               sif_get_meta_data_num_items(sif_file *file) {
  return file->header->n_keys;
}

/* See sif-io.h for detailed documentation of public functions. */
void              sif_get_meta_data_keys(sif_file *file,
                                         const char *** key_strs,
                                         int *num_keys) {
  int n, i, j;
  sif_meta_data *cur;
  SIF_CHECK_FILE_V(file);
  n = sif_get_meta_data_num_items(file);
  *key_strs = (const char**)malloc(sizeof(char*) * (n + 1));
  if (*key_strs == 0) {
    file->error = SIF_ERROR_MEM;
  }
  for (j = 0, i = 0; j < SIF_HASH_TABLE_SIZE; j++) {
    for (cur = file->meta_data[j]; cur != 0; cur = cur->next) {
      (*key_strs)[i] = cur->key;
      i++;
    }
  }
  (*key_strs)[n] = 0;
  if (num_keys != 0) {
    *num_keys = n;
  }
}

/* See sif-io.h for detailed documentation of public functions. */
void              sif_remove_meta_data_item(sif_file *file, const char *key) {
  sif_meta_data *item;
  SIF_CHECK_FILE_V(file);
  item = _sif_unlink_meta_data_pair(file, key);
  if (item) {
    free(item->value);
    free(item->key);
    free(item);
  }
}

/* See sif-io.h for detailed documentation of public functions. */
void              sif_use_file_format_version(sif_file *file, long version) {
  if (version < 1) {
    file->error = SIF_ERROR_CANNOT_WRITE_VERSION;
    return;
  }
  else {
    file->use_file_version = version;
  }
}

/* See sif-io.h for detailed documentation of public functions. */
int               sif_is_possibly_sif_file(const char *filename) {
#ifdef WIN32
  HANDLE fp = 0;
#else
  FILE *fp = 0;
#endif
```

```
  sif_header oheader;
  sif_file ofile;
  sif_header *header;
  sif_file *file;
  int retval = 0;
  /** We’re not going to return any structures to the user, so we can use the stack for the he
ader. **/
  header = &oheader;
  file = &ofile;
#ifdef WIN32
  fp = CreateFile(TEXT(filename), GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN
_EXISTING, FILE_ATTRIBUTE_READONLY, NULL);
#else
  fp = fopen64(filename, "rb");
#endif
  if (FILE_IS_OKAY(fp)) {
    file->fp = fp;
    file->error = 0;
    file->tiles = 0;

    file->meta_data = 0;
    file->read_only = 1;
    REWIND64NEC(fp);
    if (_sif_read_header(file) != 1 ||
        strncmp(header->magic_number, SIF_MAGIC_NUMBER, SIF_MAGIC_NUMBER_SIZE) != 0 ||
        (file->tiles = _sif_alloc_tile_headers(file)) == 0) {
      retval = 0;
    }
    else {
      retval = 1;
    }
    FCLOSE64(fp);
  }
  else {
    retval = -1;
  }
  return retval;
}

/* See sif-io.h for detailed documentation of public functions. */
const char *      sif_get_error_description(int code) {
  const char *str;
  switch(code) {
  case SIF_ERROR_NONE:
    str = "No error";
    break;
  case SIF_ERROR_MEM:
    str = "Memory error";
    break;
  case SIF_ERROR_NULL_FP:
    str = "Null file pointer";
    break;
  case SIF_ERROR_NULL_HDR:
    str = "Null header";
    break;
  case SIF_ERROR_INVALID_BN:
    str = "Invalid block number";
    break;
  case SIF_ERROR_INVALID_TN:
    str = "Invalid tile number";
    break;
  case SIF_ERROR_READ:
    str = "Error when reading";
    break;
  case SIF_ERROR_WRITE:
    str = "Error when writing";
    break;
  case SIF_ERROR_SEEK:
    str = "Error when seeking";
    break;
  case SIF_ERROR_TRUNCATE:
    str = "Error when truncating";
    break;
  case SIF_ERROR_INVALID_FILE_MODE:
    str = "Invalid file mode";
    break;
  case SIF_ERROR_INCOMPATIBLE_VERSION:
    str = "Cannot files of the version stored in the SIF file";
    break;
  case SIF_ERROR_META_DATA_KEY:
    str = "Cannot find a (key,value) pair with the specified key";
    break;
  case SIF_ERROR_META_DATA_VALUE:
    str = "The value of the meta-data item is invalid.";
    break;
  case SIF_ERROR_CANNOT_WRITE_VERSION:
    str = "Cannnot write files of the version requested.";
    break;
  case SIF_ERROR_INVALID_BAND:
    str = "Band index invalid (e.g. band argument).";
    break;
  case SIF_ERROR_INVALID_COORD:
    str = "Invalid coordinate (e.g. x or y).";
    break;
  case SIF_ERROR_INVALID_TILE_SIZE:
    str = "Invalid tile size (e.g. tile_width or tile_height).";
    break;
  case SIF_ERROR_INVALID_REGION_SIZE:
    str = "Invalid region size (e.g. width or height).";
    break;
  case SIF_ERROR_INVALID_BUFFER:
    str = "Invalid buffer passed (NULL?).";
    break;
  case SIF_ERROR_PNM_INCOMPATIBLE_TYPE_CODE:
    str = "Invalid type code for PNM output.";
    break;
  case SIF_ERROR_PGM_INVALID_BAND_COUNT:
    str = "Invalid band count for PGM output.";
    break;
  case SIF_ERROR_PPM_INVALID_BAND_COUNT:
    str = "Invalid band count for PPM output.";
    break;
  case SIF_ERROR_PNM_INCOMPATIBLE_DT_CONVENTION:
    str = "PNM output requires the 'simple' data type convention.";
    break;
  case SIF_SIMPLE_ERROR_UNDEFINED_DT:
    str = "Undefined data type code (simple).";
    break;
  case SIF_SIMPLE_ERROR_INCORRECT_DT:
    str = "Data type mismatch (simple).";
    break;
  case SIF_SIMPLE_ERROR_UNDEFINED_ENDIAN:
    str = "Endian code not understood (simple).";
    break;
  default:
    str = "Unknown error.";
    break;
  }
  return str;
}

void               sif_simple_set_endian(sif_file *file, int endian) {
  int simple_data_type;
  SIF_ERROR_CHECK_RETURN_V(endian < 0 || endian > 1, SIF_SIMPLE_ERROR_UNDEFINED_ENDIAN);
  SIF_CHECK_FILE_V(file);
  simple_data_type = SIF_SIMPLE_BASE_TYPE_CODE(file->header->user_data_type);
  file->header->user_data_type = simple_data_type + 10 * endian;
}

int                sif_simple_get_endian(sif_file *file) {
  SIF_CHECK_FILE(file);
  return SIF_SIMPLE_ENDIAN(file->header->user_data_type);
}

void               sif_simple_set_data_type(sif_file *file, int data_type_code) {
  int simple_endian;
  SIF_ERROR_CHECK_RETURN_V(data_type_code < 0 || data_type_code > 9, SIF_SIMPLE_ERROR_UNDEFINE
D_DT);
  SIF_CHECK_FILE_V(file);
  simple_endian = SIF_SIMPLE_ENDIAN(file->header->user_data_type);
  file->header->user_data_type = data_type_code + 10 * simple_endian;
}

int                sif_simple_get_data_type(sif_file *file) {
  SIF_CHECK_FILE(file);
  return SIF_SIMPLE_BASE_TYPE_CODE(file->header->user_data_type);
}

static const long _sif_simple_data_type_sizes_bits [] = { 8, 8, 16, 16, 32, 32, 64, 64, 32, 64
};
static const long _sif_simple_data_type_sizes_bytes [] = { 1, 1,  2,  2,  4,  4,  8,  8,  4,
8};

int               _sif_simple_alloc_region_buffer(sif_file *file, long nbytes) {
  /** If the number of bytes requested for allocation is 0 or the
      native endian is the same as the byte order of the image rasters,
      do not allocate, return successful. */
  if (nbytes == 0 || sif_simple_get_endian(file) == SIF_SIMPLE_NATIVE_ENDIAN
      || nbytes <= file->simple_region_bytes) {
    return 1;
  }
  /** If the buffer is already allocated but not big enough, try expanding its size. */
  if (file->simple_region_buffer) {
    file->simple_region_buffer = (unsigned char*)realloc(file->simple_region_buffer, nbytes);
    file->simple_region_bytes = nbytes;
  }
  /** Otherwise, allocate a new buffer. */
  else {
    file->simple_region_buffer = (unsigned char*)malloc(nbytes);
    file->simple_region_bytes = nbytes;
  }
  /** Return successful if the buffer was allocated, unsuccessful otherwise.*/
  if (file->simple_region_buffer) {
    return 1;
  }
  else {
    file->simple_region_buffer = 0;
    file->simple_region_bytes = 0;
    return 0;
  }
}

SIF_EXPORT sif_file*        sif_simple_create(const char *filename,
                                              long width, long height,
                                              long bands,
                                              int simple_data_type,
                                              int consolidate_on_close,
                                              int defragment_on_close,
                                              long tile_width, long tile_height,
                                              int intrinsic_write) {
  int user_data_type, data_unit_size;
  sif_file *retval;
  if (simple_data_type < 0 || simple_data_type > 9) {
    return 0;
  }

  user_data_type = SIF_SIMPLE_NATIVE_ENDIAN * 10 + simple_data_type;
  data_unit_size = _sif_simple_data_type_sizes_bytes[simple_data_type];

  /** Now create the file. */
  retval = sif_create(filename, width, height, bands, data_unit_size,
                      user_data_type, consolidate_on_close,
                      defragment_on_close, tile_width, tile_height, intrinsic_write);

