FreeImage

a free, open source graphics library

Documentation
Library version 3.17.0

Contents

Introduction 1

Foreword 1
Purpose of FreeImage 1
Library reference 2

Bitmap function reference 3

General functions 3
Bitmap management functions 5
Bitmap information functions 15
Filetype functions 24
Pixel access functions 26
Conversion functions 32
Tone mapping operators 43
ICC profile functions 46
Plugin functions 48
Multipage functions 56
Memory I/O streams 62
Compression functions 71
Helper functions 74

Metadata function reference 75

Introduction 75
Tag creation and destruction 78
Tag accessors 79
Metadata iterator 82
Metadata accessors 83
Metadata helper functions 85

Toolkit function reference 88

Rotation and flipping 88
Upsampling / downsampling 91
Color manipulation 95
Channel processing 101
Copy / Paste / Composite routines 102
JPEG lossless transformations 107
Background filling 111
Miscellaneous algorithms 116

Appendix 117

Supported file formats 117
Supported camera RAW file formats 118
Choosing the right resampling filter 119
Using the rotation functions 123
FreeImage metadata models 126
FIMD_ANIMATION metadata model specification 130

List of tables

Table 1: FREE_IMAGE_FORMATS constants (FreeImage format identifiers). 6

Table 2: FREE_IMAGE_TYPE constants (FreeImage data type identifiers). 7

Table 3: Optionnal decoder constants. 9

Table 4: Optionnal encoder constants. 13

Table 5: FREE_IMAGE_COLOR_TYPE constants. 18

Table 6: Pixel access macros and associated masks for 24- or 32-bit images. 27

Table 7: FREE_IMAGE_QUANTIZE constants. 33

Table 8: FREE_IMAGE_DITHER constants. 36

Table 9: Bitmap type conversions allowed by FreeImage. 40

Table 10: FREE_IMAGE_TMO constants. 43

Table 11: FreeImage FITAG structure. 75

Table 12: FreeImage tag data types (FREE_IMAGE_MDTYPE identifier). 76

Table 13: Metadata models supported by FreeImage. 77

Table 14: IMAGE_FILTER constants. 91

Table 15: FREE_IMAGE_COLOR_CHANNEL constants. 95

Table 16: FREE_IMAGE_JPEG_OPERATION constants. 107

Table 17: Background filling options for 24- or 32-bit images 111

Table 18: Background filling options for palletized images 112

Table 19: List of tag keys supported by the IPTC metadata model. 128

List of figures

Figure 1: Illustration of the FreeImage_Composite function. 104

Figure 2: Comparison of resampling filters on a 32x32 Lena image resized to 400%. 120

Figure 3: Comparison of resampling filters on a 40x46 Bikini image resized to 800%. 122

Figure 4: Parrot image rotated by 45° using FreeImage_Rotate. 123

Figure 5: Parrot image rotated by 90° using FreeImage_Rotate. 123

Figure 6: Some examples illustrating the use of FreeImage_RotateEx. 124

Introduction

Foreword

Thank you for downloading FreeImage, a free and open source graphics library for Windows, Linux and Mac OS X. FreeImage is widely used and praised for its speed and simplicity. It has been under development for more than 15 years.

FreeImage was created by Floris van den Berg. It was originally developed to provide bitmap loading support to an authoring tool named the Magenta Multimedia Tool. The major parts of the library were designed by Floris, but in its long lifetime, many people have contributed to FreeImage, adding new features and helping to test the library. Without the help of these people, FreeImage wouldn't have been where it is now. Anyone can contribute and post their changes and improvements and have them inserted in the main sources (of course on the condition that developers agree on that the fixes are good). The list of contributors in the changelog file is only a small part of all the people that every day provide us with bug reports, suggestions, ideas and source code.

In the middle of the year 2002, Floris stopped the development of the library. Since this date, the FreeImage Project continues to be developed and is maintained by Hervé Drolon.

Purpose of FreeImage

A clear picture about a project is important, because it is that picture that defines which features are implemented and which are not.

FreeImage supports:

- Loading and saving of as many bitmap types as possible
- Easy access to bitmap components, such as palettes and data bits
- Converting bitmap's bit depths from one to another
- Accessing pages in a bitmap when there are multiple, such as in TIFF
- Basic manipulation of bitmaps, such as rotation, flipping and resampling or point operations such as brightness and contrast adjustment
- Alpha compositing and alpha blending

FreeImage does not support:

- Advanced image processing operations such as convolution and transforms
- Bitmap drawing
- Vector graphics

Library reference

Each function name in FreeImage starts with "FreeImage_", for instance FreeImage_Load, FreeImage_Save, FreeImage_Unload ...

A detailed description of each function supported by the FreeImage library is given in the Bitmap function reference, Metadata function reference and Toolkit function reference chapters. For each entry, the function prototype is shown for C/C++ and the function arguments and explanations are listed.

Throughout these chapters, you will see numbers in colored boxes at the top of some functions. These numbers indicate the pixel depth of the input image that the function can operate on.

This may be:

1-, 4-, 8-, 16-, 24-, 32-bit per pixel for **standard bitmap** (green boxes),

16-, 48-, 64-bit per pixel for **UINT16, RGB16 and RGBA16 image types** (blue boxes),

32-, 96-, 128-bit per pixel for **FLOAT, RGBF and RGBAF image types** (orange boxes),

16-, 32-, 64-, 2x64-bit for **other special image types** (yellow boxes).

If boxed numbers are not displayed the function operation is independent of the image pixel depth (e.g. for load / save and plugins functions).

Bitmap function reference

General functions

The following functions don't have anything to do with the bitmap support provided by FreeImage. They are internal library management functions. That doesn't mean they are not important. Without them you won't be able to load any bitmap at all.

FreeImage_Initialise

```
DLL_API void DLL_CALLCONV FreeImage_Initialise(BOOL load_local_plugins_only
FI_DEFAULT(FALSE));
```

Initialises the library. When the *load_local_plugins_only* parameter is TRUE, FreeImage won't make use of external plugins.

When using the FreeImage DLL, this function is called **automatically** with the load_local_plugins_only parameter set to FALSE. When using FreeImage as a static linked library, you must call this function **exactly once** at the start of your program.

FreeImage_DeInitialise

```
DLL_API void DLL_CALLCONV FreeImage_DeInitialise();
```

Deinitialises the library.

When using the FreeImage DLL, this function is called **automatically**. When using FreeImage as a static linked library, you must call this function **exactly once** at the end of your program to clean up allocated resources in the FreeImage library.

Under Linux or under any *nix OS (i.e. under Unix or MacOSX), you need to call FreeImage_Initialise at the beginning of your main function and you need to call FreeImage_DeInitialise at the end of this main function (this is not needed when using FreeImage as a .SO).

FreeImage_GetVersion

```
DLL_API const char *DLL_CALLCONV FreeImage_GetVersion();
```

Returns a string containing the current version of the library.

FreeImage_GetCopyrightMessage

```
DLL_API const char *DLL_CALLCONV FreeImage_GetCopyrightMessage();
```

Returns a string containing a standard copyright message you can show in your program.

FreeImage_SetOutputMessage

```
DLL_API void DLL_CALLCONV FreeImage_SetOutputMessage(FreeImage_OutputMessageFunction
omf);
```

When a certain bitmap cannot be loaded or saved there is usually an explanation for it. For example a certain bitmap format might not be supported due to patent restrictions, or there might be a known issue with a certain bitmap subtype. Whenever something fails in FreeImage internally a log-string is generated, which can be captured by an application driving FreeImage. You use the function FreeImage_SetOutputMessage to capture the log string so that you can show it to the user of the program.

```
/**
FreeImage error handler
@param fif Format / Plugin responsible for the error
@param message Error message
*/
void FreeImageErrorHandler(FREE_IMAGE_FORMAT fif, const char *message) {
  printf("\n*** ");
  if(fif != FIF_UNKNOWN) {
    printf("%s Format\n", FreeImage_GetFormatFromFIF(fif));
  }
  printf(message);
  printf(" ***\n");
}

// In your main program …

FreeImage_SetOutputMessage(FreeImageErrorHandler);
```

The *fif* parameter passed in the callback first argument may be equal to *FIF_UNKNOWN* when an error that is not related to a plugin is generated. In this case, calling *FreeImage_GetFormatFromFIF(FIF_UNKNOWN)* will return NULL.
Giving a NULL value to functions such as "*printf(…)*" may crash your application so just be careful …

Bitmap management functions

The bitmap management functions in FreeImage are definitely the most used ones. They allow you to allocate new bitmaps, import bitmaps so that they can be edited in memory and export bitmaps to disc. As you will see, the FreeImage bitmap management functions are very easy to use.

Although FreeImage can handle more than 30 bitmap types, there are only 4 bitmap handling functions. A special parameter, an enum named FREE_IMAGE_FORMAT, is used to specify the bitmap format that will be loaded or saved. This enum is defined in the header file FREEIMAGE.H. The following FREE_IMAGE_FORMATS constants are currently available:

FIF	Description
FIF_UNKNOWN	Unknown format (**returned value only, never use it as input value**)
FIF_BMP	Windows or OS/2 Bitmap File (*.BMP)
FIF_CUT	Dr. Halo (*.CUT)
FIF_DDS	DirectDraw Surface (*.DDS)
FIF_EXR	ILM OpenEXR (*.EXR)
FIF_FAXG3	Raw Fax format CCITT G3 (*.G3)
FIF_GIF	Graphics Interchange Format (*.GIF)
FIF_HDR	High Dynamic Range (*.HDR)
FIF_ICO	Windows Icon (*.ICO)
FIF_IFF	Amiga IFF (*.IFF, *.LBM)
FIF_J2K	JPEG-2000 codestream (*.J2K, *.J2C)
FIF_JNG	JPEG Network Graphics (*.JNG)
FIF_JP2	JPEG-2000 File Format (*.JP2)
FIF_JPEG	Independent JPEG Group (*.JPG, *.JIF, *.JPEG, *.JPE)
FIF_JXR	JPEG XR image format (*.JXR, *.WDP, *.HDP)
FIF_KOALA	Commodore 64 Koala format (*.KOA)
FIF_MNG	Multiple Network Graphics (*.MNG)
FIF_PBM	Portable Bitmap (ASCII) (*.PBM)
FIF_PBMRAW	Portable Bitmap (BINARY) (*.PBM)
FIF_PCD	Kodak PhotoCD (*.PCD)
FIF_PCX	Zsoft Paintbrush PCX bitmap format (*.PCX)
FIF_PFM	Portable Floatmap (*.PFM)
FIF_PGM	Portable Graymap (ASCII) (*.PGM)
FIF_PGMRAW	Portable Graymap (BINARY) (*.PGM)
FIF_PICT	Macintosh PICT (*.PCT, *.PICT, *.PIC)
FIF_PNG	Portable Network Graphics (*.PNG)
FIF_PPM	Portable Pixelmap (ASCII) (*.PPM)
FIF_PPMRAW	Portable Pixelmap (BINARY) (*.PPM)
FIF_PSD	Adobe Photoshop (*.PSD)
FIF_RAS	Sun Rasterfile (*.RAS)
FIF_RAW	RAW camera image (*many extensions*)
FIF_SGI	Silicon Graphics SGI image format (*.SGI)
FIF_TARGA	Truevision Targa files (*.TGA, *.TARGA)
FIF_TIFF	Tagged Image File Format (*.TIF, *.TIFF)
FIF_WBMP	Wireless Bitmap (*.WBMP)
FIF_WEBP	Google WebP image format (*.WEBP)
FIF_XBM	X11 Bitmap Format (*.XBM)

FIF	Description
FIF_XPM	X11 Pixmap Format (*.XPM)

Table 1: FREE_IMAGE_FORMATS constants (FreeImage format identifiers).

As an extension to the FREE_IMAGE_FORMATs, you can register your own bitmap formats. Registering bitmaps can be done manually, by calling one of the plugin management functions (see Plugin functions), or automatically by copying a precompiled FreeImage bitmap plugin DLL into the same directory where FREEIMAGE.DLL is residing. When a new bitmap type is registered it is assigned a new, unique plugin identification number that you can pass to the same place that you would pass a FREE_IMAGE_FORMAT.

FreeImage_Allocate

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Allocate(int width, int height, int bpp,
unsigned red_mask FI_DEFAULT(0), unsigned green_mask FI_DEFAULT(0), unsigned blue_mask
FI_DEFAULT(0));
```

If you want to create a new bitmap in memory from scratch, without loading a pre-made bitmap from disc, you use this function. FreeImage_Allocate takes a width and height parameter, and a bpp parameter to specify the bit depth of the image and returns a FIBITMAP. The optional last three parameters (red_mask, green_mask and blue_mask) are used to tell FreeImage the bit-layout of the color components in the bitmap, e.g. where in a pixel the red, green and blue components are stored. To give you an idea about how to interpret the color masks: when red_mask is 0xFF000000 this means that the last 8 bits in one pixel are used for the color red. When green_mask is 0x000000FF, it means that the first 8 bits in a pixel are used for the color green.

FreeImage_Allocate allocates an *empty* bitmap, i.e. a bitmap that is filled completely with zeroes. Zero in a bitmap is usually interpreted as black. This means that if your bitmap is palletised it will contain a completely black palette. You can access, and hence populate the palette by using the function FreeImage_GetPalette.
For 8-bit images only, FreeImage_Allocate will build a default greyscale palette.

```
FIBITMAP *bitmap = FreeImage_Allocate(320, 240, 32);

if (bitmap) {
    // bitmap successfully created!

    FreeImage_Unload(bitmap);
}
```

FreeImage_Allocate is an alias for FreeImage_AllocateT and can be replaced by this call:
FreeImage_AllocateT(FIT_BITMAP, width, height, bpp, red_mask, green_mask, blue_mask);

FreeImage_AllocateT

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_AllocateT(FREE_IMAGE_TYPE type, int width,
int height, int bpp FI_DEFAULT(8), unsigned red_mask FI_DEFAULT(0), unsigned
green_mask FI_DEFAULT(0), unsigned blue_mask FI_DEFAULT(0));
```

While most imaging applications only deal with photographic images, many scientific applications need to deal with high resolution images (e.g. 16-bit greyscale images), with real

valued pixels or even with complex pixels (think for example about the result of a Fast Fourier Transform applied to a 8-bit greyscale image: the result is a complex image).

A special parameter, an enum named FREE_IMAGE_TYPE, is used to specify the bitmap type of a FIBITMAP. This enum is defined in the header file FREEIMAGE.H. The following FREE_IMAGE_TYPE constants are currently available:

FIT	Description
FIT_UNKNOWN	Unknown format (**returned value only, never use it as input value**)
FIT_BITMAP	Standard image: 1-, 4-, 8-, 16-, 24-, 32-bit
FIT_UINT16	Array of unsigned short: unsigned 16-bit
FIT_INT16	Array of short: signed 16-bit
FIT_UINT32	Array of unsigned long: unsigned 32-bit
FIT_INT32	Array of long: signed 32-bit
FIT_FLOAT	Array of float: 32-bit IEEE floating point
FIT_DOUBLE	Array of double: 64-bit IEEE floating point
FIT_COMPLEX	Array of FICOMPLEX: 2 x 64-bit IEEE floating point
FIT_RGB16	48-bit RGB image: 3 x unsigned 16-bit
FIT_RGBA16	64-bit RGBA image: 4 x unsigned 16-bit
FIT_RGBF	96-bit RGB float image: 3 x 32-bit IEEE floating point
FIT_RGBAF	128-bit RGBA float image: 4 x 32-bit IEEE floating point

Table 2: FREE_IMAGE_TYPE constants (FreeImage data type identifiers).

When you need to know the data type of a bitmap, you can use the FreeImage_GetImageType function.

```
FIBITMAP *bitmap = FreeImage_AllocateT(FIT_RGB16, 512, 512);

if (bitmap) {
    // bitmap successfully created!

    FreeImage_Unload(bitmap);
}
```

FreeImage_Load

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Load(FREE_IMAGE_FORMAT fif, const char
*filename, int flags FI_DEFAULT(0));
```

This function decodes a bitmap, allocates memory for it and then returns it as a FIBITMAP. The first parameter defines the type of bitmap to be loaded. For example, when FIF_BMP is passed, a BMP file is loaded into memory (an overview of possible FREE_IMAGE_FORMAT constants is available in Table 1). The second parameter tells FreeImage the file it has to decode. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

```
FIBITMAP *bitmap = FreeImage_Load(FIF_BMP, "mybitmap.bmp", BMP_DEFAULT);

if (bitmap) {
    // bitmap successfully loaded!

    FreeImage_Unload(bitmap);
}
```

Some bitmap loaders can receive parameters to change the loading behaviour. When the parameter is not available or unused you can pass the value 0 or <TYPE_OF_BITMAP>_DEFAULT (e.g. BMP_DEFAULT, ICO_DEFAULT, etc).

Bitmap type	Flag	Description
<any type>	FIF_LOAD_NOPIXELS	When this flag is supported by a plugin, load only header data and possibly metadata (including embedded thumbnail)[1] When the flag is not supported, pixels are loaded.
FIF_GIF	GIF_DEFAULT	
	GIF_LOAD256	Load the image as a 256 color image with unused palette entries, if it's 16 or 2 color
	GIF_PLAYBACK	'Play' the GIF to generate each frame (as 32bpp) instead of returning raw frame data when loading
FIF_ICO	ICO_MAKEALPHA	Convert to 32-bit and create an alpha channel from the AND-mask when loading
FIF_JPEG	JPEG_DEFAULT	Load the file as fast as possible, sacrificing some quality
	JPEG_FAST	Load the file as fast as possible, sacrificing some quality
	JPEG_ACCURATE	Load the file with the best quality, sacrificing some speed
	JPEG_CMYK	This flag will load CMYK bitmaps as 32-bit separated CMYK (use | to combine with other load flags)
	JPEG_GREYSCALE	Load and convert to a 8-bit greyscale image (faster than loading as 24-bit and converting to 8-bit)
	Integer X such that flags = flags | (X << 16)	Load and resize the file such that size/X = max(width, height)/X will return an image scaled by 2, 4 or 8 (i.e. the most appropriate requested size).[2]
	JPEG_EXIFROTATE	Load and rotate according to Exif 'Orientation' tag if available
FIF_JXR	JXR_DEFAULT	
FIF_PCD	PCD_DEFAULT	A PhotoCD picture comes in many sizes. This flag will load the one sized 768 x 512
	PCD_BASE	This flag will load the one sized 768 x 512
	PCD_BASEDIV4	This flag will load the bitmap sized 384 x 256
	PCD_BASEDIV16	This flag will load the bitmap sized 192 x 128
FIF_PNG	PNG_IGNOREGAMMA	Avoid gamma correction on loading
FIF_PSD	PSD_DEFAULT	Load and convert to RGB[A]
	PSD_CMYK	Reads tags for separated CMYK (default is conversion to RGB)
	PSD_LAB	Reads tags for CIELab (default is conversion to RGB)
FIF_RAW	RAW_DEFAULT	Load the file as linear RGB 48-bit
	RAW_PREVIEW	Try to load the embedded JPEG preview with included Exif data or default to RGB 24-bit
	RAW_DISPLAY	Load the file as RGB 24-bit
	RAW_HALFSIZE	Output a half-size color image
	RAW_UNPROCESSED	Output a FIT_UINT16 raw Bayer image[3]
FIF_TARGA	TARGA_LOAD_RGB888	If set the loader converts RGB555 and ARGB8888 -> RGB888
FIF_TIFF	TIFF_CMYK	This flag will load CMYK bitmaps as separated CMYK (default is conversion to RGB)
FIF_WEBP	WEBP_DEFAULT	

Table 3: Optionnal decoder constants.

[1] See the FreeImage_HasPixels sample code for a sample use.
[2] See the FreeImage_MakeThumbnail sample code for a sample use.
[3] See the FIMD_COMMENTS metatata model for more information.

Instead of hardcoding the FREE_IMAGE_FORMAT when calling FreeImage_Load, it is advised to use one of the FreeImage Filetype functions such as FreeImage_GetFileType in order to write a generic code, independent of possible future API changes.

FreeImage_LoadU

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_LoadU(FREE_IMAGE_FORMAT fif, const wchar_t
*filename, int flags FI_DEFAULT(0));
```

This function works exactly like FreeImage_Load but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns NULL.

FreeImage_LoadFromHandle

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_LoadFromHandle(FREE_IMAGE_FORMAT fif,
FreeImageIO *io, fi_handle handle, int flags FI_DEFAULT(0));
```

FreeImage has the unique feature to load a bitmap from an arbitrary source. This source might for example be a cabinet file, a zip file or an Internet stream. Handling of these arbitrary sources is not directly handled in the FREEIMAGE.DLL, but can be easily added by using a FreeImageIO structure as defined in FREEIMAGE.H.

FreeImageIO is a structure that contains 4 function pointers: one to *read* from a source, one to *write* to a source, one to *seek* in the source and one to *tell* where in the source we currently are. When you populate the FreeImageIO structure with pointers to functions and pass that structure to FreeImage_LoadFromHandle, FreeImage will call *your* functions to read, seek and tell in a file. The handle-parameter (third parameter from the left) is used in this to differentiate between different contexts, e.g. different files or different Internet streams.

The function pointers in FreeImageIO use the stdcall calling convention. This means that the functions pointed to must also use the stdcall calling convention. The calling convention was chosen to be compatible with programming language other than C++, such as Visual Basic.

```
FreeImageIO io;
io.read_proc = ReadProc; // pointer to function that calls fread
io.write_proc = NULL;    // not needed for loading
io.seek_proc = SeekProc; // pointer to function that calls fseek
io.tell_proc = TellProc; // pointer to function that calls ftell

FILE *f = fopen("mybitmap.bmp", "rb");

FIBITMAP *bitmap = FreeImage_LoadFromHandle(FIF_BMP, &io, (fi_handle)f, 0);

fclose(f);

if (bitmap) {
    // bitmap successfully loaded!

    FreeImage_Unload(bitmap);
}
```

FreeImage_Save

```
DLL_API BOOL DLL_CALLCONV FreeImage_Save(FREE_IMAGE_FORMAT fif, FIBITMAP *dib, const
char *filename, int flags FI_DEFAULT(0));
```

This function saves a previously loaded FIBITMAP to a file. The first parameter defines the type of the bitmap to be saved. For example, when FIF_BMP is passed, a BMP file is saved (an overview of possible FREE_IMAGE_FORMAT constants is available in Table 1). The second parameter is the name of the bitmap to be saved. If the file already exists it is overwritten. Note that some bitmap save plugins have restrictions on the bitmap types they can save. For example, the JPEG plugin can only save 24 bit and 8 bit greyscale bitmaps*. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

* In the FreeImage JPEG plugin, 8 bit palletised bitmaps are transparently converted to 24 bit when saving.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

if (FreeImage_Save(FIF_BMP, bitmap, "mybitmap.bmp", 0)) {
    // bitmap successfully saved!
}
```

Some bitmap savers can receive parameters to change the saving behaviour. When the parameter is not available or unused you can pass the value 0 or <TYPE_OF_BITMAP>_DEFAULT (e.g. BMP_DEFAULT, ICO_DEFAULT, etc).

Bitmap type	Flag	Description
FIF_BMP	BMP_DEFAULT	Save without any compression
	BMP_SAVE_RLE	Compress the bitmap using RLE when saving
FIF_EXR	EXR_DEFAULT	Save data as half with piz-based wavelet compression
	EXR_FLOAT	Save data as float instead of as half (not recommended)
	EXR_NONE	Save with no compression
	EXR_ZIP	Save with zlib compression, in blocks of 16 scan lines
	EXR_PIZ	Save with piz-based wavelet compression
	EXR_PXR24	Save with lossy 24-bit float compression
	EXR_B44	Save with lossy 44% float compression - goes to 22% when combined with EXR_LC
	EXR_LC	Save images with one luminance and two chroma channels, rather than as RGB (lossy compression)
FIF_J2K	J2K_DEFAULT	Save with a 16:1 rate
	Integer X in [1..512]	Save with a X:1 rate

Bitmap type	Flag	Description
FIF_JP2	JP2_DEFAULT	Save with a 16:1 rate
	Integer X in [1..512]	Save with a X:1 rate
FIF_JPEG	JPEG_DEFAULT	Saves with good quality (75:1)
	JPEG_QUALITYSUPERB	Saves with superb quality (100:1)
	JPEG_QUALITYGOOD	Saves with good quality (75:1)
	JPEG_QUALITYNORMAL	Saves with normal quality (50:1)
	JPEG_QUALITYAVERAGE	Saves with average quality (25:1)
	JPEG_QUALITYBAD	Saves with bad quality (10:1)
	Integer X in [0..100]	Save with quality X:1
	JPEG_PROGRESSIVE	Saves as a progressive JPEG file (use | to combine with JPEG quality flags)
	JPEG_SUBSAMPLING_411	Save with high 4x1 chroma subsampling (4:1:1)
	JPEG_SUBSAMPLING_420	Save with medium 2x2 chroma subsampling (4:2:0) - default value
	JPEG_SUBSAMPLING_422	Save with low 2x1 chroma subsampling (4:2:2)
	JPEG_SUBSAMPLING_444	Save with no chroma subsampling (4:4:4)
	JPEG_OPTIMIZE	On saving, compute optimal Huffman coding tables (can reduce a few percent of file size)
	JPEG_BASELINE	Save basic JPEG, without metadata or any markers (use | to combine with other JPEG flags)
FIF_JXR	JXR_DEFAULT	Save with quality 80 and no chroma subsampling (4:4:4) - quality is between [1..100), 100 means lossless compression
	Integer X in [1..100)	Save with quality X:1, chroma subsampling is automatically adjusted according to the quality. Using X=100 means lossless.
	JXR_LOSSLESS	Save lossless (quality = 100)
	JXR_PROGRESSIVE	Saves as a progressive JPEG-XR file (use | to combine with JPEG_XR quality flags)
FIF_PNG	PNG_DEFAULT	Save with ZLib level 6 compression and no interlacing
	PNG_Z_BEST_SPEED	Save using ZLib level 1 compression (default value is 6)
	PNG_Z_DEFAULT_COMPRESSION	Save using ZLib level 6 compression (default recommended value)
	PNG_Z_BEST_COMPRESSION	Save using ZLib level 9 compression (default value is 6)
	PNG_Z_NO_COMPRESSION	Save without ZLib compression
	PNG_INTERLACED	Save using Adam7 interlacing (use | to combine with other save flags)
FIF_PBM, FIF_PGM, FIF_PPM	PNM_DEFAULT	Saves the bitmap as a binary file
	PNM_SAVE_RAW	Saves the bitmap as a binary file
	PNM_SAVE_ASCII	Saves the bitmap as an ASCII file
FIF_TIFF	TIFF_DEFAULT	Save using CCITTFAX4 compression for 1-bit bitmaps and LZW compression for any other bitmaps
	TIFF_CMYK	Stores tags for separated CMYK (use | to combine with TIFF compression flags)
	TIFF_PACKBITS	Save using PACKBITS compression.
	TIFF_DEFLATE	Save using DEFLATE compression (also known as ZLIB compression)
	TIFF_ADOBE_DEFLATE	Save using ADOBE DEFLATE compression
	TIFF_NONE	Save without any compression
	TIFF_CCITTFAX3	Save using CCITT Group 3 fax encoding
	TIFF_CCITTFAX4	Save using CCITT Group 4 fax encoding

Bitmap type	Flag	Description
	TIFF_LZW	Save using LZW compression
	TIFF_JPEG	Save using JPEG compression (8-bit greyscale and 24-bit only. Default to LZW for other bitdepths).
	TIFF_LOGLUV	Save using LogLuv compression (only available with RGBF images) – default to LZW compression.
FIF_TARGA	TARGA_DEFAULT	Save without compression
	TARGA_SAVE_RLE	Save with RLE compression
FIF_WEBP	WEBP_DEFAULT	Save with good quality (75:1)
	Integer X in [1..100]	Save with quality X:1, where X is between 0 (smallest file) and 100 (biggest file).
	WEBP_LOSSLESS	Save in lossless mode

Table 4: Optionnal encoder constants.

FreeImage_SaveU

```
DLL_API BOOL DLL_CALLCONV FreeImage_SaveU(FREE_IMAGE_FORMAT fif, FIBITMAP *dib, const
wchar_t *filename, int flags FI_DEFAULT(0));
```

This function works exactly like FreeImage_Save but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FALSE.

FreeImage_SaveToHandle

```
DLL_API BOOL DLL_CALLCONV FreeImage_SaveToHandle(FREE_IMAGE_FORMAT fif, FIBITMAP *dib,
FreeImageIO *io, fi_handle handle, int flags FI_DEFAULT(0));
```

The FreeImageIO structure described earlier to load a bitmap from an arbitrary source can also be used to save bitmaps. Once again, FreeImage does not implement the way the bitmap is saved but lets you implement the desired functionality by populating a FreeImageIO structure with pointers to functions. FreeImage will now call *your* functions to write, seek and tell in a stream.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

FreeImageIO io;
io.read_proc = NULL;       // usually not needed for saving
io.write_proc = WriteProc; // pointer to function that calls fwrite
io.seek_proc = SeekProc;   // pointer to function that calls fseek
io.tell_proc = TellProc;   // pointer to function that calls ftell

FILE *f = fopen("mybitmap.bmp", "wb");

if (FreeImage_SaveToHandle(FIF_BMP, bitmap, &io, (fi_handle)f, 0)) {
    // bitmap successfully saved!
}

fclose(f);
```

FreeImage_Clone

```
DLL_API FIBITMAP * DLL_CALLCONV FreeImage_Clone(FIBITMAP *dib);
```

Makes an exact reproduction of an existing bitmap, including metadata and attached profile if any.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

FIBITMAP *clone = FreeImage_Clone(dib);

if (clone) {
    // clone succeeded!

    FreeImage_Unload(clone);
}
```

FreeImage_Unload

```
DLL_API void DLL_CALLCONV FreeImage_Unload(FIBITMAP *dib);
```

Deletes a previously loaded FIBITMAP from memory.

You *always* need to call this function once you're done with a bitmap, or you will have a memory leak.

Bitmap information functions

Once a bitmap is loaded into memory, you can retrieve all kinds of information from it or access specific parts from the bitmap, such as the pixel bits and the palette.

FreeImage_GetImageType

```
DLL_API FREE_IMAGE_TYPE DLL_CALLCONV FreeImage_GetImageType(FIBITMAP *dib);
```

Returns the data type of a bitmap (see Table 2).

FreeImage_GetColorsUsed

1 4 8 16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetColorsUsed(FIBITMAP *dib);
```

Returns the number of colors used in a bitmap. This function returns the palette-size for palletised bitmaps, and 0 for high-colour bitmaps.

There has been some criticism on the name of this function. Some users expect this function to return the actual number of colors being used in a bitmap, while the function actually returns the **size of the palette**. The name of this function originates from a member in BITMAPINFOHEADER named biClrUsed. The function actually returns the content of this member.

FreeImage_GetBPP

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetBPP(FIBITMAP *dib);
```

Returns the size of one pixel in the bitmap in bits. For example when each pixel takes 32-bits of space in the bitmap, this function returns 32. Possible bit depths are 1, 4, 8, 16, 24, 32 for standard bitmaps and 16-, 32-, 48-, 64-, 96- and 128-bit for non standard bitmaps.

FreeImage_GetWidth

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetWidth(FIBITMAP *dib);
```

Returns the width of the bitmap in pixel units.

FreeImage_GetHeight

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetHeight(FIBITMAP *dib);
```

Returns the height of the bitmap in pixel units.

FreeImage_GetLine

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetLine(FIBITMAP *dib);
```

Returns the width of the bitmap in bytes.

See also: FreeImage_GetPitch.

There has been some criticism on the name of this function. Some people expect it to return a scanline in the pixel data, while it actually returns the **width of the bitmap in bytes**. As far as I know the term Line is common terminology for the width of a bitmap in bytes. It is at least used by Microsoft DirectX.

FreeImage_GetPitch

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetPitch(FIBITMAP *dib);
```

Returns the width of the bitmap in bytes, rounded to the next 32-bit boundary, also known as pitch or stride or scan width.

In FreeImage each scanline starts at a **32-bit boundary** for performance reasons. This accessor is **essential** when using low level pixel manipulation functions (see also the chapter on Pixel access functions).

FreeImage_GetDIBSize

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetDIBSize(FIBITMAP *dib);
```

Returns the size of the DIB-element of a FIBITMAP in memory, i.e. the BITMAPINFOHEADER + palette + data bits (note that this is not the *real* size of a FIBITMAP, only the size of its DIB-element).

FreeImage_GetMemorySize

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetMemorySize(FIBITMAP *dib);
```

Calculates (or at least estimates) the total memory usage of a FreeImage bitmap. This includes the ICC profile size, the size of the embedded thumbnail (if any), the memory required for all the metadata, as well as any FreeImage internal (housekeeping) memory.

Due to implementation differences, this value may vary between using different C++ standard libraries. Also, the returned value may be just an approximation of the actual memory usage for some C++ standard libraries but, should be exact for the most commonly used C++ standard libraries (Microsoft C++ Standard Library, GNU Standard C++ Library v3, "libc++" C++ Standard Library). For adding support for other libraries and getting more information, see file MapIntrospector.h in the source tree.

There is another difference with the function FreeImage_GetDIBSize. FreeImage_GetMemorySize returns meaningful values for views, that are images that share the DIB data with other FIBITMAPs (see FreeImage_CreateView) or even with other non-FreeImage memory (see FreeImage_ConvertFromRawBitsEx).

Additionally, FreeImage_GetMemorySize works with LOAD_NOPIXELS images as well.

FreeImage_GetPalette

1 4 8 16 24 32

```
DLL_API RGBQUAD *DLL_CALLCONV FreeImage_GetPalette(FIBITMAP *dib);
```

Returns a pointer to the bitmap's palette. If the bitmap doesn't have a palette (i.e. when the pixel bit depth is greater than 8), this function returns NULL.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'
if(FreeImage_GetBPP(dib) == 8) {
  // Build a greyscale palette
  RGBQUAD *pal = FreeImage_GetPalette(dib);
  for (int i = 0; i < 256; i++) {
    pal[i].rgbRed = i;
    pal[i].rgbGreen = i;
    pal[i].rgbBlue = i;
  }
```

FreeImage_GetDotsPerMeterX

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetDotsPerMeterX(FIBITMAP *dib);
```

Returns the horizontal resolution, in pixels-per-meter, of the target device for the bitmap.

