|  |  |
| --- | --- |
|  | Using The TIFF Library libtiff is a set of C functions (a library) that support the manipulation of TIFF image files. The library requires an ANSI C compilation environment for building and presumes an ANSI C environment for use. |

libtiff provides interfaces to image data at several layers of abstraction (and cost). At the highest level image data can be read into an 8-bit/sample, ABGR pixel raster format without regard for the underlying data organization, colorspace, or compression scheme. Below this high-level interface the library provides scanline-, strip-, and tile-oriented interfaces that return data decompressed but otherwise untransformed. These interfaces require that the application first identify the organization of stored data and select either a strip-based or tile-based API for manipulating data. At the lowest level the library provides access to the raw uncompressed strips or tiles, returning the data exactly as it appears in the file.

The material presented in this chapter is a basic introduction to the capabilities of the library; it is not an attempt to describe everything a developer needs to know about the library or about TIFF. Detailed information on the interfaces to the library are given in the [UNIX manual pages](http://www.simplesystems.org/libtiff/man/index.html) that accompany this software.

Michael Still has also written a useful introduction to libtiff for the IBM DeveloperWorks site available at <http://www.ibm.com/developerworks/linux/library/l-libtiff>.

The following sections are found in this chapter:

* [How to tell which version you have](#gjdgxs)
* [Library Datatypes](#30j0zll)
* [Memory Management](#1fob9te)
* [Error Handling](#3znysh7)
* [Basic File Handling](#2et92p0)
* [TIFF Directories](#tyjcwt)
* [TIFF Tags](#3dy6vkm)
* [TIFF Compression Schemes](#1t3h5sf)
* [Byte Order](#4d34og8)
* [Data Placement](#2s8eyo1)
* [TIFFRGBAImage Support](#17dp8vu)
* [Scanline-based Image I/O](#3rdcrjn)
* [Strip-oriented Image I/O](#26in1rg)
* [Tile-oriented Image I/O](#lnxbz9)
* [Other Stuff](#35nkun2)

## How to tell which version you have

The software version can be found by looking at the file named VERSION that is located at the top of the source tree; the precise alpha number is given in the file dist/tiff.alpha. If you have need to refer to this specific software, you should identify it as:

TIFF <*version*> <*alpha*>

where <*version*> is whatever you get from "cat VERSION" and <*alpha*> is what you get from "cat dist/tiff.alpha".

Within an application that uses libtiff the TIFFGetVersion routine will return a pointer to a string that contains software version information. The library include file <tiffio.h> contains a C pre-processor define TIFFLIB\_VERSION that can be used to check library version compatiblity at compile time.

## Library Datatypes

libtiff defines a portable programming interface through the use of a set of C type definitions. These definitions, defined in in the files **tiff.h** and **tiffio.h**, isolate the libtiff API from the characteristics of the underlying machine. To insure portable code and correct operation, applications that use libtiff should use the typedefs and follow the function prototypes for the library API.

## Memory Management

libtiff uses a machine-specific set of routines for managing dynamically allocated memory. \_TIFFmalloc, \_TIFFrealloc, and \_TIFFfree mimic the normal ANSI C routines. Any dynamically allocated memory that is to be passed into the library should be allocated using these interfaces in order to insure pointer compatibility on machines with a segmented architecture. (On 32-bit UNIX systems these routines just call the normal malloc, realloc, and free routines in the C library.)

To deal with segmented pointer issues libtiff also provides \_TIFFmemcpy, \_TIFFmemset, and \_TIFFmemmove routines that mimic the equivalent ANSI C routines, but that are intended for use with memory allocated through \_TIFFmalloc and \_TIFFrealloc.

## Error Handling

libtiff handles most errors by returning an invalid/erroneous value when returning from a function call. Various diagnostic messages may also be generated by the library. All error messages are directed to a single global error handler routine that can be specified with a call to TIFFSetErrorHandler. Likewise warning messages are directed to a single handler routine that can be specified with a call to TIFFSetWarningHandler

## Basic File Handling

The library is modeled after the normal UNIX stdio library. For example, to read from an existing TIFF image the file must first be opened:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("foo.tif", "r");

    ... do stuff ...