  /** If successful, set the file's region buffer and bytes fields. **/
  if (retval != 0) {
    sif_set_agreement(retval, "simple");
  }
  return retval;
}

sif_file*                  sif_simple_create_defaults(const char *filename, long width, long he
ight,
                                                     long bands, int simple_data_type) {
  return sif_simple_create(filename, width, height, bands, simple_data_type, 1, 1, 64, 64, 1);
}

void                       sif_simple_set_raster(sif_file* file, const void *data,
                                                 long x, long y, long w, long h,
                                                 long band) {
  long region_bytes;
  int file_endian;
  SIF_CHECK_FILE_V(file);
  if (file->error) { return; }
  region_bytes = file->header->data_unit_size * w * h;
  file_endian = sif_simple_get_endian(file);
  /** If the byte order of the data elements in the file is not the same as the
      byte order of the current architecture, we need to do some byte swapping. */
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    /** See if the file's block buffer is big enough for the raster. If not, reallocate
        and anticipate more of the same big writes so keep the buffer at its increased
        size.*/
    SIF_ERROR_CHECK_RETURN_V(_sif_simple_alloc_region_buffer(file, region_bytes) == 0, SIF_ERR
OR_MEM);
    memcpy(file->simple_region_buffer, data, region_bytes);
    _sif_buffer_host_to_code(file->simple_region_buffer, region_bytes,
                             file->header->data_unit_size, file_endian);
    sif_set_raster(file, file->simple_region_buffer, x, y, w, h, band);
  }
  /** Otherwise, our task is much easier, just write the bytes to the file in native order. */
  else {
    sif_set_raster(file, data, x, y, w, h, band);
  }
}

void              sif_simple_get_raster(sif_file* file, void *data, long x, long y,
                                                   long w, long h, long band) {
  long region_bytes;
  int file_endian;
  SIF_CHECK_FILE_V(file);
  file_endian = sif_simple_get_endian(file);
  region_bytes = file->header->data_unit_size * w * h;
  /** Get the raster from the file, being ignorant about the byte order. */
  sif_get_raster(file, data, x, y, w, h, band);
  /** If an error occured during the read, just return. */
  if (file->error) { return; }
  /** Do a byte swap if the data elements stored in the file are in a different byte
      order than the native byte order. */
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    _sif_buffer_code_to_host(file->simple_region_buffer, region_bytes,
                             file->header->data_unit_size, SIF_SIMPLE_NATIVE_ENDIAN);
  }
}

void              sif_simple_fill_tiles(sif_file *file, long band, const void *value) {
  int file_endian;
  char v[8]; /** A char array with size=maximum size of any simple data type. */
  SIF_CHECK_FILE_V(file);
  if (file->read_only) { return; }
  file_endian = sif_simple_get_endian(file);
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    memcpy(&v, value, file->header->data_unit_size);
    _sif_buffer_host_to_code((unsigned char *)&v, file->header->data_unit_size,
                             file->header->data_unit_size, file_endian);
    sif_fill_tiles(file, band, &v);
  }
}

void              sif_simple_get_tile_slice(sif_file *file, void *buffer, long tx, long ty,
                                                       long band) {
  long region_bytes;
  int file_endian;
  SIF_CHECK_FILE_V(file);
  file_endian = sif_simple_get_endian(file);
  region_bytes = file->header->tile_bytes / file->header->bands;
  /** Get the raster from the file, being ignorant about the byte order. */
  sif_get_tile_slice(file, buffer, tx, ty, band);
  /** If an error occured during the read, just return. */
  if (file->error) { return; }
  /** Do a byte swap if the data elements stored in the file are in a different byte
      order than the native byte order. */
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    _sif_buffer_code_to_host((unsigned char *)file->simple_region_buffer, region_bytes,
                             file->header->data_unit_size, SIF_SIMPLE_NATIVE_ENDIAN);
  }
}

void              sif_simple_set_tile_slice(sif_file *file, const void *buffer,
                                                       long tx, long ty, long band) {
  long region_bytes;
  int file_endian;
  SIF_CHECK_FILE_V(file);
  if (file->read_only) { return; }
  region_bytes = file->header->tile_bytes / file->header->bands;
  file_endian = sif_simple_get_endian(file);
  /** If the byte order of the data elements in the file is not the same as the
      byte order of the current architecture, we need to do some byte swapping. */
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    /** See if the file’s block buffer is big enough for the raster. If not, reallocate
        and anticipate more of the same big writes so keep the buffer at its increased
        size.*/
    SIF_ERROR_CHECK_RETURN_V(_sif_simple_alloc_region_buffer(file, region_bytes) == 0, SIF_ERR
OR_MEM);
    memcpy(file->simple_region_buffer, buffer, region_bytes);
    _sif_buffer_host_to_code((unsigned char *)file->simple_region_buffer, region_bytes,
                             file->header->data_unit_size, file_endian);
    sif_set_tile_slice(file, file->simple_region_buffer, tx, ty, band);
  }
  /** Otherwise, our task is much easier, just write the bytes to the file in native order. */
  else {
    sif_set_tile_slice(file, buffer, tx, ty, band);
  }
}

void              sif_simple_fill_tile_slice(sif_file *file, long tx, long ty,
                                                        long band, const void *value) {
  int file_endian;
  char v[8];
  SIF_CHECK_FILE_V(file);
  if (file->read_only) {
    return;
  }
  file_endian = sif_simple_get_endian(file);
  if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
    memcpy(&v, value, file->header->data_unit_size);
    _sif_buffer_host_to_code((unsigned char *)&v, file->header->data_unit_size,
                             file->header->data_unit_size, file_endian);
    sif_fill_tile_slice(file, tx, ty, band, &v);
  }
}

sif_file*        sif_simple_open(const char* filename, int read_only) {
  sif_file *retval;
  long simple_region_bytes;
  retval = sif_open(filename, read_only);
  if (retval) {
    if (strcmp("simple", sif_get_agreement(retval)) != 0) {
      sif_close(retval);
      retval = 0;
    }
  }
  return retval;
}

int              sif_simple_is_shallow_uniform(sif_file *file, long x, long y, long w, long h,
 long band, void *uniform_value) {
  int retcode;
  int file_endian;
  char v[8];
  retcode = sif_is_shallow_uniform(file, x, y, w, h, band, uniform_value);
  if (retcode != 0) {
    file_endian = sif_simple_get_endian(file);
    if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
      _sif_buffer_code_to_host((unsigned char*)uniform_value, file->header->data_unit_size,
                               file->header->data_unit_size, file_endian);
    }
  }
  return retcode;
```

```
}

int              sif_simple_is_slice_shallow_uniform(sif_file *file, long tx, long ty, long ba
nd, void *uniform_value) {
  int retcode;
  int file_endian;
  char v[8];
  retcode = sif_is_slice_shallow_uniform(file, tx, ty, band, uniform_value);
  if (retcode != 0) {
    file_endian = sif_simple_get_endian(file);
    if (file_endian != SIF_SIMPLE_NATIVE_ENDIAN) {
      _sif_buffer_code_to_host((unsigned char*)uniform_value, file->header->data_unit_size,
                               file->header->data_unit_size, file_endian);
    }
  }
  return retcode;
}

int              sif_is_simple(sif_file *file) {
  int retval = 0;
  const char *agree;
  if (file) {
    agree = sif_get_agreement(file);
    if (agree) {
      if (strcmp(agree, "simple") == 0) {
        retval = 1;
      }
    }
  }
  return retval;
}

int              sif_is_simple_by_name(const char *filename) {
  int retval = 0;
  int retval2 = 0;
  const char *agree;
  sif_file *file;
  if (filename) {
    retval2 = sif_is_possibly_sif_file(filename);
    if (retval2 > 0) {
      /** Open the file. */
      file = sif_open(filename, 1);
      /** It may be the case that the file is no longer openable, for some odd reason.
          Let’s check. */
      if (file) {
        retval2 = sif_is_simple_file(file);
        /** If the file is a simple file, set the return value accordingly. */
        if (retval2 == 0) {
          retval = -2; /** -2: file is a SIF file but does not conform to the simple data type
 convention. */
        }
        else {
          retval = 1;  /** 1: file could be opened, is a SIF file, & conforms to the "simple"
convention. */
        }
        sif_close(file);
      }
      else {
        retval = -1; /** -1: file could not be opened. */
      }
    }
    else {
      retval = retval2; /**  0: file exists and openable but not SIF file
```

```
                             -1: file could not be opened. */
    }
  }
  return retval;
}

typedef struct {
#ifdef WIN32
  HANDLE fp;
#else
  FILE *fp;
#endif
} _sif_file_ptr;

int           _sif_create_blank_file(const char *filename, _sif_file_ptr *ptr) {
#ifdef WIN32
  HANDLE fp = 0;
#else
  FILE *fp = 0;
#endif
  ptr->fp = 0;

#ifdef WIN32
  fp = CreateFile(filename, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
#else
  fp = fopen64(filename, "wb+");
#endif
  if (!FILE_IS_OKAY(fp)) {
    return 0;
  }
  ftr->fp = fp;
  return 1;
}

int          sif_export_to_pgm_file(sif_file *file, const char *filename) {
  _sif_file_ptr ptr;
  int user_data_type;
  if (file == 0) {
    return 0;
  }
  if (sif_is_simple(file) < 1) {
    file->error = SIF_ERROR_PNM_INCOMPATIBLE_DT_CONVENTION;
    return 0;
  }
  if (file->header->n_bands != 1) {
    file->error = SIF_ERROR_PGM_INVALID_BAND_COUNT;
    return 0;
  }
  if (!(user_data_type == 0 || user_data_type == 2)) {
    file->error = SIF_ERROR_PNM_INCOMPATIBLE_TYPE_CODE;
    return 0;
  }
  if (_sif_create_blank_file(filename, &ptr)) {

  }
}
```

```
/**
 * \internal
 * File:            sif-io.h
 * Date:            December 17, 2004
 * Author:          Damian Eads
 * Description:     A C library for manipulating Sparse Image Format (SIF) files.
 *
 * Copyright (C) 2004-2006 The Regents of the University of California.
 *
 * Copyright (C) 2006-2008 Los Alamos National Security, LLC.
 *
 * This material was produced under U.S. Government contract
 * DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is
 * operated by Los Alamos National Security, LLC for the U.S.
 * Department of Energy. The U.S. Government has rights to use,
 * reproduce, and distribute this software.  NEITHER THE
 * GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY,
 * EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS
 * SOFTWARE.  If software is modified to produce derivative works, such
 * modified software should be clearly marked, so as not to confuse it
 * with the version available from LANL.
 *
 * Additionally, this library is free software; you can redistribute it
 * and/or modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either version 2.1
 * of the License, or (at your option) any later version. Accordingly, this
 * library is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * Los Alamos Computer Code LA-CC-06-105
 */

/**
 * \file sif-io.h
 * @brief The only header file to include for using sif-io library functions.
 */

#ifndef __SIF_IO_H_
#define __SIF_IO_H_

#include "SIFExport.h"
#if defined(_MSC_VER)
#define HAVE_LONG_LONG
#include <windows.h>
#endif

#define LONGLONG long long

#include <sys/types.h>
#include <stdio.h>

/**
 * \defgroup sif_ec SIF Error Codes
 */

/**
 * \def SIF_ERROR_NONE
 * \ingroup sif_ec
 *
 * @brief A status code indicating no error has been detected for the processing of
 * the target file.
 */