FreeImage_GetDotsPerMeterY

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetDotsPerMeterY(FIBITMAP *dib);
```

Returns the vertical resolution, in pixels-per-meter, of the target device for the bitmap.

FreeImage_SetDotsPerMeterX

```
DLL_API void DLL_CALLCONV FreeImage_SetDotsPerMeterX(FIBITMAP *dib, unsigned res);
```

Set the horizontal resolution, in pixels-per-meter, of the target device for the bitmap.

FreeImage_SetDotsPerMeterY

```
DLL_API void DLL_CALLCONV FreeImage_SetDotsPerMeterY(FIBITMAP *dib, unsigned res);
```

Set the vertical resolution, in pixels-per-meter, of the target device for the bitmap.

FreeImage_GetInfoHeader

1 4 8 16 24 32

```
DLL_API BITMAPINFOHEADER *DLL_CALLCONV FreeImage_GetInfoHeader(FIBITMAP *dib);
```

Returns a pointer to the BITMAPINFOHEADER of the DIB-element in a FIBITMAP.

FreeImage_GetInfo

1 4 8 16 24 32

```
DLL_API BITMAPINFO *DLL_CALLCONV FreeImage_GetInfo(FIBITMAP *dib);
```

Alias for FreeImage_GetInfoHeader that returns a pointer to a BITMAPINFO rather than to a BITMAPINFOHEADER.

FreeImage_GetColorType

```
DLL_API FREE_IMAGE_COLOR_TYPE DLL_CALLCONV FreeImage_GetColorType(FIBITMAP *dib);
```

Investigates the color type of the bitmap by reading the bitmap's pixel bits and analysing them. FreeImage_GetColorType can return one of the following values:

Value	Description
FIC_MINISBLACK	Monochrome bitmap (1-bit) : first palette entry is black. Palletised bitmap (4 or 8-bit) and single channel non standard bitmap: the bitmap has a greyscale palette
FIC_MINISWHITE	Monochrome bitmap (1-bit) : first palette entry is white. Palletised bitmap (4 or 8-bit) : the bitmap has an inverted greyscale palette
FIC_PALETTE	Palettized bitmap (1, 4 or 8 bit)
FIC_RGB	High-color bitmap (16, 24 or 32 bit), RGB16 or RGBF
FIC_RGBALPHA	High-color bitmap with an alpha channel (32 bit bitmap, RGBA16 or RGBAF)
FIC_CMYK	CMYK bitmap (32 bit only)

Table 5: FREE_IMAGE_COLOR_TYPE constants.

To be judged greyscale (i.e. FIC_MINISBLACK), a bitmap must have a palette with these characteristics:
- The red, green, and blue values of each palette entry must be equal,
- The interval between adjacent palette entries must be positive and equal to 1.

The CMYK color model (i.e. FIC_CMYK) is the preferred one, if one needs a picture for the print industry or press. In almost every case, this is done by graphic artists: they take a RGB picture (e.g. from a digital camera) and correct the values as appropriate for the picture (single pixel, brightness, contrast...). Finally, they export an CMYK separated image. This will go directly to a layout program and then to the print machines. Most FreeImage users will never need to use CMYK separated images, because the printer drivers will do the conversion job. But in the professional print, the proofed conversion is essential to get a brilliant print result (where no driver will do something like conversion). That's why printed pictures in some magazines look so much better than our home-made prints.

FreeImage_GetRedMask

16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetRedMask(FIBITMAP *dib);
```

Returns a bit pattern describing the red color component of a pixel in a FIBITMAP, returns 0 otherwise.

FreeImage_GetGreenMask

16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetGreenMask(FIBITMAP *dib);
```

Returns a bit pattern describing the green color component of a pixel in a FIBITMAP, returns 0 otherwise.

FreeImage_GetBlueMask

16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetBlueMask(FIBITMAP *dib);
```

Returns a bit pattern describing the blue color component of a pixel in a FIBITMAP, returns 0 otherwise.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'
unsigned red_mask, green_mask, blue_mask;
red_mask = FreeImage_GetRedMask(dib);
green_mask = FreeImage_GetGreenMask(dib);
blue_mask = FreeImage_GetBlueMask(dib);
if(FreeImage_GetBPP(dib) == 16) {
  if ((red_mask == FI16_565_RED_MASK) && (green_mask == FI16_565_GREEN_MASK) &&
      (blue_mask == FI16_565_BLUE_MASK)) {
    // We are in RGB16 565 mode
  } else {
    // We are in RGB16 555 mode
  }
}
```

FreeImage_GetTransparencyCount

1 4 8 16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetTransparencyCount(FIBITMAP *dib);
```

Returns the number of transparent colors in a palletised bitmap. When the bitmap is not palletised, FreeImage_GetTransparencyCount always returns 0.

FreeImage_GetTransparencyTable

1 4 8

```
DLL_API BYTE * DLL_CALLCONV FreeImage_GetTransparencyTable(FIBITMAP *dib);
```

Returns a pointer to the bitmap's transparency table. Only palletised bitmaps have a transparency table. High-color bitmaps store the transparency values directly in the bitmap bits. FreeImage_GetTransparencyTable returns NULL for these bitmaps.

FreeImage_SetTransparencyTable

1 4 8

```
DLL_API void DLL_CALLCONV FreeImage_SetTransparencyTable(FIBITMAP *dib, BYTE *table,
int count);
```

Set the bitmap's transparency table. Only palletised bitmaps have a transparency table. High-color bitmaps store the transparency values directly in the bitmap bits. FreeImage_SetTransparencyTable does nothing for these bitmaps.

```
#include "FreeImage.h"

int main(int argc, char* argv[]) {

 FIBITMAP *hDIB24bpp = FreeImage_Load(FIF_BMP, "test.bmp", 0);
 if (hDIB24bpp) {
  // color-quantize 24bpp (results in a 8bpp bitmap to set transparency)
  FIBITMAP *hDIB8bpp = FreeImage_ColorQuantize(hDIB24bpp, FIQ_WUQUANT);
  // get palette and find bright green
  RGBQUAD  *Palette  = FreeImage_GetPalette(hDIB8bpp);
  BYTE      Transparency[256];
  for (unsigned i = 0; i < 256; i++) {
    Transparency[i] = 0xFF;
    if (Palette[i].rgbGreen >= 0xFE &&
      Palette[i].rgbBlue  == 0x00 &&
      Palette[i].rgbRed   == 0x00) {
      Transparency[i] = 0x00;
    }
  }
  // set the tranparency table
  FreeImage_SetTransparencyTable(hDIB8bpp, Transparency, 256);
  // save 8bpp image as transparent PNG
  FreeImage_Save(FIF_PNG, hDIB8bpp, "test.png", 0);

  FreeImage_Unload(hDIB24bpp);
  FreeImage_Unload(hDIB8bpp);
 }
 return 0;
}
```

FreeImage_SetTransparent

1 4 8 32

```
DLL_API void DLL_CALLCONV FreeImage_SetTransparent(FIBITMAP *dib, BOOL enabled);
```

Tells FreeImage if it should make use of the transparency table or the alpha channel that may accompany a bitmap. When calling this function with a bitmap whose bitdepth is different from 1-, 4-, 8- or 32-bit, transparency is disabled whatever the value of the Boolean parameter.

FreeImage_IsTransparent

```
DLL_API BOOL DLL_CALLCONV FreeImage_IsTransparent(FIBITMAP *dib);
```

Returns TRUE when the transparency table is enabled (1-, 4- or 8-bit images) or when the input dib contains alpha values (32-bit images, RGBA16 or RGBAF images). Returns FALSE otherwise.

FreeImage_SetTransparentIndex

1 4 8

```
DLL_API void DLL_CALLCONV FreeImage_SetTransparentIndex(FIBITMAP *dib, int index);
```

Sets the index of the palette entry to be used as transparent color for the image specified. Does nothing on high color images.

This method sets the index of the palette entry to be used as single transparent color for the image specified. This works on palletised images only and does nothing for high color images.

Although it is possible for palletised images to have more than one transparent color, this method sets the palette entry specified as the single transparent color for the image. All other colors will be set to be non-transparent by this method.

As with FreeImage_SetTransparencyTable, this method also sets the image's transparency property to TRUE (as it is set and obtained by FreeImage_SetTransparent and FreeImage_IsTransparent respectively) for palletised images.

FreeImage_GetTransparentIndex

1 4 8

```
DLL_API int DLL_CALLCONV FreeImage_GetTransparentIndex(FIBITMAP *dib);
```

Returns the palette entry used as transparent color for the image specified. Works for palletised images only and returns -1 for high color images or if the image has no color set to be transparent.

Although it is possible for palletised images to have more than one transparent color, this function always returns the index of the first palette entry, set to be transparent.

FreeImage_HasBackgroundColor

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_HasBackgroundColor(FIBITMAP *dib);
```

Returns TRUE when the image has a file background color, FALSE otherwise.

FreeImage_GetBackgroundColor

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetBackgroundColor(FIBITMAP *dib, RGBQUAD
*bkcolor);
```

Retrieves the file background color of an image. Returns TRUE if successful, FALSE otherwise. For 8-bit images, the color index in the palette is returned in the rgbReserved member of the bkcolor parameter.

FreeImage_SetBackgroundColor

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetBackgroundColor(FIBITMAP *dib, RGBQUAD
*bkcolor);
```

Set the file background color of an image. When saving an image to PNG, this background color is transparently saved to the PNG file.

When the bkcolor parameter is NULL, the background color is removed from the image.

FreeImage_HasPixels

```
DLL_API BOOL DLL_CALLCONV FreeImage_HasPixels(FIBITMAP *dib);
```

Returns FALSE if the bitmap does not contain pixel data (i.e. if it contains only header and possibly some metadata).

Header only bitmap can be loaded using the FIF_LOAD_NOPIXELS load flag (see Table 3). This load flag will tell the decoder to read header data and available metadata and skip pixel data decoding. The memory size of the dib is thus reduced to the size of its members, excluding the pixel buffer. Reading metadata only information is fast since no pixel decoding occurs.

Header only bitmap can be used with Bitmap information functions, Metadata iterator. They cannot be used with any pixel processing function or by saving function.

A plugin can be asked for “header only” support using FreeImage_FIFSupportsNoPixels.

```
BOOL testHeaderData(const char *lpszPathName) {
  int flags = FIF_LOAD_NOPIXELS;

  FIBITMAP *dib = NULL;

  try {
    // load a file using the FIF_LOAD_NOPIXELS flag
    FREE_IMAGE_FORMAT fif = FreeImage_GetFIFFromFilename(lpszPathName);
    assert(FreeImage_FIFSupportsNoPixels(fif) == TRUE);

    dib = FreeImage_Load(fif, lpszPathName, flags);
    if(!dib) throw(1);

    // check that dib does not contains pixels
    BOOL bHasPixel = FreeImage_HasPixels(dib);
    assert(bHasPixel == FALSE);

    // use accessors
    FREE_IMAGE_TYPE type = FreeImage_GetImageType(dib);
    unsigned width = FreeImage_GetWidth(dib);
    unsigned height = FreeImage_GetHeight(dib);
    unsigned bpp = FreeImage_GetBPP(dib);

    // parse some metadata (see e.g. FreeImage_FindFirstMetadata)
    ParseMetadata(dib, FIMD_COMMENTS);
    ParseMetadata(dib, FIMD_EXIF_MAIN);
    ParseMetadata(dib, FIMD_EXIF_EXIF);
    ParseMetadata(dib, FIMD_EXIF_GPS);
    ParseMetadata(dib, FIMD_EXIF_MAKERNOTE);
    ParseMetadata(dib, FIMD_IPTC);
    ParseMetadata(dib, FIMD_XMP);

    // check for a possible embedded thumbnail
    if(FreeImage_GetThumbnail(dib)) {
      // thumbnail is present
      FIBITMAP *thumbnail = FreeImage_GetThumbnail(dib);
    }

    // you cannot access pixels
    BYTE *bits = FreeImage_GetBits(dib);
    assert(bits == NULL);

    FreeImage_Unload(dib);

    return TRUE;
  }
  catch(int) {
    if(dib) FreeImage_Unload(dib);
  }

  return FALSE;
}
```

FreeImage_GetThumbnail

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_GetThumbnail(FIBITMAP *dib);
```

Some image formats allow a thumbnail image to be embedded together with the output image file. When this thumbnail image is present in a file, it is automatically loaded by FreeImage (whatever the loading flag, even when using the FIF_LOAD_NOPIXEL flag).

Image formats that currently support thumbnail loading are JPEG (Exif or JFIF formats), PSD, EXR, TGA and TIFF.

FreeImage_GetThumbnail retrieves a link to the thumbnail that may be available with a dib.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

if( FreeImage_GetThumbnail(bitmap) ) {
  // a thumbnail is available: get a link to it
  FIBITMAP *thumbnail = FreeImage_GetThumbnail(bitmap);
  unsigned width = FreeImage_GetWidth(thumbnail);
  unsigned height = FreeImage_GetHeight(thumbnail);
  FIBITMAP *clone = FreeImage_Clone(thumbnail);
  // ... process 'clone' ...
  FreeImage_Unload(clone);
  // never call FreeImage_Unload on a thumbnail as its lifecycle is managed
internally
}

// calling FreeImage_Unload on the bitmap will destroy everything
// (including its attached thumbnail)
FreeImage_Unload(bitmap);
```

FreeImage_SetThumbnail

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetThumbnail(FIBITMAP *dib, FIBITMAP *thumbnail);
```

Attach a thumbnail image to a dib, so that it can be later stored together with the dib to an ouput image file format.

If input parameter *thumbnail* is NULL then the thumbnail is deleted from the dib.

Image formats that currently support thumbnail saving are JPEG (JFIF formats), EXR, TGA and TIFF.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

// create a thumbnail and convert to a standard bitmap type
FIBITMAP *thumbnail = FreeImage_MakeThumbnail(bitmap, 100, TRUE);
if(thumbnail) {
  // attach the thumbnail to 'bitmap'
  FreeImage_SetThumbnail(bitmap, thumbnail);
  // thumbnail is no longer needed
  FreeImage_Unload(thumbnail);
}

// save the bitmap as JPEG, together with its embedded thumbnail
FreeImage_Save(FIF_JPEG, bitmap, "test_thumb.jpg", 0);

// save the bitmap as JPEG, without embedded thumbnail
FreeImage_SetThumbnail(bitmap, NULL);
FreeImage_Save(FIF_JPEG, bitmap, "test_no_thumb.jpg", 0);

// clear and exit
FreeImage_Unload(bitmap);
```

Thumbnail images are almost always standard bitmaps (e.g. images with a FIT_BITMAP image type). The JPEG format supports 8- or 24-bit thumbnails, while the EXR format only supports 32-bit thumbnails. The TGA format needs a thumbnail with the same bit depth as the image. The TIF format has no restriction regarding the thumbnail bit depth, but a standard bitmap type is recommended.

Filetype functions

The following functions retrieve the FREE_IMAGE_FORMAT from a bitmap by reading up to 16 bytes and analysing it.

Note that for some bitmap types no FREE_IMAGE_FORMAT can be retrieved. This has to do with the bit-layout of the bitmap-types, which are sometimes not compatible with FreeImage's file-type retrieval system. The unidentifiable formats are: CUT, MNG, PCD, TARGA and WBMP. However, these formats can be identified using the FreeImage_GetFIFFromFilename function.

FreeImage_GetFileType

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFileType(const char *filename, int
size FI_DEFAULT(0));
```

Orders FreeImage to analyze the bitmap signature. The function then returns one of the predefined FREE_IMAGE_FORMAT constants or a bitmap identification number registered by a plugin. The *size* parameter is currently not used and can be set to 0.

Because not all formats can be identified by their header (some images don't have a header or one at the end of the file), FreeImage_GetFileType may return FIF_UNKNOWN whereas a plugin is available for the file being analysed. In this case, you can use FreeImage_GetFIFFromFilename to guess the file format from the file extension, but this last function is slower and less accurate.

```
/** Generic image loader
@param lpszPathName Pointer to the full file name
@param flag Optional load flag constant
@return Returns the loaded dib if successful, returns NULL otherwise
*/
FIBITMAP* GenericLoader(const char* lpszPathName, int flag) {
  FREE_IMAGE_FORMAT fif = FIF_UNKNOWN;
  // check the file signature and deduce its format
  // (the second argument is currently not used by FreeImage)
  fif = FreeImage_GetFileType(lpszPathName, 0);
  if(fif == FIF_UNKNOWN) {
    // no signature ?
    // try to guess the file format from the file extension
    fif = FreeImage_GetFIFFromFilename(lpszPathName);
  }
  // check that the plugin has reading capabilities ...
  if((fif != FIF_UNKNOWN) && FreeImage_FIFSupportsReading(fif)) {
    // ok, let's load the file
    FIBITMAP *dib = FreeImage_Load(fif, lpszPathName, flag);
    // unless a bad file format, we are done !
    return dib;
  }
  return NULL;
}
```

FreeImage_GetFileTypeU

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFileTypeU(const wchar_t *filename,
int size FI_DEFAULT(0));
```

This function works exactly like FreeImage_GetFileType but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FIF_UNKNOWN.

FreeImage_GetFileTypeFromHandle

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFileTypeFromHandle(FreeImageIO
*io, fi_handle handle, int size FI_DEFAULT(0));
```

Uses the FreeImageIO structure as described in the topic Bitmap management functions to identify a bitmap type. Now the bitmap bits are retrieved from an arbitrary place.

FreeImage_GetFileTypeFromMemory

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFileTypeFromMemory(FIMEMORY
*stream, int size FI_DEFAULT(0));
```

Uses a memory handle to identify a bitmap type. The bitmap bits are retrieved from an arbitrary place (see the chapter on Memory I/O streams for more information on memory handles).

Pixel access functions

The pixel access functions provide you with an easy way to read, write and work pixel-by-pixel with FIBITMAP data.

FreeImage is able to work not only with standard bitmap data (e.g. 1-, 4-, 8-, 16-, 24- and 32-bit) but also with scientific data such as 16-bit greyscale images, or images made up of long, double or complex values (often used in signal and image processing algorithms). An overview of the supported data types is given in Table 2.

In FreeImage, FIBITMAP are based on a coordinate system that is upside down relative to usual graphics conventions. Thus, the **scanlines are stored upside down**, with the first scan in memory being the bottommost scan in the image.

Bit Formats

In a FIBITMAP the format of the bits are defined by a pixel's bit depth that can be read via a call to FreeImage_GetBPP (see also FreeImage_GetImageType). Possible bit depths include 1-, 4-, 8-, 16-, 24-, 32-, 48-, 64-, 96- and 128-bit. All formats share the following rules:

- Every scanline is DWORD-aligned. The scanline is buffered to alignment; the buffering is set to 0.
- The scanlines are stored upside down, with the first scan (scan 0) in memory being the bottommost scan in the image.

Each format has the following specifics:

- 1-bit DIBs are stored using each bit as an index into the color table. The most significant bit is the leftmost pixel.
- 4-bit DIBs are stored with each 4 bits representing an index into the color table. The most significant nibble is the leftmost pixel.
- 8-bit DIBs are the easiest to store because each byte is an index into the color table.
- 24-bit DIBs have every 3 bytes representing a color, using the same ordering as the RGBTRIPLE structure.
- 32-bit DIB have every 4 bytes representing a color associated to a alpha value (used to indicate transparency), using the same ordering as the RGBQUAD structure.
- Non standard image types such as short, long, float or double do not have a color table. Pixels are stored in a similar way as 8-bit DIB.
- Complex image types are stored in a similar way as 24- or 32bit DIB, using the same ordering as the FICOMPLEX structure.
- 16-bit RGB[A] or float RGB[A] image types are stored in a similar way as 24- or 32bit DIB, using the same ordering as the FIRGB[A]16 or FIRGB[A]F structures.

Color model

A color model is an abstract mathematical model describing the way colors can be represented as tuples of numbers, typically as three or four values or color components (e.g. RGB and CMYK are color models). FreeImage mainly uses the RGB[A] color model to represent pixels in memory.

However, the pixel layout used by this model is OS dependant. Using a byte by byte memory order to label the pixel layout, then FreeImage uses a BGR[A] pixel layout under a Little Endian processor (Windows, Linux) and uses a RGB[A] pixel layout under a Big Endian

processor (Mac OS X or any Big Endian Linux / Unix). This choice was made to ease the use of FreeImage with graphics API.

This subtle difference is however transparent to the user. In order to make pixel access OS independent, FreeImage defines a set of macros used to set or get individual color components in a 24- or 32-bit DIB.

Channel	Pixel position	Associated mask
Red	FI_RGBA_RED	FI_RGBA_RED_MASK
Green	FI_RGBA_GREEN	FI_RGBA_GREEN_MASK
Blue	FI_RGBA_BLUE	FI_RGBA_BLUE_MASK
Alpha	FI_RGBA_ALPHA	FI_RGBA_ALPHA_MASK

Table 6: Pixel access macros and associated masks for 24- or 32-bit images.

When accessing to individual color components of a 24- or 32-bit DIB, you should always use FreeImage macros or RGBTRIPLE / RGBQUAD structures in order to write OS independent code.

The following sample shows how to use these macros when working with a 32-bit dib:

```
 // Allocate a 32-bit dib
 FIBITMAP *dib = FreeImage_Allocate(512, 512, 32, FI_RGBA_RED_MASK,
FI_RGBA_GREEN_MASK, FI_RGBA_BLUE_MASK);

 // Calculate the number of bytes per pixel (3 for 24-bit or 4 for 32-bit)
 int bytespp = FreeImage_GetLine(dib) / FreeImage_GetWidth(dib);

 for(unsigned y = 0; y < FreeImage_GetHeight(dib); y++) {
   BYTE *bits = FreeImage_GetScanLine(dib, y);

  for(unsigned x = 0; x < FreeImage_GetWidth(dib); x++) {
    // Set pixel color to green with a transparency of 128
    bits[FI_RGBA_RED]   = 0;
    bits[FI_RGBA_GREEN] = 255;
    bits[FI_RGBA_BLUE]  = 0;
    bits[FI_RGBA_ALPHA] = 128;

    // jump to next pixel
    bits += bytespp;
  }
 }
```

FreeImage_GetBits

```
DLL_API BYTE *DLL_CALLCONV FreeImage_GetBits(FIBITMAP *dib);
```

Returns a pointer to the data-bits of the bitmap. It is up to you to interpret these bytes correctly, according to the results of FreeImage_GetBPP, FreeImage_GetRedMask, FreeImage_GetGreenMask and FreeImage_GetBlueMask.

For a performance reason, the address returned by FreeImage_GetBits is aligned on a 16 bytes alignment boundary.

Note: FreeImage_GetBits will return NULL if the bitmap does not contain pixel data (i.e. if it contains only header and possibly some or all metadata). See also FreeImage_HasPixels.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

unsigned width  = FreeImage_GetWidth(dib);
unsigned height = FreeImage_GetHeight(dib);
unsigned pitch  = FreeImage_GetPitch(dib);

FREE_IMAGE_TYPE image_type = FreeImage_GetImageType(dib);

// test pixel access avoiding scanline calculations
// to speed-up the image processing

if(image_type == FIT_RGBF) {
  BYTE *bits = (BYTE*)FreeImage_GetBits(dib);
  for(y = 0; y < height; y++) {
    FIRGBF *pixel = (FIRGBF*)bits;
    for(x = 0; x < width; x++) {
      pixel[x].red = 128;
      pixel[x].green = 128;
      pixel[x].blue = 128;
    }
    // next line
    bits += pitch;
  }
}
else if((image_type == FIT_BITMAP) && (FreeImage_GetBPP(dib) == 24)) {
  BYTE *bits = (BYTE*)FreeImage_GetBits(dib);
  for(y = 0; y < height; y++) {
    BYTE *pixel = (BYTE*)bits;
    for(x = 0; x < width; x++) {
      pixel[FI_RGBA_RED]   = 128;
      pixel[FI_RGBA_GREEN] = 128;
      pixel[FI_RGBA_BLUE]  = 128;
      pixel += 3;
    }
    // next line
    bits += pitch;
  }
}
```

FreeImage_GetScanLine

```
DLL_API BYTE *DLL_CALLCONV FreeImage_GetScanLine(FIBITMAP *dib, int scanline);
```

Returns a pointer to the start of the given scanline in the bitmap's data-bits.

It is up to you to interpret these bytes correctly, according to the results of FreeImage_GetBPP and FreeImage_GetImageType (see the following sample).

When working with FIT_INT32 or FIT_UINT32 image types, you should cast the result of FreeImage_GetScanLine with respectively a LONG or a DWORD data type. This is because the size of a long is 32-bit under Windows and is 64-bit under Unix or Linux. Using LONG or DWORD data type ensure that you are working with 32-bit types, whatever the platform.

Note: FreeImage_GetScanLine will return NULL if the bitmap does not contain pixel data (i.e. if it contains only header and possibly some or all metadata). See also FreeImage_HasPixels.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'image'

unsigned x, y;

FREE_IMAGE_TYPE image_type = FreeImage_GetImageType(image);

// test pixel access
switch(image_type) {
  case FIT_BITMAP:
    if(FreeImage_GetBPP(image) == 8) {
      for(y = 0; y < FreeImage_GetHeight(image); y++) {
        BYTE *bits = (BYTE *)FreeImage_GetScanLine(image, y);
        for(x = 0; x < FreeImage_GetWidth(image); x++) {
          bits[x] = 128;
        }
      }
    }
    break;
  case FIT_UINT16:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      unsigned short *bits = (unsigned short *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_INT16:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      short *bits = (short *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_UINT32:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      DWORD *bits = (DWORD *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_INT32:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      LONG *bits = (LONG *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_FLOAT:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      float *bits = (float *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_DOUBLE:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      double *bits = (double *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x] = 128;
      }
    }
    break;
  case FIT_COMPLEX:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      FICOMPLEX *bits = (FICOMPLEX *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x].r = 128;
        bits[x].i = 128;
      }
    }
    break;
  case FIT_RGB16:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      FIRGB16 *bits = (FIRGB16 *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x].red = 128;
        bits[x].green = 128;
```

```
        bits[x].blue = 128;
      }
    }
    break;
  case FIT_RGBF:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      FIRGBF *bits = (FIRGBF *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x].red = 128;
        bits[x].green = 128;
        bits[x].blue = 128;
      }
    }
    break;
  case FIT_RGBA16:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      FIRGBA16 *bits = (FIRGBA16 *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x].red = 128;
        bits[x].green = 128;
        bits[x].blue = 128;
        bits[x].alpha = 128;
      }
    }
    break;
  case FIT_RGBAF:
    for(y = 0; y < FreeImage_GetHeight(image); y++) {
      FIRGBAF *bits = (FIRGBAF *)FreeImage_GetScanLine(image, y);
      for(x = 0; x < FreeImage_GetWidth(image); x++) {
        bits[x].red = 128;
        bits[x].green = 128;
        bits[x].blue = 128;
        bits[x].alpha = 128;
      }
    }
    break;
}
```

FreeImage_GetPixelIndex

1 4 8

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetPixelIndex(FIBITMAP *dib, unsigned x, unsigned
y, BYTE *value);
```

Get the pixel index of a palettized image at position (x, y), including range check (slow access). Parameter x is the pixel position in horizontal direction, and parameter y is the pixel position in vertical direction. The function returns TRUE on success, and returns FALSE otherwise (e.g. for RGB[A] images).

FreeImage_GetPixelColor

16 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetPixelColor(FIBITMAP *dib, unsigned x, unsigned
y, RGBQUAD *value);
```

Get the pixel color of a 16-, 24- or 32-bit image at position (x, y), including range check (slow access). Parameter x is the pixel position in horizontal direction, and parameter y is the pixel position in vertical direction. The function returns TRUE on success, and returns FALSE otherwise (e.g. for palettized images).

FreeImage_SetPixelIndex

1 4 8

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetPixelIndex(FIBITMAP *dib, unsigned x, unsigned
y, BYTE *value);
```

Set the pixel index of a palettized image at position (x, y), including range check (slow access). Parameter x is the pixel position in horizontal direction, and parameter y is the pixel position in vertical direction. The function returns TRUE on success, and returns FALSE otherwise (e.g. for RGB[A] images).

FreeImage_SetPixelColor

16 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetPixelColor(FIBITMAP *dib, unsigned x, unsigned y, RGBQUAD *value);
```

Set the pixel color of a 16-, 24- or 32-bit image at position (x, y), including range check (slow access). Parameter x is the pixel position in horizontal direction, and parameter y is the pixel position in vertical direction. The function returns TRUE on success, and returns FALSE otherwise (e.g. for palettized images).

Conversion functions

The following functions make it possible to convert a bitmap from one bit depth to another.

Under a Little Endian OS (Windows, Linux on PC), bitmaps are always stored in memory as blue first, then green then red, then alpha (BGR[A] convention). Under a Big Endian OS, FreeImage uses the RGB[A] convention. However, these portability considerations are transparently handled by the conversion functions, so that you can later save converted bitmaps in an OS independent manner.

FreeImage_ConvertTo4Bits

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo4Bits(FIBITMAP *dib);
```

Converts a bitmap to 4 bits. If the bitmap was a high-color bitmap (16, 24 or 32-bit) or if it was a monochrome or greyscale bitmap (1 or 8-bit), the end result will be a greyscale bitmap, otherwise (1-bit palletised bitmaps) it will be a palletised bitmap. A clone of the input bitmap is returned for 4-bit bitmaps.

NB: here “greyscale” means that the resulting bitmap will have grey colors, but the palette won’t be a linear greyscale palette. Thus, FreeImage_GetColorType will return FIC_PALETTE.

FreeImage_ConvertTo8Bits

1 4 8 16 24 32 16$_{UINT16}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo8Bits(FIBITMAP *dib);
```

Converts a bitmap to 8 bits. If the bitmap was a high-color bitmap (16, 24 or 32-bit) or if it was a monochrome or greyscale bitmap (1 or 4-bit), the end result will be a greyscale bitmap, otherwise (1 or 4-bit palletised bitmaps) it will be a palletised bitmap. A clone of the input bitmap is returned for 8-bit bitmaps.

When creating the greyscale palette, the greyscale intensity of a result pixel is based on red, green, and blue levels of the corresponding source pixel using the following formula (known as Rec. 709 formula):
grey = (0.2126 x R + 0.7152 x G + 0.0722 x B)
The values 0.2126, 0.7152 and 0.0722 represent the relative red, green, and blue intensities.

For 16-bit greyscale images (images whose type is FIT_UINT16), conversion is done by dividing the 16-bit channel by 256 (see also FreeImage_ConvertToStandardType). A NULL value is returned for other non-standard bitmap types.

FreeImage_ConvertToGreyscale

1 4 8 16 24 32 16$_{UINT16}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToGreyscale(FIBITMAP *dib);
```

Converts a bitmap to a 8-bit greyscale image with a linear ramp. Contrary to the FreeImage_ConvertTo8Bits function, 1-, 4- and 8-bit palletised images are correctly converted, as well as images with a FIC_MINISWHITE color type.

FreeImage_ConvertTo16Bits555

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo16Bits555(FIBITMAP *dib);
```

Converts a bitmap to 16 bits, where each pixel has a color pattern of 5 bits red, 5 bits green and 5 bits blue. One bit in each pixel is unused. A clone of the input bitmap is returned for 16-bit 555 bitmaps

FreeImage_ConvertTo16Bits565

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo16Bits565(FIBITMAP *dib);
```

Converts a bitmap to 16 bits, where each pixel has a color pattern of 5 bits red, 6 bits green and 5 bits blue. A clone of the input bitmap is returned for 16-bit 565 bitmaps

FreeImage_ConvertTo24Bits

1 4 8 16 24 32 48_{RGB16} 64_{RGBA16}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo24Bits(FIBITMAP *dib);
```

Converts a bitmap to 24 bits. A clone of the input bitmap is returned for 24-bit bitmaps.

For 48-bit RGB images, conversion is done by dividing each 16-bit channel by 256. For 64-bit RGBA images, conversion is done by dividing each 16-bit channel by 256, ignoring the alpha channel. A NULL value is returned for other non-standard bitmap types.

FreeImage_ConvertTo32Bits

1 4 8 16 24 32 48_{RGB16} 64_{RGBA16}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertTo32Bits(FIBITMAP *dib);
```

Converts a bitmap to 32 bits. A clone of the input bitmap is returned for 32-bit bitmaps.

For 48-bit RGB images, conversion is done by dividing each 16-bit channel by 256 and by setting the alpha channel to an opaque value (0xFF). For 64-bit RGBA images, conversion is done by dividing each 16-bit channel by 256. A NULL value is returned for other non-standard bitmap types.

FreeImage_ColorQuantize

24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ColorQuantize(FIBITMAP *dib,
FREE_IMAGE_QUANTIZE quantize);
```

Quantizes a high-color 24- or 32-bit bitmap to an 8-bit palette color bitmap. The quantize parameter specifies the color reduction algorithm to be used:

Parameter	Quantization method
FIQ_WUQUANT	Xiaolin Wu color quantization algorithm
FIQ_NNQUANT	NeuQuant neural-net quantization algorithm by Anthony Dekker (**24-bit only**)
FIQ_LFPQUANT	Lossless Fast Pseudo-Quantization Algorithm by Carsten Klein

Table 7: FREE_IMAGE_QUANTIZE constants.

References

Wu, Xiaolin, Efficient Statistical Computations for Optimal Color Quantization. In Graphics Gems, vol. II, p. 126-133. [Online] http://www.ece.mcmaster.ca/~xwu/

Dekker A. H., Kohonen neural networks for optimal color quantization. Network: Computation in Neural Systems, Volume 5, Number 3, Institute of Physics Publishing, 1994. [Online] http://members.ozemail.com.au/~dekker/NEUQUANT.HTML

About the Lossless Fast Pseudo-Quantization Algorithm (Carsten Klein, 2014)

The Lossless Fast Pseudo-Quantization algorithm is no real quantization algorithm, since it makes no attempt to create a palette, that is suitable for all colors of the 24-bit source image. However, it provides very fast conversions from 24-bit to 8-bit images, if the number of distinct colors in the source image is *not greater than* the desired palette size (i.e. less than 256). If the number of colors in the source image is exceeded, the Quantize method of this implementation stops the process and returns NULL.

This implementation uses a very fast hash map implementation to collect the source image's colors. It turned out that a customized implementation of a hash table with open addressing (using linear probing) provides the best performance. The hash table has 512 entries, which prevents the load factor to exceed 0.5 as we have 256 entries at most. Each entry consumes 64 bits, so the whole hash table takes 4KB of memory.

For large images, the LFPQuantizer is typically up to three times faster than the Wu Quantizer.

FreeImage_ColorQuantizeEx

24

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ColorQuantizeEx(FIBITMAP *dib,
FREE_IMAGE_QUANTIZE quantize FI_DEFAULT(FIQ_WUQUANT), int PaletteSize FI_DEFAULT(256),
int ReserveSize FI_DEFAULT(0), RGBQUAD *ReservePalette FI_DEFAULT(NULL));
```

FreeImage_ColorQuantizeEx is an extension to the FreeImage_ColorQuantize function that provides additional options used to quantize a 24-bit image to any number of colors (up to 256), as well as quantize a 24-bit image using a partial or full provided palette.