    TIFFClose(tif);

}

The handle returned by TIFFOpen is *opaque*, that is the application is not permitted to know about its contents. All subsequent library calls for this file must pass the handle as an argument.

To create or overwrite a TIFF image the file is also opened, but with a "w" argument:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("foo.tif", "w");

    ... do stuff ...

    TIFFClose(tif);

}

If the file already exists it is first truncated to zero length.

|  |  |
| --- | --- |
|  | *Note that unlike the stdio library TIFF image files may not be opened for both reading and writing; there is no support for altering the contents of a TIFF file.* |

libtiff buffers much information associated with writing a valid TIFF image. Consequently, when writing a TIFF image it is necessary to always call TIFFClose or TIFFFlush to flush any buffered information to a file. Note that if you call TIFFClose you do not need to call TIFFFlush.

## TIFF Directories

TIFF supports the storage of multiple images in a single file. Each image has an associated data structure termed a *directory* that houses all the information about the format and content of the image data. Images in a file are usually related but they do not need to be; it is perfectly alright to store a color image together with a black and white image. Note however that while images may be related their directories are not. That is, each directory stands on its own; their is no need to read an unrelated directory in order to properly interpret the contents of an image.

libtiff provides several routines for reading and writing directories. In normal use there is no need to explicitly read or write a directory: the library automatically reads the first directory in a file when opened for reading, and directory information to be written is automatically accumulated and written when writing (assuming TIFFClose or TIFFFlush are called).

For a file open for reading the TIFFSetDirectory routine can be used to select an arbitrary directory; directories are referenced by number with the numbering starting at 0. Otherwise the TIFFReadDirectory and TIFFWriteDirectory routines can be used for sequential access to directories. For example, to count the number of directories in a file the following code might be used:

#include "tiffio.h"

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

{

    TIFF\* tif = TIFFOpen(argv[1], "r");

    if (tif) {

        int dircount = 0;

        do {

            dircount++;

        } while (TIFFReadDirectory(tif));

        printf("%d directories in %s\n", dircount, argv[1]);

        TIFFClose(tif);

    }

    exit(0);

}

Finally, note that there are several routines for querying the directory status of an open file: TIFFCurrentDirectory returns the index of the current directory and TIFFLastDirectory returns an indication of whether the current directory is the last directory in a file. There is also a routine, TIFFPrintDirectory, that can be called to print a formatted description of the contents of the current directory; consult the manual page for complete details.

## TIFF Tags

Image-related information such as the image width and height, number of samples, orientation, colorimetric information, etc. are stored in each image directory in *fields* or *tags*. Tags are identified by a number that is usually a value registered with the Aldus (now Adobe) Corporation. Beware however that some vendors write TIFF images with tags that are unregistered; in this case interpreting their contents is usually a waste of time.

libtiff reads the contents of a directory all at once and converts the on-disk information to an appropriate in-memory form. While the TIFF specification permits an arbitrary set of tags to be defined and used in a file, the library only understands a limited set of tags. Any unknown tags that are encountered in a file are ignored. There is a mechanism to extend the set of tags the library handles without modifying the library itself; this is described [elsewhere](http://docs.google.com/addingtags.html).

libtiff provides two interfaces for getting and setting tag values: TIFFGetField and TIFFSetField. These routines use a variable argument list-style interface to pass parameters of different type through a single function interface. The *get interface* takes one or more pointers to memory locations where the tag values are to be returned and also returns one or zero according to whether the requested tag is defined in the directory. The *set interface* takes the tag values either by-reference or by-value. The TIFF specification defines *default values* for some tags. To get the value of a tag, or its default value if it is undefined, the TIFFGetFieldDefaulted interface may be used.

The manual pages for the tag get and set routines specifiy the exact data types and calling conventions required for each tag supported by the library.

## TIFF Compression Schemes

libtiff includes support for a wide variety of data compression schemes. In normal operation a compression scheme is automatically used when the TIFF Compression tag is set, either by opening a file for reading, or by setting the tag when writing.