#define SIF_ERROR_NONE 0

/**
 * \def SIF_ERROR_MEM
 * \ingroup sif_ec
 *
 * @brief A status code indicating an error occurred while allocating or freeing
 * memory.
 */

#define SIF_ERROR_MEM 1

/**
 * \def SIF_ERROR_NULL_FP
 * \ingroup sif_ec
 *
 * @brief A status code indicating a file could not be processed because the
 * file pointer is null. Admittedly, there is no way to store this
 * in the sif_file struct passed since it is null. However, setting a static
 * variable for the caller to check is under consideration for a future version.
 */

#define SIF_ERROR_NULL_FP 2

/**
 * \def SIF_ERROR_NULL_HDR
 * \ingroup sif_ec
 *
 * @brief A status code indicating a file could not be processed because the
 * header pointer is null.
 */

#define SIF_ERROR_NULL_HDR 3

/**
 * \def SIF_ERROR_INVALID_BN
 * \ingroup sif_ec
 *
 * @brief A status code indicating a block number passed to a sif-io function
 * was invalid (i.e. negative or out-of-bounds).
 */

#define SIF_ERROR_INVALID_BN 4

/**
 * \def SIF_ERROR_INVALID_TN
 * \ingroup sif_ec
 *
 * @brief A status code indicating a tile number passed to a sif-io function
 * was invalid (i.e. negative or out-of-bounds).
 */

#define SIF_ERROR_INVALID_TN 5

/**
 * \def SIF_ERROR_READ
 * \ingroup sif_ec
 *
 * @brief A status code indicating an error occurred when reading from the
 * file.
 */
```

```
#define SIF_ERROR_READ 6

/**
 * \def SIF_ERROR_WRITE
 * \ingroup sif_ec
 *
 * @brief A status code indicating an error occurred when writing to the
 * file.
 */

#define SIF_ERROR_WRITE 7

/**
 * \def SIF_ERROR_SEEK
 * \ingroup sif_ec
 *
 * @brief A status code indicating an error occurred when seeking in the
 * file.
 */

#define SIF_ERROR_SEEK 8

/**
 * \def SIF_ERROR_TRUNCATE
 * \ingroup sif_ec
 *
 * @brief A status code indicating an error occurred when truncating the file.
 */

#define SIF_ERROR_TRUNCATE 9

/**
 * \def SIF_ERROR_INVALID_FILE_MODE
 * \ingroup sif_ec
 *
 * @brief A status code indicating that the file mode chosen is invalid. This
 * usually occurs when a file is opened for update that is read-only or a
 * opened when the permissions do not permit reading.
 */

#define SIF_ERROR_INVALID_FILE_MODE 10

/**
 * \def SIF_ERROR_INCOMPATIBLE_VERSION
 * \ingroup sif_ec
 *
 * @brief A status code indicating that the currently loaded sif-io library
 * is not capable of processing the version of a file. This is usually due
 * to the fact that the file was written with a later version of the SIF
 * format than the loaded library.
 */


#define SIF_ERROR_INCOMPATIBLE_VERSION 11

/**
 * \def SIF_ERROR_META_DATA_KEY
 * \ingroup sif_ec
 *
 * @brief Returned when a call is made that expects a key to be present
 * when the key cannot be found.
 */
```

```
#define SIF_ERROR_META_DATA_KEY 12

/**
 * \def SIF_ERROR_META_DATA_VALUE
 * \ingroup sif_ec
 *
 * @brief Returned by <code>sif_get_meta_data</code> when the meta-data does not contain
 * a null-terminated string.
 */

#define SIF_ERROR_META_DATA_VALUE 13

/**
 * \def SIF_ERROR_CANNOT_WRITE_VERSION
 * \ingroup sif_ec
 *
 * @brief Returned by <code>sif_use_file_version</code> when the library is not capable
 * of writing the file in the requested version.
 */

#define SIF_ERROR_CANNOT_WRITE_VERSION 14

/**
 * \def SIF_ERROR_INVALID_BAND
 * \ingroup sif_ec
 *
 * @brief Returned if a band argument passed is invalid.
 */

#define SIF_ERROR_INVALID_BAND 15

/**
 * \def SIF_ERROR_INVALID_COORD
 * \ingroup sif_ec
 *
 * @brief Returned if a coordinate argument (e.g. <code>x</code> or
 * <code>y</code>) is invalid.
 */

#define SIF_ERROR_INVALID_COORD 16

/**
 * \def SIF_ERROR_INVALID_TILE_SIZE
 * \ingroup sif_ec
 *
 * @brief Returned if a tile size argument (e.g. <code>tile_width</code> or
 * <code>tile_height</code>) is invalid.
 */

#define SIF_ERROR_INVALID_TILE_SIZE 17

/**
 * \def SIF_ERROR_INVALID_REGION_SIZE
 * \ingroup sif_ec
 *
 * @brief Returned if a region size argument (e.g. <code>width</code> or
 * <code>height</code>) is invalid.
 */

#define SIF_ERROR_INVALID_REGION_SIZE 18
```

```
/**
 * \def SIF_ERROR_INVALID_BUFFER
 * \ingroup sif_ec
 *
 * @brief Returned if a tile size argument (e.g. <code>width</code> or
 * <code>height</code>) is invalid.
 */

#define SIF_ERROR_INVALID_BUFFER 19

/**
 * \def SIF_ERROR_PNM_INCOMPATIBLE_TYPE_CODE
 * \ingroup sif_ec
 *
 * @brief Returned if the type code is not supported for PNM output.
 */

#define SIF_ERROR_PNM_INCOMPATIBLE_TYPE_CODE 20

/**
 * \def SIF_ERROR_PGM_INVALID_BAND_COUNT
 *
 * @brief Returned if the number of bands is not equal to one, which is required
 * for PGM output.
 */

#define SIF_ERROR_PGM_INVALID_BAND_COUNT 21

/**
 * \def SIF_ERROR_PPM_INVALID_BAND_COUNT
 *
 * @brief Returned if the number of bands is not equal to three, which is required
 * for PPM output.
 */

#define SIF_ERROR_PPM_INVALID_BAND_COUNT 22

/**
 * \def SIF_ERROR_PNM_INCOMPATIBLE_DT_CONVENTION
 * \ingroup sif_ec
 *
 * @brief Returned if the data type convention is not "simple".
 */

#define SIF_ERROR_PNM_INCOMPATIBLE_DT_CONVENTION 23

/**
 * \defgroup simpdecs Simple Data Type Convention Macro Definitions
 */

/**
 * \def SIF_AGREEMENT_SIMPLE
 *
 * @brief A value to set the data-type convention agreement (i.e. "_sif_agree")
 * meta-data field to indicate that the <code>simple</code> data-type convention
 * is used.
 */

#define SIF_AGREEMENT_SIMPLE "simple"

/**
 * \def SIF_AGREEMENT_GDAL
 *
 * @brief A value to set the data-type convention agreement (i.e. "_sif_agree")
 * meta-data field to indicate that the <code>gdal</code> data-type convention
 * is used.
 */

#define SIF_AGREEMENT_GDAL "gdal"

/**
 * \def SIF_MAGIC_NUMBER
 *
 * @brief A string representing the magic number. The obscure string used to easily
 * identify a file as a file in SIF format.
 */

#define SIF_MAGIC_NUMBER "!**SIF**"

/**
 * \def SIF_MAGIC_NUMBER_SIZE
 *
 * @brief The number of bytes needed to store the magic number. The obscure string is
 * used to easily identify that a file is likely to be in SIF format.
 */

#define SIF_MAGIC_NUMBER_SIZE 8

/**
 * @brief A type of function pointer, instances of which are stored internally in a sif_file o
bject.
 *
 * It is used for preprocessing a buffer and then writing it to disk. Either no preprocessing
is
 * done or bytes are properly swapped to return a buffer with native byte ordering. This funct
ion
 * only serves a purpose with the "simple" data type interface.
 *
 * The \ref sif_file::simple_region_buffer in the \sif_file object may be used or resized,
 * as needed.
 *
 * @param file    The SIF file to which the preprocessed buffer will be written.
 * @param buffer  The buffer to preprocess.
 * @param size    The size of an element (in bytes).
 * @param nmemb   The number of elements to write.
 *
 * @return The number of bytes written.
 */

typedef int (*sif_buffer_preprocessor) (sif_file *file, size_t size, size_t nmemb, const void
*buffer);

/**
 * @brief A type of function pointer, instances of which are stored internally in a sif_file o
bject.
 * It is used for postprocessing a buffer after reading it from disk. Either no preprocessing
is
 * done or bytes are properly swapped so the buffer is stored on disk using a particular byte
 * order that may be different from the host. This function only serves a purpose with the "si
mple" data
 * type interface.
 *
 * The \ref sif_file::simple_region_buffer in the \sif_file object may be used or resized,
 * as needed.
 *
 * @param file    The SIF file from which to read.
 * @param buffer  The buffer to read.
 * @param size    The size of an element (in bytes).
 * @param nmemb   The number of elements to read.
 *
 * @return The number of bytes read.
 */

typedef int (*sif_buffer_postprocessor) (sif_file *file, size_t size, size_t nmemb, void *buff
er);

/**
 * \struct sif_header
 * @brief A struct for storing a SIF file header in memory.
 *
 * @warning Changing its fields does not result in an immediate change to the header
 * stored in the file to which it points. The file must be flushed with
 * \ref sif_flush or closed with \ref sif_close. Integers are stored
 * with a sign bit in big-endian form.
 */

typedef struct SIF_EXPORT {

  /**
   * @brief This field identifies whether the header read from a file is likely
   * to correspond to a SIF file.
   *
   * These bytes must equal the string "!**SIF**" or an error will occur
   * when the header is processed by a SIF function. The byte offset of this
   * field is 0.
   *
   * @warning Do not edit this field directly.
   */

  char                   magic_number[SIF_MAGIC_NUMBER_SIZE];

  /**
   * @brief The minimum version of the SIF library needed to read this file.
   *
   * @warning Do not edit this field directly. Changing the value of this field
   * without a corresponding change to the organization of the file may make
   * it unreadable.
   */

  long                   version;

  /**
   * @brief The width of the image in pixels.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   width;

  /**
   * @brief The height of the image in pixels.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   height;
  /**
   * @brief The number of bands of the image.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   bands;