The *PaletteSize* parameter is the size of the desired output palette. *ReserveSize* is the size of the provided palette, given by the *ReservePalette* input array.

```
// this code assumes there is a 24-bit bitmap loaded and
// present in a variable called 'dib'

RGBQUAD web_palette[216]; // list of the 216 "web-safe" colors (RGB increments of 51)
// ...

// Perform a color quantization using a user supplied palette
// The goal of FreeImage_ColorQuantizeEx will be to fill in
// the remaining 39 palette entries with the best choices based
// on the input image, then use the palette of size 255 to quantize the image.

// The output palette will contain a mix of the 216 and 39 colors,
// but not in any particular order. Palette entry 255 (the 256th entry)
// is unused in the image, and will be black in the palette.
// This allows the user to use the palette entry # 255 for transparency
// without worrying about making valid pixel data become transparent.

FIBITMAP *dib8_a = FreeImage_ColorQuantizeEx(dib, FIQ_NNQUANT, 255, 216,
web_palette);

// Other uses of the function

// Only use 255 colors, so the 256th can be used for transparency
FIBITMAP *dib8_b = FreeImage_ColorQuantizeEx(dib, FIQ_NNQUANT, 255, 0, NULL);

 // Generate no additional colors, only use the web-safe colors
FIBITMAP *dib8_c = FreeImage_ColorQuantizeEx(dib, FIQ_NNQUANT, 216, 216,
web_palette);

// Quantize using a palette from a different dib
RGBQUAD another_palette[256];
// ...
FIBITMAP *dib8_d = FreeImage_ColorQuantizeEx(dib, FIQ_NNQUANT, 256, 256,
                                             another_palette);

// ...

FreeImage_Unload(dib8_a);
FreeImage_Unload(dib8_b);
FreeImage_Unload(dib8_c);
FreeImage_Unload(dib8_d);
```

When using FreeImage_ColorQuantizeEx, the *PaletteSize* setting works on all quantizers, with any value 2-256, but the *ReserveSize*/*ReservePalette* settings work on the NN quantizer and the LFP quantizer only.

FreeImage_Threshold

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Threshold(FIBITMAP *dib, BYTE T);
```

Converts a bitmap to 1-bit monochrome bitmap using a threshold T between [0..255]. The function first converts the bitmap to a 8-bit greyscale bitmap. Then, any brightness level that is less than T is set to zero, otherwise to 1. For 1-bit input bitmaps, the function clones the input bitmap and builds a monochrome palette.

FreeImage_Dither

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Dither(FIBITMAP *dib, FREE_IMAGE_DITHER
algorithm);
```

Converts a bitmap to 1-bit monochrome bitmap using a dithering algorithm. For 1-bit input bitmaps, the function clones the input bitmap and builds a monochrome palette.

The *algorithm* parameter specifies the dithering algorithm to be used. The function first converts the bitmap to a 8-bit greyscale bitmap. Then, the bitmap is dithered using one of the following algorithms:

Parameter	Dithering method
FID_FS	Floyd & Steinberg error diffusion algorithm
FID_BAYER4x4	Bayer ordered dispersed dot dithering (order 2 – 4x4 -dithering matrix)
FID_BAYER8x8	Bayer ordered dispersed dot dithering (order 3 – 8x8 -dithering matrix)
FID_BAYER16x16	Bayer ordered dispersed dot dithering (order 4 – 16x16 dithering matrix)
FID_CLUSTER6x6	Ordered clustered dot dithering (order 3 - 6x6 matrix)
FID_CLUSTER8x8	Ordered clustered dot dithering (order 4 - 8x8 matrix)
FID_CLUSTER16x16	Ordered clustered dot dithering (order 8 - 16x16 matrix)

Table 8: FREE_IMAGE_DITHER constants.

References

Ulichney, R., Digital Halftoning. The MIT Press, Cambridge, MA, 1987.

Hawley S., Ordered Dithering. Graphics Gems, Academic Press, 1990.

FreeImage_ConvertFromRawBits

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertFromRawBits(BYTE *bits, int width, int height, int pitch, unsigned bpp, unsigned red_mask, unsigned green_mask, unsigned blue_mask, BOOL topdown FI_DEFAULT(FALSE));
```

Converts a raw bitmap somewhere in memory to a FIBITMAP. The parameters in this function are used to describe the raw bitmap. The first parameter is a pointer to the start of the raw bits. The width and height parameter describe the size of the bitmap. The pitch defines the total width of a scanline in the source bitmap, including padding bytes that may be applied. The bpp parameter tells FreeImage what the bit depth of the bitmap is. The red_mask, green_mask and blue_mask parameters tell FreeImage the bit-layout of the color components in the bitmap. The last parameter, topdown, will store the bitmap top-left pixel first when it is TRUE or bottom-left pixel first when it is FALSE.

When the source bitmap uses a 32-bit padding, you can calculate the pitch using the following formula:
int pitch = ((((bpp * width) + 31) / 32) * 4);

FreeImage_ConvertFromRawBitsEx

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertFromRawBitsEx(BOOL copySource, BYTE *bits, FREE_IMAGE_TYPE type, int width, int height, int pitch, unsigned bpp, unsigned red_mask, unsigned green_mask, unsigned blue_mask, BOOL topdown FI_DEFAULT(FALSE));
```

Converts a raw bitmap somewhere in memory to a FIBITMAP. Conversion can be done without allocating an internal pixel buffer. The parameters in this function are used to describe the raw bitmap. A description of parameters follows:

- *copySource*: if FALSE, wrap the user's pixel buffer, otherwise, make a deep copy
- *bits*: pointer to the start of the raw bits
- *type*: image type
- *width*: image width

- *height*: image height
- *pitch*: image pitch
- *bpp*: image pixel's bit depth
- *red_mask*: bit-layout of the red color components in the image
- *green_mask*: bit-layout of the green color components in the image
- *blue_mask*: bit-layout of the blue color components in the image
- *topdown*: store the bitmap top-left pixel first when it is TRUE or bottom-left pixel first when it is FALSE

```
static void testBasicWrapper(BOOL copySource, BYTE *bits, FREE_IMAGE_TYPE type, int
width, int height, int pitch, unsigned bpp) {
  FIBITMAP *src = NULL;
  FIBITMAP *clone = NULL;
  FIBITMAP *dst = NULL;

  // allocate a wrapper
  src = FreeImage_ConvertFromRawBitsEx(copySource, bits, type, width, height, pitch,
bpp, FI_RGBA_RED_MASK, FI_RGBA_GREEN_MASK, FI_RGBA_BLUE_MASK, FALSE);
  assert(src != NULL);

  // test clone
  clone = FreeImage_Clone(src);
  assert(clone != NULL);

  // test in-place processing
  FreeImage_Invert(src);

  // test processing
  dst = FreeImage_ConvertToFloat(src);
  assert(dst != NULL);

  FreeImage_Unload(dst);
  FreeImage_Unload(clone);

  // unload the wrapper
  FreeImage_Unload(src);
}

static void testViewport(FIBITMAP *dib) {
  FIBITMAP *src = NULL;

  // define a viewport as [vp_x, vp_y, vp_width, vp_height]
  // (assume the image is larger than the viewport)
  int vp_width = 300;
  int vp_height = 200;
  int vp_x = FreeImage_GetWidth(dib) / 2 - vp_width / 2;
  int vp_y = FreeImage_GetHeight(dib) / 2 - vp_height / 2;

  // point the viewport data
  unsigned bytes_per_pixel = FreeImage_GetLine(dib) / FreeImage_GetWidth(dib);
  BYTE *data = FreeImage_GetBits(dib) + vp_y * FreeImage_GetPitch(dib) + vp_x *
bytes_per_pixel;

  // wrap a section (no copy)
  src = FreeImage_ConvertFromRawBitsEx(FALSE/*copySource*/, data, FIT_BITMAP,
vp_width, vp_height, FreeImage_GetPitch(dib), FreeImage_GetBPP(dib),
FI_RGBA_RED_MASK, FI_RGBA_GREEN_MASK, FI_RGBA_BLUE_MASK);

  // save the section (note that the image is inverted due to previous processing)
  FreeImage_Save(FIF_PNG, src, "viewport png");

  // unload the wrapper
  FreeImage_Unload(src);
}

// Main test functions
// --------------------------------------------------------

void testWrappedBuffer(const char *lpszPathName, int flags) {
  FIBITMAP *dib = NULL;

  // simulate a user provided buffer
  // -------------------------------

  // load the dib
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(lpszPathName);
  dib = FreeImage_Load(fif, lpszPathName, flags);
  assert(dib != NULL);

  // get data info
  FREE_IMAGE_TYPE type = FreeImage_GetImageType(dib);
  unsigned width = FreeImage_GetWidth(dib);
  unsigned height = FreeImage_GetHeight(dib);
  unsigned pitch = FreeImage_GetPitch(dib);
  unsigned bpp = FreeImage_GetBPP(dib);
  BYTE *bits = FreeImage_GetBits(dib);

  // test wrapped buffer manipulations
  // -------------------------------
```

```
  testBasicWrapper(TRUE /*copySource*/, bits, type, width, height, pitch, bpp);

  testBasicWrapper(FALSE /*copySource*/, bits, type, width, height, pitch, bpp);

  // test another use-case : viewport
  testViewport(dib);

  // unload the user provided buffer
  // -------------------------------
  FreeImage_Unload(dib);
}
```

FreeImage_ConvertToRawBits

1 4 8 16 24 32

```
DLL_API void DLL_CALLCONV FreeImage_ConvertToRawBits(BYTE *bits, FIBITMAP *dib, int
pitch, unsigned bpp, unsigned red_mask, unsigned green_mask, unsigned blue_mask, BOOL
topdown FI_DEFAULT(FALSE));
```

Converts a FIBITMAP to a raw piece of memory. The layout of the memory is described in the passed parameters, which are the same as in the previous function. The last parameter, *topdown*, will store the bitmap top-left pixel first when it is TRUE or bottom-left pixel first when it is FALSE.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

// convert a bitmap to a 32-bit raw buffer (top-left pixel first)
// --------------------------------------------------------------
FIBITMAP *src = FreeImage_ConvertTo32Bits(dib);
FreeImage_Unload(dib);

// Allocate a raw buffer
int width = FreeImage_GetWidth(src);
int height = FreeImage_GetHeight(src);
int scan_width = FreeImage_GetPitch(src);
BYTE *bits = (BYTE*)malloc(height * scan_width);

// convert the bitmap to raw bits (top-left pixel first)
FreeImage_ConvertToRawBits(bits, src, scan_width, 32,
                      FI_RGBA_RED_MASK, FI_RGBA_GREEN_MASK, FI_RGBA_BLUE_MASK,
TRUE);
FreeImage_Unload(src);

// convert a 32-bit raw buffer (top-left pixel first) to a FIBITMAP
// ----------------------------------------------------------------
FIBITMAP *dst = FreeImage_ConvertFromRawBits(bits, width, height, scan_width,
                32, FI_RGBA_RED_MASK, FI_RGBA_GREEN_MASK, FI_RGBA_BLUE_MASK, FALSE);
```

FreeImage_ConvertToStandardType

1 4 8 16 24 32 16_{UINT16} 32_{FLOAT} 64_{DOUBLE} 16_{INT16} $32_{UINT32/INT32}$ $2x64_{COMPLEX}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToStandardType(FIBITMAP *src, BOOL
scale_linear FI_DEFAULT(TRUE));
```

Converts a non standard image whose color type is FIC_MINISBLACK to a standard 8-bit greyscale image (see Table 9 for allowed conversions). When the *scale_linear* parameter is TRUE, conversion is done by scaling linearly each pixel value from [min, max] to an integer value between [0..255], where min and max are the minimum and maximum pixel values in the image. When *scale_linear* is FALSE, conversion is done by rounding each pixel value to an integer between [0..255]. Rounding is done using the following formula:

dst_pixel = (BYTE) MIN(255, MAX(0, q)) where int q = int(src_pixel + 0.5);

The function returns the converted 8-bit greyscale image. For standard images, a clone of the input image is returned.

For complex images, the magnitude is extracted as a double image and then converted according to the scale parameter.

FreeImage_ConvertToType

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToType(FIBITMAP *src, FREE_IMAGE_TYPE
dst_type, BOOL scale_linear FI_DEFAULT(TRUE));
```

Converts an image of any type to type *dst_type*. When *dst_type* is equal to FIT_BITMAP, the function calls FreeImage_ConvertToStandardType. Otherwise, conversion is done using standard C language casting convention. When a conversion is not allowed, a NULL value is returned and an error message is thrown (it can be catched using FreeImage_SetOutputMessage). The following conversions are currently allowed by the library (other conversions may be added easily if needed):

➔	FIT_BITMAP	FIT_UINT16	FIT_INT16	FIT_UINT32	FIT_INT32	FIT_FLOAT	FIT_DOUBLE	FIT_COMPLEX	FIT_RGB16	FIT_RGBA16	FIT_RGBF	FIT_RGBAF
FIT_BITMAP	•	•	•	•	•	•	•	•	•	•	•	•
FIT_UINT16	•	•				•	•	•	•	•	•	•
FIT_INT16	•		•			•	•	•				
FIT_UINT32	•			•		•	•	•				
FIT_INT32	•				•	•	•	•				
FIT_FLOAT	•					•	•	•			•	•
FIT_DOUBLE	•						•	•				
FIT_COMPLEX								•				
FIT_RGB16	•	•				•			•	•	•	•
FIT_RGBA16	•	•				•			•	•	•	•
FIT_RGBF						•					•	•
FIT_RGBAF						•					•	•

color legend

- • returns FreeImage_Clone
- • returns FreeImage_ConvertTo24Bits
- • returns FreeImage_ConvertTo32Bits
- • returns FreeImage_ConvertToUINT16
- • returns FreeImage_ConvertToRGB16
- • returns FreeImage_ConvertToRGBA16
- • returns FreeImage_ConvertToFloat
- • returns FreeImage_ConvertToRGBF
- • returns FreeImage_ConvertToRGBAF
- • returns FreeImage_ConvertToStandardType
- • conversion is done using standard C language casting convention

Table 9: Bitmap type conversions allowed by FreeImage.

FreeImage_ConvertToFloat

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToFloat(FIBITMAP *dib);
```

Converts an image to a FIT_FLOAT image type. Conversion is done as follows:

- FIT_BITMAP type: the input dib is first converted to a 8-bit greyscale image using FreeImage_ConvertToGreyscale, then each pixel value is divided by 255 so that the output image is in the range [0..1].
- FIT_UINT16 type: conversion is done by copying the source integer pixel values into the destination float pixel values, then each pixel value is divided by 65535 so that the output image is in the range [0..1].

- FIT_RGB[A]16 type: the input dib is first converted to a 16-bit greyscale image, then each pixel value is divided by 65535 so that the output image is in the range [0..1]. When an alpha channel is present in the source, it is simply ignored by the conversion function.
- FIT_RGB[A]F type: the input dib is converted to a 32-bit float image. When an alpha channel is present in the source, it is simply ignored by the conversion function.

For 32-bit float input images, a clone of the input is returned.

When creating the greyscale float image, the luminance L (or greyscale intensity) of a result pixel is calculated from the sRGB model using a D65 white point, using the Rec.709 formula :

L = (0.2126 * r) + (0.7152 * g) + (0.0722 * b)

The values 0.2126, 0.7152 and 0.0722 represent the relative red, green, and blue intensities.

FreeImage_ConvertToRGBF

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToRGBF(FIBITMAP *dib);
```

Converts a 24- or 32-bit RGB(A) standard image or a 48- or 64-bit RGB(A) image to a FIT_RGBF type image. Conversion is done by copying the source integer pixel values into the destination float pixel values, and dividing by the maximum source pixel value (i.e. 255 or 65535) so that the output image is in the range [0..1]. When an alpha channel is present in the source, it is simply ignored by the conversion function. For 96-bit RGBF input images, a clone of the input is returned. For 128-bit RGBAF images, conversion is done by copying the source float pixel values into the destination float pixel values, skipping the alpha channel.

FreeImage_ConvertToRGBAF

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToRGBAF(FIBITMAP *dib);
```

Converts a 24- or 32-bit RGB(A) standard image or a 48- or 64-bit RGB(A) image to a FIT_RGBAF type image. Conversion is done by copying the source integer pixel values into the destination float pixel values, and dividing by the maximum source pixel value (i.e. 255 or 65535) so that the output image is in the range [0..1]. For 128-bit RGBAF input images, a clone of the input is returned.

FreeImage_ConvertToUINT16

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToUINT16(FIBITMAP *dib);
```

Converts a bitmap to an unsigned 16-bit greyscale image (i.e. image whose type is FIT_UINT16). Standard bitmaps are first converted (if needed) to 8-bit greyscale images and then conversion is done by multiplying the 8-bit channel by 256. Other bitmap types are converted by using a greyscale conversion formula.

For 16-bit FIT_UINT16 images, a clone of the input is returned.

When creating the greyscale uint16 image from a RGB[A] image, the luminance L (or greyscale intensity) of a result pixel is calculated from the sRGB model using a D65 white point, using the Rec.709 formula :

L = (0.2126 * r) + (0.7152 * g) + (0.0722 * b)

The values 0.2126, 0.7152 and 0.0722 represent the relative red, green, and blue intensities.

FreeImage_ConvertToRGB16

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToRGB16(FIBITMAP *dib);
```

Converts a bitmap to an unsigned 16-bit RGB image (i.e. image whose type is FIT_RGB16). Standard bitmaps are first converted (if needed) to 24-bit RGB images and then conversion is done by multiplying the 8-bit channel by 256. When an alpha channel is present in the source, it is simply ignored by the conversion function. For 16-bit FIT_RGB16 images, a clone of the input is returned.

FreeImage_ConvertToRGBA16

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ConvertToRGBA16(FIBITMAP *dib);
```

Converts a bitmap to an unsigned 16-bit RGBA image (i.e. image whose type is FIT_RGBA16). Standard bitmaps are first converted (if needed) to 32-bit RGBA images and then conversion is done by multiplying the 8-bit channel by 256. For 16-bit FIT_RGBA16 images, a clone of the input is returned.

Tone mapping operators

Tone mapping operators are used to compress a large range of pixel luminances into a smaller range that is suitable for display on devices with limited dynamic range (e.g. display devices such as CRTs or LCDs and print media).

In principle this problem is simple: we need to turn an image with a large range of numbers into an image containing integers in the range of 0 to 255 such that we can display it on a printer or a monitor. This suggests linear scaling as a possible solution. However, this approach is flawed because details in the light or dark areas of the image will be lost due to subsequent quantization, and the displayed image will therefore not be perceived the same as the scene that was photographed. For this reason, more elaborate algorithms, called tone mapping operators, have been proposed to accurately render High Dynamic Range images.

FreeImage_ToneMapping

48_{RGB16} 64_{RGBA16} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_ToneMapping(FIBITMAP *dib, FREE_IMAGE_TMO
tmo, double first_param FI_DEFAULT(0), double second_param FI_DEFAULT(0));
```

Converts a High Dynamic Range image (48-bit RGB or 96-bit RGBF) to a 24-bit RGB image, suitable for display. The *tmo* parameter specifies the tone mapping operator to be used. The function first converts the input image to a 96-bit RGBF image (using the FreeImage_ConvertToRGBF function). Then, the bitmap is tone mapped using one of the following algorithms:

Parameter	Tone mapping operator
FITMO_DRAGO03	Adaptive logarithmic mapping (F. Drago, 2003)
FITMO_REINHARD05	Dynamic range reduction inspired by photoreceptor physiology (E. Reinhard, 2005)
FITMO_FATTAL02	Gradient domain High Dynamic Range compression (R. Fattal, 2002)

Table 10: FREE_IMAGE_TMO constants.

The meaning of the *first_param* and *second_param* parameters depends on the choosen algorithm (see the definition of each tone mapping operator below). When both parameters are set to zero, a default set of parameters is used.

```
// load a HDR RGB Float image
FIBITMAP *src = FreeImage_Load(FIF_HDR, "memorial.hdr", 0);
// create a 24-bit tone mapped image suitable for display
FIBITMAP *dst = FreeImage_ToneMapping(src, FITMO_DRAGO03);
// …
FreeImage_Unload(src);
FreeImage_Unload(dst);
```

FreeImage_TmoDrago03

48_{RGB16} 64_{RGBA16} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP* DLL_CALLCONV FreeImage_TmoDrago03(FIBITMAP *src, double gamma
FI_DEFAULT(2.2), double exposure FI_DEFAULT(0));
```

Converts a High Dynamic Range image to a 24-bit RGB image using a global operator based on logarithmic compression of luminance values, imitating the human response to light. A bias power function is introduced to adaptively vary logarithmic bases, resulting in good preservation of details and contrast.

Upon entry, *gamma* (where gamma > 0) is a gamma correction that is applied *after* the tone mapping. A value of 1 means no correction. The default 2.2 value, used in the original author's paper, is recommended as a good starting value.

The *exposure* parameter, in the range [-8, 8], is an exposure scale factor allowing users to adjust the brightness of the output image to their displaying conditions. The default value (0) means that no correction is applied. Higher values will make the image lighter whereas lower values make the image darker.

Reference

F. Drago, K. Myszkowski, T. Annen and N. Chiba, Adaptive logarithmic mapping for displaying high contrast scenes. Proceedings of Eurographics2003, Vol.22, No, 3, pp. 419-426, 2003.

FreeImage_TmoReinhard05

48$_{RGB16}$ 64$_{RGBA16}$ 96$_{RGBF}$ 128$_{RGBAF}$

```
DLL_API FIBITMAP* DLL_CALLCONV FreeImage_TmoReinhard05(FIBITMAP *src, double intensity
FI_DEFAULT(0), double contrast FI_DEFAULT(0));
```

Converts a High Dynamic Range image to a 24-bit RGB image using a global operator inspired by photoreceptor physiology of the human visual system.

Upon entry, the *intensity* parameter, in the range [-8, 8], controls the overall image intensity. The default value 0 means no correction. Higher values will make the image lighter whereas lower values make the image darker.

The *contrast* parameter, in the range [0.3, 1.0[, controls the overall image contrast. When using the default value (0), this parameter is calculated automatically.

References

E. Reinhard and K. Devlin, Dynamic Range Reduction Inspired by Photoreceptor Physiology. IEEE Transactions on Visualization and Computer Graphics, 11(1), Jan/Feb 2005.

E. Reinhard, Parameter estimation for photographic tone reproduction, Journal of Graphics Tools, vol. 7, no. 1, pp. 45–51, 2003.

FreeImage_TmoReinhard05Ex

48$_{RGB16}$ 64$_{RGBA16}$ 96$_{RGBF}$ 128$_{RGBAF}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_TmoReinhard05Ex(FIBITMAP *src, double
intensity FI_DEFAULT(0), double contrast FI_DEFAULT(0), double adaptation
FI_DEFAULT(1), double color_correction FI_DEFAULT(0));
```

Converts a High Dynamic Range image to a 24-bit RGB image using a global / local operator inspired by photoreceptor physiology of the human visual system. User parameters control intensity, contrast, and level of adaptation.

Upon entry, the *intensity* parameter, in the range [-8, 8], controls the overall image intensity. The default value 0 means no correction. Higher values will make the image lighter whereas lower values make the image darker.

The *contrast* parameter, in the range [0.3, 1.0[, controls the overall image contrast. When using the default value (0), this parameter is calculated automatically.

The *adaptation* parameter, in range [0:1], controls the level of light adaptation. When using the default value (1), adaptation is local and is based on the pixel intensity. When using a value 0, the adaptation is global and is based on the average channel intensity.

The *color_correction* parameter, in range [0:1], controls the level of chromatic adaptation. Using the default value (0) means no chromatic adaptation, i.e. the *adaptation* level is the same for all 3 color channels. Setting this value to 1 means that each R, G, B channel is treated independently.

References

E. Reinhard and K. Devlin, Dynamic Range Reduction Inspired by Photoreceptor Physiology. IEEE Transactions on Visualization and Computer Graphics, 11(1), Jan/Feb 2005.

E. Reinhard, Parameter estimation for photographic tone reproduction, Journal of Graphics Tools, vol. 7, no. 1, pp. 45–51, 2003.

FreeImage_TmoFattal02

48$_{RGB16}$ 64$_{RGBA16}$ 96$_{RGBF}$ 128$_{RGBAF}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_TmoFattal02(FIBITMAP *src, double
color_saturation FI_DEFAULT(0.5), double attenuation FI_DEFAULT(0.85));
```

Converts a High Dynamic Range image to a 24-bit RGB image using a local operator that manipulate the gradient field of the luminance image by attenuating the magnitudes of large gradients. A new, low dynamic range image is then obtained by solving a Poisson equation on the modified gradient field.

Upon entry, the *color_saturation* parameter, in the range [0.4, 0.6], controls color saturation in the resulting image.

The *attenuation* parameter, in the range [0.8, 0.9], controls the amount of attenuation.

The algorithm works by solving as many Partial Differential Equations as there are pixels in the image, using a Poisson solver based on a multigrid algorithm. Thus, the algorithm may take many minutes (up to 5 or more) before to complete.

Reference

R. Fattal, D. Lischinski, and M. Werman, Gradient domain high dynamic range compression. ACM Transactions on Graphics, 21(3):249–256, 2002.

ICC profile functions

Whenever an ICC profile is available in a bitmap file it is transparently loaded and stored in the FIBITMAP. On the other side, whenever an ICC profile is stored in a FIBITMAP, it is transparently stored in the bitmap file when saving, provided the output FREEIMAGE_FORMAT supports ICC profiles (a plugin can be asked for ICC profile support using FreeImage_FIFSupportsICCProfiles).

FreeImage defines a structure called FIICCPROFILE, that is used to access this ICC profile. The structure can then be used with any color management engine to perform bitmap transformations between two ICC profiles.

If the FIICCPROFILE is flagged with FIICC_COLOR_IS_CMYK the bitmap is a representation of a CMYK separation. Together with color management this information is important, because the profile data and the bitmap must reside in the same color model (e.g. RGB or CMYK).

In almost all cases, the bitmap is loaded as an RGB representation. It may depend on special flags to FreeImage_Load, whether the original color representation is preserved or not.

```
// load a bitmap from file, enforce to preserve the
// CMYK separated data from TIFF (no RGB conversion done)

FIBITMAP *bitmap = FreeImage_Load (FIF_TIFF, name, TIFF_CMYK);

if (bitmap) {

// test for RGB or CMYK colour space

  if ((FreeImage_GetICCProfile(bitmap)->flags &
       FIICC_COLOR_IS_CMYK) == FIICC_COLOR_IS_CMYK)

    // we are in CMYK colour space

  else
    // we are in RGB colour space

}
```

ICC profiles are currently supported by TIFF, PNG and JPEG plugins.

FreeImage_GetICCProfile

```
DLL_API FIICCPROFILE *DLL_CALLCONV FreeImage_GetICCProfile(FIBITMAP *dib);
```

Retrieves a pointer to the FIICCPROFILE data of the bitmap. This function can also be called safely, when the original format does not support profiles.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

// retrieve a pointer to FIICCPROFILE structure

FIICCPROFILE *profile = FreeImage_GetICCProfile(bitmap);

If (profile->data) {

  // profile data present

}
```

FreeImage_CreateICCProfile

```
DLL_API FIICCPROFILE *DLL_CALLCONV FreeImage_CreateICCProfile(FIBITMAP *dib, void
*data, long size);
```

Creates a new FIICCPROFILE block from ICC profile data previously read from a file or built by a color management system. The profile data is attached to the bitmap. The function returns a pointer to the FIICCPROFILE structure created.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

DWORD size = _filelength(fileno(hProfile));

// read profile data from file and zero-terminate

if (size && (data = (void *)malloc(size + 1))) {
  size = fread(data, 1, size, hProfile);
  *(data + size) = 0;

  // attach retrieved profile data to bitmap

  FIICCPROFILE *profile = FreeImage_CreateICCProfile (bitmap, data, size);

  free (data);
}
```

FreeImage_DestroyICCProfile

```
DLL_API void DLL_CALLCONV FreeImage_DestroyICCProfile(FIBITMAP *dib);
```

This function destroys an FIICCPROFILE previously created by FreeImage_CreateICCProfile. After this call the bitmap will contain no profile information. This function should be called to ensure that a stored bitmap will not contain any profile information.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'bitmap'

// destroy profile possibly present

FreeImage_DestroyICCProfile(bitmap);

// store profile-less bitmap

FreeImage_Save (FIF_TIFF, bitmap, name, flags);
```

Plugin functions

Through average use you won't probably notice it, FreeImage is plugin driven. Each bitmap loader/saver is in fact a plugin module that is linked inside the integrated plugin manager. You won't notice it, until you decide to write your own plugins.

Almost every plugin in FreeImage is incorporated directly into the DLL. The reason why this is done this way is a mixture of evolution and design. The first versions of FreeImage (actually, about the whole first year of its existence) it had no notion of plugins. This meant that all bitmap functionality was available only from the main DLL. In the second year Floris decided to create plugins, because he wanted to support some bitmaps formats that have license restrictions on them, such as GIF. In fear that he would put all its bitmap loaders/savers in tiny DLLs that would splatter the hard drive, his most important 'customer' strongly encouraged him to keep as much bitmap formats in one DLL as possible. He took his word for it and it lead to the design you see here today.

The actual plugin system evolved from something very simple to a very flexible mechanism that he now often reuses in other software. At this moment it's possible to have plugins in the main FREEIMAGE.DLL, in external DLLs, and even directly in an application that drives FreeImage.

FreeImage_GetFIFCount

```
DLL_API int DLL_CALLCONV FreeImage_GetFIFCount();
```

Retrieves the number of FREE_IMAGE_FORMAT identifiers being currently registered. In FreeImage FREE_IMAGE_FORMAT became, through evolution, synonymous with plugin.

FreeImage_SetPluginEnabled

```
DLL_API int DLL_CALLCONV FreeImage_SetPluginEnabled(FREE_IMAGE_FORMAT fif, BOOL
enable);
```

Enables or disables a plugin. A disabled plugin cannot be used to import and export bitmaps, nor will it identify bitmaps. When called, this function returns the previous plugin state (TRUE / 1 or FALSE / 0), or –1 if the plugin doesn't exist.

FreeImage_IsPluginEnabled

```
DLL_API int DLL_CALLCONV FreeImage_IsPluginEnabled(FREE_IMAGE_FORMAT fif);
```

Returns TRUE when the plugin is enabled, FALSE when the plugin is disabled, -1 otherwise.

FreeImage_GetFIFFromFormat

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFIFFromFormat(const char *format);
```

Returns a FREE_IMAGE_FORMAT identifier from the format string that was used to register the FIF.

FreeImage_GetFIFFromMime

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFIFFromMime(const char *mime);
```

Returns a FREE_IMAGE_FORMAT identifier from a MIME content type string (MIME stands for Multipurpose Internet Mail Extension).

```
FREE_IMAGE_FORMAT fif = FreeImage_GetFIFFromMime("image/png");
If(fif != FIF_UNKNOWN) {
  assert(fif == FIF_PNG);
}
```

FreeImage_GetFIFMimeType

```
DLL_API const char *DLL_CALLCONV FreeImage_GetFIFMimeType(FREE_IMAGE_FORMAT fif);
```

Given a FREE_IMAGE_FORMAT identifier, returns a MIME content type string (MIME stands for Multipurpose Internet Mail Extension).

FreeImage_GetFormatFromFIF

```
DLL_API const char *DLL_CALLCONV FreeImage_GetFormatFromFIF(FREE_IMAGE_FORMAT fif);
```

Returns the string that was used to register a plugin from the system assigned FREE_IMAGE_FORMAT.

FreeImage_GetFIFExtensionList

```
DLL_API const char *DLL_CALLCONV FreeImage_GetFIFExtensionList(FREE_IMAGE_FORMAT fif);
```

Returns a comma-delimited file extension list describing the bitmap formats the given plugin can read and/or write.

```
/**
Builds a series of string pairs that specify filters you can apply to load a file.
The filter string is to be used by a 'File Open' dialog box
(GetOpenFileName or CFileDialog).
@param szFilter Input and output parameter. szFilter is an array of char whose length
should be 2048 or more.
@return Returns the number of supported import formats
*/
int GetOpenFilterString(char *szFilter) {
  int i, iCount;
  char Filter[2048];
  char *token;

  // Build a string for 'All image files'
  Filter[0] = '\0';
  for(i = 0; i < FreeImage_GetFIFCount(); i++) {
    if(FreeImage_FIFSupportsReading((FREE_IMAGE_FORMAT)i)) {
      strcat(Filter, FreeImage_GetFIFExtensionList((FREE_IMAGE_FORMAT)i));
      strcat(Filter, ",");
    }
  }
  Filter[strlen(Filter)-1] = '\0';
  strcpy(szFilter, "All image files|");
  token = strtok(Filter, ",");
  while(token != NULL) {
    strcat(szFilter, "*.");
    strcat(szFilter, token);
    strcat(szFilter, ";");
    // get next token
    token = strtok(NULL, ",");
  }
  szFilter[strlen(szFilter)-1] = '|';

  // Build a string for 'All files'
  strcat(szFilter, "All Files (*.*)|*.*|");

  // Build a string for each format
  Filter[0] = '\0';
  iCount = 0;
  for(i = 0; i < FreeImage_GetFIFCount(); i++) {
    if(FreeImage_FIFSupportsReading((FREE_IMAGE_FORMAT)i)) {
      // Description
      sprintf(Filter, "%s (%s)|", FreeImage_GetFIFDescription((FREE_IMAGE_FORMAT)i),
      FreeImage_GetFIFExtensionList((FREE_IMAGE_FORMAT)i));
      strcat(szFilter, Filter);
      // Extension(s)
      strcpy(Filter, FreeImage_GetFIFExtensionList((FREE_IMAGE_FORMAT)i));
      token = strtok(Filter, ",");
      while(token != NULL) {
        strcat(szFilter, "*.");
        strcat(szFilter, token);
        strcat(szFilter, ";");
        // get next token
        token = strtok(NULL, ",");
      }
      szFilter[strlen(szFilter)-1] = '|';
      iCount++;
    }
  }
  strcat(szFilter, "|");

  return iCount;
}
```

FreeImage_GetFIFDescription

```
DLL_API const char *DLL_CALLCONV FreeImage_GetFIFDescription(FREE_IMAGE_FORMAT fif);
```

Returns a descriptive string that describes the bitmap formats the given plugin can read and/or write.