Compression schemes are implemented by software modules termed *codecs* that implement decoder and encoder routines that hook into the core library i/o support. Codecs other than those bundled with the library can be registered for use with the TIFFRegisterCODEC routine. This interface can also be used to override the core-library implementation for a compression scheme.

## Byte Order

The TIFF specification says, and has always said, that *a correct TIFF reader must handle images in big-endian and little-endian byte order*. libtiff conforms in this respect. Consequently there is no means to force a specific byte order for the data written to a TIFF image file (data is written in the native order of the host CPU unless appending to an existing file, in which case it is written in the byte order specified in the file).

## Data Placement

The TIFF specification requires that all information except an 8-byte header can be placed anywhere in a file. In particular, it is perfectly legitimate for directory information to be written after the image data itself. Consequently TIFF is inherently not suitable for passing through a stream-oriented mechanism such as UNIX pipes. Software that require that data be organized in a file in a particular order (e.g. directory information before image data) does not correctly support TIFF. libtiff provides no mechanism for controlling the placement of data in a file; image data is typically written before directory information.

## TIFFRGBAImage Support

libtiff provides a high-level interface for reading image data from a TIFF file. This interface handles the details of data organization and format for a wide variety of TIFF files; at least the large majority of those files that one would normally encounter. Image data is, by default, returned as ABGR pixels packed into 32-bit words (8 bits per sample). Rectangular rasters can be read or data can be intercepted at an intermediate level and packed into memory in a format more suitable to the application. The library handles all the details of the format of data stored on disk and, in most cases, if any colorspace conversions are required: bilevel to RGB, greyscale to RGB, CMYK to RGB, YCbCr to RGB, 16-bit samples to 8-bit samples, associated/unassociated alpha, etc.

There are two ways to read image data using this interface. If all the data is to be stored in memory and manipulated at once, then the routine TIFFReadRGBAImage can be used:

#include "tiffio.h"

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

{

    TIFF\* tif = TIFFOpen(argv[1], "r");

    if (tif) {

        uint32 w, h;

        size\_t npixels;

        uint32\* raster;

        TIFFGetField(tif, TIFFTAG\_IMAGEWIDTH, &w);

        TIFFGetField(tif, TIFFTAG\_IMAGELENGTH, &h);

        npixels = w \* h;

        raster = (uint32\*) \_TIFFmalloc(npixels \* sizeof (uint32));

        if (raster != NULL) {

            if (TIFFReadRGBAImage(tif, w, h, raster, 0)) {

                ...process raster data...

            }

            \_TIFFfree(raster);

        }

        TIFFClose(tif);

    }

    exit(0);

}

Note above that \_TIFFmalloc is used to allocate memory for the raster passed to TIFFReadRGBAImage; this is important to insure the ``appropriate type of memory'' is passed on machines with segmented architectures.

Alternatively, TIFFReadRGBAImage can be replaced with a more low-level interface that permits an application to have more control over this reading procedure. The equivalent to the above is:

#include "tiffio.h"

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

{

    TIFF\* tif = TIFFOpen(argv[1], "r");

    if (tif) {

        TIFFRGBAImage img;

        char emsg[1024];

        if (TIFFRGBAImageBegin(&img, tif, 0, emsg)) {

            size\_t npixels;

            uint32\* raster;

            npixels = img.width \* img.height;

            raster = (uint32\*) \_TIFFmalloc(npixels \* sizeof (uint32));

            if (raster != NULL) {

                if (TIFFRGBAImageGet(&img, raster, img.width, img.height)) {

                    ...process raster data...

                }

                \_TIFFfree(raster);

            }

            TIFFRGBAImageEnd(&img);

        } else

            TIFFError(argv[1], emsg);

        TIFFClose(tif);

    }

    exit(0);

}

However this usage does not take advantage of the more fine-grained control that's possible. That is, by using this interface it is possible to:

* repeatedly fetch (and manipulate) an image without opening and closing the file
* interpose a method for packing raster pixel data according to application-specific needs (or write the data at all)
* interpose methods that handle TIFF formats that are not already handled by the core library

The first item means that, for example, image viewers that want to handle multiple files can cache decoding information in order to speedup the work required to display a TIFF image.