  /**
   * @brief The number of keys stored in the meta-data.
   *
   * @warning Do not edit this field directly.
   * Instead use the \ref sif_get_meta_data, \ref sif_get_meta_data_binary,
   * \ref sif_set_meta_data, and \ref sif_get_meta_data_binary functions.
   */

  long                   n_keys;

  /**
   * @brief The number of tiles that comprise this image.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   n_tiles;

  /**
   * @brief The width of each tile in pixels.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   *
   * @warning Non-square tiles have not been tested.
   */

  long                   tile_width;

  /**
   * @brief The height of each tile in pixels.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   *
   * @warning Non-square tiles have not been tested.
   */

  long                   tile_height;

  /**
   * @brief The number of bytes required to store a single tile raster. This is equal
   * to tile_width * tile_height * n_bands * data_unit_size.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   tile_bytes;

  /**
   * @brief The number of tiles across the width of an image.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   n_tiles_across;

  /**
   * @brief The number of bytes required to store each pixel.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   *
   * @warning Non-square tiles have not been tested.
   */

  long                   data_unit_size;

  /**
   * @brief A number that is only read from and written to the SIF file header. It has no
   * meaning to the sif-io functions since sif-io processes images without regard to the
   * data type of the pixels. The caller to the library function can use the field to store
   * an integer that represents the data type of the pixels in the image.
   *
   * @warning Do not edit this field directly. Instead use the \ref sif_set_user_data_type fun
ction.
   */

  long                   user_data_type;

  /**
   * @brief A field that, when nonzero, indicates the file should be defragmented when its clo
sed.
   *
   * @warning Do not edit this field directly. Instead use the \ref sif_set_defragment or
   * \ref sif_unset_defragment functions.
   */

  long                   defragment;

  /**
   * @brief A field, that when nonzero, indicates that the file should be consolidated when it
s closed.
   *
   * This involves performing pixel uniformity checks on each dirty tile during close.
   *
   * @warning Do not edit this field directly. Instead use the \ref sif_set_defragment or
   * \ref sif_unset_defragment functions.
   */

  long                   consolidate;

  /**
   * @brief A field, that when nonzero, indicates that when each tile is written, a uniformity
 check
   * should be performed.
   *
   * @warning Do not edit this field directly. Instead use the \ref sif_set_intrinsic_write
   * or \ref sif_unset_intrinsic_write functions.
   */

  long                   intrinsic_write;

  /**
```

```
   * @brief The number of bytes needed to store the header for each tile.
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   tile_header_bytes;

  /**
   * @brief The number of bytes to store the uniformity flags, i.e.
   * Ceil(number_of_flags / 8).
   *
   * @warning Do not edit this field directly. Changing its value without changing
   * the image layout on disk will make the file unreadable.
   */

  long                   n_uniform_flags;

  /**
   * @brief Six doubles representing the affine georeferencing transform parameters.
   *
   * The georeferenced coordinates of the pixel coordinate (Xpixel, Yline)
   * are computed as follows (from GDAL documentation):
   * \code
   *  const double *GT = &(hd->affine_geo_transform);
   *  Xgeo = GT[0] + Xpixel * GT[1] + Yline * GT[2];
   *  Ygeo = GT[3] + Xpixel * GT[4] + Yline * GT[5];
   * \endcode
   *
   * The transform is set to {0.0, 1.0, 0.0, 0.0, 0.0, 1.0} by default
   * by sif_create so that x and y are just mapped to themselves.
   *
   * @warning Do not edit this field directly. Instead use the
   * \ref sif_set_affine_geo_transform function.
   */

  double                 affine_geo_transform[6];
} sif_header;

/**
 * \struct sif_tile
 * @brief A struct for storing a SIF tile header in memory. It
 * stores important information related to a tile, including
 * which of its bands are uniform, and the uniform pixel values
 * of the bands.
 */

typedef struct SIF_EXPORT {
  //  long                   block_num;
  //  char                   uniform_pixel_value[16];

  /**
   * @brief A byte sequence where the i’th bit in the sequence
   * indicates whether the i’th band in the tile is uniform. The
   * number of bytes is Ceil(n_bands / 8).
   */
  u_char                 *uniform_flags;

  /**
   * @brief A sequence of pixel data units. The i’th data unit
   * represents the uniform pixel value for the i’th band. The
   * number of bytes is n_bands * data_unit_size.
   */
  u_char                  *uniform_pixel_values;

  /**
   * @brief The block location of the file where the tile is stored.
   *
   * This number is -1 if the tile is completely uniform, i.e. each
   * band in the tile is completely uniform. Note that the bands
   * of a tile (i.e. slices) may have different uniform pixel values.
   * A tile or block is uniform iff each of its slices is uniform.
   */

  long                    block_num;

} sif_tile;

/**
 * \struct sif_meta_data
 * @brief A struct for storing meta-data in memory. It stores a node
 * in a linked list of meta-data.
 *
 * @warning Do not modify this data structure directly. Instead use
 * the \ref sif_set_meta_data and \ref sif_set_meta_data_binary
 * functions.
 */

typedef struct SIF_EXPORT sif_meta_data {

  /**
   * @brief The key identifier of this meta-data field.
   */

  char*                   key;

  /**
   * @brief The value of this meta-data field.
   */

  char*                   value;

  /**
   * @brief The number of bytes to store the key and its null terminator.
   */

  unsigned long           key_length;

  /**
   * @brief The number of bytes to store the value. If the value is binary,
   * the null terminator is included in this count.
   */

  unsigned long           value_length;

  /**
   * @brief A pointer to the next meta-data field. The value is NULL if
   * there is no next meta-data field.
   */

  struct sif_meta_data*  next;

} sif_meta_data;

/** \internal
```

```
   We need the CFile class for 64-bit file support in windows. Microsoft does
   not conform to the LFS standard.
*/

/**
 * \struct sif_file
 * @brief A struct for storing necessary data for the processing of an
 * open file.
 *
 * @warning Do not modify this data structure directly.
 */


typedef struct SIF_EXPORT {
#ifdef WIN32
  /** @brief The handle to the internal file pointer. */
  HANDLE                    fp;
#else
  /** @brief The handle to the internal file pointer. */
  FILE*                     fp;
#endif
  /**
   * @brief The header corresponding to the target file.
   */

  sif_header*               header;

  /**
   * @brief An array of tiles to store.
   */

  sif_tile*                 tiles;

  /**
   * @brief The meta-data for the file. This structure is a linked list
   * of (key, value) pairs. Meta-data in SIF can be null-terminated strings
   * or binary data blocks.
   */
  sif_meta_data**           meta_data;

  /**
   * @brief A flag indicating whether the file is open in
   * read-only mode.
   */
  int                       read_only;

  /**
   * @brief An array where the i'th value is the tile index of the
   * tile stored in data block i.

   * If no tile is stored in data block i, the block is unused,
   * and the corresponding value in this array is set to -1. Unused
   * blocks can be reclaimed for use or truncated when the file is
   * consolidated or defragmented.
   */

  long*                     blocks_to_tiles;

  /**
   * @brief An array of booleans where the i'th value is one iff the
   * i'th tile has been written and no uniformity check was made
   * during the write. In this future, the type of the values contained
   * in the array will be changed to char*, pending confirmation that
   * the change does not break regression tests.
   */

  long*                    dirty_tiles;

  /**
   * @brief Two buffers with enough memory to each store one block. The
   * number of bytes for one block is computed by,
   * \code
   *    tile_width * tile_height * n_bands * data_unit_size .
   * \endcode
   */

  void*                    buffer[2];

  /**
   * \internal @brief Stores the projection string, which is expected
   * to be empty ("") or in  OpenGIS WKT format.
   *
   * @warning Do not modify this value directly. Instead use the
   * \ref sif_set_projection function.
   */

  /**  char*                  proj;**/

  /**
   * @brief Stores the byte offset of the first block in the file.
   */

  LONGLONG                 base_location;

  /**
   * @brief An error code for the last error that occurred. The value
   * is non-zero if an error occurred during the last sif-io call.
   */

  int                      error;

  /**
   * @brief When the C standard library is used, it represents
   * the last errno encountered when executing a libc function in
   * a sif-io function. Otherwise, it represents the WIN32 error code
   * returned by the GetLastErr function.
   */

  long                     sys_error_no;

  /**
   * @brief The number of pixels per band in a tile (slice). This value
   * is simply tile_width * tile_height. The term slice differs slightly
   * from the term band, it is a band within a tile.
   */
  long                     units_per_slice;

  /**
   * @brief The number of pixels per tile. This value is simply the number
   * of units_per_slice times the number of bands in the image. This number
   * is also the number of units in a block.
   */

  long                     units_per_tile;

  /**
```

```
   * @brief The number of bytes to store the header.
   */

  long                      header_bytes;

  /**
   * @brief A character buffer with enough bytes to store a 64-bit integer.
   */

  u_char                    ubuf[8];

  /**
   * @brief A integer representing the SIF file version to use when
   * writing the file. This is used to ensure that the file is written
   * with an earlier version so that it can be read by previous versions
   * of this library.
   */

  long                      use_file_version;

  /**
   * @brief A buffered only used by the SIF "simple" interface for byte
   * swapping prior to writing to a file. It is initially holds the number
   * of bytes needed to store a tile slice but grows as larger regions are
   * written.
   */

  void*                     simple_region_buffer;

  /**
   * @brief The size of the simple region buffer (in bytes).
   */

  long                      simple_region_bytes;

  /**
   * @brief The line number of sif-io.c where the last SIF error occurred.
   */

  int                       error_line_no;

  /**
   * @brief The function to call to write image buffers or uniform pixel value
   * buffers to a disk.
   */

  sif_buffer_preprocessor  preprocessor;

  /**
   * @brief The function to call to read image buffers or uniform pixel value
   * buffers to a disk.
   */

  sif_buffer_postprocessor postprocessor;

} sif_file;

/**
 * @brief Return the latest version of the SIF file format that the
 * currently loaded SIF library can process.
 *
 * @return The latest version number of a SIF file this library can
 *         process.
 */

SIF_EXPORT long             sif_get_version();

/**
 * @brief Open a Sparse Image File (SIF) format file for reading or update.
 *
 * @param filename  The filename of the SIF file to open.
 * @param read_only A flag indicating whether to open as read-only (1)
 *                  or update (0).
 *
 * @return A file structure containing all the constructs needed to
 *         manipulate the opened SIF file is returned. NULL is returned
 *         if an error occured during open.
 */

SIF_EXPORT sif_file*        sif_open(const char* filename, int read_only);