FreeImage_GetFIFRegExpr

```
DLL_API const char * DLL_CALLCONV FreeImage_GetFIFRegExpr(FREE_IMAGE_FORMAT fif);
```

Returns a regular expression string that can be used by a regular expression engine to identify the bitmap. FreeImageQt makes use of this function.

FreeImage_GetFIFFromFilename

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFIFFromFilename(const char
*filename);
```

This function takes a filename or a file-extension and returns the plugin that can read/write files with that extension in the form of a FREE_IMAGE_FORMAT identifier.

```
/** Generic image loader
@param lpszPathName Pointer to the full file name
@param flag Optional load flag constant
@return Returns the loaded dib if successful, returns NULL otherwise
*/
FIBITMAP* GenericLoader(const char* lpszPathName, int flag) {
  FREE_IMAGE_FORMAT fif = FIF_UNKNOWN;
  // check the file signature and deduce its format
  // (the second argument is currently not used by FreeImage)
  fif = FreeImage_GetFileType(lpszPathName, 0);
  if(fif == FIF_UNKNOWN) {
    // no signature ?
    // try to guess the file format from the file extension
    fif = FreeImage_GetFIFFromFilename(lpszPathName);
  }
  // check that the plugin has reading capabilities ...
  if((fif != FIF_UNKNOWN) && FreeImage_FIFSupportsReading(fif)) {
    // ok, let's load the file
    FIBITMAP *dib = FreeImage_Load(fif, lpszPathName, flag);
    // unless a bad file format, we are done !
    return dib;
  }
  return NULL;
}
```

FreeImage_GetFIFFromFilenameU

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_GetFIFFromFilenameU(const wchar_t
*filename);
```

This function works exactly like FreeImage_GetFIFFromFilename but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FIF_UNKNOWN.

FreeImage_FIFSupportsReading

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsReading(FREE_IMAGE_FORMAT fif);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can be used to load bitmaps, FALSE otherwise.

FreeImage_FIFSupportsWriting

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsWriting(FREE_IMAGE_FORMAT fif);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can be used to save bitmaps, FALSE otherwise.

```
/** Generic image writer
@param dib Pointer to the dib to be saved
@param lpszPathName Pointer to the full file name
@param flag Optional save flag constant
@return Returns true if successful, returns false otherwise
*/
bool GenericWriter(FIBITMAP* dib, const char* lpszPathName, int flag) {
  FREE_IMAGE_FORMAT fif = FIF_UNKNOWN;
  BOOL bSuccess = FALSE;

  // Try to guess the file format from the file extension
  fif = FreeImage_GetFIFFromFilename(lpszPathName);
  if(fif != FIF_UNKNOWN ) {
    // Check that the dib can be saved in this format
    BOOL bCanSave;

    FREE_IMAGE_TYPE image_type = FreeImage_GetImageType(dib);
    if(image_type == FIT_BITMAP) {
      // standard bitmap type
      // check that the plugin has sufficient writing
      // and export capabilities ...
      WORD bpp = FreeImage_GetBPP(dib);
      bCanSave = (FreeImage_FIFSupportsWriting(fif) &&
                  FreeImage_FIFSupportsExportBPP(fif, bpp));
    } else {
      // special bitmap type
      // check that the plugin has sufficient export capabilities
      bCanSave = FreeImage_FIFSupportsExportType(fif, image_type);
    }

    if(bCanSave) {
      bSuccess = FreeImage_Save(fif, dib, lpszPathName, flag);
    }
  }
  return (bSuccess == TRUE) ? true : false;
}
```

FreeImage_FIFSupportsExportType

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsExportType(FREE_IMAGE_FORMAT fif,
FREE_IMAGE_TYPE type);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can save a bitmap in the desired data type, returns FALSE otherwise. See the list of Supported file formats in the appendix for a list of plugins that can save non-standard images.

FreeImage_FIFSupportsExportBPP

1 4 8 16 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsExportBPP(FREE_IMAGE_FORMAT fif, int
bpp);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can save a bitmap in the desired bit depth, returns FALSE otherwise.

```
/**
Builds a series of string pairs that specify filters you can apply to save a file.
The filter string is to be used by a 'File Save As' dialog box (GetSaveFileName or
CFileDialog).
@param szFilter Input and output parameters. szFilter is an array of char whose
length should be 2048 or more.
@param bpp The bit depth of the image to be saved.
@param image_type The image type to be saved
@return Return the number of supported export formats
*/
int GetSaveAsFilterString(char *szFilter, WORD bpp, FREE_IMAGE_TYPE image_type) {
  int i, iCount;
  char Filter[2048];
  char *token;

  szFilter[0] = '\0';
  iCount = 0;

  // Build a string for each format
  for(i = 0; i < FreeImage_GetFIFCount(); i++) {
    // Check that the dib can be saved in this format
    BOOL bCanSave;

    FREE_IMAGE_FORMAT fif = (FREE_IMAGE_FORMAT)i;

    if(image_type == FIT_BITMAP) {
      // standard bitmap type
      bCanSave = (FreeImage_FIFSupportsWriting(fif) &&
                  FreeImage_FIFSupportsExportBPP(fif, bpp));
    } else {
      // special bitmap type
      bCanSave = FreeImage_FIFSupportsExportType(fif, image_type);
    }

    if(bCanSave) {
      // Handle the special case of PNM files
      strcpy(Filter, FreeImage_GetFormatFromFIF((FREE_IMAGE_FORMAT)i));
      if((bpp == 1) && (!strncmp(Filter, "PGM", 3) || !strncmp(Filter, "PPM", 3)))
        continue;
      if((bpp == 8) && (!strncmp(Filter, "PBM", 3) || !strncmp(Filter, "PPM", 3)))
        continue;
      if((bpp == 24) && (!strncmp(Filter, "PGM", 3) || !strncmp(Filter, "PBM", 3)))
        continue;

      // Description
      sprintf(Filter, "%s (%s)|", FreeImage_GetFIFDescription((FREE_IMAGE_FORMAT)i),
      FreeImage_GetFIFExtensionList((FREE_IMAGE_FORMAT)i));
      strcat(szFilter, Filter);
      // Extension(s)
      strcpy(Filter, FreeImage_GetFIFExtensionList((FREE_IMAGE_FORMAT)i));
      token = strtok(Filter, ",");
      while(token != NULL) {
        strcat(szFilter, "*.");
        strcat(szFilter, token);
        strcat(szFilter, ";");
        // get next token
        token = strtok(NULL, ",");
      }
      szFilter[strlen(szFilter)-1] = '|';
      iCount++;
    }
  }
  strcat(szFilter, "|");
  return iCount;
}
```

FreeImage_FIFSupportsICCProfiles

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsICCProfiles(FREE_IMAGE_FORMAT fif);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can load or save an ICC profile, returns FALSE otherwise.

```
// determine, whether profile support is present

if (FreeImage_FIFSupportsICCProfiles(FIF_TIFF)) {

  // profile support present

}
```

FreeImage_FIFSupportsNoPixels

```
DLL_API BOOL DLL_CALLCONV FreeImage_FIFSupportsNoPixels(FREE_IMAGE_FORMAT fif);
```

Returns TRUE if the plugin belonging to the given FREE_IMAGE_FORMAT can load a file using the FIF_LOAD_NOPIXELS load flag. If TRUE, a loader can load header only data and possibly some metadata;

```
// determine, whether 'header only' support is present

if (FreeImage_FIFSupportsNoPixels(FIF_JPEG)) {

  // 'header only' support present

}
```

FreeImage_RegisterLocalPlugin

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_RegisterLocalPlugin(FI_InitProc
proc_address, const char *format FI_DEFAULT(0), const char *description FI_DEFAULT(0),
const char *extension FI_DEFAULT(0), const char *regexpr FI_DEFAULT(0));
```

Registers a new plugin to be used in FreeImage. The plugin is residing directly in the application driving FreeImage. The first parameter is a pointer to a function that is used to initialise the plugin. The initialization function is responsible for filling in a Plugin structure and storing a system-assigned format identification number used for message logging.

```
static int s_format_id;

void stdcall
Init(Plugin *plugin, int format_id) {
    s_format_id = format_id;

    // pointer to a function that returns a type-string
    // for the bitmap. For example, a plugin that loads
    // BMPs returns the string "BMP".

    plugin->format_proc = Format;

    // pointer to a function that returns a descriptive
    // string for the bitmap type. For example, a plugin
    // that loads BMPs may return "Windows or OS/2 Bitmap"

    plugin->description_proc = Description;

    // pointer to a function that returns a comma delimited
    // list of possible file extension that are valid for
    // this plugin. A JPEG plugin would return "jpeg,jif,jfif"

    plugin->extension_proc = Extension;

    // pointer to a function that is used to load the bitmap

    plugin->load_proc = Load;

    // pointer to a function that is used to save the bitmap

    plugin->save_proc = Save;

    // pointer to a function that will try to identify a
    // bitmap by looking at the first few bytes of the bitmap.

    plugin->validate_proc = Validate;
}
```

FreeImage_RegisterExternalPlugin

```
DLL_API FREE_IMAGE_FORMAT DLL_CALLCONV FreeImage_RegisterExternalPlugin(const char
*path, const char *format FI_DEFAULT(0), const char *description FI_DEFAULT(0), const
char *extension FI_DEFAULT(0), const char *regexpr FI_DEFAULT(0));
```

Registers a new plugin to be used in FreeImage. The plugin is residing in a DLL. Functionally this function is the same as FreeImage_RegisterLocalPlugin, but now FreeImage calls an Init function in a DLL instead of a local function in an application. The Init function must be called "Init" and must use the stdcall calling convention.

Multipage functions

FreeImage features a set of functions that can be used to manipulate pages in a multi-page bitmap format. Currently TIFF, ICO and GIF formats are supported for this. The multi-page API makes it possible to access and change pages in a multi-bitmap, delete pages and change the order of pages. All of this is offered with a minimum implementation in a plugin and low requirement of memory through a sophisticated, compressing cache mechanism.

In the multipage API, whenever a 'page' parameter is needed by a function, it is always 0-based.

FreeImage_OpenMultiBitmap

```
DLL_API FIMULTIBITMAP * DLL_CALLCONV FreeImage_OpenMultiBitmap(FREE_IMAGE_FORMAT fif,
const char *filename, BOOL create_new, BOOL read_only, BOOL keep_cache_in_memory
FI_DEFAULT(FALSE), int flags FI_DEFAULT(0));
```

Opens a multi-page bitmap.

The first parameter tells FreeImage the bitmap-type of bitmap to be opened. Currently FIF_TIFF, FIF_ICO and FIF_GIF are supported. The second parameter specifies the name of the bitmap. When the third parameter is TRUE, it means that a new bitmap will be created rather than an existing one being opened. When the fourth parameter is TRUE the bitmap is opened read-only. The keep_cache_in_memory parameter is one purely for performance. When it is TRUE, all gathered bitmap data in the page manipulation process is kept in memory, otherwise it is lazily flushed to a temporary file on the hard disk in 64 Kb blocks. Note that depending on the amount of manipulation being performed and the size of the bitmap, the temporary data can become quite large. It's advised to lazily flush to disc. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

FreeImage_OpenMultiBitmapFromHandle

```
DLL_API FIMULTIBITMAP * DLL_CALLCONV
FreeImage_OpenMultiBitmapFromHandle(FREE_IMAGE_FORMAT fif, FreeImageIO *io, fi_handle
handle, int flags FI_DEFAULT(0));
```

This function lets one open an existing multi-page bitmap from a handle in *read-only mode*. As with FreeImage_LoadFromHandle, a pointer to a FreeImageIO structure and a fi_handle must be specified. The actual implementation of FreeImage_CloseMultiBitmap is sufficient, to close such a multipage bitmap opened from a handle.

Altough the source handle is opened in read-only mode, using this function, multi-page handles support *read* or *read*/*write* operations. When you modify a multi-page file using functions such as FreeImage_AppendPage, FreeImage_InsertPage, FreeImage_MovePage or FreeImage_DeletePage, changes are transparently stored into a *memory cache* so that these changes can be later saved to an output stream. The source stream is left unmodified: closing the source stream will not change it. You thus need to use a kind of "save as …" function to save your changes.

```
static unsigned DLL_CALLCONV
myReadProc(void *buffer, unsigned size, unsigned count, fi_handle handle) {
    return (unsigned)fread(buffer, size, count, (FILE *)handle);
}

static unsigned DLL_CALLCONV
myWriteProc(void *buffer, unsigned size, unsigned count, fi_handle handle) {
    return (unsigned)fwrite(buffer, size, count, (FILE *)handle);
}

static int DLL_CALLCONV
mySeekProc(fi_handle handle, long offset, int origin) {
    return fseek((FILE *)handle, offset, origin);
}

static long DLL_CALLCONV
myTellProc(fi_handle handle) {
    return ftell((FILE *)handle);
}

BOOL testStreamMultiPageOpen(const char *input, int flags) {
    // initialize your own IO functions

    FreeImageIO io;

    io.read_proc  = myReadProc;
    io.write_proc = myWriteProc;
    io.seek_proc  = mySeekProc;
    io.tell_proc  = myTellProc;

    BOOL bSuccess = FALSE;

    // Open src stream in read-only mode
    FILE *file = fopen(input, "r+b");
    if (file != NULL) {
        // Open the multi-page file
        FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromHandle(&io,
(fi_handle)file);
        FIMULTIBITMAP *src = FreeImage_OpenMultiBitmapFromHandle(fif, &io,
(fi_handle)file, flags);

        if(src) {
            // get the page count
            int count = FreeImage_GetPageCount(src);
            assert(count > 1);

            // delete page 0 (modifications are stored to the cache)
            FreeImage_DeletePage(src, 0);

            // Close src file (nothing is done, the cache is cleared)
            bSuccess = FreeImage_CloseMultiBitmap(src, 0);
            assert(bSuccess);
        }

        // Close the src stream
        fclose(file);

        return bSuccess;
    }

    return bSuccess;
}
```

FreeImage_SaveMultiBitmapToHandle

```
DLL_API BOOL DLL_CALLCONV FreeImage_SaveMultiBitmapToHandle(FREE_IMAGE_FORMAT fif,
FIMULTIBITMAP *bitmap, FreeImageIO *io, fi_handle handle, int flags FI_DEFAULT(0));
```

Saves a multi-page bitmap into the specified handle. The handle must be set to the correct position before calling the function.

As with FreeImage_SaveToHandle, a pointer to a FreeImageIO structure and a fi_handle must be specified.

```
BOOL testStreamMultiPageSave(const char *input, const char *output, int input_flag,
int output_flag) {
  // initialize your own IO functions

  FreeImageIO io;

  io.read_proc  = myReadProc;
  io.write_proc = myWriteProc;
  io.seek_proc  = mySeekProc;
  io.tell_proc  = myTellProc;

  BOOL bCreateNew = FALSE;
  BOOL bReadOnly = TRUE;
  BOOL bMemoryCache = TRUE;

  // Open src file (read-only, use memory cache)
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(input);
  FIMULTIBITMAP *src = FreeImage_OpenMultiBitmap(fif, input, bCreateNew, bReadOnly,
bMemoryCache, input_flag);

  if(src) {
    // Open dst stream in read/write mode
    FILE *file = fopen(output, "w+b");
    if (file != NULL) {
      // Save the multi-page file to the stream
      BOOL bSuccess = FreeImage_SaveMultiBitmapToHandle(fif, src, &io,
(fi_handle)file, output_flag);
      assert(bSuccess);

      // Close the dst stream
      fclose(file);

      // Close src file
      FreeImage_CloseMultiBitmap(src, 0);

      return TRUE;
    }

    // Close src
    FreeImage_CloseMultiBitmap(src, 0);
  }

  return FALSE;
}
```

The following sample shows how to modify an input stream and save your modifications into an output stream. Note that the input stream is left unchanged: all modifications are stored into a cache; this cache is then used to apply your modification on saving.

```
BOOL testStreamMultiPageOpenSave(const char *input, const char *output, int
input_flag, int output_flag) {
  // initialize your own IO functions

  FreeImageIO io;

  io.read_proc  = myReadProc;
  io.write_proc = myWriteProc;
  io.seek_proc  = mySeekProc;
  io.tell_proc  = myTellProc;

  BOOL bSuccess = FALSE;

  // Open src stream in read-only mode
  FILE *src_file = fopen(input, "r+b");
  assert(src_file);
  if (src_file != NULL) {
    // Open the multi-page file
    FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromHandle(&io,
(fi_handle)src_file);
    FIMULTIBITMAP *src = FreeImage_OpenMultiBitmapFromHandle(fif, &io,
(fi_handle)src_file, input_flag);

    if(src) {
      // get the page count
      int count = FreeImage_GetPageCount(src);
      assert(count > 2);

      // Load the bitmap at position '2'
      FIBITMAP *dib = FreeImage_LockPage(src, 2);
      if(dib) {
        FreeImage_Invert(dib);
        // Unload the bitmap (apply change to src, modifications are stored to the
cache)
        FreeImage_UnlockPage(src, dib, TRUE);
      }

      // delete page 0 (modifications are stored to the cache)
      FreeImage_DeletePage(src, 0);

      // insert a new page at position '0' (modifications are stored to the cache)
      FIBITMAP *page = createZonePlateImage(512, 512, 128);
      FreeImage_InsertPage(src, 0, page);
      FreeImage_Unload(page);

      // Open dst stream in read/write mode
      FILE *dst_file = fopen(output, "w+b");
      assert(dst_file);
      if (dst_file != NULL) {
        // Save the multi-page file to the stream (modifications are applied)
        BOOL bSuccess = FreeImage_SaveMultiBitmapToHandle(fif, src, &io,
(fi_handle)dst_file, output_flag);
        assert(bSuccess);

        // Close the dst stream
        fclose(dst_file);
      }

      // Close src file (nothing is done, the cache is cleared)
      bSuccess = FreeImage_CloseMultiBitmap(src, 0);
      assert(bSuccess);
    }

    // Close the src stream
    fclose(src_file);

    return bSuccess;
  }

  return FALSE;
}
```

FreeImage_CloseMultiBitmap

```
DLL_API BOOL DLL_CALLCONV FreeImage_CloseMultiBitmap(FIMULTIBITMAP *bitmap, int flags
FI_DEFAULT(0));
```

Closes a previously opened multi-page bitmap and, when the bitmap was not opened read-only, applies any changes made to it.

The flags parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters (see Table 4). Some bitmap savers can receive parameters to change the saving behaviour. When the parameter is not available or unused you can pass the value 0 or <TYPE_OF_BITMAP>_DEFAULT (e.g. TIFF_DEFAULT, ICO_DEFAULT, etc).

FreeImage_GetPageCount

```
DLL_API int DLL_CALLCONV FreeImage_GetPageCount(FIMULTIBITMAP *bitmap);
```

Returns the number of pages currently available in the multi-paged bitmap.

FreeImage_AppendPage

```
DLL_API void DLL_CALLCONV FreeImage_AppendPage(FIMULTIBITMAP *bitmap, FIBITMAP *data);
```

Appends a new page to the end of the bitmap.

FreeImage_InsertPage

```
DLL_API void DLL_CALLCONV FreeImage_InsertPage(FIMULTIBITMAP *bitmap, int page,
FIBITMAP *data);
```

Inserts a new page before the given position in the bitmap. Page has to be a number smaller than the current number of pages available in the bitmap.

FreeImage_DeletePage

```
DLL_API void DLL_CALLCONV FreeImage_DeletePage(FIMULTIBITMAP *bitmap, int page);
```

Deletes the page on the given position.

FreeImage_LockPage

```
DLL_API FIBITMAP * DLL_CALLCONV FreeImage_LockPage(FIMULTIBITMAP *bitmap, int page);
```

Locks a page in memory for editing. The page can now be saved to a different file or inserted into another multi-page bitmap. When you are done with the bitmap you have to call FreeImage_UnlockPage to give the page back to the bitmap and/or apply any changes made in the page.

It is forbidden to use *FreeImage_Unload* on a locked page: you must use *FreeImage_UnlockPage* instead.

FreeImage_UnlockPage

```
DLL_API void DLL_CALLCONV FreeImage_UnlockPage(FIMULTIBITMAP *bitmap, FIBITMAP *data,
BOOL changed);
```

Unlocks a previously locked page and gives it back to the multi-page engine. When the last parameter is TRUE, the page is marked changed and the new page data is applied in the multi-page bitmap.

```
bool CloneMultiPage(FREE_IMAGE_FORMAT fif, char *input, char *output, int output_flag)
{

  BOOL bMemoryCache = TRUE;

  // Open src file (read-only, use memory cache)
  FIMULTIBITMAP *src = FreeImage_OpenMultiBitmap(fif, input, FALSE, TRUE,
bMemoryCache);

  if(src) {
    // Open dst file (creation, use memory cache)
    FIMULTIBITMAP *dst = FreeImage_OpenMultiBitmap(fif, output, TRUE, FALSE,
bMemoryCache);

    // Get src page count
    int count = FreeImage_GetPageCount(src);

    // Clone src to dst
    for(int page = 0; page < count; page++) {
      // Load the bitmap at position 'page'
      FIBITMAP *dib = FreeImage_LockPage(src, page);
      if(dib) {
        // add a new bitmap to dst
        FreeImage_AppendPage(dst, dib);
        // Unload the bitmap (do not apply any change to src)
        FreeImage_UnlockPage(src, dib, FALSE);
      }
    }

    // Close src
    FreeImage_CloseMultiBitmap(src, 0);
    // Save and close dst
    FreeImage_CloseMultiBitmap(dst, output_flag);

    return true;
  }

  return false;
}
```

FreeImage_MovePage

```
DLL_API BOOL DLL_CALLCONV FreeImage_MovePage(FIMULTIBITMAP *bitmap, int target, int
source);
```

Moves the source page to the position of the target page. Returns TRUE on success, FALSE on failure.

FreeImage_GetLockedPageNumbers

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetLockedPageNumbers(FIMULTIBITMAP *bitmap, int
*pages, int *count);
```

Returns an array of page-numbers that are currently locked in memory. When the pages parameter is NULL, the size of the array is returned in the count variable. You can then allocate the array of the desired size and call FreeImage_GetLockedPageNumbers again to populate the array.

Memory I/O streams

Memory I/O routines use a specialized version of the FreeImageIO structure, targeted to save/load FIBITMAP images to/from a memory stream. Just like you would do with a file stream. Memory file streams support loading and saving of FIBITMAP in a memory file (managed internally by FreeImage). They also support seeking and telling in the memory file.

Examples of using these functions would be to store image files as blobs in a database, or to write image files to a Internet stream.

FreeImage_OpenMemory

```
DLL_API FIMEMORY *DLL_CALLCONV FreeImage_OpenMemory(BYTE *data FI_DEFAULT(0), DWORD
size_in_bytes FI_DEFAULT(0));
```

Open a memory stream. The function returns a pointer to the opened memory stream.

When called with default arguments (0), this function opens a memory stream for read / write access. The stream will support loading and saving of FIBITMAP in a memory file (managed internally by FreeImage). It will also support seeking and telling in the memory file.

This function can also be used to wrap a memory buffer provided by the application driving FreeImage. A buffer containing image data is given as function arguments *data* (start of the buffer) and *size_in_bytes* (buffer size in bytes). A memory buffer wrapped by FreeImage is read only. Images can be loaded but cannot be saved.

FreeImage_CloseMemory

```
DLL_API void DLL_CALLCONV FreeImage_CloseMemory(FIMEMORY *stream);
```

Close and free a memory stream.

When the stream is managed by FreeImage, the memory file is destroyed. Otherwise (wrapped buffer), it's destruction is left to the application driving FreeImage.

You always need to call this function once you're done with a memory stream (whatever the way you opened the stream), or you will have a memory leak.

FreeImage_LoadFromMemory

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_LoadFromMemory(FREE_IMAGE_FORMAT fif,
FIMEMORY *stream, int flags FI_DEFAULT(0));
```

This function does for memory streams what FreeImage_Load does for file streams. FreeImage_LoadFromMemory decodes a bitmap, allocates memory for it and then returns it as a FIBITMAP. The first parameter defines the type of bitmap to be loaded. For example, when FIF_BMP is passed, a BMP file is loaded into memory (an overview of possible FREE_IMAGE_FORMAT constants is available in Table 1). The second parameter tells FreeImage the memory stream it has to decode. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

Some bitmap loaders can receive parameters to change the loading behaviour (see Table 3). When the parameter is not available or unused you can pass the value 0 or <TYPE_OF_BITMAP>_DEFAULT (e.g. BMP_DEFAULT, ICO_DEFAULT, etc).

```
void testLoadMemIO(const char *lpszPathName) {
  struct stat buf;
  int result;

  // get data associated with lpszPathName
  result = stat(lpszPathName, &buf);
  if(result == 0) {
    // allocate a memory buffer and load temporary data
    BYTE *mem_buffer = (BYTE*)malloc(buf.st_size * sizeof(BYTE));
    if(mem_buffer) {
      FILE *stream = fopen(lpszPathName, "rb");
      if(stream) {
        fread(mem_buffer, sizeof(BYTE), buf.st_size, stream);
        fclose(stream);

        // attach the binary data to a memory stream
        FIMEMORY *hmem = FreeImage_OpenMemory(mem_buffer, buf.st_size);

        // get the file type
        FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromMemory(hmem, 0);

        // load an image from the memory stream
        FIBITMAP *check = FreeImage_LoadFromMemory(fif, hmem, 0);

        // save as a regular file
        FreeImage_Save(FIF_PNG, check, "blob.png", PNG_DEFAULT);
        FreeImage_Unload(check);

        // always close the memory stream
        FreeImage_CloseMemory(hmem);

      }
    }
    // user is responsible for freeing the data
    free(mem_buffer);
  }
}
```

FreeImage_SaveToMemory

```
DLL_API BOOL DLL_CALLCONV FreeImage_SaveToMemory(FREE_IMAGE_FORMAT fif, FIBITMAP *dib,
FIMEMORY *stream, int flags FI_DEFAULT(0));
```

This function does for memory streams what FreeImage_Save does for file streams. FreeImage_SaveToMemory saves a previously loaded FIBITMAP to a memory file managed by FreeImage. The first parameter defines the type of the bitmap to be saved. For example, when FIF_BMP is passed, a BMP file is saved (an overview of possible FREE_IMAGE_FORMAT constants is available in Table 1). The second parameter is the memory stream where the bitmap must be saved. When the memory file pointer point to the beginning of the memory file, any existing data is overwritten. Otherwise, you can save multiple images on the same stream.

Note that some bitmap save plugins have restrictions on the bitmap types they can save. For example, the JPEG plugin can only save 24-bit and 8-bit greyscale bitmaps. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

Some bitmap savers can receive parameters to change the saving behaviour (see Table 4). When the parameter is not available or unused you can pass the value 0 or <TYPE_OF_BITMAP>_DEFAULT (e.g. BMP_DEFAULT, ICO_DEFAULT, etc).

```
void testSaveMemIO(const char *lpszPathName) {
  FIMEMORY *hmem = NULL;

  // load and decode a regular file
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(lpszPathName);
  FIBITMAP *dib = FreeImage_Load(fif, lpszPathName, 0);

  // open a memory stream
  hmem = FreeImage_OpenMemory();

  // encode and save the image to the memory
  FreeImage_SaveToMemory(fif, dib, hmem, 0);

  // at this point, hmem contains the entire data in memory stored in fif format.
  // the amount of space used by the memory is equal to file_size
  FreeImage_SeekMemory(hmem, 0, SEEK_END);
  long file_size = FreeImage_TellMemory(hmem);
  printf("File size : %ld\n", file_size);


  // its easy to load an image from memory as well

  // seek to the start of the memory stream
  FreeImage_SeekMemory(hmem, 0L, SEEK_SET);

  // get the file type
  FREE_IMAGE_FORMAT mem_fif = FreeImage_GetFileTypeFromMemory(hmem, 0);

  // load an image from the memory handle
  FIBITMAP *check = FreeImage_LoadFromMemory(mem_fif, hmem, 0);

  // save as a regular file
  FreeImage_Save(FIF_PNG, check, "dump.png", PNG_DEFAULT);

  // make sure to close the stream since FreeImage_SaveToMemory
  // will cause internal memory allocations and this is the only
  // way to free this allocated memory
  FreeImage_CloseMemory(hmem);

  FreeImage_Unload(check);
  FreeImage_Unload(dib);
}
```

FreeImage_AcquireMemory

```
DLL_API BOOL DLL_CALLCONV FreeImage_AcquireMemory(FIMEMORY *stream, BYTE **data, DWORD
*size_in_bytes);
```

Provides a direct buffer access to a memory stream. Upon entry, *stream* is the target memory stream, returned value *data* is a pointer to the memory buffer, returned value *size_in_bytes* is the buffer size in bytes. The function returns TRUE if successful, FALSE otherwise.

When the memory stream is managed internally by FreeImage, the data pointer returned by *FreeImage_AcquireMemory* may become invalid as soon as you call FreeImage_SaveToMemory.

```
void testAcquireMemIO(const char *lpszPathName) {
  FIMEMORY *hmem = NULL;

  // load a regular file
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(lpszPathName);
  FIBITMAP *dib = FreeImage_Load(fif, lpszPathName, 0);

  // open and allocate a memory stream
  hmem = FreeImage_OpenMemory();

  // save the image to a memory stream
  FreeImage_SaveToMemory(FIF_PNG, dib, hmem, PNG_DEFAULT);

  FreeImage_Unload(dib);

  // get the buffer from the memory stream
  BYTE *mem_buffer = NULL;
  DWORD size_in_bytes = 0;

  FreeImage_AcquireMemory(hmem, &mem_buffer, &size_in_bytes);

  // save the buffer to a file stream
  FILE *stream = fopen("buffer.png", "wb");
  if(stream) {
    fwrite(mem_buffer, sizeof(BYTE), size_in_bytes, stream);
    fclose(stream);
  }

  // close and free the memory stream
  FreeImage_CloseMemory(hmem);

}
```

FreeImage_TellMemory

```
DLL_API long DLL_CALLCONV FreeImage_TellMemory(FIMEMORY *stream);
```

Gets the current position of a memory pointer. Upon entry, *stream* is the target memory stream. The function returns the current file position if successful, -1 otherwise.

FreeImage_SeekMemory

```
DLL_API BOOL DLL_CALLCONV FreeImage_SeekMemory(FIMEMORY *stream, long offset, int
origin);
```

Moves the memory pointer to a specified location. A description of parameters follows:

stream Pointer to the target memory stream

offset Number of bytes from *origin*

origin Initial position

The function returns TRUE if successful, returns FALSE otherwise

The *FreeImage_SeekMemory* function moves the memory file pointer (if any) associated with *stream* to a new location that is *offset* bytes from *origin*. The next operation on the stream takes place at the new location. On a stream managed by FreeImage, the next operation can be either a read or a write. The argument *origin* must be one of the following constants, defined in STDIO.H (and also in FreeImage.h) :

SEEK_CUR Current position of file pointer.

SEEK_END End of file.

SEEK_SET Beginning of file.

FreeImage_ReadMemory

```
DLL_API unsigned DLL_CALLCONV FreeImage_ReadMemory(void *buffer, unsigned size,
unsigned count, FIMEMORY *stream);
```

Reads data from a memory stream.

The FreeImage_ReadMemory function reads up to *count* items of *size* bytes from the input memory stream and stores them in *buffer*. The memory pointer associated with *stream* is increased by the number of bytes actually read.

The function returns the number of full items actually read, which may be less than *count* if an error occurs or if the end of the stream is encountered before reaching *count*.

FreeImage_WriteMemory

```
DLL_API unsigned DLL_CALLCONV FreeImage_WriteMemory(const void *buffer, unsigned size,
unsigned count, FIMEMORY *stream);
```

Writes data to a memory stream.

The FreeImage_WriteMemory function writes up to *count* items, of *size* length each, from *buffer* to the output memory stream. The memory pointer associated with *stream* is incremented by the number of bytes actually written.

The function returns the number of full items actually written, which may be less than *count* if an error occurs.

FreeImage_ReadMemory and *FreeImage_WriteMemory* are useful whenever you need a memory stream to store temporary data. When combined with *FreeImage_SeekMemory* and *FreeImage_TellMemory*, this represents an alternative to the use of temporary files.

FreeImage_LoadMultiBitmapFromMemory

```
DLL_API FIMULTIBITMAP *DLL_CALLCONV
FreeImage_LoadMultiBitmapFromMemory(FREE_IMAGE_FORMAT fif, FIMEMORY *stream, int flags
FI_DEFAULT(0));
```

Open a multi-page bitmap from a memory stream.

This function is similar to the FreeImage_OpenMultiBitmap function, with the difference that you will be able to open a multi-page file, in *read-only mode*, from a memory stream instead of a file stream.

The first parameter tells FreeImage the bitmap-type of bitmap to be opened. Currently FIF_TIFF, FIF_ICO and FIF_GIF are supported. The second parameter tells FreeImage the memory stream it has to decode. The last parameter is used to change the behaviour or enable a feature in the bitmap plugin. Each plugin has its own set of parameters.

Altough the source memory handle is opened in read-only mode, using this function, multi-page memory handles support read or read/write operations. When you modify a multi-page file using functions such as FreeImage_AppendPage, FreeImage_InsertPage, FreeImage_MovePage or FreeImage_DeletePage, changes are transparently stored into a memory cache so that these changes can be later saved to an output stream. The source memory stream is left unmodified: closing the source stream will not change it. You thus need to use a kind of “save as …” function to save your changes.