The second item is the main reason for this interface. By interposing a "put method" (the routine that is called to pack pixel data in the raster) it is possible share the core logic that understands how to deal with TIFF while packing the resultant pixels in a format that is optimized for the application. This alternate format might be very different than the 8-bit per sample ABGR format the library writes by default. For example, if the application is going to display the image on an 8-bit colormap display the put routine might take the data and convert it on-the-fly to the best colormap indices for display.

The last item permits an application to extend the library without modifying the core code. By overriding the code provided an application might add support for some esoteric flavor of TIFF that it needs, or it might substitute a packing routine that is able to do optimizations using application/environment-specific information.

The TIFF image viewer found in **tools/sgigt.c** is an example of an application that makes use of the TIFFRGBAImage support.

## Scanline-based Image I/O

The simplest interface provided by libtiff is a scanline-oriented interface that can be used to read TIFF images that have their image data organized in strips (trying to use this interface to read data written in tiles will produce errors.) A scanline is a one pixel high row of image data whose width is the width of the image. Data is returned packed if the image data is stored with samples packed together, or as arrays of separate samples if the data is stored with samples separated. The major limitation of the scanline-oriented interface, other than the need to first identify an existing file as having a suitable organization, is that random access to individual scanlines can only be provided when data is not stored in a compressed format, or when the number of rows in a strip of image data is set to one (RowsPerStrip is one).

Two routines are provided for scanline-based i/o: TIFFReadScanline and TIFFWriteScanline. For example, to read the contents of a file that is assumed to be organized in strips, the following might be used:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        uint32 imagelength;

        tdata\_t buf;

        uint32 row;

        TIFFGetField(tif, TIFFTAG\_IMAGELENGTH, &imagelength);

        buf = \_TIFFmalloc(TIFFScanlineSize(tif));

        for (row = 0; row < imagelength; row++)

            tiffreadscanline(tif, buf, row);

        \_tifffree(buf);

        tiffclose(tif);

    }

}

TIFFScanlineSize returns the number of bytes in a decoded scanline, as returned by TIFFReadScanline. Note however that if the file had been create with samples written in separate planes, then the above code would only read data that contained the first sample of each pixel; to handle either case one might use the following instead:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        uint32 imagelength;

        tdata\_t buf;

        uint32 row;

        TIFFGetField(tif, TIFFTAG\_IMAGELENGTH, &imagelength);

        TIFFGetField(tif, TIFFTAG\_PLANARCONFIG, &config);

        buf = \_TIFFmalloc(TIFFScanlineSize(tif));

        if (config == PLANARCONFIG\_CONTIG) {

            for (row = 0; row < imagelength; row++)

                tiffreadscanline(tif, buf, row);

        } else if (config == planarconfig\_separate) {

            uint16 s, nsamples;

            tiffgetfield(tif, tifftag\_samplesperpixel, &nsamples);

            for (s = 0; s < nsamples; s++)

                for (row = 0; row < imagelength; row++)

                    tiffreadscanline(tif, buf, row, s);

        }

        \_tifffree(buf);

        tiffclose(tif);

    }

}

Beware however that if the following code were used instead to read data in the case PLANARCONFIG\_SEPARATE,...

            for (row = 0; row < imagelength; row++)

                for (s = 0; s < nsamples; s++)

                    tiffreadscanline(tif, buf, row, s);

...then problems would arise if RowsPerStrip was not one because the order in which scanlines are requested would require random access to data within strips (something that is not supported by the library when strips are compressed).

## Strip-oriented Image I/O

The strip-oriented interfaces provided by the library provide access to entire strips of data. Unlike the scanline-oriented calls, data can be read or written compressed or uncompressed. Accessing data at a strip (or tile) level is often desirable because there are no complications with regard to random access to data within strips.