/**
 * @brief Create a new Sparse Image Format (SIF) file with a given filename
 * and attributes. The file’s header and tile headers are written. No
 * space is preallocated for data blocks.
 *
 * @param filename                The filename of the new file.
 * @param width                   The width of the image to store in the file to create.
 * @param height                  The height of the image to store in the file to create.
 * @param bands                   The number of bands of the image to store in the file to creat
e.
 * @param data_unit_size          The size of a single pixel in bytes, e.g. <code>sizeof(pixel_d
ata_type)</code>.
 * @param user_data_type          A user-defined data type. The SIF I/O functions do not
 *                                look at this value. This is strictly for the user’s reference
 *                                when opening a pre-existing file.
 * @param consolidate_on_close    Defines whether an intrinsic uniformity check should be applie
d to
 *                                dirty tiles during each close.
 * @param defragment_on_close     Defines whether the file should be defragmented during each cl
ose.
 * @param tile_width              The width of a single tile.
 * @param tile_height             The height of a single tile.
 * @param intrinsic_write         Defines whether intrinsic uniformity checks should be performe
d
 *                                when rasters are written to a file.
 *
 * @return  A file structure containing the constructs needed to manipulate the file created
 *          by this function. This function returns NULL if an error occurs during creation.
 */

SIF_EXPORT sif_file*        sif_create(const char *filename, long width, long height,
                                       long bands, int data_unit_size,
                                       int user_data_type, int consolidate_on_close,
                                       int defragment_on_close,
                                       long tile_width, long tile_height,
                                       int intrinsic_write);

/**
 * @brief Create a copy of a SIF file.
 *
 * @warning Note that this function has neither been tested nor ported for use with WIN32+MSVS
.
 *
 * @param file      The file structure pointing to the file to copy. This
 *                  file is flushed before its contents are read.
 * @param filename  The filename of the file to store the copy.
 *
 * @return  A file structure containing the constructs needed to manipulate the file copied
 *          by this function. This function returns NULL if an error occurs during file
 *          creation.
 */

SIF_EXPORT sif_file*        sif_create_copy(sif_file *file, const char *filename);

/**
 * @brief Close a SIF file.

 * If the file is open for reading and writing, defragmentation
 * and consolidation occur only if the defragment and consolidate
 * flags are set in the file’s header. The file header, tile headers, and
 * meta data are written upon close.
 *
 * @param file   The SIF file to close.
 *
 * @return       The status of the close.
 *
 * @see sif_header::consolidate
 * @see sif_header::defragment
 * @see sif_set_consolidate
 * @see sif_unset_consolidate
 * @see sif_is_consolidate_set
 */

SIF_EXPORT int              sif_close(sif_file* file);

/**
 * @brief Check all tiles in a file for intrinsic uniformity.
 *
 * If a tile is found to be intrinsically uniform, its common pixel values
 * for each slice is stored in its header and the physical storage block it
 * is using is freed. If the consolidation flag in the file’s header is
 * turned off or the file is read only, this method does nothing.
 *
 * @param file   The file to mark for uniformity.
 */

SIF_EXPORT void             sif_consolidate(sif_file* file);

/**
 * @brief Defragment the file.
 *
 * This results in a sort of the storage blocks
 * so they are in the order of their corresponding tile indices. This
 * enables faster reading/writing of continguous blocks. No unused
 * storage blocks remain in the file (i.e. the used blocks are shifted
 * so that they write over the unused blocks). The file is
 * truncated at the position of the last used storage block byte. Meta-data
 * and the file’s header are rewritten.
 *
 * @param file   The file to defragment.
 */

SIF_EXPORT void             sif_defragment(sif_file* file);

/**
 * @brief Writes a rectangular image region to a file.
 *
 * The tiles changed by this write are not checked for pixel uniformity.
 * This results in the dirty flags in their respective tile headers being
 * set to true. This results in a uniformity check during the file’s close unless
 * the uniformity check flag is set to false in the file’s header. Also, any
 * fragmentation caused by this function is not resolved until the file is closed.
 *
 * @warning This function has not been tested.
 *
 * @param file   The file on which to write the raster plane.
 * @param data   The buffer containing the raster to write.
 * @param x      The starting horizontal pixel offset (0..N-1 indexed) to write.
 * @param y      The starting vertical pixel offset (0..N-1 indexed) to write.
 * @param w      The width of the region.
 * @param h      The height of the region.
 * @param band   The band offset (0..N-1 indexed).
 *
 * @see sif_get_raster
 * @see sif_get_tile_slice
 * @see sif_set_tile_slice
 */

SIF_EXPORT void             sif_set_raster(sif_file* file, const void *data,
                                long x, long y, long w, long h, long band);

/**
 * @brief Reads a rectangular raster region from a file. It may overlap
 * multiple tiles in the file.
 *
 * @warning This function has not been tested.
 *
 * @param file   The file on which to read the raster plane out.
 * @param data   The buffer to store the raster plane.
 * @param x      The starting horizontal pixel offset (0..N-1 indexed) to read.
 * @param y      The starting vertical pixel offset (0..N-1 indexed) to read.
 * @param w      The width of the region.
 * @param h      The height of the region.
 * @param band   The band offset (0..N-1 indexed).
 *
 * @see sif_set_raster
 * @see sif_get_tile_slice
 * @see sif_set_tile_slice
 */


SIF_EXPORT void             sif_get_raster(sif_file* file, void *data,
                                long x, long y, long w, long h, long band);

/**
 * @brief Fill all tiles of a particular band with a constant value.
 *
 * If uniformity results as a result of this fill, the corresponding
 * tiles are marked appropriately and the block space they use is
 * freed.
 *
 * @param file  The file on which to perform the fill.
 * @param band  The band index of the tile to retrieve (0..N-1 indexed).
 * @param value The value to fill all values of the slice. It
 *              must be data_unit_size in bytes.
 */

SIF_EXPORT void             sif_fill_tiles(sif_file *file, long band, const void *value);

/**
 * @brief Retrieve a tile slice.
 *
 * If the tile is uniform, no access to the disk is made; instead,
 * the uniform pixel value for the band in the tile’s header is
 * used to fill the buffer. The buffer must contain enough bytes
 * to hold a slice.
 *
 * @param file   The file on which to perform the fill.
 * @param tx     The horizontal index of the slice to retrieve (0..N-1 indexed).
 * @param ty     The vertical index of the slice to retrieve (0..N-1 indexed).
 * @param band   The band index of the slice to retrieve (0..N-1 indexed).
 * @param buffer The buffer to store the tile slice. It must be data_unit_size in
 *               bytes.
 *
 * @see sif_fill_tile_slice
 */

SIF_EXPORT void             sif_get_tile_slice(sif_file *file, void *buffer, long tx, long ty,
 long band);

/**
 * @brief Store a tile slice.
 *
 * A check is not made to determine pixel uniformity. The tile’s dirty flag
 * is set to true. This results in a uniformity check during the file’s close,
 * unless uniformity check flag is disabled in the file’s header. Also, any
 * fragmentation caused by this function is not resolved until the file is closed.
 *
 * @param file    The file on which to perform the write.
 * @param tx      The horizontal index (0..N-1 indexed) of the slice to write.
 * @param ty      The vertical index (0..N-1 indexed) of the slice to write.
 * @param band    The band index (0..N-1 indexed) of the slice to write.
 * @param buffer  The buffer to write. It must have enough bytes for an entire
 *                tile slice, e.g. \ref sif_header::tile_width *
 *                \ref sif_header::tile_height * \ref sif_header::data_unit_size.
 */

SIF_EXPORT void             sif_set_tile_slice(sif_file *file, const void *buffer, long tx, lo
ng ty, long band);

/**
 * @brief Fill a tile slice with a constant value.
 *
 * If all bands become uniform as a result of this fill, the block for this
 * slice’s corresponding cube will be freed.
 *
 * @warning This function has not been tested.
 *
 * @param file  The file on which to perform the fill.
 * @param tx    The horizontal index of the tile to fill (0..N-1 indexed).
 * @param ty    The vertical index of the tile to fill (0..N-1 indexed).
 * @param band  The band index of the tile to fill (0..N-1 indexed).
 * @param value The value to fill all values of the slice. It
 *              must be \ref sif_header::data_unit_size bytes in size.
 *
 * @see sif_fill_tiles
 */

SIF_EXPORT void             sif_fill_tile_slice(sif_file *file, long tx, long ty, long band, c
onst void *value);

/**
 * @brief Set a meta-data field with a given key to a value defined by
 * a null-terminated character string.
 *
 * @param file      The file to set the meta-data.
 * @param key       The key of the field to set.
 * @param value     The value to set the field.
 *
 * @see sif_set_meta_data_binary
 * @see sif_get_meta_data
 * @see sif_get_meta_data_binary
 */

SIF_EXPORT void             sif_set_meta_data(sif_file *file, const char *key, const char *val
ue);

/**
 * @brief Set a meta-data field with a given key to a given
 * sequence of bytes.
 *
 * The length of the value is passed here, thereby allowing for
 * binary, non-null-terminated, meta-data.
 *
 * @param file      The file on which to set the meta-data field.
 * @param key       The key of the field to set.
 * @param buffer    The value to set the field.
 * @param n_bytes   The length of the value (in bytes).
 *
 * @warning The meta-data is not written to the file until the file
 * is closed or flushed.
 *
 * @see sif_set_meta_data
 * @see sif_get_meta_data
 * @see sif_get_meta_data_binary
 */

SIF_EXPORT void             sif_set_meta_data_binary(sif_file *file, const char *key, const vo
id *buffer, int n_bytes);

/**
 * @brief Get a string meta-data field with a given key. This function
 * returns 0 and sets the error field in the file's header if the buffer
 * stored for this meta-data is not a null-terminated string or if the
 * field with the given key string could not be found.
 *
 * @param file      The file on which to set the meta-data field.
 * @param key       The key of the field to set.
 *
 * @return          The value of the field.
 *
 * @see sif_get_meta_data_binary
 * @see sif_set_meta_data
 * @see sif_set_meta_data_binary
 */

SIF_EXPORT const char*      sif_get_meta_data(sif_file *file, const char *key);

/**
 * @brief Get a string meta-data field with a given key. This function
 * returns 0 and sets the error field in the file's header.
 *
 * @param file      The file to set the meta-data.
 * @param key       The key of the field to set.
 * @param n_bytes   A pointer to an integer value. This
 *                  value is set to the size of the buffer
 *                  returned.
 *
 * @return          The value of the field.
 */


SIF_EXPORT const void*      sif_get_meta_data_binary(sif_file *file, const char *key, int *n_b
ytes);