```
static BOOL
extractPagesFromMemory(FREE_IMAGE_FORMAT fif, FIMEMORY *stream) {
  char filename[256];
  // open the multipage bitmap stream as read-only
  FIMULTIBITMAP *src = FreeImage_LoadMultiBitmapFromMemory(fif, stream, 0);
  if(src) {
    // get the page count
    int count = FreeImage_GetPageCount(src);
    // extract all pages
    for(int page = 0; page < count; page++) {
      // load the bitmap at position 'page'
      FIBITMAP *dib = FreeImage_LockPage(src, page);
      if(dib) {
        // save the page
        sprintf(filename, "page%d.%s", page, FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif, dib, filename, 0);
        // Unload the bitmap (do not apply any change to src)
        FreeImage_UnlockPage(src, dib, FALSE);
      } else {
        // an error occured: free the multipage bitmap handle and return
        FreeImage_CloseMultiBitmap(src, 0);
        return FALSE;
      }
    }
  }
  // make sure to close the multipage bitmap handle on exit
  return FreeImage_CloseMultiBitmap(src, 0);
}

void testLoadMultiBitmapFromMemory(const char *lpszPathName) {
  struct stat buf;
  int result;

  // get data associated with lpszPathName
  result = stat(lpszPathName, &buf);
  if(result == 0) {
    // allocate a memory buffer and load temporary data
    BYTE *mem_buffer = (BYTE*)malloc(buf.st_size * sizeof(BYTE));
    if(mem_buffer) {
      FILE *stream = fopen(lpszPathName, "rb");
      if(stream) {
        fread(mem_buffer, sizeof(BYTE), buf.st_size, stream);
        fclose(stream);

        // attach the binary data to a memory stream
        FIMEMORY *hmem = FreeImage_OpenMemory(mem_buffer, buf.st_size);

        // get the file type
        FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromMemory(hmem, 0);

        // extract pages
        BOOL bSuccess = extractPagesFromMemory(fif, hmem);
        assert(bSuccess);

        // close the stream
        FreeImage_CloseMemory(hmem);

      }
    }
    // user is responsible for freeing the data
    free(mem_buffer);
  }
}
```

FreeImage_SaveMultiBitmapToMemory

```
DLL_API BOOL DLL_CALLCONV FreeImage_SaveMultiBitmapToMemory(FREE_IMAGE_FORMAT fif,
FIMULTIBITMAP *bitmap, FIMEMORY *stream, int flags);
```

Saves a multi-page bitmap into the specified memory handle. The handle must be set to the correct position before calling the function.

```
BOOL testSaveMultiBitmapToMemory(const char *input, const char *output, int
output_flag) {
  BOOL bSuccess;

  BOOL bCreateNew = FALSE;
  BOOL bReadOnly = TRUE;
  BOOL bMemoryCache = TRUE;

  // Open src file (read-only, use memory cache)
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(input);
  FIMULTIBITMAP *src = FreeImage_OpenMultiBitmap(fif, input, bCreateNew, bReadOnly,
bMemoryCache);

  if(src) {
    // open and allocate a memory stream
    FIMEMORY *dst_memory = FreeImage_OpenMemory();

    // save the file to memory
    bSuccess = FreeImage_SaveMultiBitmapToMemory(fif, src, dst_memory, output_flag);
    assert(bSuccess);

    // src is no longer needed: close and free src file
    FreeImage_CloseMultiBitmap(src, 0);

    // get the buffer from the memory stream
    BYTE *mem_buffer = NULL;
    DWORD size_in_bytes = 0;

    bSuccess = FreeImage_AcquireMemory(dst_memory, &mem_buffer, &size_in_bytes);
    assert(bSuccess);

    // save the buffer in a file stream
    FILE *stream = fopen(output, "wb");
    if(stream) {
      fwrite(mem_buffer, sizeof(BYTE), size_in_bytes, stream);
      fclose(stream);
    }

    // close and free the memory stream
    FreeImage_CloseMemory(dst_memory);

    return TRUE;
  }

  return FALSE;
}
```

The following sample shows how to modify an input memory stream and save your modifications into an output memory stream. Note that the input stream is left unchanged: all modifications are stored into a cache; this cache is then used to apply your modification on saving.

```
static BOOL
loadBuffer(const char *lpszPathName, BYTE **buffer, DWORD *length) {
  struct stat file_info;
  int result;

  // get data associated with lpszPathName
  result = stat(lpszPathName, &file_info);
  if(result == 0) {
    // allocate a memory buffer and load temporary data
    *buffer = (BYTE*)malloc(file_info.st_size * sizeof(BYTE));
    if(*buffer) {
      FILE *stream = fopen(lpszPathName, "rb");
      if(stream) {
        *length = (DWORD)fread(*buffer, sizeof(BYTE), file_info.st_size, stream);
        fclose(stream);

        return TRUE;
      }
    }
  }

  return FALSE;
}

BOOL testMemoryStreamMultiPageOpenSave(const char *lpszPathName, char *output, int
input_flag, int output_flag) {
  BOOL bSuccess = FALSE;

  BYTE *buffer = NULL;
  DWORD buffer_size = 0;

  // load source stream as a buffer, i.e.
  // allocate a memory buffer and load temporary data
  bSuccess = loadBuffer(lpszPathName, &buffer, &buffer_size);
  assert(bSuccess);

  // attach the binary data to a memory stream
  FIMEMORY *src_stream = FreeImage_OpenMemory(buffer, buffer_size);
  assert(src_stream);

  // open the multipage bitmap stream
  FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromMemory(src_stream, 0);
  FIMULTIBITMAP *src = FreeImage_LoadMultiBitmapFromMemory(fif, src_stream,
input_flag);

  // apply some modifications (everything being stored to the cache) ...

  if(src) {
    // get the page count
    int count = FreeImage_GetPageCount(src);
    assert(count > 2);

    // Load the bitmap at position '2'
    FIBITMAP *dib = FreeImage_LockPage(src, 2);
    if(dib) {
      FreeImage_Invert(dib);
      // Unload the bitmap (apply change to src, modifications are stored to the
cache)
      FreeImage_UnlockPage(src, dib, TRUE);
    }

    // delete page 0 (modifications are stored to the cache)
    FreeImage_DeletePage(src, 0);

    // insert a new page at position '0' (modifications are stored to the cache)
    FIBITMAP *page = createZonePlateImage(512, 512, 128);
    FreeImage_InsertPage(src, 0, page);
    FreeImage_Unload(page);
  }

  // save the modification into the output stream ...

  if(src) {
    // open and allocate a memory stream
    FIMEMORY *dst_stream = FreeImage_OpenMemory();
    assert(dst_stream);

    // save the file to memory
    FreeImage_SaveMultiBitmapToMemory(fif, src, dst_stream, output_flag);

    // src is no longer needed
```

```
    // close and free the memory stream
    FreeImage_CloseMemory(src_stream);
    // close and free src file (nothing is done, the cache is cleared)
    FreeImage_CloseMultiBitmap(src, 0);

    // at this point, the input buffer is no longer needed
    // !!! user is responsible for freeing the initial source buffer !!!
    free(buffer); buffer = NULL;

    // get the dst buffer from the memory stream
    BYTE *dst_buffer = NULL;
    DWORD size_in_bytes = 0;

    FreeImage_AcquireMemory(dst_stream, &dst_buffer, &size_in_bytes);

    // save the buffer in a file stream
    FILE *stream = fopen(output, "wb");
    if(stream) {
      fwrite(dst_buffer, sizeof(BYTE), size_in_bytes, stream);
      fclose(stream);
    }

    // close and free the memory stream
    FreeImage_CloseMemory(dst_stream);

    return TRUE;
  }

  if(buffer) {
    free(buffer);
  }

  return FALSE;
}
```

Compression functions

FreeImage uses many Open Source third party libraries in order to load or save complex image file formats. Among these libraries, some of them, such as the ZLib library, deal with compression / decompression of memory buffers. Since this feature may be useful in many applications and not only for image compression, FreeImage provides an interface to the main functionalities of these libraries.

Currently, only ZLib compression is supported. Other compression algorithms may be added with future releases of FreeImage.

FreeImage_ZLibCompress

```
DLL_API DWORD DLL_CALLCONV FreeImage_ZLibCompress(BYTE *target, DWORD target_size,
BYTE *source, DWORD source_size);
```

Compresses a source buffer into a target buffer, using the ZLib library. Upon entry, target_size is the total size of the destination buffer, which must be at least 0.1% larger than source_size plus 12 bytes.

The function returns the actual size of the compressed buffer, or returns 0 if an error occurred.

```
BYTE *data = NULL;
DWORD original_size = 0;
// ...
data = (BYTE*)malloc(original_size * sizeof(BYTE));
// ...

// compress data
DWORD compressed_size = (DWORD)((double) original_size + (0.1 * (double)
original_size) + 12);
BYTE *compressed_data = (BYTE*)malloc(compressed_size * sizeof(BYTE));
compressed_size = FreeImage_ZLibCompress(compressed_data, compressed_size, data,
original_size);

// write data to disk
fwrite(&original_size, sizeof(DWORD), 1, stream);
fwrite(&compressed_size, sizeof(DWORD), 1, stream);
fwrite(compressed_data, sizeof(BYTE), compressed_size, stream);

free(compressed_data);
```

FreeImage_ZLibUncompress

```
DLL_API DWORD DLL_CALLCONV FreeImage_ZLibUncompress(BYTE *target, DWORD target_size,
BYTE *source, DWORD source_size);
```

Decompresses a source buffer into a target buffer, using the ZLib library. Upon entry, target_size is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.

The function returns the actual size of the uncompressed buffer, or returns 0 if an error occurred.

```
BYTE *data = NULL;
DWORD original_size = 0, compressed_size = 0;
// ...
// read data from disk
fread(&original_size, sizeof(DWORD), 1, stream);
fread(&compressed_size, sizeof(DWORD), 1, stream);
data = (BYTE*)malloc(original_size * sizeof(BYTE));
compressed_data = (BYTE*)malloc(compressed_size * sizeof(BYTE));
fread(compressed_data, sizeof(BYTE), compressed_size, stream);

// decompress data
DWORD size = 0;
size = FreeImage_ZLibUncompress(data, original_size, compressed_data,
compressed_size);
assert(size == original_size);
free(compressed_data);
```

FreeImage_ZLibGZip

```
DLL_API DWORD DLL_CALLCONV FreeImage_ZLibGZip(BYTE *target, DWORD target_size, BYTE
*source, DWORD source_size);
```

Compresses a source buffer into a target buffer, using the ZLib library. Upon entry, target_size is the total size of the destination buffer, which must be at least 0.1% larger than source_size plus 24 bytes. On success, the target buffer contains a GZIP compatible layout.

```
BYTE *data = NULL;
DWORD original_size = 0;
// ...
data = (BYTE*)malloc(original_size * sizeof(BYTE));
// ...

// initial size is original plus overhead & gzip-header
DWORD compressed_size =
  (DWORD)((double) original_size + (0.1 * (double) original_size) + 24);

BYTE *compressed_data = (BYTE*)malloc(compressed_size * sizeof(BYTE));

compressed_size =
  FreeImage_ZLibGZip(compressed_data, compressed_size, data, original_size);

// now compressed_data contains 'compressed_size' bytes of GZIP compressed data

// write data to a stream
// ...

free(compressed_data);
```

This function is useful in conjunction with the memory-i/o functions, if one is using this to compress something to send it gzip-compressed over the internet (where a simple zip-layout will not be accepted). Custom or more complex layouts may be obtained using a CRC32 builder in conjuction with the existing zip compression function (see the FreeImage_ZLibCRC32 function).

FreeImage_ZLibCRC32

```
DLL_API DWORD DLL_CALLCONV FreeImage_ZLibCRC32(DWORD crc, BYTE *source, DWORD
source_size);
```

Updates a running *crc* from *source* (whose size in bytes is given *source_size*) and returns the updated crc, using the ZLib library.

If source is NULL, this function returns the required initial value for the crc. Otherwise, it returns the new crc value.

FreeImage_ZlibGUnzip

```
DLL_API DWORD DLL_CALLCONV FreeImage_ZLibGUnzip(BYTE *target, DWORD target_size, BYTE
*source, DWORD source_size);
```

Decompresses a gzipped source buffer into a target buffer, using the ZLib library. Upon entry, target_size is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.

The function returns the actual size of the uncompressed buffer or returns 0 if an error occurred.

Helper functions

FreeImage_IsLittleEndian

```
DLL_API BOOL DLL_CALLCONV FreeImage_IsLittleEndian();
```

This function returns TRUE if the platform running FreeImage uses the Little Endian convention (Intel processors) and returns FALSE if it uses the Big Endian convention (Motorola processors).

FreeImage_LookupX11Color

```
DLL_API BOOL DLL_CALLCONV FreeImage_LookupX11Color(const char *szColor, BYTE *nRed,
BYTE *nGreen, BYTE *nBlue);
```

Converts a X11 color name into a corresponding RGB value. Upon entry, szColor is the color name. On output, nRed, nGreen and nBlue are the color components in the range [0..255]. The function returns TRUE if successful, FALSE otherwise.

```
BYTE red, green, blue;
BOOL bResult;
bResult = FreeImage_LookupX11Color("papaya whip", &red, &green, &blue);
if(bResult) {
  assert((red == 255) && (green == 239) && (blue == 213));
}
```

FreeImage_LookupSVGColor

```
DLL_API BOOL DLL_CALLCONV FreeImage_LookupSVGColor(const char *szColor, BYTE *nRed,
BYTE *nGreen, BYTE *nBlue);
```

Converts a SVG color name into a corresponding RGB value. Upon entry, szColor is the color name. On output, nRed, nGreen and nBlue are the color components in the range [0..255]. The function returns TRUE if successful, FALSE otherwise.

```
BYTE red, green, blue;
BOOL bResult;
bResult = FreeImage_LookupSVGColor("lemonchiffon", &red, &green, &blue);
if(bResult) {
  assert((red == 255) && (green == 250) && (blue == 205));
}
```

Reference

Scalable Vector Graphics (SVG) 1.1 Specification.
[Online] http://www.w3.org/TR/SVG/types.html

Metadata function reference

Introduction

Metadata or "data about data" describe the content, quality, condition, and other characteristics of data such as images. In FreeImage, metadata is information attached to an image in the form of keywords, free text or other data types. This information can be relatively straightforward such as author, date of creation or subject content of a resource. It can also be more complex and less easily defined (e.g. picture taking conditions, GPS information for recording position information, etc.).

Storage and retrieval of metadata *usually* refers to a standard or a specification. Examples of metadata standards used to describe images include IPTC/NAA, EXIF, GeoTIFF or Adobe XMP. Standards are not always used however. Many image formats use their own proprietary way to store metadata, either as simple text strings or in a more complex way (e.g. 8BIM markers used by Adobe Photoshop).

Although many standards or proprietary formats are used to describe images with metadata, FreeImage provides you with a simple interface to deal with all information attached to your images.

FreeImage Tag

FreeImage uses a structure known as a *tag* to store metadata information. The notion of tag originates from the TIFF specification and because of its universality, it is widely used to store metadata information in a file.

FreeImage provides an enhanced version of the standard TIFF or Exif tag structure. This version is described below:

Field name	Data type	Description
key	Pointer to a C string (char *)	Tag field name (unique inside a metadata model)
description	Pointer to a C string (char *)	Tag description if available, NULL otherwise
id	WORD (unsigned 16-bit)	Tag ID if available, 0 otherwise
type	WORD (unsigned 16-bit)	Tag data type (see FREE_IMAGE_MDTYPE below)
count	DWORD (unsigned 32-bit)	Number of *type* components in the tag
length	DWORD (unsigned 32-bit)	Length of the tag value in bytes
value	Pointer to a 32-bit value (void *)	Tag value

Table 11: FreeImage FITAG structure.

Given a metadata model (e.g. Exif, Exif GPS, IPTC/NAA), the tag key (or tag field name) is unique inside this data model. This uniqueness allows FreeImage to use this key to index the tag inside a hash table, in order to speed up tag access. Whenever you store a tag inside a metadata model, you thus need to provide a unique key with the tag to store.

A FreeImage tag may be used to store *any* kind of data (e.g. strings, integers, doubles, rational numbers, etc.). The complete list of data type supported by FreeImage is given in Table 12. For example, when the tag data type indicates a double and the tag count is 8, then the tag value is an array of 8 doubles. Its length should be 64 bytes (8 x sizeof(double)). If the tag data type indicates a rational and the length is 48 bytes, then there are (48 bytes / (2 x 4-bytes)) = 6 rational values in the tag.

As for ASCII strings, the value of the count part of an ASCII tag entry includes the NULL.

Tag data type	Description
0 = FIDT_NOTYPE	Placeholder (do not use this type)
1 = FIDT_BYTE	8-bit unsigned integer
2 = FIDT_ASCII	8-bit byte that contains a 7-bit ASCII code; the last byte must be NUL (binary zero)
3 = FIDT_SHORT	16-bit (2-byte) unsigned integer
4 = FIDT_LONG	32-bit (4-byte) unsigned integer
5 = FIDT_RATIONAL	Two LONGs: the first represents the numerator of a fraction; the second, the denominator
6 = FIDT_SBYTE	An 8-bit signed (twos-complement) integer
7 = FIDT_UNDEFINED	An 8-bit byte that may contain anything, depending on the definition of the field.
8 = FIDT_SSHORT	A 16-bit (2-byte) signed (twos-complement) integer
9 = FIDT_SLONG	A 32-bit (4-byte) signed (twos-complement) integer
10 = FIDT_SRATIONAL	Two SLONG's: the first represents the numerator of a fraction, the second the denominator
11 = FIDT_FLOAT	Single precision (4-byte) IEEE format
12 = FIDT_DOUBLE	Double precision (8-byte) IEEE format
13 = FIDT_IFD	FIDT_IFD data type is identical to LONG, but is only used to store offsets
14 = FIDT_PALETTE	32-bit (4-byte) RGBQUAD
16 = FIDT_LONG8	64-bit unsigned integer
17 = FIDT_SLONG8	64-bit signed integer
18 = FIDT_IFD8	FIDT_IFD8 data type is identical to LONG8, but is only used to store offsets

Table 12: FreeImage tag data types (FREE_IMAGE_MDTYPE identifier).

FreeImage metadata model

The metadata models currently recognized by the library are listed in Table 13, together with the FreeImage plugins that can load or save the corresponding metadata.

These metadata models are described in more detail in the appendix (see FreeImage metadata models).

Metadata model / FIF	FIF_JPEG	FIF_TIFF	FIF_PNG	FIF_GIF	FIF_RAW	FIF_JXR	FIF_WEBP
0 = FIMD_COMMENTS	R/W	-	R/W	R/W	-	-	-
1 = FIMD_EXIF_MAIN	R	R/W	-	-	R	R/W	R
2 = FIMD_EXIF_EXIF	R	R	-	-	R	R/W	R
3 = FIMD_EXIF_GPS	R	-	-	-	R	R/W	R
4 = FIMD_EXIF_MAKERNOTE	R	-	-	-	R	R	R
5 = FIMD_EXIF_INTEROP	R	-	-	-	R	R	R
6 = FIMD_IPTC	R/W	R/W	-	-	-	R/W	-
7 = FIMD_XMP	R/W	R/W	R/W	-	-	R/W	R/W
8 = FIMD_GEOTIFF	-	R/W	-	-	-	-	-
9 = FIMD_ANIMATION	-	-	-	R/W	-	-	-
10 = FIMD_CUSTOM	-	-	-	-	-	-	-
11 = FIMD_EXIF_RAW	R/W	-	-	-	-	-	R/W

R = Read, **W** = Write, **-** = Not implemented

Table 13: Metadata models supported by FreeImage.

Tag creation and destruction

FreeImage_CreateTag

```
DLL_API FITAG *DLL_CALLCONV FreeImage_CreateTag();
```

Allocates a new FITAG object. This object must be destroyed with a call to FreeImage_DeleteTag when no longer in use.

Tag creation and destruction functions are only needed when you use the FreeImage_SetMetadata function.

FreeImage_DeleteTag

```
DLL_API void DLL_CALLCONV FreeImage_DeleteTag(FITAG *tag);
```

Delete a previously allocated FITAG object.

FreeImage_CloneTag

```
DLL_API FITAG *DLL_CALLCONV FreeImage_CloneTag(FITAG *tag);
```

Creates and returns a copy of a FITAG object. This copy must be destroyed with a call to FreeImage_DeleteTag when no longer in use.

Tag accessors

FreeImage_GetTagKey

```
DLL_API const char *DLL_CALLCONV FreeImage_GetTagKey(FITAG *tag);
```

Returns the tag field name (unique inside a metadata model).

FreeImage_GetTagDescription

```
DLL_API const char *DLL_CALLCONV FreeImage_GetTagDescription(FITAG *tag);
```

Returns the tag description if available, returns NULL otherwise.

FreeImage_GetTagID

```
DLL_API WORD DLL_CALLCONV FreeImage_GetTagID(FITAG *tag);
```

Returns the tag ID if available, returns 0 otherwise.

FreeImage_GetTagType

```
DLL_API FREE_IMAGE_MDTYPE DLL_CALLCONV FreeImage_GetTagType(FITAG *tag);
```

Returns the tag data type (see Table 12 for a list of known data types).

FreeImage_GetTagCount

```
DLL_API DWORD DLL_CALLCONV FreeImage_GetTagCount(FITAG *tag);
```

Returns the number of components in the tag (in *tag type* units). For example, when the tag data type indicates a double (i.e. a FIDT_DOUBLE type) and the tag count is 8, then the tag value is an array of 8 doubles.

FreeImage_GetTagLength

```
DLL_API DWORD DLL_CALLCONV FreeImage_GetTagLength(FITAG *tag);
```

Returns the length of the tag value in bytes.

FreeImage_GetTagValue

```
DLL_API const void *DLL_CALLCONV FreeImage_GetTagValue(FITAG *tag);
```

Returns the tag value.

It is up to you to interpret the returned pointer correctly, according to the results of FreeImage_GetTagType and FreeImage_GetTagCount.

FreeImage_SetTagKey

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagKey(FITAG *tag, const char *key);
```

Set the tag field name (**always required**, must be unique inside a metadata model). The function returns TRUE if successful and returns FALSE otherwise.

FreeImage_SetTagDescription

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagDescription(FITAG *tag, const char
*description);
```

Set the (usually optional) tag description. The function returns TRUE if successful and returns FALSE otherwise.

The tag description is never stored in a file. FreeImage maintains an internal table for all known tags, together with their description when available. Whenever you read a known tag, the library is able to give the tag description (provided that the tag is known by the library) using FreeImage_GetTagDescription. However, you will never have to provide a tag description when storing a tag.

FreeImage_SetTagID

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagID(FITAG *tag, WORD id);
```

Set the (usually optional) tad ID. The function returns TRUE if successful and returns FALSE otherwise.

FreeImage_SetTagType

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagType(FITAG *tag, FREE_IMAGE_MDTYPE type);
```

Set the tag data type (**always required**, see Table 12 for a list of available data types). The function returns TRUE if successful and returns FALSE otherwise.

FreeImage_SetTagCount

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagCount(FITAG *tag, DWORD count);
```

Set the number of data in the tag (**always required**, expressed in *tag type* unit). The function returns TRUE if successful and returns FALSE otherwise.

FreeImage_SetTagLength

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagLength(FITAG *tag, DWORD length);
```

Set the length of the tag value, in bytes (**always required**). The function returns TRUE if successful and returns FALSE otherwise.

FreeImage_SetTagValue

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetTagValue(FITAG *tag, const void *value);
```

Set the tag value (**always required**). The function returns TRUE if successful and returns FALSE otherwise.

This function must be called *after* the tag data type, tag count and tag length have been filled. Otherwise, you will be unable to successfully call FreeImage_SetMetadata.

Metadata iterator

FreeImage_FindFirstMetadata

```
DLL_API FIMETADATA *DLL_CALLCONV FreeImage_FindFirstMetadata(FREE_IMAGE_MDMODEL model,
FIBITMAP *dib, FITAG **tag);
```

Provides information about the first instance of a tag that matches the metadata model specified in the *model* argument.

If successful, FreeImage_FindFirstMetadata returns a unique search handle identifying the group of tags matching the *model* specification, which can be used in a subsequent call to FreeImage_FindNextMetadata or to FreeImage_FindCloseMetadata.

When the metadata model does not exist in the input dib, FreeImage_FindFirstMetadata returns NULL.

FreeImage_FindNextMetadata

```
DLL_API BOOL DLL_CALLCONV FreeImage_FindNextMetadata(FIMETADATA *mdhandle, FITAG
**tag);
```

Find the next tag, if any, that matches the metadata *model* argument in a previous call to FreeImage_FindFirstMetadata, and then alters the tag object contents accordingly.

If successful, FreeImage_FindNextMetadata returns TRUE. Otherwise, it returns FALSE, indicating that no more matching tags could be found.

FreeImage_FindCloseMetadata

```
DLL_API void DLL_CALLCONV FreeImage_FindCloseMetadata(FIMETADATA *mdhandle);
```

Closes the specified metadata search handle and releases associated resources

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

FITAG *tag = NULL;
FIMETADATA *mdhandle = NULL;

mdhandle = FreeImage_FindFirstMetadata(FIMD_EXIF_MAIN, dib, &tag);

if(mdhandle) {
  do {
    // process the tag
    printf("%s\n", FreeImage_GetTagKey(tag));
    // ...

  } while(FreeImage_FindNextMetadata(mdhandle, &tag));

  FreeImage_FindCloseMetadata(mdhandle);
}
```

Metadata accessors

FreeImage_GetMetadata

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetMetadata(FREE_IMAGE_MDMODEL model, FIBITMAP
*dib, const char *key, FITAG **tag);
```

Retrieve a metadata attached to a dib. Upon entry, *model* is the metadata model to look for, *dib* is the image that contains metadata, *key* is the metadata field name (unique inside a metadata model) and *tag* is a FITAG structure returned by the function.

When the searched tag doesn't exist, the tag object is left unchanged and the function returns FALSE. Otherwise, the function returns TRUE and the tag object is populated with the metadata information.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

// Get the camera model
FITAG *tagMake = NULL;
FreeImage_GetMetadata(FIMD_EXIF_MAIN, dib, "Make", &tagMake);
if(tagMake != NULL) {

  // here we know (according to the Exif specifications) that tagMake is a C string
  printf("Camera model : %s\n", (char*)FreeImage_GetTagValue(tagMake));

  // if we don't have the specifications, we can still convert the tag to a C string
  printf("Camera model : %s\n", FreeImage_TagToString(FIMD_EXIF_MAIN, tagMake));

}
```

When a tag returned by *FreeImage_GetMetadata* or by the metadata iterator functions is modified, changes will be applied to the corresponding tag attached to the bitmap. Saving the bitmap will thus save the modified tag (provided that the library can save the corresponding metadata model).

FreeImage_SetMetadata

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetMetadata(FREE_IMAGE_MDMODEL model, FIBITMAP
*dib, const char *key, FITAG *tag);
```

Attach a new FreeImage tag to a dib. Upon entry, *model* is the metadata model used to store the tag, *dib* is the target image, *key* is the tag field name and *tag* is the FreeImage tag to be attached.

If tag is NULL then the metadata is deleted.

If both key and tag are NULL then the metadata model is deleted.

The function returns TRUE on success and returns FALSE otherwise.

The tag field name (or tag key) used by FreeImage to index a tag is given by the metadata model specification (e.g. EXIF specification or Adobe XMP specification).

```
char *xmp_profile = NULL;
DWORD profile_size = 0;
// ...

// the following assumes that you have a XML packet stored in
// the (null terminated) variable 'xmp_profile'.
// The size of the packet is given
// by 'profile_size' and includes the NULL value

// create a tag
FITAG *tag = FreeImage_CreateTag();
if(tag) {
  // fill the tag members
  // note that the FIMD_XMP model accept a single key named "XMLPacket"
  FreeImage_SetTagKey(tag, "XMLPacket");
  FreeImage_SetTagLength(tag, profile_size);
  FreeImage_SetTagCount(tag, profile_size);
  FreeImage_SetTagType(tag, FIDT_ASCII);
  // the tag value must be stored after
  // the tag data type, tag count and tag length have been filled.
  FreeImage_SetTagValue(tag, xmp_profile);

  // store the tag
  FreeImage_SetMetadata(FIMD_XMP, dib, FreeImage_GetTagKey(tag), tag);

  // destroy the tag
  FreeImage_DeleteTag(tag);
}
```

```
/**
Add a single IPTC tag to a FIBITMAP
NB: The tag ID is not needed as it is filled automatically by FreeImage_SetMetadata
@param image Your image to be saved
@param key Tag key
@param value Tag value
*/
void add_IPTC_tag(FIBITMAP *image, const char *key, const char *value) {
  // create a tag
  FITAG *tag = FreeImage_CreateTag();
  if(tag) {
    // fill the tag
    FreeImage_SetTagKey(tag, key);
    FreeImage_SetTagLength(tag, strlen(value) + 1);
    FreeImage_SetTagCount(tag, strlen(value) + 1);
    FreeImage_SetTagType(tag, FIDT_ASCII);
    FreeImage_SetTagValue(tag, value);
    FreeImage_SetMetadata(FIMD_IPTC, image, FreeImage_GetTagKey(tag), tag);
    // destroy the tag
    FreeImage_DeleteTag(tag);
  }
}

/**
Add some IPTC tags to a FIBITMAP
*/
void add_IPTC_Metadata(FIBITMAP *dib) {
  // !!! IPTC data is ignored by Photoshop when there is a XML packet in the dib !!!
  add_IPTC_tag(dib, "ObjectName", "my title");
  add_IPTC_tag(dib, "Caption-Abstract", "my caption");
  add_IPTC_tag(dib, "Writer-Editor", "myself");
  add_IPTC_tag(dib, "By-line", "my name");
  add_IPTC_tag(dib, "By-lineTitle", "my position");
  add_IPTC_tag(dib, "Keywords", "FreeImage;Library;Images;Compression");
}
```

FreeImage_SetMetadataKeyValue

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetMetadataKeyValue(FREE_IMAGE_MDMODEL model,
FIBITMAP *dib, const char *key, const char *value);
```

This helper function builds and set a FITAG whose type is FIDT_ASCII. A description of parameters follows:

- *model* Metadata model to be filled
- *dib* Image to be filled
- *key* Tag key
- *value* Tag value as a ASCII string (NULL-terminated)

The function returns TRUE if successful, returns FALSE otherwise.

```
char *xmp_profile = NULL;

// the following assumes that you have a XML packet stored in
// the (null terminated) variable 'xmp_profile'.

// store the tag
FreeImage_SetMetadataKeyValue(FIMD_XMP, dib, "XMLPacket", xmp_profile);
```

```
/**
Add some IPTC tags to a FIBITMAP
*/
void add_IPTC_Metadata(FIBITMAP *dib) {
  // !!! IPTC data is ignored by Photoshop when there is a XML packet in the dib !!!
  FreeImage_SetMetadataKeyValue(FIMD_IPTC, dib, "ObjectName", "my title");
  FreeImage_SetMetadataKeyValue(FIMD_IPTC, dib, "Caption-Abstract", "my caption");
  FreeImage_SetMetadataKeyValue(FIMD_IPTC, dib, "Writer-Editor", "myself");
  FreeImage_SetMetadataKeyValue(FIMD_IPTC, dib, "By-lineTitle", "my position");
  FreeImage_SetMetadataKeyValue(FIMD_IPTC, dib, "Keywords",
"FreeImage;Library;Images;Compression");
}
```

Metadata helper functions

FreeImage_GetMetadataCount

```
DLL_API unsigned DLL_CALLCONV FreeImage_GetMetadataCount(FREE_IMAGE_MDMODEL model,
FIBITMAP *dib);
```

Returns the number of tags contained in the *model* metadata model attached to the input dib.

```
unsigned count = FreeImage_GetMetadataCount(FIMD_EXIF_GPS, dib);
if(count > 0) {
  // process GPS data
}
```

FreeImage_CloneMetadata

```
DLL_API BOOL DLL_CALLCONV FreeImage_CloneMetadata(FIBITMAP *dst, FIBITMAP *src);
```

Copy all metadata contained in *src* into *dst*, with the exception of FIMD_ANIMATION metadata (these metadata are not copied because this may cause problems when saving to GIF). When a *src* metadata model already exists in *dst*, the *dst* metadata model is first erased

before copying the *src* one. When a metadata model already exists in *dst* and not in *src*, it is left untouched.

Horizontal and vertical resolution info (returned by FreeImage_GetDotsPerMeterX and by FreeImage_GetDotsPerMeterY) is also copied from *src* to *dst*.

The function returns TRUE on success and returns FALSE otherwise (e.g. when *src* or *dst* are invalid).

Every FreeImage function with a signature such as
`FIBITMAP *dst = FreeImage_Function(FIBITMAP *src, ...)`
will internally clone src metadata into dst.
Thus, you will never have to use this function when using FreeImage functions. However, you may have to use this function when creating your *own* processing functions.

The FreeImage_CloneMetadata functions clones metadata from one image to another. Although the target image contains all metadata of the source image, some metadata can't be saved to disk, see e.g. this pseudo-code:

```
FreeImage_CloneMetadata(dst_jpeg, src_jpeg);
// dst_jpeg has the same metadata as src_jpeg
FreeImage_Save(FIF_JPEG, dst_jpeg, "example.jpg", 0);
FIBITMAP *new_jpeg = FreeImage_Load(FIF_JPEG, "example.jpg", 0);
// new_jpeg is missing some metadata
```

FreeImage_TagToString

```
DLL_API const char* DLL_CALLCONV FreeImage_TagToString(FREE_IMAGE_MDMODEL model, FITAG
*tag, char *Make FI_DEFAULT(NULL));
```

Converts a FreeImage tag structure to a string that represents the interpreted tag value. The tag value is interpreted according to the metadata model specification. For example, consider a tag extracted from the FIMD_EXIF_EXIF metadata model, whose ID is 0x9209 and whose key is “Flash”. Then if the tag value is 0x0005, the function will return “Strobe return light not detected”.

Upon entry, *model* is the metadata model from which the tag was extracted, *tag* is the FreeImage tag to interpret and *Make* is the camera model. This last parameter is currently not used by the library but will be used in the future to interpret the camera maker notes (FIMD_EXIF_MAKERNOTE metadata model).

FreeImage_TagToString is not thread safe.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

FITAG *tag = NULL;
FIMETADATA *mdhandle = NULL;

mdhandle = FreeImage_FindFirstMetadata(model, dib, &tag);

if(mdhandle) {
  do {
    // convert the tag value to a human readable string
    const char *value = FreeImage_TagToString(model, tag);

    // print the tag
    // note that most tags do not have a description,
    // especially when the metadata specifications are not available
    if(FreeImage_GetTagDescription(tag)) {
      cout << "key : " << FreeImage_GetTagKey(tag) << "; value : " << value
           << "; description : " << FreeImage_GetTagDescription(tag) << "\n";
    } else {
      cout << "key : " << FreeImage_GetTagKey(tag) << "; value : " << value
           << "; description : (none)\n";
    }

  } while(FreeImage_FindNextMetadata(mdhandle, &tag));
}

FreeImage_FindCloseMetadata(mdhandle);
```

Toolkit function reference

Rotation and flipping

FreeImage_Rotate

1 8 24 32 16$_{UINT16}$ 48$_{RGB16}$ 64$_{RGBA16}$ 32$_{FLOAT}$ 96$_{RGBF}$ 128$_{RGBAF}$

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Rotate(FIBITMAP *dib, double angle, const
void *bkcolor FI_DEFAULT(NULL));
```

This function rotates a standard image (1-, 8-bit greyscale or a 24-, 32-bit color), a RGB(A)16 or RGB(A)F image by means of 3 shears. The angle of rotation is specified by the *angle* parameter in degrees. Rotation occurs around the center of the image area. Rotated image retains size and aspect ratio of source image (destination image size is usually bigger), so that this function should be used when rotating an image by 90°, 180° or 270°.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

// perform a 90° rotation (CCW rotation)
FIBITMAP *rotated = FreeImage_Rotate(dib, 90);
```

For **1-bit images**, rotation is limited to angles whose value is an integer multiple of 90° (e.g. –90, 90, 180, 270). A NULL value is returned for other angles.