A simple example of reading an image by strips is:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        tdata\_t buf;

        tstrip\_t strip;

        buf = \_TIFFmalloc(TIFFStripSize(tif));

        for (strip = 0; strip < tiffnumberofstrips(tif); strip++)

            tiffreadencodedstrip(tif, strip, buf, (tsize\_t) -1);

        \_tifffree(buf);

        tiffclose(tif);

    }

}

Notice how a strip size of -1 is used; TIFFReadEncodedStrip will calculate the appropriate size in this case.

The above code reads strips in the order in which the data is physically stored in the file. If multiple samples are present and data is stored with PLANARCONFIG\_SEPARATE then all the strips of data holding the first sample will be read, followed by strips for the second sample, etc.

Finally, note that the last strip of data in an image may have fewer rows in it than specified by the RowsPerStrip tag. A reader should not assume that each decoded strip contains a full set of rows in it.

The following is an example of how to read raw strips of data from a file:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        tdata\_t buf;

        tstrip\_t strip;

        uint32\* bc;

        uint32 stripsize;

        TIFFGetField(tif, TIFFTAG\_STRIPBYTECOUNTS, &bc);

        stripsize = bc[0];

        buf = \_TIFFmalloc(stripsize);

        for (strip = 0; strip < tiffnumberofstrips(tif); strip++) {

            if (bc[strip] > stripsize) {

                buf = \_TIFFrealloc(buf, bc[strip]);

                stripsize = bc[strip];

            }

            TIFFReadRawStrip(tif, strip, buf, bc[strip]);

        }

        \_TIFFfree(buf);

        TIFFClose(tif);

    }

}

As above the strips are read in the order in which they are physically stored in the file; this may be different from the logical ordering expected by an application.

## Tile-oriented Image I/O

Tiles of data may be read and written in a manner similar to strips. With this interface, an image is broken up into a set of rectangular areas that may have dimensions less than the image width and height. All the tiles in an image have the same size, and the tile width and length must each be a multiple of 16 pixels. Tiles are ordered left-to-right and top-to-bottom in an image. As for scanlines, samples can be packed contiguously or separately. When separated, all the tiles for a sample are colocated in the file. That is, all the tiles for sample 0 appear before the tiles for sample 1, etc.

Tiles and strips may also be extended in a z dimension to form volumes. Data volumes are organized as "slices". That is, all the data for a slice is colocated. Volumes whose data is organized in tiles can also have a tile depth so that data can be organized in cubes.

There are actually two interfaces for tiles. One interface is similar to scanlines, to read a tiled image, code of the following sort might be used:

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        uint32 imageWidth, imageLength;

        uint32 tileWidth, tileLength;

        uint32 x, y;

        tdata\_t buf;

        TIFFGetField(tif, TIFFTAG\_IMAGEWIDTH, &imageWidth);

        TIFFGetField(tif, TIFFTAG\_IMAGELENGTH, &imageLength);

        TIFFGetField(tif, TIFFTAG\_TILEWIDTH, &tileWidth);

        TIFFGetField(tif, TIFFTAG\_TILELENGTH, &tileLength);

        buf = \_TIFFmalloc(TIFFTileSize(tif));

        for (y = 0; y < imagelength; y += tilelength)

            for (x = 0; x < imagewidth; x += tilewidth)

                tiffreadtile(tif, buf, x, y, 0);

        \_tifffree(buf);

        tiffclose(tif);

    }

}

(once again, we assume samples are packed contiguously.)

Alternatively a direct interface to the low-level data is provided a la strips. Tiles can be read with TIFFReadEncodedTile or TIFFReadRawTile, and written with TIFFWriteEncodedTile or TIFFWriteRawTile. For example, to read all the tiles in an image:

#include "tiffio.h"

main()

{

    TIFF\* tif = TIFFOpen("myfile.tif", "r");

    if (tif) {

        tdata\_t buf;

        ttile\_t tile;

        buf = \_TIFFmalloc(TIFFTileSize(tif));

        for (tile = 0; tile < tiffnumberoftiles(tif); tile++)

            tiffreadencodedtile(tif, tile, buf, (tsize\_t) -1);

        \_tifffree(buf);

        tiffclose(tif);

    }

}

## Other Stuff

Some other stuff will almost certainly go here...

Last updated: $Date: 2016-09-25 20:05:44 $