/**
 * @brief Determine if the tiles comprising a region are shallow uniform.
 *
 * @param file          The file to perform the check.
 * @param x             The starting horizontal pixel offset (0..N-1 indexed)
 *                      of the region to check.
 * @param y             The starting vertical pixel offset (0..N-1 indexed)
 *                      of the region to check.
 * @param w             The width of the region.to check.
 * @param h             The height of the region to check.
 * @param band          The band offset (0..N-1 indexed).
 * @param uniform_value This value is only meaningful when this function
 *                      returns true (non-zero). It is expected that the
 *                      pointer passed point to at least data_unit_size bytes.
 *                      When the region is completely uniform, the uniform
 *                      pixel value is stored here.
 *
 * @return 0 if the tiles are not shallow uniform or a memory allocation error
 *           occurred, otherwise a non-zero value.
 */

SIF_EXPORT int              sif_is_shallow_uniform(sif_file *file, long x, long y, long w, lon
g h, long band, void *uniform_value);

/**
 * @brief Determine if a tile has shallow uniformity.
 *
 * @param file          The file to perform the check.
 * @param tx            The horizontal tile index (0..N-1 indexed)
 *                      of the region to check.
 * @param ty            The vertical tile index (0..N-1 indexed)
 *                      of the region to check.
 * @param band          The band offset (0..N-1 indexed).
 * @param uniform_value This value is only meaningful when this function
 *                      returns true (non-zero). It is expected that the
 *                      pointer passed point to at least data_unit_size bytes.
 *                      When the region is completely uniform, the uniform
 *                      pixel value is stored here.
 *
 * @return 0 if the tiles are non-uniform or a memory allocation error
 *           occurred, otherwise a non-zero value.
 */

SIF_EXPORT int              sif_is_slice_shallow_uniform(sif_file *file, long tx, long ty, lon
g band, void *uniform_value);

/**
 * @brief Flush all remaining unwritten data to the file.
 *
 * This function immediately returns if the file passed is read-only.
 *
 * @param file   The SIF file to flush.
 *
 * @return A non-zero value if no error occurred during the flush.
 */
```

```
SIF_EXPORT int              sif_flush(sif_file* file);

/**
 * @brief Set the user data type for the file.
 *
 * This value does not change the behavior of any sif-io functions. The user
 * may use it to store an integer representing the data type of the pixel
 * values in the image.
 *
 * @param file            The file to change the data type flag.
 * @param user_data_type  The value of the new user-defined data type flag.
 *
 */

SIF_EXPORT void             sif_set_user_data_type(sif_file *file, long user_data_type);

/**
 * @brief Get the user data type integer for the file.
 *
 * This value does not change the behavior of any \ref sif-io.h functions. The user
 * may use it to store an integer representing the data type of the pixel
 * values in the image.
 *
 * @param file            The file on which to get the user-defined data type flag.
 *
 * @return                The user-defined data type of the data units in the
 *                        file.
 */

SIF_EXPORT long             sif_get_user_data_type(sif_file *file);

/**
 * @brief Set the uniformity flag. This results in a pixel uniformity check on
 * all dirty tiles.
 *
 * @param file            The file to change.
 * @see sif_is_intrinsic_write_set
 * @see sif_unset_intrinsic_write
 * @see sif_is_shallow_uniform
 * @see sif_is_slice_shallow_uniform
 */

void                         sif_set_intrinsic_write(sif_file *file);

/**
 * @brief Return the value of the uniformity flag. When true, a uniformity
 * check is perfomed on all dirty tiles during close.
 *
 * @param file            The file to check.
 * @see sif_unset_intrinsic_write
 * @see sif_is_intrinsic_write_set
 * @see sif_is_shallow_uniform
 * @see sif_is_slice_shallow_uniform
 *
 * @return The value of the flag.
 */

SIF_EXPORT int              sif_is_intrinsic_write_set(sif_file *file);

/**
 * @brief Unset the uniformity flag.
 *
 * This cancels pixel uniformity checks during close.
 *
 * @param file            The file to change.
 * @see sif_unset_intrinsic_write
 * @see sif_set_intrinsic_write
 * @see sif_is_shallow_uniform
 * @see sif_is_slice_shallow_uniform
 */

SIF_EXPORT void             sif_unset_intrinsic_write(sif_file *file);

/**
 * @brief Set the defragmentation flag.
 *
 * Defragmentation is then performed during the file’s close. Data blocks
 * are rearranged in the order they appear in the image.
 *
 * @param file            The file to change.
 * @see sif_unset_defragment
 * @see sif_is_defragment_set
 */

SIF_EXPORT void             sif_set_defragment(sif_file *file);

/**
 * @brief Unsets the defragmentation flag.
 *
 * This cancels defragmentation when the file is closed.
 *
 * @param file            The file to change.
 *
 * @return The value of the defragmentation flag.
 * @see sif_set_defragment
 * @see sif_unset_defragment
 * @see sif_defragment
 * @see sif_close
 * @see sif_header::defragment
 */

SIF_EXPORT int               sif_is_defragment_set(sif_file *file);

/**
 * @brief Set the defragmentation flag.
 *
 * Defragmentation on the file’s close is cancelled.
 *
 * @param file            The file to change.
 * @see sif_set_defragment
 * @see sif_is_defragment_set
 * @see sif_defragment
 * @see sif_close
 * @see sif_header::defragment
 */

SIF_EXPORT void              sif_unset_defragment(sif_file *file);

/**
 * @brief Set the consolidation flag.
 *
 * Consolidation is then performed during the files close.
 *
 * @param file            The file to change.
 * @see sif_unset_consolidate
 * @see sif_is_consolidate_set
 * @see sif_consolidate
 * @see sif_close
 * @see sif_header::consolidate
 */

SIF_EXPORT void              sif_set_consolidate(sif_file *file);

/**
 * @brief Return whether the file will be scheduled for consolidation
 * on its close. Used blocks are moved toward the begining of the
 * file, taking up the space of unused blocks before them. If there
 * are no unused blocks or all of the unused blocks are at the end of
 * the file, this file is simply truncated at the location of the first
 * byte of the unused block and the meta-data is rewritten.
 *
 * @param file            The file to check.
 *
 * @return The value of the consolidation flag.
 * @see sif_unset_consolidate
 * @see sif_set_consolidate
 * @see sif_consolidate
 * @see sif_close
 * @see sif_header::consolidate
 */

SIF_EXPORT int               sif_is_consolidate_set(sif_file *file);

/**
 * @brief Unset the consolidation flag.
 *
 * Consolidation is then performed during the files close.
 *
 * @param file            The file to change.
 * @see sif_set_consolidate
 * @see sif_is_consolidate_set
 * @see sif_consolidate
 * @see sif_close
 * @see sif_header::consolidate
 */

SIF_EXPORT void              sif_unset_consolidate(sif_file *file);

/**
 * @brief Set the affine georeferencing transform of an open file.
 *
 * @param file            The file to set the affine georeferencing transform.
 * @param trans           A double array of size 6 with the new value of the
 *                        georeferencing transform.
 *
 * @see sif_get_affine_geo_transform
 * @see sif_header::affine_geo_transform
 */

SIF_EXPORT void              sif_set_affine_geo_transform(sif_file *file,
                                                          const double *trans);

/**
 * @brief Get the affine georeferencing transform of an open file.
 *
 * @param file            The file to get the transform.
 * @return                An array of six doubles representing the transform.
 *
 * @see sif_set_affine_geo_transform
 * @see sif_header::affine_geo_transform
 */

SIF_EXPORT const double *    sif_get_affine_geo_transform(sif_file *file);

/**
 * @brief Return the projection string of an open file. It is expected
 * to be in OpenGIS WKT format. The string is stored in the "_sif_proj"
 * field in the meta-data region of the file. If the projection string
 * cannot be found, the empty string is returned.
 *
 * @param file             The file from which to get the projection string.
 *
 * @return The projection string.
 *
 * @see sif_set_projection
 */

SIF_EXPORT const char *      sif_get_projection(sif_file *file);

/**
 * @brief Set the projection string of an open file. This is expected
 * to be empty ("") or in OpenGIS WKT format. The string is stored in
 * the "_sif_proj" field in the meta-data region of the file.
 *
 * @param file             The file to set the projection string.
 * @param proj             The new projection string value.
 *
 * @see sif_get_projection
 */

SIF_EXPORT void              sif_set_projection(sif_file *file, const char *proj);

/**
 * @brief Return a string indicating the data type convention used in
 *        this file. If the string is <code>"gdal"</code> then the GDT type codes in
 *        the GDAL library are used. If the string is <code>"simple"</code> then
 *        the convention presented earlier in this document is used.
 *
 * @param file             The file from which to get the projection string.
 *
 * @return The convention agreement string.
 *
 * @see sif_set_agreement
 */

SIF_EXPORT const char *      sif_get_agreement(sif_file *file);

/**
 * @brief Set a string indicating the data type convention used in this
 *        file. If the string is "gdal" then the GDT type codes in the
 *        GDAL library are used. If the string is "simple" then the
 *        convention presented earlier in this document is used.
 *
 * @param file             The file to set the projection string.
 * @param agree            The new projection string value.
 *
 * @see sif_get_agreement
 */

SIF_EXPORT void              sif_set_agreement(sif_file *file, const char *agree);
```

```
/**
 * @brief Get the number of meta data (key, value) pairs in this file.
 *
 * @param file              The file from which to get the number of meta-data items.
 */

SIF_EXPORT int                sif_get_meta_data_num_items(sif_file *file);

/**
 * @brief Retrieve the keys of the meta data stored in the file. It is the
 * responsibility of the caller to free the memory to which *key_strs points
 * but not (*key_strs)[i] for any i, non-negative i < \ref sif_header::n_keys.
 *
 * @param file              The file from which to retrieve the meta-data keys.
 * @param key_strs          A pointer pointing to the pointer to set to the location
 *                          of the array of strings. The last value of the array of
 *                          strings is set to 0 (sentinel).
 * @param num_keys          A pointer to the int to store the number of keys retrieved
 *                          by this function.
 */

SIF_EXPORT void               sif_get_meta_data_keys(sif_file *file, const char *** key_strs, i
nt *num_keys);

/**
 * @brief Removes a meta-data item by its key string.
 *
 * @param file              The file from which to remove a meta-data item.
 * @param key               The key of the meta-data item to remove.
 */

SIF_EXPORT void               sif_remove_meta_data_item(sif_file *file, const char *key);