When the *angle* value isn't an integer multiple of 90°, the background is filled with the supplied *bkcolor* parameter. When *bkcolor* is NULL (default value), the background is filled with a black color. The data type of *bkcolor* depends on the image type (see the following example).

```
FIBITMAP* testRotateWithBackground(FIBITMAP *src, double angle) {
  FREE_IMAGE_TYPE image_type = FreeImage_GetImageType(src);

  switch(image_type) {
    case FIT_BITMAP:
      switch(FreeImage_GetBPP(src)) {
        case 8:
        {
          BYTE color = 128;
          return FreeImage_Rotate(src, angle, &color);
        }
        break;
        case 24:  // we could also use 'RGBTRIPLE color' here
        case 32:
        {
          RGBQUAD color = { 0, 0, 255, 0 };
          // for 24-bit images, the first 3 bytes will be read
          // for 32-bit images, the first 4 bytes will be read
          return FreeImage_Rotate(src, angle, &color);
        }
        break;
      }
      break;
    case FIT_UINT16:
    {
      WORD color = 128;
      return FreeImage_Rotate(src, angle, &color);
    }
    break;

    case FIT_RGB16:  // we could also use 'FIRGB16 color' here
    case FIT_RGBA16:
    {
      FIRGBA16 color = { 0, 0, 255, 0 };
      // for RGB16 images, the first 3 WORD will be read
      // for RGBA16 images, the first 4 WORD will be read
      return FreeImage_Rotate(src, angle, &color);
    }
    break;

    case FIT_FLOAT:
    {
      float color = 0.5F;
      return FreeImage_Rotate(src, angle, &color);
    }
    break;

    case FIT_RGBF:  // we could also use 'FIRGBF color' here
    case FIT_RGBAF:
    {
      FIRGBAF color = { 0, 0, 1, 0 };
      // for RGBF images, the first 3 float will be read
      // for RGBAF images, the first 4 float will be read
      return FreeImage_Rotate(src, angle, &color);
    }
    break;
  }

  return NULL;
}
```

A demonstration of this function is given in the Appendix (see Using the rotation functions).

References

Paeth A., A Fast Algorithm for General Raster Rotation. Graphics Gems, p. 179, Andrew Glassner editor, Academic Press, 1990.

Yariv E., High quality image rotation (rotate by shear).
[Online] http://www.codeproject.com/bitmap/rotatebyshear.asp

FreeImage_RotateEx

8 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_RotateEx(FIBITMAP *dib, double angle, double
x_shift, double y_shift, double x_origin, double y_origin, BOOL use_mask);
```

This function performs a rotation and / or translation of an 8-bit greyscale, 24- or 32-bit image, using a 3rd order (cubic) B-Spline. The rotated image will have the same width and height as the source image, so that this function is better suited for computer vision and robotics.

The angle of rotation is specified by the angle parameter in degrees. Horizontal and vertical image translations (in pixel units) are specified by the x_shift and y_shift parameters. Rotation occurs around the center specified by x_origin and y_origin, also given in pixel units. When use_mask is set to TRUE, the irrelevant part of the image is set to a black color, otherwise, a mirroring technique is used to fill irrelevant pixels.

```
// this code assumes there is a bitmap loaded and
// present in a variable called 'dib'

// rotate the image about the center of the image area

double x_orig = FreeImage_GetWidth(dib) / (double)2;
double y_orig = FreeImage_GetHeight(dib) / (double)2;

// perform a 15° CCW rotation using a mask (no translation)
FIBITMAP *rotated = FreeImage_RotateEx(dib, 15, 0, 0, x_orig, y_orig, TRUE);
```

A demonstration of this function is given in the Appendix (see Using the rotation functions).

References

Philippe Thévenaz, Spline interpolation, a C source code implementation. [Online] http://bigwww.epfl.ch/thevenaz/

Unser M., Splines: A Perfect Fit for Signal and Image Processing. IEEE Signal Processing Magazine, vol. 16, no. 6, pp. 22-38, November 1999.

Unser M., Aldroubi A., Eden M., B-Spline Signal Processing: Part I--Theory. IEEE Transactions on Signal Processing, vol. 41, no. 2, pp. 821-832, February 1993.

Unser M., Aldroubi A., Eden M., B-Spline Signal Processing: Part II--Efficient Design and Applications. IEEE Transactions on Signal Processing, vol. 41, no. 2, pp. 834-848, February 1993.

FreeImage_FlipHorizontal

```
DLL_API BOOL DLL_CALLCONV FreeImage_FlipHorizontal(FIBITMAP *dib);
```

Flip the input dib horizontally along the vertical axis.

FreeImage_FlipVertical

```
DLL_API BOOL DLL_CALLCONV FreeImage_FlipVertical(FIBITMAP *dib);
```

Flip the input dib vertically along the horizontal axis.

Upsampling / downsampling

FreeImage_Rescale

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP * DLL_CALLCONV FreeImage_Rescale(FIBITMAP *dib, int dst_width, int
dst_height, FREE_IMAGE_FILTER filter);
```

This function performs resampling (or scaling, zooming) of a greyscale or RGB(A) image to the desired destination width and height. A NULL value is returned when the bitdepth cannot be handled or when there's not enough memory (this may happen with very large images).

FIT_BITMAP types

Images whose image type is FIT_BITMAP are returned as 8-bit or 24-bit, or as 32-bit if they contain transparency. For example, 16-bit RGB bitmap are returned as 24-bit. Non transparent palettized and 4-bit bitmap are returned as 24-bit images. The algorithm tries to produce destination images with the smallest possible bit depth.

If you have transparency, you'll get a 32-bit image. If you have real colors, you'll get a 24-bit image. For all other cases, you'll get an 8-bit image with a linear color palette (which defaults to MINISBLACK; it is MINISWHITE only, if the source image was of type MINISWHITE).

Resampling refers to changing the pixel dimensions (and therefore display size) of an image. When you downsample (or decrease the number of pixels), information is deleted from the image. When you upsample (or increase the number of pixels), new pixels are added based on color values of existing pixels. You specify an interpolation filter to determine how pixels are added or deleted.

The following filters can be used as resampling filters:

Filter flag	Description
FILTER_BOX	Box, pulse, Fourier window, 1st order (constant) B-Spline
FILTER_BILINEAR	Bilinear filter
FILTER_BSPLINE	4th order (cubic) B-Spline
FILTER_BICUBIC	Mitchell and Netravali's two-param cubic filter
FILTER_CATMULLROM	Catmull-Rom spline, Overhauser spline
FILTER_LANCZOS3	Lanczos-windowed sinc filter

Table 14: IMAGE_FILTER constants.

Some hints on how to use these filters are given in the Appendix (see Choosing the right resampling filter).

References

Catmull E., Rom R., A class of local interpolating splines. In Computer Aided Geometric Design, R. E. Barnhill and R. F. Reisenfeld, Eds. Academic Press, New York, 1974, pp. 317–326.

Hou H.S., Andrews H.C., Cubic Splines for Image Interpolation and Digital Filtering. IEEE Trans. Acoustics, Speech, and Signal Proc., vol. ASSP-26, no. 6, pp. 508-517, Dec. 1978.

Keys R.G., Cubic Convolution Interpolation for Digital Image Processing. IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 29, no. 6, pp. 1153-1160, Dec. 1981.

Mitchell Don P., Netravali Arun N., Reconstruction filters in computer graphics. In John Dill, editor, Computer Graphics (SIGGRAPH '88 Proceedings), Vol. 22, No. 4, pp. 221-228, August 1988.

Paul Heckbert, C code to zoom raster images up or down, with nice filtering. UC Berkeley, August 1989.

[Online] http://www-2.cs.cmu.edu/afs/cs.cmu.edu/Web/People/ph/heckbert.html

Glassner A.S., Principles of digital image synthesis. Morgan Kaufmann Publishers, Inc, San Francisco, Vol. 2, 1995.

FreeImage_RescaleRect

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_RescaleRect(FIBITMAP *dib, int dst_width, int
dst_height, int left, int top, int right, int bottom, FREE_IMAGE_FILTER filter
FI_DEFAULT(FILTER_CATMULLROM), unsigned flags FI_DEFAULT(0));
```

FreeImage_RescaleRect provides support for rescaling only a rectangular area of an image. Basically, that's a much faster solution than coding:

```
FIBITMAP *dibTmp = FreeImage_Copy(dib, 10, 10, 140, 140);
FIBITMAP *dibDst = FreeImage_Rescale(dibTmp, 260, 260);
FreeImage_Unload(dibTmp);
```

Much faster and easier to write is:

```
FIBITMAP *dibDst = FreeImage_RescaleRect(dib, 260, 260, 10, 10, 140, 140);
```

Of course, the function does not rely on FreeImage_Copy but reads and rescales the bits inside the specified rectangle only.

Additionally, function FreeImage_RescaleRect takes a flags parameter. Currently, there are 3 flags/options defined and implemented:

RescaleRect flag	Description
FI_RESCALE_DEFAULT	Default options; none of the following other options apply.
FI_RESCALE_TRUE_COLOR	For non-transparent greyscale images, convert to 24-bit if src bitdepth <= 8 (default is a 8-bit greyscale image).
FI_RESCALE_OMIT_METADATA	Do not copy metadata to the rescaled image.

About the FI_RESCALE_TRUE_COLOR option:

By using flag FI_RESCALE_TRUE_COLOR, one can ensure **not** getting a palletized 8-bit image but a *true color* result image (24- or 32-bit, depending on transparency), when rescaling an image with a bit depth smaller than or equal to 8 (src bpp <= 8), that consists of gray colors only (r=g=b). By default, FreeImage_Rescale returns an image with the smallest possible bit depth (which is 8-bit for a grayscale image). However, it can be annoying to get either a true color image or a palletized image for a palletized source image, depending on

the *color-awareness* of the source image only (especially, if the rescaled image needs further processing).

FreeImage_MakeThumbnail

1 4 8 16 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16} 32_{FLOAT} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_MakeThumbnail(FIBITMAP *dib, int
max_pixel_size, BOOL convert FI_DEFAULT(TRUE));
```

Creates a thumbnail from a greyscale or RGB(A) image so that the output image fits inside a square of size *max_pixel_size*, keeping aspect ratio.

Downsampling is done using a bilinear filter (see FreeImage_Rescale). 16-bit RGB bitmap are returned as 24-bit. Palettized and 4-bit bitmap are returned as 8-bit or as 32-bit if they contain transparency.

When the *convert* parameter is set to TRUE, High Dynamic Range images (FIT_UINT16, FIT_RGB16, FIT_RGBA16, FIT_FLOAT) are transparently converted to standard images (i.e. 8-, 24 or 32-bit images), using one of the FreeImage_ConvertToXXX conversion function. As for RBG[A]F images, they are converted to 24-bit using the FreeImage_TmoDrago03 function with default options.

```
#define THUMBNAIL_SIZE 90 // fit inside a square whose size is 90 pixels

FIBITMAP * makeThumbnail(const char *szPathName) {
  FIBITMAP *dib = NULL;
  int flags = 0;              // default load flag
  int originalWidth = 0;      // original image width
  int originalHeight = 0;     // original image height

  FREE_IMAGE_FORMAT fif = FreeImage_GetFileType(szPathName);
  if(fif == FIF_UNKNOWN) return NULL;

  if(fif == FIF_JPEG) {
    FITAG *tag = NULL;
    // for JPEG images, we can speedup the loading part
    // using LibJPEG downsampling feature while loading the image...
    flags |= THUMBNAIL_SIZE << 16;
    // load the dib
    dib = FreeImage_Load(fif, szPathName, flags);
    if(!dib) return NULL;
    // the dib may have been downscaled by 2x, 4x or 8x
    // retrieve the original width & height (stored as comments for this special
case)
    if(FreeImage_GetMetadata(FIMD_COMMENTS, dib, "OriginalJPEGWidth", &tag)) {
      originalWidth = atoi( (char*)FreeImage_GetTagValue(tag) );
    } else {
      originalWidth = FreeImage_GetWidth(dib);
    }
    if(FreeImage_GetMetadata(FIMD_COMMENTS, dib, "OriginalJPEGHeight", &tag)) {
      originalHeight = atoi( (char*)FreeImage_GetTagValue(tag) );
    } else {
      originalHeight = FreeImage_GetHeight(dib);
    }
  } else {
    // any cases other than the JPEG case: load the dib ...

    if(fif == FIF_RAW) {
      // ... except for RAW images, try to load the embedded JPEG preview
      // or default to RGB 24-bit ...
      flag = RAW_PREVIEW;
    }

    dib = FreeImage_Load(fif, szPathName, flags);
    if(!dib) return NULL;
    originalWidth = FreeImage_GetWidth(dib);
    originalHeight = FreeImage_GetHeight(dib);
  }

  // store 'originalWidth' and 'originalHeight' for later use …
  // store any other metadata (such as Exif) for later use …
  // ...

  // create the requested thumbnail
  FIBITMAP *thumbnail = FreeImage_MakeThumbnail(dib, THUMBNAIL_SIZE, TRUE);
  FreeImage_Unload(dib);
  return thumbnail;
}
```

Color manipulation

FreeImage uses the RGB(A) color model to represent color images in memory. A 8-bit greyscale image has a single channel, often called the black channel. A 24-bit image is made up of three 8-bit channels: one for each of the red, green and blue colors. For 32-bit images, a fourth 8-bit channel, called alpha channel, is used to create and store masks, which let you manipulate, isolate, and protect specific parts of an image. Unlike the others channels, the alpha channel doesn't convey color information, in a physical sense.

Color manipulation functions used in FreeImage allow you to modify the histogram of a specific channel. This transformation is known as a *point operation*, and may be used to adjust brightness, contrast or gamma of an image, to perform image enhancement (e.g. histogram equalization, non-linear contrast adjustment) or even to invert or threshold an image.

Currently, the following channels are defined in FreeImage:

Channel flag	Description
FICC_RGB	Function applies to red, green and blue channels
FICC_RED	Function applies to red channel only
FICC_GREEN	Function applies to green channel only
FICC_BLUE	Function applies to blue channel only
FICC_ALPHA	Function applies to alpha channel only
FICC_BLACK	Function applies to black channel
FICC_REAL	Complex images: function applies to the real part
FICC_IMAG	Complex images: function applies to the imaginary part
FICC_MAG	Complex images: function applies to the magnitude
FICC_PHASE	Complex images: function applies to the phase

Table 15: FREE_IMAGE_COLOR_CHANNEL constants.

FreeImage_AdjustCurve

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_AdjustCurve(FIBITMAP *dib, BYTE *LUT,
FREE_IMAGE_COLOR_CHANNEL channel);
```

Perfoms an histogram transformation on a 8-, 24- or 32-bit image according to the values of a lookup table (LUT). The transformation changes one or more channels according to the following equation:

channel(x, y) = LUT[channel(x, y)]

The size of 'LUT' is assumed to be 256. The color channel to be transformed is specified by the *channel* parameter. The transformation is done as follows:

- 8-bit images: if the image has a color palette, the LUT is applied to this palette, otherwise, it is applied to the grey values. The channel parameter is not used.
- 24-bit & 32-bit images: if channel is equal to FICC_RGB, the same LUT is applied to each color plane (R, G, and B). Otherwise, the LUT is applied to the specified channel only (R, G, B or A).

The function returns TRUE on success, FALSE otherwise (e.g. when the bitdepth of the source dib cannot be handled).

FreeImage_AdjustGamma

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_AdjustGamma(FIBITMAP *dib, double gamma);
```

Performs gamma correction on a 8-, 24- or 32-bit image. The gamma parameter represents the gamma value to use (gamma > 0). A value of 1.0 leaves the image alone, less than one darkens it, and greater than one lightens it.

The function returns TRUE on success. It returns FALSE when gamma is less than or equal to zero or when the bitdepth of the source dib cannot be handled.

FreeImage_AdjustBrightness

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_AdjustBrightness(FIBITMAP *dib, double
percentage);
```

Adjusts the brightness of a 8-, 24- or 32-bit image by a certain amount. This amount is given by the percentage parameter, where percentage is a value between [-100..100]. A value 0 means no change, less than 0 will make the image darker and greater than 0 will make the image brighter.

The function returns TRUE on success, FALSE otherwise (e.g. when the bitdepth of the source dib cannot be handled).

FreeImage_AdjustContrast

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_AdjustContrast(FIBITMAP *dib, double percentage);
```

Adjusts the contrast of a 8-, 24- or 32-bit image by a certain amount. This amount is given by the percentage parameter, where percentage is a value between [-100..100]. A value 0 means no change, less than 0 will decrease the contrast and greater than 0 will increase the contrast of the image.

The function returns TRUE on success, FALSE otherwise (e.g. when the bitdepth of the source dib cannot be handled).

FreeImage_Invert

1 4 8 24 32 16_{UINT16} 48_{RGB16} 64_{RGBA16}

```
DLL_API BOOL DLL_CALLCONV FreeImage_Invert(FIBITMAP *dib);
```

Inverts each pixel data.

FreeImage_GetHistogram

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_GetHistogram(FIBITMAP *dib, DWORD *histo,
FREE_IMAGE_COLOR_CHANNEL channel FI_DEFAULT(FICC_BLACK));
```

Computes the image histogram. For 24-bit and 32-bit images, histogram can be computed from red, green, blue and black channels. For 8-bit images, histogram is computed from the black channel. Other bit depth is not supported (nothing is done and the function returns

FALSE). The histo variable must be allocated by the application driving FreeImage. **Its size is assumed to be equal to 256**.

FreeImage_GetAdjustColorsLookupTable

8 24 32

```
DLL_API int DLL_CALLCONV FreeImage_GetAdjustColorsLookupTable(BYTE *LUT, double
brightness, double contrast, double gamma, BOOL invert);
```

This function creates a lookup table to be used with FreeImage_AdjustCurve which may adjust brightness and contrast, correct gamma and invert the image with a single call to FreeImage_AdjustCurve. If more than one of these image display properties need to be adjusted, using a combined lookup table should be preferred over calling each adjustment function separately. That's particularly true for huge images or if performance is an issue. Then, the expensive process of iterating over all pixels of an image is performed only once and not up to four times.

Furthermore, the lookup table created does not depend on the order, in which each single adjustment operation is performed. Due to rounding and byte casting issues, it actually matters in which order individual adjustment operations are performed. Both of the following snippets most likely produce different results:

```
// snippet 1: contrast, brightness
FreeImage_AdjustContrast(dib, 15.0);
FreeImage_AdjustBrightness(dib, 50.0);

// snippet 2: brightness, contrast
FreeImage_AdjustBrightness(dib, 50.0);
FreeImage_AdjustContrast(dib, 15.0);
```

Better and even faster would be snippet 3:

```
// snippet 3:
BYTE LUT[256];
FreeImage_GetAdjustColorsLookupTable(LUT, 50.0, 15.0, 1.0, FALSE);
FreeImage_AdjustCurve(dib, LUT, FICC_RGB);
```

This function is also used internally by FreeImage_AdjustColors, which does not return the lookup table, but uses it to call FreeImage_AdjustCurve on the passed image.

A description of parameters follows:

- *LUT* Output lookup table to be used with FreeImage_AdjustCurve. **The size of 'LUT' is assumed to be 256**.
- *brightness* Percentage brightness value in [-100..100]. A value of 0 means no change, less than 0 will make the image darker and greater than 0 will make the image brighter.
- *contrast* Percentage contrast value in [-100..100]. A value of 0 means no change, less than 0 will decrease the contrast and greater than 0 will increase the contrast of the image.
- *gamma* Gamma value to be used for gamma correction. A value of 1.0 leaves the image alone, less than one darkens it, and greater than one lightens it. This parameter must greater than zero. Otherwise, it will be ignored and no gamma correction will be performed using the lookup table created.
- *invert* If set to TRUE, the image will be inverted.

The function returns the number of adjustments applied to the resulting lookup table compared to a blind lookup table.

FreeImage_AdjustColors

8 24 32

```
DLL_API BOOL DLL_CALLCONV FreeImage_AdjustColors(FIBITMAP *dib, double brightness,
double contrast, double gamma, BOOL invert FI_DEFAULT(FALSE));
```

This function adjusts an image's brightness, contrast and gamma as well as it may optionally invert the image within a single operation. If more than one of these image display properties need to be adjusted, using this function should be preferred over calling each adjustment function separately. That's particularly true for huge images or if performance is an issue.

This function relies on FreeImage_GetAdjustColorsLookupTable, which creates a single lookup table, that combines all adjustment operations requested. Furthermore, the lookup table created by FreeImage_GetAdjustColorsLookupTable does not depend on the order, in which each single adjustment operation is performed. Due to rounding and byte casting issues, it actually matters in which order individual adjustment operations are performed. Both of the following snippets most likely produce different results:

```
// snippet 1: contrast, brightness
FreeImage_AdjustContrast(dib, 15.0);
FreeImage_AdjustBrightness(dib, 50.0);

// snippet 2: brightness, contrast
FreeImage_AdjustBrightness(dib, 50.0);
FreeImage_AdjustContrast(dib, 15.0);
```

Better and even faster would be snippet 3:

```
// snippet 3:
FreeImage_AdjustColors(dib, 50.0, 15.0, 1.0, FALSE);
```

A description of parameters follows:

- *dib* Input/output image to be processed.
- *brightness* Percentage brightness value in [-100..100]. A value of 0 means no change, less than 0 will make the image darker and greater than 0 will make the image brighter.
- *contrast* Percentage contrast value in [-100..100]. A value of 0 means no change, less than 0 will decrease the contrast and greater than 0 will increase the contrast of the image.
- *gamma* Gamma value to be used for gamma correction. A value of 1.0 leaves the image alone, less than one darkens it, and greater than one lightens it. This parameter must greater than zero. Otherwise, it will be ignored and no gamma correction will be performed using the lookup table created.
- *invert* If set to TRUE, the image will be inverted.

The function returns TRUE on success, FALSE otherwise (e.g. when the bitdeph of the source dib cannot be handled).

FreeImage_ApplyColorMapping

1 4 8 16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_ApplyColorMapping(FIBITMAP *dib, RGBQUAD
*srccolors, RGBQUAD *dstcolors, unsigned count, BOOL ignore_alpha, BOOL swap);
```

Applies color mapping for one or several colors on a 1-, 4- or 8-bit palletized or a 16-, 24- or 32-bit high color image.

This function maps up to *count* colors specified in *srccolors* to these specified in *dstcolors*. Thereby, color *srccolors[N]*, if found in the image, will be replaced by color *dstcolors[N]*.

Both arrays *srccolors* and *dstcolors* are assumed not to hold less than *count* colors.

If parameter *swap* is TRUE, additionally all colors specified in *dstcolors* are also mapped to these specified in *srccolors*. For high color images, the actual image data will be modified whereas, for palletized images only the palette will be changed.

For 16-bit images, all colors specified are transparently converted to their proper 16-bit representation (either in RGB555 or RGB565 format, which is determined by the image's red-green- and blue-mask).

Note, that this behaviour is different from what FreeImage_ApplyPaletteIndexMapping does, which modifies the actual image data on palletized images.

A description of parameters follows:

- *dib* Input/output image to be processed.
- *srccolors* Array of colors to be used as the mapping source.
- *dstcolors* Array of colors to be used as the mapping destination.
- *count* The number of colors to be mapped. This is the size of both *srccolors* and *dstcolors*.
- *ignore_alpha* If TRUE, 32-bit images and colors are treated as 24-bit.
- *swap* If TRUE, source and destination colors are swapped, that is, each destination color is also mapped to the corresponding source color.

The function returns the number of pixels changed or zero, if no pixels were changed.

FreeImage_SwapColors

1 4 8 16 24 32

```
DLL_API unsigned DLL_CALLCONV FreeImage_SwapColors(FIBITMAP *dib, RGBQUAD *color_a,
RGBQUAD *color_b, BOOL ignore_alpha);
```

Swaps two specified colors on a 1-, 4- or 8-bit palletized or a 16-, 24- or 32-bit high color image.

This function swaps the two specified colors *color_a* and *color_b* on a palletized or high color image *dib*. For high color images, the actual image data will be modified whereas, for palletized images only the palette will be changed. When the *ignore_alpha* parameter is set to TRUE, 32-bit images and colors are treated as 24-bit.

The function returns the number of pixels changed or zero, if no pixels were changed.

Note, that this behaviour is different from what FreeImage_SwapPaletteIndices does, which modifies the actual image data on palletized images.

This function is just a thin wrapper for FreeImage_ApplyColorMapping and resolves to:

return FreeImage_ApplyColorMapping(dib, color_a, color_b, 1, ignore_alpha, TRUE);

FreeImage_ApplyPaletteIndexMapping

1 4 8

```
DLL_API unsigned DLL_CALLCONV FreeImage_ApplyPaletteIndexMapping(FIBITMAP *dib, BYTE
*srcindices, BYTE *dstindices, unsigned count, BOOL swap);
```

Applies palette index mapping for one or several indices on a 1-, 4- or 8-bit palletized image.

This function maps up to *count* palette indices specified in *srcindices* to these specified in *dstindices*. Thereby, index *srcindices[N]*, if present in the image, will be replaced by index *dstindices[N]*. If parameter *swap* is TRUE, additionally all indices specified in *dstindices* are also mapped to these specified in *srcindices*. Both arrays *srcindices* and *dstindices* are assumed not to hold less than *count* indices.

Note, that this behaviour is different from what FreeImage_ApplyColorMapping does, which modifies the actual image data on palletized images.

A description of parameters follows:

- *dib* Input/output image to be processed.
- *srcindices* Array of palette indices to be used as the mapping source.
- *dstindices* Array of palette indices to be used as the mapping destination.
- *count* The number of palette indices to be mapped. This is the size of both *srcindices* and *dstindices*.
- *swap* If TRUE, source and destination palette indices are swapped, that is, each destination index is also mapped to the corresponding source index.

The function returns the number of pixels changed or zero, if no pixels were changed.

FreeImage_SwapPaletteIndices

1 4 8

```
DLL_API unsigned DLL_CALLCONV FreeImage_SwapPaletteIndices(FIBITMAP *dib, BYTE
*index_a, BYTE *index_b);
```

Swaps two specified palette indices on a 1-, 4- or 8-bit palletized image.

This function swaps the two specified palette indices *index_a* and *index_b* on a palletized image. Therefore, not the palette, but the actual image data will be modified.

The function returns the number of pixels changed or zero, if no pixels were changed.

Note, that this behaviour is different from what FreeImage_SwapColors does on palletized images, which only swaps the colors in the palette.

This function is just a thin wrapper for FreeImage_ApplyPaletteIndexMapping and resolves to:

return FreeImage_ApplyPaletteIndexMapping(dib, index_a, index_b, 1, TRUE);

Channel processing

FreeImage_GetChannel

24 32 48_{RGB16} 64_{RGBA16} 96_{RGBF} 128_{RGBAF}

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_GetChannel(FIBITMAP *dib,
FREE_IMAGE_COLOR_CHANNEL channel);
```

Retrieves the red, green, blue or alpha channel of a RGB[A] image. dib is the input image to be processed and channel is the color channel to extract. The function returns the extracted channel if successful and returns NULL otherwise.

FreeImage_SetChannel

24 32 48_{RGB16} 64_{RGBA16} 96_{RGBF} 128_{RGBAF}

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetChannel(FIBITMAP *dst, FIBITMAP *src,
FREE_IMAGE_COLOR_CHANNEL channel);
```

Insert a greyscale dib into a RGB[A] image. *src* and *dst* must have the same width and height. *dst* is the destination image to modify, *src* is the greyscale image to insert and *channel* is the color channel to replace. The function returns TRUE if successful, FALSE otherwise.

FreeImage_GetComplexChannel

$2x64_{COMPLEX}$

```
DLL_API FIBITMAP * DLL_CALLCONV FreeImage_GetComplexChannel(FIBITMAP *src,
FREE_IMAGE_COLOR_CHANNEL channel);
```

Retrieves the real part, imaginary part, magnitude or phase of a complex image (image whose type is FIT_COMPLEX). The function returns the extracted channel as a FIT_DOUBLE image if successful and returns NULL otherwise.

FreeImage_SetComplexChannel

$2x64_{COMPLEX}$

```
DLL_API BOOL DLL_CALLCONV FreeImage_SetComplexChannel(FIBITMAP *dst, FIBITMAP *src,
FREE_IMAGE_COLOR_CHANNEL channel);
```

Set the real or imaginary part of a complex image (image whose type is FIT_COMPLEX). Both src and dst must have the same width and height. Upon entry, dst is the image to modify (image of type FIT_COMPLEX) and src is the channel to replace (image of type FIT_DOUBLE). The function returns TRUE if successful, FALSE otherwise.

Copy / Paste / Composite routines

FreeImage_Copy

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Copy(FIBITMAP *dib, int left, int top, int
right, int bottom);
```

Copy a sub part of the current dib image. The rectangle defined by the (left, top, right, bottom) parameters is first normalized such that the value of the left coordinate is less than the right and the top is less than the bottom. Then, the returned bitmap is defined by a width equal to (right - left) and a height equal to (bottom - top).

A description of parameters follows:

left : specifies the left position of the cropped rectangle.

top : specifies the top position of the cropped rectangle.

right : specifies the right position of the cropped rectangle.

bottom : specifies the bottom position of the cropped rectangle.

The function returns the subimage if successful and returns NULL otherwise.

Normalization of the rectangle defined by the (left, top, right, bottom) parameters means that the coordinate system uses usual graphics conventions. This is used to ease interactions with a mouse.

FreeImage_Paste

```
DLL_API BOOL DLL_CALLCONV FreeImage_Paste(FIBITMAP *dst, FIBITMAP *src, int left, int
top, int alpha);
```

Alpha blend or combine a sub part image with the current dib image.

For images of type FITBITMAP only: The bit depth of the *dst* bitmap must be greater than or equal to the bit depth of the *src*. Upper promotion of *src* is done internally, without modifying *src*. Supported *dst* bit depth equals to 1, 4, 8, 16, 24 or 32.

For any other image type: The image type of the *dst* bitmap must be equal to the image type of the *src*. The *alpha* parameter is always ignored and the source image is combined to the destination image.

The parameters are described below:

dst :destination image

src : source subimage

left : specifies the left position of the sub image.

top : specifies the top position of the sub image.

alpha : alpha blend factor. The source and destination images are alpha blended if alpha=0..255. If alpha > 255, then the source image is combined to the destination image.

The function returns TRUE if successful, FALSE otherwise.

You cannot perform paste operations between palettized images, *unless* both src and dst images use the same palette. For bit depths less than or equal to 8-bit, paste operations usually only work with greyscale images.

Coordinate system defined by the (left, top) parameters uses usual graphics conventions. This is used to ease interactions with a mouse.

FreeImage_CreateView

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_CreateView(FIBITMAP *dib, unsigned left,
unsigned top, unsigned right, unsigned bottom);
```

Creates a dynamic read/write view into a FreeImage bitmap.

A dynamic view is a FreeImage bitmap with its own width and height, that, however, shares its bits with another FreeImage bitmap. Typically, views are used to define one or more rectangular sub-images of an existing bitmap. All FreeImage operations, like saving, displaying and all the toolkit functions, when applied to the view, only affect the view's rectangular area.

Although the view's backing image's bits not need to be copied around, which makes the view much faster than similar solutions using FreeImage_Copy, a view uses some private memory that needs to be freed by calling FreeImage_Unload on the view's handle to prevent memory leaks.

Only the backing image's pixels are shared by the view. For all other image data, notably for the resolution, background color, color palette, transparency table and for the ICC profile, the view gets a private copy of the data. By default, the backing image's metadata is NOT copied to the view.

As with all FreeImage functions that take a rectangle region, top and left positions are included, whereas right and bottom positions are excluded from the rectangle area.

Since the memory block shared by the backing image and the view must start at a byte boundary, the value of parameter *left* must be a multiple of 8 for 1-bit images and a multiple of 2 for 4-bit images.

The parameters are described below:

dib The FreeImage bitmap on which to create the view.

left The left position of the view's area.

top The top position of the view's area.

right The right position of the view's area.

bottom The bottom position of the view's area.

The function returns a handle to the newly created view or returns NULL if the view was not created.

FreeImage_Composite

8 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_Composite(FIBITMAP *fg, BOOL useFileBkg
FI_DEFAULT(FALSE), RGBQUAD *appBkColor FI_DEFAULT(NULL), FIBITMAP *bg
FI_DEFAULT(NULL));
```

This function composite a transparent foreground image against a single background color or against a background image. Upon entry, *fg* defines the foreground image and the transparency mask (implicitly included in the foreground image as a transparency table for 8-bit dib or as a alpha channel for 32-bit dib).

The equation for computing a composited sample value is:

*output = alpha * foreground + (1-alpha) * background*

where alpha and the input and output sample values are expressed as fractions in the range 0 to 1. For colour images, the computation is done separately for R, G, and B samples.

The following pseudo-code illustrates the internal use of the other parameters:

```
if(useFileBkg && FreeImage_HasBackgroundColor(fg)) {
  // Use the file background as the single background color
} else {
  // no file background color …
  // use application background color ?
  if(appBkColor) {
    // use the application background as the single background color
  }
  // no application background color …
  // use a background image ?
  else if(bg) {
    // use bg as the background image
    // bg MUST BE a 24-bit image with the same width and height as fg
  }
  else {
    // default case
    // use a checkerboard as the background image
  }
}
```

(a) (b) (c)

Figure 1: Illustration of the FreeImage_Composite function.

An illustration of the FreeImage_Composite function is given in Figure 1. This sample image is a 8-bit transparent PNG with a light-blue file color background. Each image was generated using the following calls:

```
FIBITMAP *fg = FreeImage_Load(FIF_PNG, "test.png", PNG_DEFAULT);
// image (a) : use a checkerboard background
FIBITMAP *display_dib_a = FreeImage_Composite(fg);
// image (b) : use the image file background if there is one
FIBITMAP *display_dib_b = FreeImage_Composite(fg, TRUE);
// image (c) : use a user specified background
RGBQUAD appColor = { 0, 255, 0, 0 };
FIBITMAP *display_dib_c = FreeImage_Composite(fg, FALSE, &appColor);
```

Reference

Portable Network Graphics (PNG) Specification (Second Edition).
[Online] http://www.w3.org/TR/PNG/

FreeImage_PreMultiplyWithAlpha

32

```
DLL_API BOOL DLL_CALLCONV FreeImage_PreMultiplyWithAlpha(FIBITMAP *dib);
```

Pre-multiplies a 32-bit image's red-, green- and blue channels with it's alpha channel for to be used with the Windows GDI function AlphaBlend(). The transformation changes the red-, green- and blue channels according to the following equation:

*channel(x, y) = channel(x, y) * alpha_channel(x, y) / 255*

The function returns TRUE on success, FALSE otherwise (e.g. when the bitdepth of the source dib cannot be handled).

JPEG lossless transformations

FreeImage_JPEGTransform

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGTransform(const char *src_file, const char
*dst_file, FREE_IMAGE_JPEG_OPERATION operation, BOOL perfect FI_DEFAULT(TRUE));
```

Performs a lossless rotation or flipping on a JPEG file. Upon entry, *src_file* is the source JPEG file and *dst_file* the destination JPEG file. Using the same file for source and destination is allowed: the source file will be transformed and overwritten. The *operation* parameter specifies the kind of transformation to apply. The following transformations are possible:

Operation	Description
FIJPEG_OP_NONE	No transformation (nothing is done)
FIJPEG_OP_FLIP_H	Horizontal flip
FIJPEG_OP_FLIP_V	Vertical flip
FIJPEG_OP_TRANSPOSE	Transpose across upper-left to lower-right axis
FIJPEG_OP_TRANSVERSE	Transpose across upper-right to lower-left axis
FIJPEG_OP_ROTATE_90	90-degree clockwise rotation
FIJPEG_OP_ROTATE_180	180-degree rotation
FIJPEG_OP_ROTATE_270	270-degree clockwise rotation (or 90-degree ccw)

Table 16: FREE_IMAGE_JPEG_OPERATION constants.

FreeImage_JPEGTransform works by rearranging the compressed data (DCT coefficients), without ever fully decoding the image. Therefore, its transformations are lossless: there is no image degradation at all, which would not be true if you used *FreeImage_Load* followed by *FreeImage_Save* to accomplish the same conversion.

The FIJPEG_OP_TRANSPOSE transformation has no restrictions regarding image dimensions. The other transformations operate rather oddly if the image dimensions are not a multiple of the iMCU size (usually 8 or 16 pixels), because they can only transform complete blocks of DCT coefficient data in the desired way.

The default function behavior when transforming an odd-size image is designed to discard any untransformable edge pixels rather than having a strange-looking strip along the right and/or bottom edges of a transformed image. Obviously, when applied to odd-size images, the transformation is not reversible, so strictly speaking, the operation is not lossless.

In order to avoid lossy transformation, you can set the *perfect* parameter to TRUE. When using this parameter, any non reversible transform is avoided, an error message is thrown (that you can log using FreeImage_SetOutputMessage) and the function will return FALSE.

FreeImage_JPEGTransformU

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGTransformU(const wchar_t *src_file, const
wchar_t *dst_file, FREE_IMAGE_JPEG_OPERATION operation, BOOL perfect
FI_DEFAULT(TRUE));
```

This function works exactly like FreeImage_JPEGTransform but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FALSE.

FreeImage_JPEGCrop

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGCrop(const char *src_file, const char
*dst_file, int left, int top, int right, int bottom);
```

Performs a lossless crop on a JPEG file. Upon entry, *src_file* is the source JPEG file and *dst_file* the destination JPEG file. Using the same file for source and destination is allowed: the source file will be transformed and overwritten.

The rectangle defined by the (*left, top, right, bottom*) parameters is first normalized such that the value of the left coordinate is less than the right and the top is less than the bottom. Then, the returned bitmap is defined by a width greater than or equal to (*right* - *left*) and a height greater than or equal to (*bottom* - *top*) – see the explanation below.

A description of parameters follows:

left : specifies the left position of the cropped rectangle.

top : specifies the top position of the cropped rectangle.

right : specifies the right position of the cropped rectangle.

bottom : specifies the bottom position of the cropped rectangle.

FreeImage_JPEGCrop works by rearranging the compressed data (DCT coefficients), without ever fully decoding the image. Therefore, the crop operation is lossless: there is no image degradation at all, which would not be true if you used *FreeImage_Load* followed by *FreeImage_Copy*, then *FreeImage_Save* to accomplish the same conversion.

To perform this lossless operation, however, the width and height of the cropped rectangle must be adjusted so that the image dimensions are a multiple of the iMCU size (usually 8 or 16 pixels), because the function can only transform complete blocks of DCT coefficient data in the desired way. That's why the output width or height can be slightly greater than the requested image size.

FreeImage_JPEGCropU

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGCropU(const wchar_t *src_file, const wchar_t
*dst_file, int left, int top, int right, int bottom);
```

This function works exactly like FreeImage_JPEGCrop but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FALSE.

FreeImage_JPEGTransformCombined

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGTransformCombined(const char *src_file, const
char *dst_file, FREE_IMAGE_JPEG_OPERATION operation, int* left, int* top, int* right,
int* bottom, BOOL perfect FI_DEFAULT(TRUE));
```

Performs a combination of lossless rotation or flipping and lossless crop on a JPEG file. Upon entry, *src_file* is the source JPEG file and *dst_file* the destination JPEG file. Using the same file for source and destination is allowed: the source file will be transformed and overwritten. The *operation* parameter specifies the kind of transformation to apply (see Table 16). Any non reversible transformation can be avoided using the *perfect* parameter. The cropped rectangle is defined by the (*left, top, right, bottom*) parameters (see FreeImage_JPEGCrop).

The function works by first performing the transformation using the *operation* parameter, then by adjusting the cropped rectangle, i.e. the *left, top, right, bottom* parameters, to the iMCU size and to the destination width and height, and lastly, by cropping the transformed image.

You can specify *FIJPEG_OP_NONE* for the *operation* parameter if you don't need any transformation, and you can specify *0* for the (*left, top, right, bottom*) parameters if you don't need any crop operation.

You can simulate the effect of a combined transform and crop operation by setting the *dst_file* parameter to NULL.

```
void testJPEGTransformation() {
  BOOL bResult;
  // assume source has a width x height = 600x400 and a iMCU size = 16
  const char *src_file = "input_test.jpg";
  const char *dst_file = "output_test.jpg";

  // cropped rectangle
  int left, top, right, bottom;

  // we require a perfect transformation
  BOOL perfect = TRUE;

  left = 50; top = 100; right = 650; bottom = 500;

  // (optional) simulate the transform and get the new rectangle coordinates
  // (adjusted to the iMCU size and to the dst image width & height)
  bResult = FreeImage_JPEGTransformCombined(src_file, NULL, FIJPEG_OP_ROTATE_90,
&left, &top, &right, &bottom, perfect);

  assert(bResult == TRUE);

  // check the cropped rectangle, adjusted for a WxH = 400x600 after rotation
  assert( (left == 48) && (right == 400) && (top == 96) && (bottom == 500) );

  // then apply the transform
  bResult = FreeImage_JPEGTransformCombined(src_file, dst_file, FIJPEG_OP_ROTATE_90,
&left, &top, &right, &bottom, perfect);

  assert(bResult == TRUE);
}
```

FreeImage_JPEGTransformCombinedU

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGTransformCombinedU(const wchar_t *src_file,
const wchar_t *dst_file, FREE_IMAGE_JPEG_OPERATION operation, int* left, int* top,
int* right, int* bottom, BOOL perfect FI_DEFAULT(TRUE));
```

This function works exactly like FreeImage_JPEGTransformCombined but supports UNICODE filenames. Note that this function only works on MS Windows operating systems. On other systems, the function does nothing and returns FALSE.

FreeImage_JPEGTransformCombinedFromMemory

```
BOOL DLL_CALLCONV FreeImage_JPEGTransformCombinedFromMemory(FIMEMORY* src_stream,
FIMEMORY* dst_stream, FREE_IMAGE_JPEG_OPERATION operation, int* left, int* top, int*
right, int* bottom, BOOL perfect FI_DEFAULT(TRUE));
```

This function works exactly like FreeImage_JPEGTransformCombined but supports memory handles instead of filenames.

Note that you cannot save a JPEG file to a destination memory stream when the destination memory handle is a (read_only) wrapped user's buffer. Thus, the *dst_stream* parameter MUST be a read / write memory handle, i.e. a handle opened using *FreeImage_OpenMemory(NULL, 0)*.

The *src_stream* and *dst_stream* parameters can point to the same stream, provided the stream is not a read-only stream.

FreeImage_JPEGTransformFromHandle

```
DLL_API BOOL DLL_CALLCONV FreeImage_JPEGTransformFromHandle(FreeImageIO* src_io,
fi_handle src_handle, FreeImageIO* dst_io, fi_handle dst_handle,
FREE_IMAGE_JPEG_OPERATION operation, int* left, int* top, int* right, int* bottom,
BOOL perfect FI_DEFAULT(TRUE));
```

This function works exactly like FreeImage_JPEGTransformCombined but supports your own handles instead of filenames. Using the same handle for the source and the destination is allowed, provided this handle is a read / write handle. Using a different kind of handle for the source and for the destination is also possible, so that for example, you can read a memory and write to a file, or read a file and write to a memory.

Background filling

FreeImage_FillBackground

```
DLL_API BOOL DLL_CALLCONV FreeImage_FillBackground(FIBITMAP *dib, const void *color,
int options FI_DEFAULT(0));
```

This function sets all pixels of an image to the color provided through the *color* parameter. A description of parameters follows:

- *dib* The image to be filled.
- *color* A pointer to the color value to be used for filling the image. The memory pointed to by this pointer is always assumed to be at least as large as the image's color value, but never smaller than the size of an RGBQUAD structure.
- *options* Options that affect the color search process for palletized images.

The function returns TRUE on success, FALSE otherwise. This function fails if any of *dib* and *color* is NULL.

The *color* pointer must point to a memory location which is at least 4-bytes for FIT_BITMAP types, and at least as large as the image's color value, if this size is greater than 4 bytes.

FIT_BITMAP image type

The *color* parameter MUST BE specified through a RGBQUAD structure for all images of type FIT_BITMAP (including all palletized images), the size of this memory is thus the size of the RGBQUAD structure, which uses 4 bytes.

24- and 32-bit images

For 24- and 32-bit images, the red, green and blue members of the RGBQUAD structure are directly used for the image's red, green and blue channel respectively. Although alpha transparent RGBQUAD colors are supported, the alpha channel of a 32-bit image never gets modified by this function. A fill color with an alpha value smaller than 255 gets blended with the image's actual background color, which is determined from the image's bottom-left pixel. So, currently using alpha enabled colors, assumes the image to be unicolor before the fill operation. However, the RGBQUAD's rgbReserved member is only taken into account, if option *FI_COLOR_IS_RGBA_COLOR* has been specified.

Parameter	Description
FI_COLOR_IS_RGBA_COLOR	RGBQUAD color is a RGBA color (contains a valid alpha channel)

Table 17: Background filling options for 24- or 32-bit images

16-bit RGB images

For 16-bit RGB images, the red-, green- and blue components of the specified color are transparently translated into either the 16-bit 555 or 565 representation. This depends on the image's actual red- green- and blue masks.

1-, 4- or 8-bit palletized images

Special attention must be payed for palletized images. Generally, the RGB color specified is looked up in the image's palette. The found palette index is then used to fill the image. There are some *option* flags that affect this lookup process.

Parameter	Description
FI_COLOR_IS_RGB_COLOR	Uses the color that is nearest to the specified color. This is the default behavior and should always find a color in the palette. However, the visual result may be far from what was expected and mainly depends on the image's palette.
FI_COLOR_FIND_EQUAL_COLOR	Searches the image's palette for the specified color but only uses the returned palette index, if the specified color exactly matches the palette entry. Of course, depending on the image's actual palette entries, this operation may fail. In this case, the function falls back to option FI_COLOR_ALPHA_IS_INDEX and uses the RGBQUAD's rgbReserved member (or its low nibble for 4-bit images or its least significant bit (LSB) for 1-bit images) as the palette index used for the fill operation.
FI_COLOR_ALPHA_IS_INDEX	The color's rgbReserved member (alpha) contains the palette index to be used
FI_COLOR_PALETTE_SEARCH_MASK	Combination of `(FI_COLOR_FIND_EQUAL_COLOR | FI_COLOR_ALPHA_IS_INDEX)` No color lookup is performed

Table 18: Background filling options for palletized images

Any other image types

The *color* parameter MUST point to a double, if the image to be filled is of type FIT_DOUBLE, point to a RGB[A]16 structure if the image is of type RGB[A]16, point to a RGB[A]F structure if the image is of type FIT_RGB[A]F and so on.

FreeImage_EnlargeCanvas

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_EnlargeCanvas(FIBITMAP *src, int left, int
top, int right, int bottom, const void *color, int options);
```

Enlarges or shrinks an image selectively per side and fills newly added areas with the specified background color. A description of parameters follows:

- *dib* The image to be enlarged or shrunken.
- *left* The number of pixels, the image should be enlarged on its left side. Negative values shrink the image on its left side.
- *top* The number of pixels, the image should be enlarged on its top side. Negative values shrink the image on its top side.
- *right* The number of pixels, the image should be enlarged on its right side. Negative values shrink the image on its right side.
- *bottom* The number of pixels, the image should be enlarged on its bottom side. Negative values shrink the image on its bottom side.
- *color* The color, the enlarged sides of the image should be filled with.
- *options* Options that affect the color search process for palletized images.

The function returns a pointer to a newly allocated enlarged or shrunken image on success, NULL otherwise. This function fails if either the input image is NULL or the pointer to the color is NULL, while at least one of left, top, right and bottom is greater than zero. This function also returns NULL, if the new image's size will be negative in either x- or y-direction.

This function enlarges or shrinks an image selectively per side. The main purpose of this function is to add borders to an image. To add a border to any of the image's sides, a positive integer value must be passed in any of the parameters left, top, right or bottom. This value represents the border's width in pixels. Newly created parts of the image (the border areas) are filled with the specified color. Specifying a negative integer value for a certain side, will shrink or crop the image on this side. Consequently, specifying zero for a certain side will not change the image's extension on that side.

So, calling this function with all parameters left, top, right and bottom set to zero, is effectively the same as calling function FreeImage_Clone; setting all parameters left, top, right and bottom to value equal to or smaller than zero, may easily be substituted by a call to function FreeImage_Copy. Both these cases produce a new image, which is guaranteed not to be larger than the input image. Thus, since the specified color is not needed in these cases, the pointer *color* may be NULL.

Both parameters *color* and *options* work according to function FreeImage_FillBackground. So, please refer to the documentation of FreeImage_FillBackground to learn more about parameters *color* and *options*. For palletized images, the palette of the input image *src* is transparently copied to the newly created enlarged or shrunken image, so any color look-ups are performed on this palette.

Here are some examples that illustrate how to use the parameters *left*, *top*, *right* and *bottom*:

```
// create a white color
RGBQUAD c;
c.rgbRed = 0xFF;
c.rgbGreen = 0xFF;
c.rgbBlue = 0xFF;
c.rgbReserved = 0x00;

// add a white, symmetric 10 pixel wide border to the image
dib2 = FreeImage_EnlargeCanvas(dib, 10, 10, 10, 10, &c, FI_COLOR_IS_RGB_COLOR);

// add white, 20 pixel wide stripes to the top and bottom side of the image
dib3 = FreeImage_EnlargeCanvas(dib, 0, 20, 0, 20, &c, FI_COLOR_IS_RGB_COLOR);

// add white, 30 pixel wide stripes to the right side of the image and
// cut off the 40 leftmost pixel columns
dib3 = FreeImage_EnlargeCanvas(dib, -40, 0, 30, 0, &c, FI_COLOR_IS_RGB_COLOR);
```

FreeImage_AllocateEx

1 4 8 16 24 32

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_AllocateEx(int width, int height, int bpp,
const RGBQUAD *color, int options, const RGBQUAD *palette, unsigned red_mask
FI_DEFAULT(0), unsigned green_mask FI_DEFAULT(0), unsigned blue_mask FI_DEFAULT(0));
```

Allocates a new image of the specified width, height and bit depth and optionally fills it with the specified color.

FreeImage_AllocateEx is an alias for FreeImage_AllocateExT and can be replaced by this call:
FreeImage_AllocateExtT(FIT_BITMAP, width, height, bpp, color, options, palette, red_mask, green_mask, blue_mask);

FreeImage_AllocateExT

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_AllocateExT(FREE_IMAGE_TYPE type, int width,
int height, int bpp, const void *color, int options, const RGBQUAD *palette, unsigned
red_mask FI_DEFAULT(0), unsigned green_mask FI_DEFAULT(0), unsigned blue_mask
FI_DEFAULT(0));
```

Allocates a new image of the specified type, width, height and bit depth and optionally fills it with the specified color. A description of parameters follows:

- *type* Specifies the image type of the new image.
- *width* The desired width in pixels of the new image.
- *height* The desired height in pixels of the new image.
- *bpp* The desired bit depth of the new image.
- *color* A pointer to the color value to be used for filling the image. The memory pointed to by this pointer is always assumed to be at least as large as the image's color value but never smaller than the size of an RGBQUAD structure.
- *options* Options that affect the color search process for palletized images.
- *red_mask* Specifies the bits used to store the red components of a pixel.
- *green_mask* Specifies the bits used to store the green components of a pixel.
- *blue_mask* Specifies the bits used to store the blue components of a pixel.

The function returns a pointer to a newly allocated image on success, returns NULL otherwise.

This function is an extension to FreeImage_AllocateT, which additionally supports specifying a palette to be set for the newly created image, as well as specifying a background color, the newly created image should initially be filled with.

Basically, this function internally relies on function FreeImage_AllocateT, followed by a call to FreeImage_FillBackground. This is why both parameters *color* and *options* behave the same as it is documented for function FreeImage_FillBackground. So, please refer to the documentation of FreeImage_FillBackground to learn more about parameters *color* and *options*.

The palette specified through parameter *palette* is only copied to the newly created image, if its image type is FIT_BITMAP and the desired bit depth is smaller than or equal to 8-bit per pixel. In other words, the *palette* parameter is only taken into account for palletized images. However, if the preceding conditions match and if *palette* is not NULL, the memory pointed to by the *palette* pointer is assumed to be at least as large as size of a fully populated palette for the desired bit depth. So, for an 8-bit image, this size is 256 x sizeof(RGBQUAD), for an 4-bit image it is 16 x sizeof(RGBQUAD) and it is 2 x sizeof(RGBQUAD) for a 1-bit image. In other words, this function does not support partial palettes.

However, specifying a palette is not necessarily needed, even for palletized images. This function is capable of implicitly creating a palette, if parameter palette is NULL. If the specified background color is a greyscale value (red = green = blue) or if option FI_COLOR_ALPHA_IS_INDEX is specified, a greyscale palette is created. For a 1-bit image, only if the specified background color is either black or white, a monochrome palette, consisting of black and white only is created. In any case, the darker colors are stored at the smaller palette indices.

If the specified background color is not a greyscale value, or is neither black nor white for a 1-bit image, solely this single color is injected into the otherwise black-initialized palette. For this operation, option FI_COLOR_ALPHA_IS_INDEX is implicit, so the specified color is applied to the palette entry, specified by the background color's rgbReserved member. The image is then filled with this palette index.

This function returns a newly created image as function FreeImage_AllocateT does, if both parameters *color* and *palette* are NULL. If only color is NULL, the palette pointed to by parameter *palette* is initially set for the new image, if a palletized image of type FIT_BITMAP is created. However, in the latter case, this function returns an image, whose pixels are all initialized with zeros so, the image will be filled with the color of the first palette entry.

Miscellaneous algorithms

FreeImage_MultigridPoissonSolver

32FLOAT

```
DLL_API FIBITMAP *DLL_CALLCONV FreeImage_MultigridPoissonSolver(FIBITMAP *Laplacian,
int ncycle FI_DEFAULT(3));
```

Poisson solver based on a multigrid algorithm.

This routine solves a Poisson equation, remap result pixels to [0..1] and returns the solution as a FIT_FLOAT image type.

Internally, the input image is first stored inside a square image whose size is $(2^j + 1) \times (2^j + 1)$ for some integer j, where j is such that 2^j is the nearest larger dimension corresponding to MAX(image width, image height). However, the resulting output image will have the same size (width and height) as the input image.

A description of parameters follows:

Laplacian Laplacian image

ncycle Number of cycles in the multigrid algorithm (usually 2 or 3)

The function returns the solved PDE equations if successful, returns NULL otherwise.

Reference

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., Numerical Recipes in C: The Art of Scientific Computing, 2nd ed. Cambridge University Press. 1992.

Appendix

Supported file formats

The following table shows the capabilities of each FreeImage plugin.

FORMAT	DESCRIPTION	EXTENSIONS	SUPPORT LOADING	SUPPORT WRITING	SUPPORT OPTIONS	SUPPORT TRANSPARENCY		SUPPORT ICC PROFILES	SUPPORT HEADER ONLY	EXPORTED BITDEPTH						EXPORTED TYPE											
						8	32			1	4	8	16	24	32	FIT_BITMAP	FIT_UINT16	FIT_INT16	FIT_UINT32	FIT_INT32	FIT_FLOAT	FIT_DOUBLE	FIT_COMPLEX	FIT_RGB16	FIT_RGBA16	FIT_RGBF	FIT_RGBAF
FIF_BMP	Windows or OS/2 Bitmap	bmp	•	•	•	-	•	-	•	•	•	•	•	•	•	•	-	-	-	-	-	-	-	-	-	-	-
FIF_CUT	Dr. Halo	cut	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_DDS	DirectX Surface	dds	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_EXR	ILM OpenEXR	exr	•	•	•	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	•	-	-	-	-	•	•
FIF_FAXG3	Raw fax format CCITT G.3	g3	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_GIF	Graphics Interchange Format	gif	•	•	•	•	-	-	-	-	-	•	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-
FIF_HDR	High Dynamic Range	hdr	•	•	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	•	-
FIF_ICO	Windows Icon	ico	•	•	•	-	•	-	•	•	•	•	•	•	•	•	-	-	-	-	-	-	-	-	-	-	-
FIF_IFF	IFF Interleaved Bitmap	iff,lbm	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_J2K	JPEG-2000 codestream	j2k,j2c	•	•	•	-	•	-	-	-	-	•	-	•	•	•	•	-	-	-	-	-	-	-	•	•	-	-
FIF_JNG	JPEG Network Graphics	jng	•	•	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_JP2	JPEG-2000 File Format	jp2	•	•	•	-	•	-	-	-	-	•	-	•	•	•	-	-	-	-	-	-	-	-	•	•	-	-
FIF_JPEG	JPEG - JFIF Compliant	jpg,jif,jpeg,jpe	•	•	•	-	-	•	•	-	-	•	-	•	-	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_JXR	JPEG XR image format	jxr,wdp,hdp	•	•	•	-	•	•	•	•	–	•	•	•	•	•	•	•	-	-	-	•	-	-	•	•	•	•
FIF_KOALA	C64 Koala Graphics	koa	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_MNG	Multiple Network Graphics	mng	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PBM	Portable Bitmap (ASCII)	pbm	•	•	•	-	-	-	•	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PBMRAW	Portable Bitmap (RAW)	pbm	•	•	•	-	-	-	•	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PCD	Kodak PhotoCD	pcd	•	-	•	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PCX	Zsoft Paintbrush	pcx	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PFM	Portable Floatmap	pfm	•	•	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	•	-	-	-	-	•	-	-
FIF_PGM	Portable Greymap (ASCII)	pgm	•	•	•	-	-	-	•	-	-	•	-	-	-	•	•	-	-	-	-	-	-	-	-	-	-	-
FIF_PGMRAW	Portable Greymap (RAW)	pgm	•	•	•	-	-	-	•	-	-	•	-	-	-	•	•	-	-	-	-	-	-	-	-	-	-	-
FIF_PICT	Macintosh PICT	pct,pict,pic	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_PNG	Portable Network Graphics	png	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	-	-	-	-	-	-	•	•	-	-
FIF_PPM	Portable Pixelmap (ASCII)	ppm	•	•	•	-	-	-	•	-	-	-	-	•	-	•	-	-	-	-	-	-	-	-	•	-	-	-
FIF_PPMRAW	Portable Pixelmap (RAW)	ppm	•	•	•	-	-	-	•	-	-	-	-	•	-	•	-	-	-	-	-	-	-	-	•	-	-	-
FIF_PSD	Adobe Photoshop	psd	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_RAS	Sun Raster Image	ras	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_RAW	RAW camera image	*many*	•	-	•	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_SGI	Silicon Graphics SGI image format	sgi	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_TARGA	Truevision Targa	tga,targa	•	•	•	•	•	-	•	-	-	•	•	•	•	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_TIFF	Tagged Image File Format	tif,tiff	•	•	•	•	•	•	•	•	•	•	-	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
FIF_WBMP	Wireless Bitmap	wap,wbmp,wbm	•	•	-	-	-	-	-	•	-	-	-	-	-	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_WEBP	Google WebP image format	webp	•	•	•	-	•	-	•	-	-	-	-	•	•	•	-	-	-	-	-	-	-	-	-	-	-	-
FIF_XBM	X11 Bitmap Format	xbm	•	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FIF_XPM	X11 Pixmap Format	xpm	•	•	-	-	-	-	•	-	-	-	-	•	-	•	-	-	-	-	-	-	-	-	-	-	-	-

Legend:
yes : •
no : -

Supported camera RAW file formats

The following table shows a **non exhaustive list** of the camera RAW formats supported by the RAW plugin. Note that others formats may be supported (the RAW file format extension list produced by camera manufacturers is not documented).

EXTENSION	DESCRIPTION
3fr	Hasselblad Digital Camera Raw Image Format.
arw	Sony Digital Camera Raw Image Format for Alpha devices.
bay	Casio Digital Camera Raw File Format.
bmq	NuCore Raw Image File.
cap	Phase One Digital Camera Raw Image Format.
cine	Phantom Software Raw Image File.
cr2	Canon Digital Camera RAW Image Format version 2.0. These images are based on the TIFF image
crw	Canon Digital Camera RAW Image Format version 1.0.
cs1	Capture Shop Raw Image File.
dc2	Kodak DC25 Digital Camera File.
dcr	Kodak Digital Camera Raw Image Format for these models: Kodak DSC Pro SLR/c, Kodak DSC Pro SLR/n, Kodak DSC Pro 14N, Kodak DSC PRO 14nx.
dng	Adobe Digital Negative: DNG is publicly available archival format for the raw files generated by digital cameras. By addressing the lack of an open standard for the raw files created by individual camera models, DNG helps ensure that photographers will be able to access their files in the future.
drf	Kodak Digital Camera Raw Image Format.
dsc	Kodak Digital Camera Raw Image Format.
erf	Epson Digital Camera Raw Image Format.
fff	Imacon Digital Camera Raw Image Format.
hdr	Leaf Raw Image File.
ia	Sinar Raw Image File.
iiq	Phase One Digital Camera Raw Image Format.
k25	Kodak DC25 Digital Camera Raw Image Format.
kc2	Kodak DCS200 Digital Camera Raw Image Format.
kdc	Kodak Digital Camera Raw Image Format.
mdc	Minolta RD175 Digital Camera Raw Image Format.
mef	Mamiya Digital Camera Raw Image Format.
mos	Mamiya Digital Camera Raw Image Format.
mrw	Minolta Dimage Digital Camera Raw Image Format.
nef	Nikon Digital Camera Raw Image Format.
nrw	Nikon Digital Camera Raw Image Format.
orf	Olympus Digital Camera Raw Image Format.
pef	Pentax Digital Camera Raw Image Format.
ptx	Pentax Digital Camera Raw Image Format.
pxn	Logitech Digital Camera Raw Image Format.
qtk	Apple Quicktake 100/150 Digital Camera Raw Image Format.
raf	Fuji Digital Camera Raw Image Format.
raw	Panasonic Digital Camera Image Format.
rdc	Digital Foto Maker Raw Image File.
rw2	Panasonic LX3 Digital Camera Raw Image Format.
rwz	Rawzor Digital Camera Raw Image Format.
sr2	Sony Digital Camera Raw Image Format.
srf	Sony Digital Camera Raw Image Format for DSC-F828 8 megapixel digital camera or Sony DSC-R1
sti	Sinar Capture Shop Raw Image File.
x3f	Sigma Digital Camera Raw Image Format for devices based on Foveon X3 direct image sensor. Unsupported because of license restrictions.

Choosing the right resampling filter

The effect of a resampling filter is highly dependant on the physical characteristics of the image being resized. Nevertheless, the following hints may prove helpful when deciding which filter to use.

Box filter

Box scaling is the simplest and fastest of the scaling algorithms, from a computational standpoint. Various names are used to denote this simple kernel. They include the box filter, sample-and-hold function, pulse function, Fourier window, 1st order (constant) B-Spline and nearest neighbour. The technique achieves magnification by pixel replication, and minification by sparse point sampling. For large-scale changes, box interpolation produces images with a blocky appearance. In addition, shift errors of up to one-half pixel are possible. These problems make this technique inappropriate when sub-pixel accuracy is required.

Bilinear filter

Bilinear scaling is the second-fastest scaling function. It employs linear interpolation to determine the output image. Bilinear scaling provides reasonably good results at moderate cost for most applications where scale factors are relatively small (4X or less). Often, though, higher fidelity is required and thus more sophisticated filters have been formulated.

B-Spline filter

The B-spline filter produces the smoothest output, but tends to smooth over fine details. This function requires the same processing time as Mitchell and Netravali's Bicubic filter. B-spline filter is recommended for applications where the smoothest output is required.

Bicubic filter

Mitchell and Netravali's bicubic filter is an advanced parameterized scaling filter. It uses a cubic function to produce very smooth output while maintaining dynamic range and sharpness. Bicubic scaling takes approximately twice the processing time as Bilinear. This filter can be used for any scaling application, especially when scaling factors are 2X or greater.

Catmull-Rom filter

When using Mitchell-Netravali filters, you have to set two parameters b and c such that b + 2c = 1, in order to use the numerically most accurate filter. The Bicubic filter uses the default values (b = 1/3, c = 1/3), which were the values recommended by Mitchell and Netravali as yielding the most visually pleasing results in subjective tests of human beings.

When b = 0, this gives the maximum value for c = 0.5, which is the **Catmull-Rom spline** and a good suggestion for sharpness.

The Catmull-Rom filter is generally accepted as the **best cubic interpolant filter**.

Lanczos filter

Lanczos uses a filter based on the sinc function. This is the most theoretically correct filter and produces the best output for photographic images that do not have sharp transitions in

them. However, Lanczos will produce ripple artefacts especially for block text, due to aliasing. Lanczos also requires three times the processing time of Bilinear. Lanczos is not recommended except in very rare applications using band-limited photographic images with no sharp edges.