/**
 * @brief Specifies that when the file is written, the file should be
 * written with using the SIF file format version specified. If the version
 * is not supported for write, a SIF_ERROR_CANNOT_WRITE_VERSION error
 * is set in the header’s error code field.
 */
SIF_EXPORT void               sif_use_file_format_version(sif_file *file, long version);

/**
 * @brief Returns a positive value if the file referred to by <code>filename</code>
 * could possibly be a SIF file.
 *
 * @param filename          The filename of the file to check.
 *
 * @return A boolean indicating the result of the check.
 */

SIF_EXPORT int                sif_is_possibly_sif_file(const char *filename);

/**
 * @brief Returns a description of a SIF error code.
 *
 * @param code              The code of the error.
 *
 * @return A description of the error as a string. Returns -1 if the file could not
 * be opened. Returns 0 if the open was successful but the file is not a SIF file.
 */

SIF_EXPORT const char *       sif_get_error_description(int code);
```

```
/**
 * \defgroup simpfuncs Simple Data-Type Convention Declarations
 */

/**
 * \def SIF_SIMPLE_ERROR_UNDEFINED_DT
 *
 * @brief An error indicating that the data type is not recognized as a data type
 * in the simple data-type convention.
 */

#define SIF_SIMPLE_ERROR_UNDEFINED_DT 100

/**
 * \def SIF_SIMPLE_ERROR_INCORRECT_DT
 *
 * @brief An error to indicate the data type of the image does not correspond to
 * the data type requested.
 */

#define SIF_SIMPLE_ERROR_INCORRECT_DT 101

/**
 * \def SIF_SIMPLE_ERROR_UNDEFINED_ENDIAN
 *
 * @brief An error to indicate an endian code is invalid.
 */

#define SIF_SIMPLE_ERROR_UNDEFINED_ENDIAN 102

/**
 * \def SIF_SIMPLE_UINT8
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing unsigned 8-
bit
 * integers.
 */

#define SIF_SIMPLE_UINT8 0

/**
 * \def SIF_SIMPLE_INT8
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing signed 8-bi
t
 * integers.
 */

#define SIF_SIMPLE_INT8 1

/**
 * \def SIF_SIMPLE_UINT16
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing unsigned 16
-bit
 * integers.
 */

#define SIF_SIMPLE_UINT16 2
```

```
/**
 * \def SIF_SIMPLE_INT16
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing signed 16-b
it
 * integers.
 */

#define SIF_SIMPLE_INT16 3

/**
 * \def SIF_SIMPLE_UINT32
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing unsigned 32
-bit
 * integers.
 */

#define SIF_SIMPLE_UINT32 4

/**
 * \def SIF_SIMPLE_INT32
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing signed 32-b
it
 * integers.
 */

#define SIF_SIMPLE_INT32 5

/**
 * \def SIF_SIMPLE_UINT64
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing unsigned 64
-bit
 * integers.
 */

#define SIF_SIMPLE_UINT64 6

/**
 * \def SIF_SIMPLE_INT64
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing signed 64-b
it
 * integers.
 */

#define SIF_SIMPLE_INT64 7

/**
 * \def SIF_SIMPLE_FLOAT32
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing IEEE-754 st
andard
 * 32-bit floats.
 */
```

```
#define SIF_SIMPLE_FLOAT32 8

/**
 * \def SIF_SIMPLE_FLOAT64
 * \ingroup simpdecs
 *
 * @brief The base type code (i.e. <code>user_data_type mod 10</code>) for storing IEEE-754 st
andard
 * 64-bit floats.
 */

#define SIF_SIMPLE_FLOAT64 9

/**
 * \def SIF_SIMPLE_LITTLE_ENDIAN
 * \ingroup simpdecs
 *
 * @brief The endian code for little endian.
 */

#define SIF_SIMPLE_LITTLE_ENDIAN 0

/**
 * \def SIF_SIMPLE_BIG_ENDIAN
 * \ingroup simpdecs
 *
 * @brief The endian code for little endian.
 */

#define SIF_SIMPLE_BIG_ENDIAN 1

/**
 * \def SIF_SIMPLE_NATIVE_ENDIAN
 * \ingroup simpdecs
 *
 * @brief The endian code for the byte order of the native machine on which this library runs.
 */

#if defined(WIN32) || __BYTE_ORDER == __LITTLE_ENDIAN
#define SIF_SIMPLE_NATIVE_ENDIAN SIF_SIMPLE_LITTLE_ENDIAN
#elif __BYTE_ORDER == __BIG_ENDIAN
#define SIF_SIMPLE_NATIVE_ENDIAN SIF_SIMPLE_BIG_ENDIAN
#endif

/**
 * \def SIF_SIMPLE_ENDIAN(t)
 * \ingroup simpdecs
 *
 * @brief A function macro that returns the endian code for a simple type code \a t.
 */

#define SIF_SIMPLE_ENDIAN(t) (((int)t)/10)

/**
 * \def SIF_SIMPLE_TYPE_CODE(bt, ec)
 * \ingroup simpdecs
 *
 * @brief A function macro that computes the compound type code from the base type code \a bt
 * and endian code \a ec.
 */

#define SIF_SIMPLE_TYPE_CODE(bt, ec) ((bt)+(ec))
```

```
/**
 * \def SIF_SIMPLE_BASE_TYPE_CODE(bt, ec)
 * \ingroup simpdecs
 *
 * @brief A function macro that computes the base type code from the compound type code.
 */

#define SIF_SIMPLE_BASE_TYPE_CODE(x) (((int)x)%10)


/**
 * @brief Create a new Sparse Image Format (SIF) file with a given filename
 * and attributes. The file’s header and tile headers are written. No
 * space is preallocated for data blocks. The simple data-type convention
 * is used. When reading, writing, or filling the image file created by
 * this function, the <code>sif_simple_*</code> functions must be used
 * to ensure the image data elements are written with the proper byte
 * order.
 *
 * Unless \ref sif_simple_set_endian is called prior to reading or writing
 * any image raster, the pixels will be stored in native byte order.
 *
 * @param filename               The filename of the new file.
 * @param width                  The width of the image to store in the file to create.
 * @param height                 The height of the image to store in the file to create.
 * @param bands                  The number of bands of the image to store in the file to creat
e.
 * @param simple_data_type       The data type code.
 * @param tile_width             The width of a single tile.
 * @param tile_height            The height of a single tile.
 * @param consolidate_on_close   Defines whether a pixel uniformity check should be applied to
 *                               dirty tiles during each close.
 * @param defragment_on_close    Defines whether the file should be defragmented during each cl
ose.
 * @param intrinsic_write        Defines whether intrinsic uniformity checks should be performe
d
 *                               when rasters are written to a file.
 *
 * @return  A file structure containing the constructs needed to manipulate the file created
 *          by this function. This function returns NULL if an error occurs during creation.
 */

SIF_EXPORT sif_file*        sif_simple_create(const char *filename,
                                              long width, long height,
                                              long bands,
                                              int simple_data_type,
                                              int consolidate_on_close,
                                              int defragment_on_close,
                                              long tile_width, long tile_height,
                                              int intrinsic_write);

/**
 * @brief Create a new Sparse Image Format (SIF) file with a given filename
 * and attributes. The file’s header and tile headers are written. No
 * space is preallocated for data blocks. The simple data-type convention
 * is used.
 *
 * When reading, writing, or filling the image file created by
 * this function, the <code>sif_simple_*</code> functions must be used
 * to ensure the image data elements are written with the proper byte
 * order.
 *
 * Unless \ref sif_simple_set_endian is called prior to reading or writing
 * any image raster, the pixels will be stored in native byte order.
 *
 * The sif_header::consolidate, sif_header::defragment and the
 * sif_header::intrisic_write flags are all set to true. The sif_header::tile_width
 * and sif_header::tile_height are both set to 64.
 *
 * @param filename              The filename of the new file.
 * @param width                 The width of the image to store in the file to create.
 * @param height                The height of the image to store in the file to create.
 * @param bands                 The number of bands of the image to store in the file to creat
e.
 * @param simple_data_type      The data type code.
 *
 * @return  A file structure containing the constructs needed to manipulate the file created
 *          by this function. This function returns NULL if an error occurs during creation.
 */

SIF_EXPORT sif_file*        sif_simple_create_defaults(const char *filename,
                                                        long width, long height,
                                                        long bands,
                                                        int simple_data_type);

/**
 * @brief Set the network byte order for the pixel. Note that this field
 * should never be set once a raster is written to a file.
 * This field must be set to one of \ref SIF_SIMPLE_LITTLE_ENDIAN or
 * \ref SIF_SIMPLE_BIG_ENDIAN.
 *
 * @param  file       The file on which to perform the operation.
 * @param  endian     The endian code to set the file.
 */

SIF_EXPORT void               sif_simple_set_endian(sif_file *file, int endian);

/**
 * @brief Get the network byte order of the pixel values in the image
 * of this file.
 *
 * @param  file       The file on which to perform the operation.
 *
 * @return            The endian code of this file.
 */

SIF_EXPORT int               sif_simple_get_endian(sif_file *file);

/**
 * @brief Set the simple data type code for the pixel. Note that this field
 * should never be set once a raster is written to a file.
 *
 * @param  file       The file on which to perform the operation.
 * @param  code       The simple data type code of the pixel values.
 * @see simpdecs
 */

SIF_EXPORT void               sif_simple_set_data_type(sif_file *file, int code);

/**
 * @brief Get the simple data type code of the pixel values in the image
 * of this file.
 *
 * @param  file       The file on which to perform the operation.
 *
 * @return            The simple data type code of the pixel values.
 * @see simpdecs
 */

SIF_EXPORT int               sif_simple_get_data_type(sif_file *file);

/**
 * @brief Write a rectangular region to a file. The byte order of the
 * data values is automatically converted from host order to the the
 * byte order of the file.
 *
 * @param file The file on which to perform the operation.
 * @param data The buffer containing the data to write.
 * @param x    The horizontal starting index of the file.
 * @param y    The vertical starting index of the file.
 * @param w    The width of the region.
 * @param h    The height of the region.
 * @param band The band of the region.
 */

SIF_EXPORT void              sif_simple_set_raster(sif_file* file,
                                                      const void *data,
                                                      long x, long y,
                                                      long w, long h,
                                                      long band);