Comparison of resampling methods

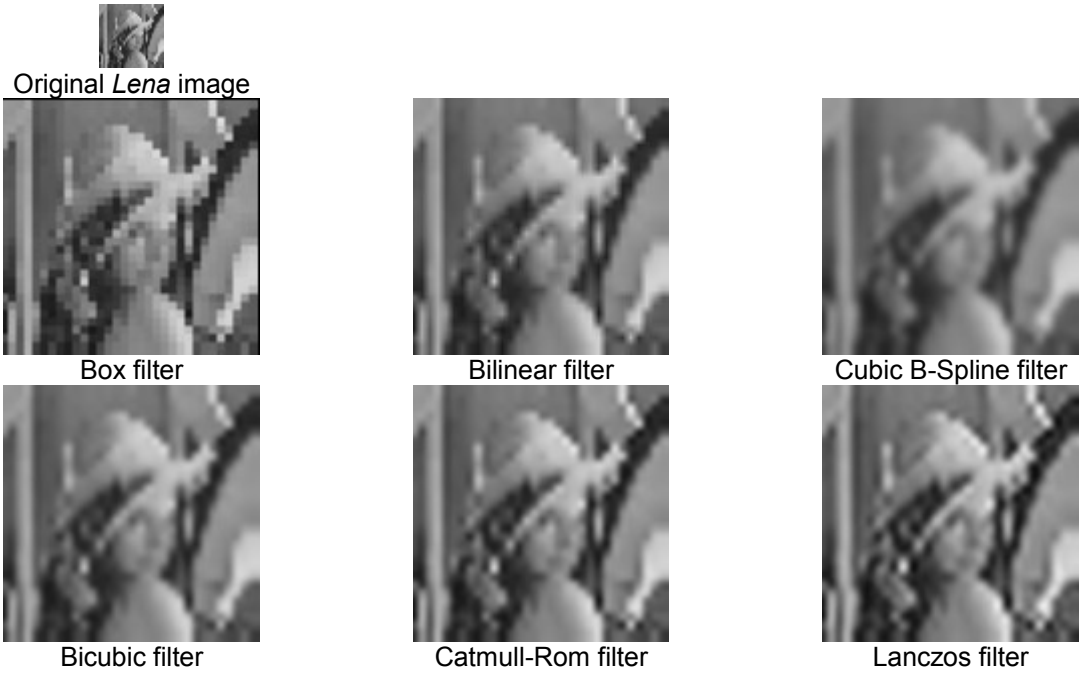
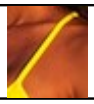
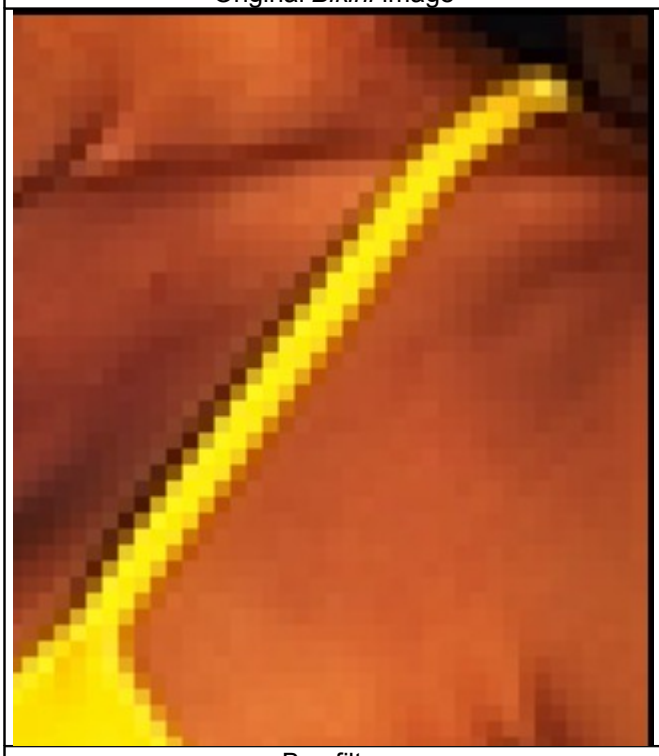


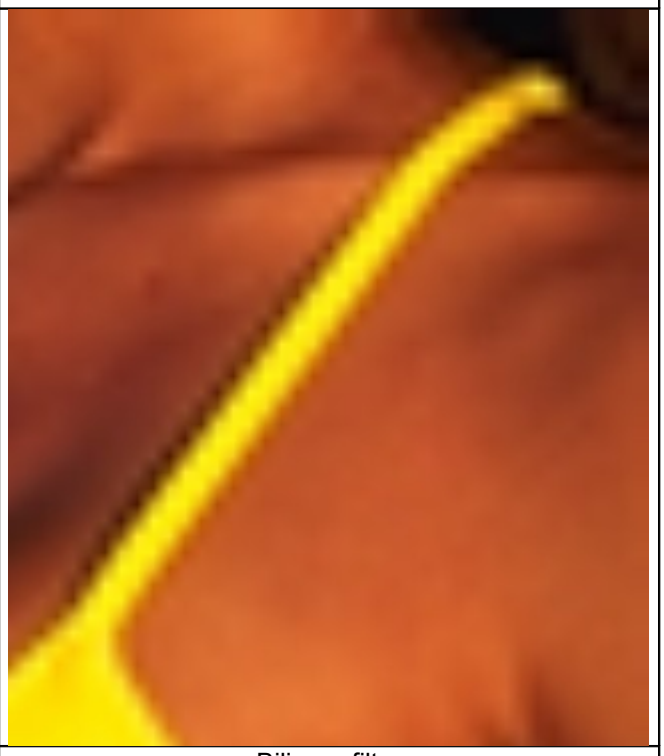
Figure 2: Comparison of resampling filters on a 32x32 Lena image resized to 400%.



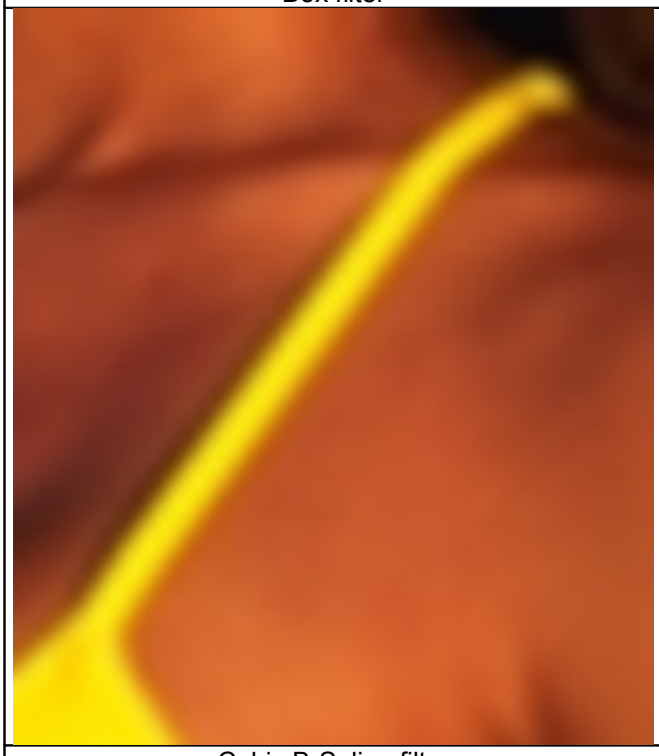
Original *Bikini* image



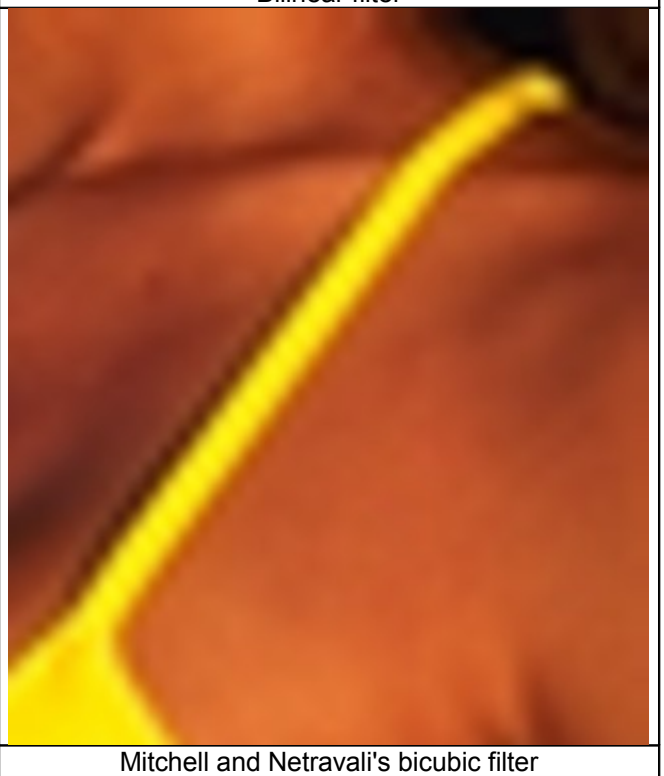
Box filter



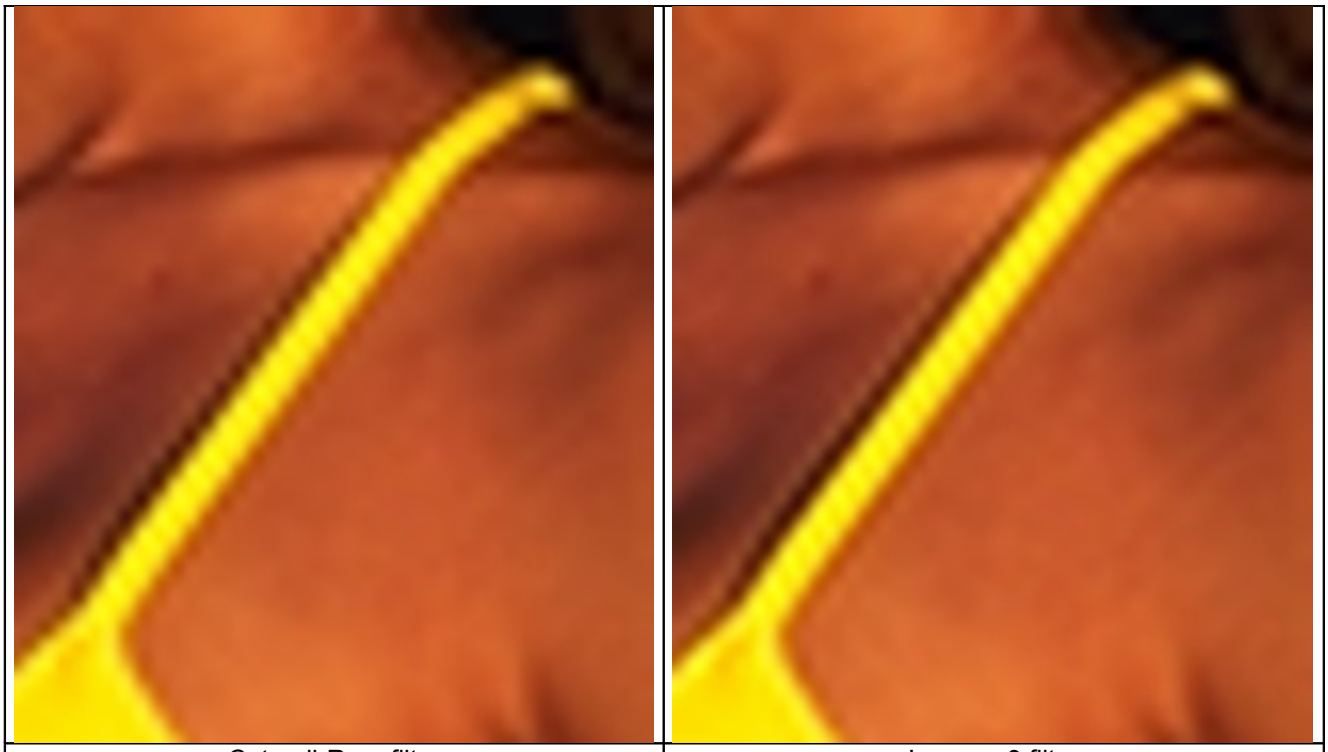
Bilinear filter



Cubic B-Spline filter



Mitchell and Netravali's bicubic filter



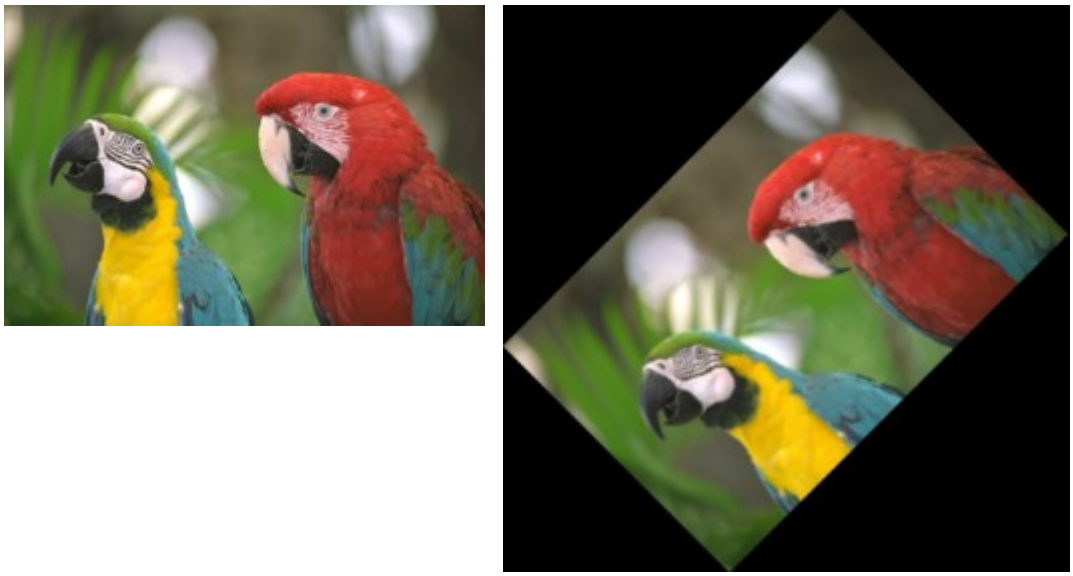
Catmull-Rom filter	Lanczos3 filter

Figure 3: Comparison of resampling filters on a 40x46 Bikini image resized to 800%.

Using the rotation functions

FreeImage_Rotate

The following figure demonstrates the result of using FreeImage_Rotate when rotating an image by an angle of 45°. Note that the rotated image is larger than the original image.



Original *Parrot* image Rotated image

Figure 4: Parrot image rotated by 45° using FreeImage_Rotate.

The same image now rotated by an angle of 90° is showed in Figure 5. This time, the rotated image has the same size as the original one.

Figure 5: Parrot image rotated by 90° using FreeImage_Rotate.

FreeImage_RotateEx

Figure 6 shows some of the results you can obtain with the FreeImage_RotateEx function.

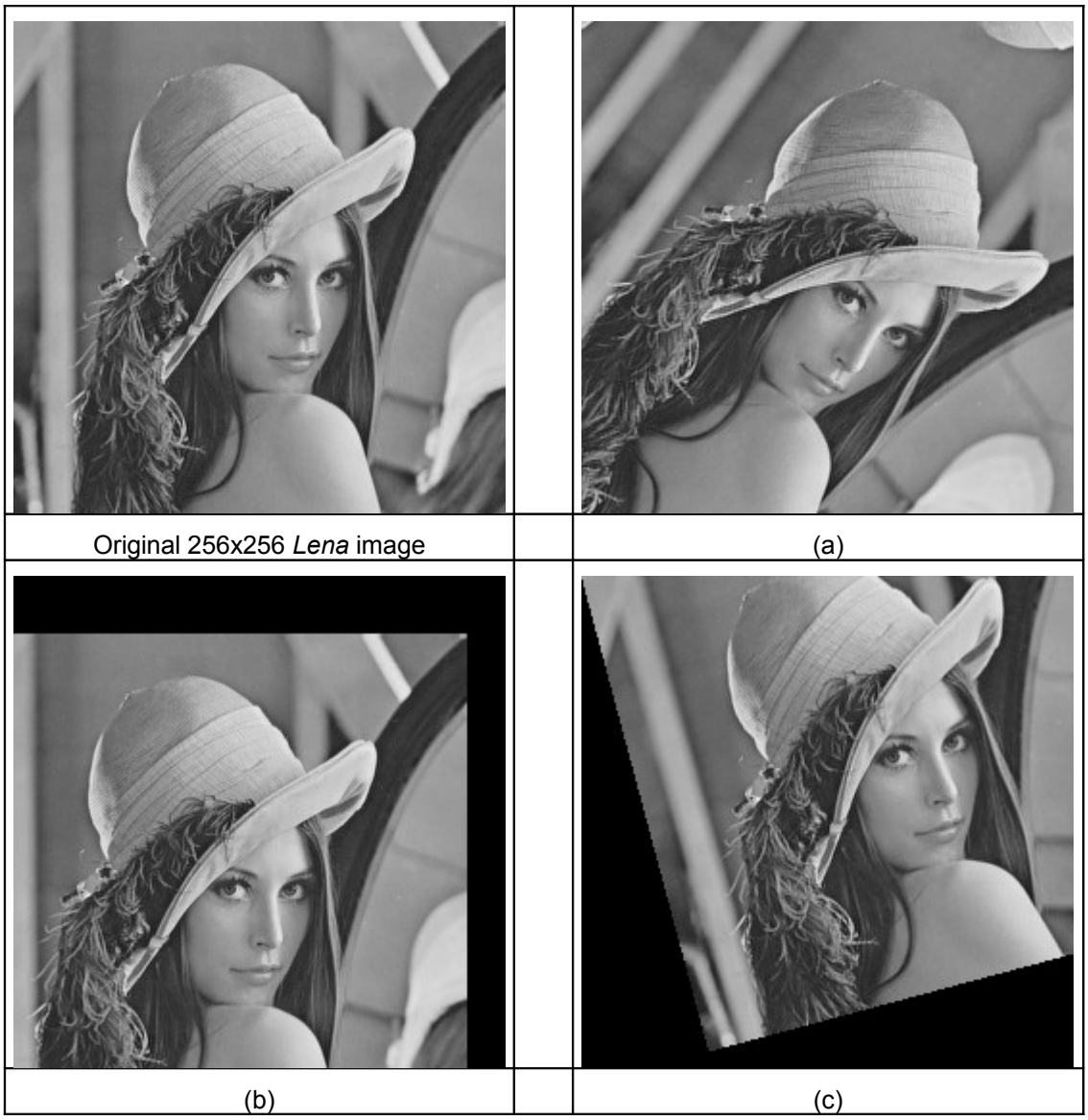


Figure 6: Some examples illustrating the use of FreeImage_RotateEx.

(a) : Image resulting from an arbitrary transformation (no masking). The image has been rotated by some angle around an arbitrary origin, while an additional translation has been thrown in for good measure. Observe the influence of mirroring the data (the function allows for the masking out of the extrapolated data, if desired).

```
FIBITMAP *dst = FreeImage_RotateEx(src, angle, x_shift, y_shift, x_origin, y_origin, FALSE);
```

(b) : Image resulting from a simple integer translation using the following code :

```
FIBITMAP *dst = FreeImage_RotateEx(src, 0, -20, 30, 0, 0, TRUE);
```

This time, we set the *use_mask* parameter to TRUE, to mask out the irrelevant part of the image.

(c) : Image resulting from a rotation around the upper-left corner :

```
FIBITMAP *dst = FreeImage_RotateEx(src, 15, 0, 0, 0, 0, TRUE);
```

Freeimage metadata models

FIMD_COMMENTS

This model is used to store image comments or image keywords.

The **JPEG** format supports a single user comment string, which can be set using the "**Comment**" tag field name.

The **PNG** format supports as many comments as you want, using any keyword as the tag field name. Each keyword is saved and loaded together with the metadata.

The PNG tIME chunk (timestamp) provides a way for the author (or image-editing software) to record the time and date the image was last modified. It is handled using the FIMD_EXIF_MAIN "**DateTime**" keyword: a string whose format is 'yyyy:MM:dd hh:mm:ss'. Whenever a tIME chunk is present in a file, it is loaded into this metadata. Whenever a FIMD_EXIF_MAIN "DateTime" metadata is present in a dib, it is written as a tIME chunk.

The **GIF** format supports as many comments as you want, using any keyword as the tag field name. The keyword is not saved with the metadata. On loading, each comment is attached to a tag key named "CommentX" where X is a number ranging from 0 to N-1, where N is the number of comments in the GIF file.

The **RAW** format loads some useful information when using the **RAW_UNPROCESSED** flag:

Tag key	Tag description
Raw.Output.Width	Output image width you should have after raw post-processing
Raw.Output.Height	Output image height you should have after raw post-processing
Raw.Frame.Left	The x-coordinate of the upper-left corner of the frame
Raw.Frame.Top	The y-coordinate of the upper-left corner of the frame
Raw.Frame.Width	Width (in pixels) of the frame
Raw.Frame.Height	Height (in pixels) of the frame
Raw.BayerPattern	Bayer pattern as a 16-byte char array. This field describe 16 pixels (8 rows with two pixels in each, from left to right and from top to bottom).

FIMD_EXIF_*

These models are used to load Exif metadata stored in JPEG images. The following sub-models are supported:

FIMD_EXIF_MAIN

This is the Exif-TIFF metadata, i.e. the metadata that are common to both TIFF and Exif files.

FIMD_EXIF_EXIF

This model represents Exif specific metadata.

FIMD_EXIF_GPS

This model represents Exif GPS metadata that are part of the Exif standard.

FIMD_EXIF_MAKERNOTE

Exif maker notes are metadata that are added by camera constructors. There is no public specification for these metadata and each constructor uses its own specification to name the tag fields.

The following makers are currently supported by the library: Asahi, Canon, Casio (type 1 and type 2), Fujifilm, Kyocera, Minolta, Nikon (type 1, type 2 and type 3), Olympus / Epson / Agfa, Panasonic, Pentax and Sony.

FIMD_EXIF_INTEROP

This model represents the Exif interoperability metadata.

Note: the Exif specifications can be downloaded at the following URL:

http://www.exif.org

FIMD_EXIF_RAW

This model store Exif data as a single undecoded raw buffer. FIMD EXIF_RAW represents a single Exif buffer and is indexed using the tag field name "**ExifRaw**".

The FIMD_EXIF_RAW metadata model does not replace the other EXIF models. This is an additional metadata used to store Exif data as a raw binary buffer. Saving Exif data can be very complex, but saving Exif raw data is quite easy (at least with the JPEG format). Thus if it is possible to preserve Exif information connected with the JPEG files if such file is e.g. loaded, resized then saved.

Exif raw loading and saving is supported by the JPEG plugin only.

FIMD_IPTC

This model represents the Information Interchange Model (IIM), also called IPTC/NAA metadata model, and was originally defined by the IPTC and the Newspaper Association of America (NAA) (see http://www.iptc.org/IIM/).

This model was widely used by Adobe Photoshop but **its support is no longer encouraged**, neither by the IPTC nor by Adobe, as it has been replaced by the XMP standard.

Tag ID	Key	Description	Photoshop compatible
0x0200	ApplicationRecordVersion	Application Record Version (not editable)	•
0x0203	ObjectTypeReference	Object Type Reference	-
0x0204	ObjectAttributeReference	Object Attribute Reference	-
0x0205	ObjectName	Title	•
0x0207	EditStatus	Edit Status	-
0x0208	EditorialUpdate	Editorial Update	-
0x020A	Urgency	Urgency	•
0x020C	SubjectReference	Subject Reference	-
0x020F	Category	Category	•
0x0214	SupplementalCategories	Supplemental Categories*	•
0x0216	FixtureIdentifier	Fixture Identifier	-
0x0219	Keywords	Keywords*	•
0x021A	ContentLocationCode	Content Location Code	-
0x021B	ContentLocationName	Content Location Name	-
0x021E	ReleaseDate	Release Date	-
0x0223	ReleaseTime	Release Time	-
0x0225	ExpirationDate	Expiration Date	-
0x0226	ExpirationTime	Expiration Time	-
0x0228	SpecialInstructions	Instructions	•
0x022A	ActionAdvised	Action Advised	-
0x022D	ReferenceService	Reference Service	-
0x022F	ReferenceDate	Reference Date	-
0x0232	ReferenceNumber	Reference Number	-
0x0237	DateCreated	Date Created	•
0x023C	TimeCreated	Time Created	-
0x023E	DigitalCreationDate	Digital Creation Date	-
0x023F	DigitalCreationTime	Digital Creation Time	-
0x0241	OriginatingProgram	Originating Program	-
0x0246	ProgramVersion	Program Version	-
0x024B	ObjectCycle	Object Cycle	-
0x0250	By-line	Author	•
0x0255	By-lineTitle	Author's Position	•
0x025A	City	City	•
0x025C	SubLocation	Sub-Location	-
0x025F	Province-State	State/Province	•
0x0264	Country-PrimaryLocationCode	Country Code	-
0x0265	Country-PrimaryLocationName	Country Name	•
0x0267	OriginalTransmissionReference	Transmission Reference	•
0x0269	Headline	Headline	•
0x026E	Credit	Credit	•
0x0273	Source	Source	•
0x0274	CopyrightNotice	Copyright Notice	•
0x0276	Contact	Contact	-
0x0278	Caption-Abstract	Caption	•
0x027A	Writer-Editor	Caption Writer	•
0x027D	RasterizedCaption	Rasterized Caption	-
0x0282	ImageType	Image Type	-
0x0283	ImageOrientation	Image Orientation	-
0x0287	LanguageIdentifier	Language Identifier	-
0x0296	AudioType	Audio Type	-
0x0297	AudioSamplingRate	Audio Sampling Rate	-
0x0298	AudioSamplingResolution	Audio Sampling Resolution	-
0x0299	AudioDuration	Audio Duration	-
0x029A	AudioOutcue	Audio Outcue	-
0x02B8	JobID	Job ID	-
0x02B9	MasterDocumentID	Master Document ID	-
0x02BA	ShortDocumentID	Short Document ID	-
0x02BB	UniqueDocumentID	Unique Document ID	-
0x02BC	OwnerID	Owner ID	-
0x02C8	ObjectPreviewFileFormat	Object Preview File Format	-
0x02C9	ObjectPreviewFileVersion	Object Preview File Version	-
0x02CA	ObjectPreviewData	Audio Outcue	-
0x02DD	Prefs	PhotoMechanic preferences	-
0x02E1	ClassifyState	Classify State	-
0x02E4	SimilarityIndex	Similarity Index	-
0x02E6	DocumentNotes	Document Notes	-
0x02E7	DocumentHistory	Document History	-
0x02E8	ExifCameraInfo	Exif Camera Info	-

* semicolon separated list of words

Legend:
yes : •
no : -

Table 19: List of tag keys supported by the IPTC metadata model.

FIMD_XMP

FIMD XMP represents a single Adobe XML packet and is indexed using the tag field name "**XMLPacket**".

The Adobe XMP standard is described at the following URL:

http://www.adobe.com/products/xmp/main.html

FIMD_GEOTIFF

This model represents the GeoTIFF metadata standard and is used to add georeferencing information to TIFF files.

The GeoTIFF specifications can be found at the following URL:

http://trac.osgeo.org/geotiff/

FIMD_ANIMATION

This model is used to load and save animation metadata attached to an animated GIF or MNG file. Metadata supported by the FIMD_ANIMATION model have been defined by FreeImage. Currently, this model is only supported by the GIF plugin.

The FIMD_ANIMATION specifications are described later in this appendix.

FIMD_CUSTOM

FIMD_CUSTOM is a placeholder metadata model that can be used to store user specific metadata. It can be used for example to store metadata that would be used by a custom plugin written by you.

FIMD_ANIMATION metadata model specification

The Animation metadata model is a generic model used to load and save animation metadata attached to an animated file (such as a GIF or a MNG file). Metadata supported by this model have been defined specifically for the FreeImage library (this is not a metadata standard such as Exif).

When saving animation metadata to an animated file, the FreeImage metadata are transparently translated to the metadata required by a given plugin specification. On the other hand, when loading an animated file, its animation metadata are transparently translated to the FreeImage animation metadata model.

Currently, this model is only supported by the GIF plugin.

The following metadata are supported by the model.

Tags relating to a single page or relating to page 0 of a multipage animated file					
Tag Name	**Field Name**	**Tag ID**		**Type**	**Count**
		Dec	**Hex**		
Logical width	LogicalWidth	1	0x0001	FIDT_SHORT	1
Logical height	LogicalHeight	2	0x0002	FIDT_SHORT	1
Global palette	GlobalPalette	3	0x0003	FIDT_PALETTE	Any
Loop	Loop	4	0x0004	FIDT_LONG	1
Tags relating to each page (including page 0) of a single page or a multipage animated file					
Tag Name	**Field Name**	**Tag ID**		**Type**	**Count**
		Dec	**Hex**		
Frame left	FrameLeft	4097	0x1001	FIDT_SHORT	1
Frame top	FrameTop	4098	0x1002	FIDT_SHORT	1
No local palette	NoLocalPalette	4099	0x1003	FIDT_BYTE	1
Interlaced	Interlaced	4100	0x1004	FIDT_BYTE	1
Frame time	FrameTime	4101	0x1005	FIDT_LONG	1
Frame disposal method	DisposalMethod	4102	0x1006	FIDT_BYTE	1

Note:

The following values are supported by the **DisposalMethod** tag:

GIF_DISPOSAL_UNSPECIFIED = 0

GIF_DISPOSAL_LEAVE = 1

GIF_DISPOSAL_BACKGROUND = 2

GIF_DISPOSAL_PREVIOUS = 3

Tags relating to a single page or relating to page 0 of a multipage animated file

LogicalWidth

Width of entire canvas area that each page is displayed in 0-65535

Tag = 1 (0001.H)

Type = FIDT_SHORT

Count = 1

Save Default = dib's width

Load: always exists in file and set

LogicalHeight

Height of entire canvas area that each page is displayed in 0-65535

Tag = 2 (0002.H)

Type = FIDT_SHORT

Count = 1

Save Default = dib's height

Load: always exists in file and set

GlobalPalette

RGBQUAD data for a "global" palette which can apply to all images with no local palette up to 256 x FIDT_PALETTE

Tag = 3 (0003.H)

Type = FIDT_PALETTE

Count = 0 to 256

Save Default = no global palette

Save Notes: rounded down to the nearest power of 2 entries

Load: set if exists in file, not set if the file has no global palette

Additional notes (GIF specific)

The palette size must be 2, 4, 8, 16, 32, 64, 128 or 256, or no global palette (0). If you specify a metadata with count 127, only the first 64 will be used, since it rounds down, and the plugin will set the global palette size in the GIF header to be 6 bits.

Loop

The number of times the animation should be played 0-65536 (0=infinite)

Tag: 4 (0004.H)

Type: FIDT_LONG

Count: 1

Save Default = 0 (infinite loops)

Save Notes:

> For GIFs specifically, the NETSCAPE2.0 application extension represents the number of times to repeat the animation, thus, 1 repeat means 2 loops (play thru the animation twice), 65535 repeats is the largest value which can be stored, and translates to 65536 loops.

Load: the metadata is always set to a value 0-65536 (set to 0 or 2-65536 if the extension is in the file, 1 if the extension is not in the file)

Tags relating to each page (including page 0) of a single page or a multipage animated file

FrameLeft

The x offset in the logical canvas area to display the image. (0-65535)

Tag = 4097 (1001.H)

Type = FIDT_SHORT

Count = 1

Save Default = 0

Load: always exists in file and set

FrameTop

The y offset in the logical canvas area to display the image. (0-65535)

Tag = 4098 (1002.H)

Type = FIDT_SHORT

Count = 1

Save Default = 0

Load: always exists in file and set

NoLocalPalette

A flag to supress saving the dib's attached palette (making it use the global palette). The local palette is the palette used by a page. This palette data is not set via metadata (like the global palette) as it is attached to the dib.

Tag = 4099 (1003.H)

Type = FIDT_BYTE

Count = 1

Save Default = 0 (meaning, yes, save the local palette data)

Load: always exists in file and set

Interlaced

Tells if the image should be stored interlaced

Tag = 4100 (1004.H)

Type = FIDT_BYTE

Count = 1

Save Default = 0

Load: always exists in file and set

FrameTime

The amount of time in ms to display the frame for (GIF specific - the value stored in the file is centiseconds (1/100th of a second)).

Tag = 4101 (1005.H)

Type = FIDT_LONG

Count = 1

Save Default = 100ms (GIF specific - the value stored is 10cs)

Save Notes:

For GIF, the value specified in ms is rounded down, such as 129ms is stored as 12cs. IE5/IE6 have a minimum and default of 100ms. Mozilla/Firefox/Netscape 6+/Opera have a minimum of 20ms and a default of 100ms if less than 20ms is specified or the GCE is absent. Netscape 4 has a minimum of 10ms if 0ms is specified, but will use 0ms if the GCE is absent. The GIF plugin always writes a GCE extension to the GIF file, and it also always uses GIF89a.

Load: always set, set to 0 if does not exist in file

DisposalMethod

What to do with the logical canvas area after displaying this image.

Tag = 4102 (1006.H)

Type = FIDT_BYTE

Count = 1

Save Default = GIF_DISPOSAL_BACKGROUND (restore to the background color, which is transparent with 0 alpha)

Save Notes:

GIF_DISPOSAL_UNSPECIFIED probably just does the same as GIF_DISPOSAL_LEAVE and should not be used.

GIF_DISPOSAL_LEAVE will leave the image in place to be entirely or partially overdrawn by the next image.

GIF_DISPOSAL_BACKGROUND will blank out the area used by the frame with the background color.

GIF_DISPOSAL_PREVIOUS will return the logical canvas to the previous state before the image was drawn.

Load: always set, set to GIF_DISPOSAL_LEAVE if does not exist in file

Additional notes (GIF specific)

Transparency is supported individually for all pages, the first entirely transparent index in the table is used, the rest of the table will be entirely opaque.

The background color is only set and stored for page 0, but requires that the global palette be set in order to use it properly.

The **GIF_PLAYBACK** load flag option (see Table 3) will load a single page as a 32bpp image with transparency by displaying each page from 0 up to the specified page, obeying the transparency overlaying and gif disposal methods. Note that it does not actually play the image animation in a displayable way. It "plays" the image internally from page 0 to the page requested, returning a single still image of what that frame would really look like.

Note that GIF_PLAYBACK will return a 32bpp image: since each individual frame may contain its own palette and transparency, a single frame of an animated GIF, when composited over top of the previous frame, may contain more than 256 colors total. It may not be possible to extract each frame and save them as a GIF if you want each still frame to look like it would look like in a web browser for example. Most GIF animation programs will "optimize" the GIF by making each individual frame contain lots of transparency for

where the pixels matched the previous frame, so if you just extract the frames normally and save them as GIF files, everything but the first frame may look like a bunch of random fuzz pixels.

The **GIF_LOAD256** load flag option is used internally by GIF_PLAYBACK, but can be used by users as well, it just prevents a lot of bitshifting and annoying things that come with 2 and 16 color images.

Using the FIMD_ANIMATION metadata model

This model is useful for generating animated GIFs with FreeImage, which web browsers will later be displaying. The metadata is used to save (and load) the various options that GIF files support for defining an animation.

The simplest of examples would not need to change any metadata. Just open a multipage GIF with create_new=TRUE and start adding pages to it.

The GIF generated when you close the multipage image will loop forever, and display each page for 1/10 of a second (100ms).

Each page of the GIF will have its own palette and fill the entire logical area.

The worst snag a user could run into is adding pages to the multipage bitmap which are larger than the first page they added, because without setting specific metadata, the logical canvas area will be set to the same size of the first page, and it is undefined (not allowed) by the GIF specification technically if you have a frame extend outside the canvas area. (IE/Firefox will simply make the image larger as needed for the largest frame, Opera will chop off any portion of the image that is outside the logical area).

```
// assume we have an array of dibs which are already 8bpp and all the same size,
// and some float called fps for frames per second
FIMULTIBITMAP *multi = FreeImage_OpenMultiBitmap(FIF_GIF, "output.gif", TRUE, FALSE);
DWORD dwFrameTime = (DWORD)((1000.0f / fps) + 0.5f);
for(int i = 0; i < 10; i++ ) {
   // clear any animation metadata used by this dib as we'll adding our own ones
   FreeImage_SetMetadata(FIMD_ANIMATION, dib[i], NULL, NULL);
   // add animation tags to dib[i]
   FITAG *tag = FreeImage_CreateTag();
   if(tag) {
       FreeImage_SetTagKey(tag, "FrameTime");
       FreeImage_SetTagType(tag, FIDT_LONG);
       FreeImage_SetTagCount(tag, 1);
       FreeImage_SetTagLength(tag, 4);
       FreeImage_SetTagValue(tag, &dwFrameTime);
       FreeImage_SetMetadata(FIMD_ANIMATION, dib[i], FreeImage_GetTagKey(tag), tag);
       FreeImage_DeleteTag(tag);
   }
   FreeImage_AppendPage(multi, dib[i]);
}
FreeImage_CloseMultiBitmap(multi);
```