/**
 * @brief Read a rectangular region from a file. The byte order of
 * the data values in the buffer is automatically converted to the
 * byte order of the file.
 *
 * @param file The file on which to perform the operation.
 * @param data The buffer into which the data will be read.
 * @param x    The horizontal starting index of the file.
 * @param y    The vertical starting index of the file.
 * @param w    The width of the region.
 * @param h    The height of the region.
 * @param band The band of the region.
 */

SIF_EXPORT void              sif_simple_get_raster(sif_file* file,
                                                      void *data,
                                                      long x, long y,
                                                      long w, long h,
                                                      long band);
/**
 * @brief Fill a band with a constant value. The byte order of the
 * value is converted to the byte order of the file’s image.
 *
 * @param file   The file on which to perform the operation.
 * @param band   The band to fill.
 * @param value  The pointer to the value with which to fill the band.
 */

SIF_EXPORT void              sif_simple_fill_tiles(sif_file *file,
                                                      long band, const void *value);

/**
 * @brief Retrieve a tile slice. The byte order of the data values
 * in the buffer are in host byte order.
 *
 * @param file   The file on which to perform the operation.
 * @param buffer The buffer to store the slice.
 * @param tx     The horizontal index of the tile slice.
 * @param ty     The vertical index of the tile slice.
 * @param band   The band of the tile to which the slice corresponds.
 */

SIF_EXPORT void              sif_simple_get_tile_slice(sif_file *file, void *buffer,
                                                        long tx, long ty, long band);
/**
 * @brief Store a tile slice. The byte order of the data values
 * in the buffer are converted to the byte order of the file.
 *
 * @param file   The file on which to perform the operation.
 * @param buffer The buffer to store the slice.
 * @param tx     The horizontal index of the tile slice.
 * @param ty     The vertical index of the tile slice.
 * @param band   The band of the tile to which the slice corresponds.
 */

SIF_EXPORT void              sif_simple_set_tile_slice(sif_file *file,
                                                       const void *buffer,
                                                       long tx, long ty, long band);

/**
 * @brief Fill a tile slice with a constant value. The value is
 * converted from host order to the byte order in the file.
 *
 * @param file   The file on which to perform the operation.
 * @param tx     The horizontal index of the tile slice.
 * @param ty     The vertical index of the tile slice.
 * @param band   The band of the tile to which the slice corresponds.
 * @param value  The pointer to the value to fill the tile slice.
 */

SIF_EXPORT void              sif_simple_fill_tile_slice(sif_file *file,
                                                         long tx, long ty,
                                                         long band, const void *value);

/**
 * @brief Open a Sparse Image File (SIF) format file for reading or update.
 * The file is expected to use the Simple data type convention.
 *
 * When reading, writing, or filling the image file opened by
 * this function, the <code>sif_simple_*</code> functions must be used
 * to ensure the image data elements are written with the proper byte
 * order.
 *
 * If the file does not use the simple data-type convention, 0 is returned.
 *
 * @param filename  The filename of the SIF file to open.
 * @param read_only A flag indicating whether to open as read-only (1)
 *                  or update (0).
 *
 * @return A file structure containing all the constructs needed to
 *         manipulate the opened SIF file is returned. NULL is returned
 *         if an error occured during open.
 */

SIF_EXPORT sif_file*        sif_simple_open(const char* filename, int read_only);

/**
 * @brief Determine if the tiles comprising a region are shallow uniform. The
 * "simple" data-type convention is assumed.
 *
 * @param file          The file to perform the check.
 * @param x             The starting horizontal pixel offset (0..N-1 indexed)
 *                      of the region to check.
 * @param y             The starting vertical pixel offset (0..N-1 indexed)
 *                      of the region to check.
 * @param w             The width of the region.to check.
 * @param h             The height of the region to check.
 * @param band          The band offset (0..N-1 indexed).
 * @param uniform_value This value is only meaningful when this function
 *                      returns true (non-zero). It is expected that the
 *                      pointer passed point to at least data_unit_size bytes.
 *                      When the region is completely uniform, the uniform
 *                      pixel value is stored here.
 *
 * @return 0 if the tiles are not shallow uniform or a memory allocation error
 *          occurred, otherwise a non-zero value.
 */

SIF_EXPORT int              sif_simple_is_shallow_uniform(sif_file *file, long x, long y, long
 w, long h, long band, void *uniform_value);

/**
 * @brief Determine if a tile slice has shallow uniformity. The simple user data
 * type convention is assumed.
 *
 * @param file          The file to perform the check.
 * @param tx            The horizontal tile index (0..N-1 indexed)
 *                      of the region to check.
 * @param ty            The vertical tile index (0..N-1 indexed)
 *                      of the region to check.
 * @param band          The band offset (0..N-1 indexed).
 * @param uniform_value This value is only meaningful when this function
 *                      returns true (non-zero). It is expected that the
 *                      pointer passed point to at least data_unit_size bytes.
 *                      When the region is completely uniform, the uniform
 *                      pixel value is stored here.
 *
 * @return 0 if the tiles are non-uniform or a memory allocation error
 *          occurred, otherwise a non-zero value.
 */

SIF_EXPORT int              sif_simple_is_slice_shallow_uniform(sif_file *file, long tx, long
ty, long band, void *uniform_value);

/**
 * Return if the file conforms to the "simple" data type convention.
 *
 * @param file              The file on which to perform the operation.
 *
 * @return A positive value if the file conforms to the "simple" data type convention.
 */

SIF_EXPORT int              sif_is_simple(sif_file *file);

/**
 * Return a positive number if the file referred to by the filename conforms to the "simple"
 * data type convention.
 *
 * @param filename          The filename of the file on which to perform the operation.
 *
 * @return A positive value if the file referred to by the filename conforms to the
 *          "simple" data type convention. If the file does not conform to the "simple"
 *         data type convention, -2 is returned. If the file could not be opened, -1 is return
```

```
ed.
 *         If the file is not a SIF file, 0 is returned.
 */

SIF_EXPORT int              sif_is_simple_by_name(const char *filename);

/**
 * Exports a region of an open SIF file to PGM (Portable Grayscale Map) format. The file must
 * conform to the "simple" data type convention. uint8 and uint16 are the only
 * supported "simple" data types for PGM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PGM output.
 * @param   x          The x pixel coordinate of the region to write (0..N-1 indexed).
 * @param   y          The y pixel coordinate of the region to write (0..N-1 indexed).
 * @param   width      The width of the region to write.
 * @param   height     The height of the region to write.
 * @param   band       The band of the region to write.
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_region_to_pgm_file(sif_file *file, const char *filename
,
                                                              int x, int y, int width, int height,
 int band);

/**
 * Exports a slice of a tile in an image of an open SIF file to PGM (Portable Grayscale Map) f
ormat.
 * The file must conform to the "simple" data type convention. uint8 and uint16 are the only
 * supported "simple" data types for PGM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PGM output.
 * @param   tx         The x tile coordinate of the slice to write (0..N-1 indexed).
 * @param   ty         The y tile coordinate of the slice to write (0..N-1 indexed).
 * @param   band       The band number of the slice to write (0..N-1 indexed).
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_slice_to_pgm_file(sif_file *file, const char *filename,
                                                             int tx, int ty, int band);

/**
 * Exports three bands of a region of an open SIF file to PPM (Portable Pixel Map) format. The
 file
 * must conform to the "simple" data type convention. The image in the file must contain 3 ban
ds.
 * uint8 and uint16 are the only supported "simple" data types for PPM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PGM output.
 * @param   x          The x pixel coordinate of the region to write (0..N-1 indexed).
 * @param   y          The y pixel coordinate of the region to write (0..N-1 indexed).
 * @param   width      The width of the region to write.
 * @param   height     The height of the region to write.
 * @param   band0      The band number of the SIF image corresponding to the red band in the P
PM file.
 * @param   band1      The band number of the SIF image corresponding to the green band in the
 PPM file.
 * @param   band2      The band number of the SIF image corresponding to the blue band in the
```

```
PPM file.
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_region_to_ppm_file(sif_file *file, const char *filename
,
                                                          int x, int y, int width, int height,
                                                          int band0, int band1, int band2);

/**
 * Exports three slices of a tile of an open SIF file to PPM (Portable Pixel Map) format. The
file
 * must conform to the "simple" data type convention. The image in the file must contain at le
ast 3 bands.
 * uint8 and uint16 are the only supported "simple" data types for PPM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PGM output.
 * @param   tx         The x tile coordinate of the slice to write (0..N-1 indexed).
 * @param   ty         The y tile coordinate of the slice to write (0..N-1 indexed).
 * @param   band0      The band number of the tile slice corresponding to the red band in the
PPM file.
 * @param   band1      The band number of the tile slice corresponding to the green band in th
e PPM file.
 * @param   band2      The band number of the tile slice corresponding to the blue band in the
 PPM file.
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_slices_to_ppm_file(sif_file *file, const char *filename
,
                                                          int tx, int ty,
                                                          int band0, int band1, int band2);

/**
 * Exports three bands of a region of an open SIF file to PAM (Portable Any Map) format. The f
ile
 * must conform to the "simple" data type convention.
 * uint8 and uint16 are the only supported "simple" data types for PAM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PAM output.
 * @param   x          The x pixel coordinate of the region to write (0..N-1 indexed).
 * @param   y          The y pixel coordinate of the region to write (0..N-1 indexed).
 * @param   width      The width of the region to write.
 * @param   height     The height of the region to write.
 * @param   bands      The band numbers of the SIF image corresponding to the bands in the PAM
 file,
 *                     in ascending order.
 * @param   nbands     The total number of bands to write.
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_region_to_ppm_file(sif_file *file, const char *filename
,
                                                          int x, int y, int width, int height,
                                                          int *bands, int nbands);

/**
 * Exports three slices of a tile of an open SIF file to PAM (Portable Any Map) format. The fi
```

```
le
 * must conform to the "simple" data type convention. The image in the file must contain at le
ast 3 bands.
 * uint8 and uint16 are the only supported "simple" data types for PAM output.
 *
 * @param   file       The SIF file to export.
 * @param   filename   The filename to write the PAM output.
 * @param   tx         The x tile coordinate of the slice to write (0..N-1 indexed).
 * @param   ty         The y tile coordinate of the slice to write (0..N-1 indexed).
 * @param   bands      The band numbers of the slices of the tile in the SIF image correspondi
ng
 *                     to the bands in the PAM file, in ascending order.
 * @param   nbands     The total number of bands to write.
 *
 * @return A nonzero value if successful.
 */

SIF_EXPORT int              sif_export_slices_to_ppm_file(sif_file *file, const char *filename
,
                                                              int tx, int ty,
                                                              int *bands, int nbands);

#endif
```