Graphics programming with libtiff

A C library for manipulating TIFF images

Level: Intermediate

Michael Still (mikal@stillhq.com), Senior Software Engineer, Tower Software Engineering

01 Mar 2002

TIFF is an extremely common but quite complex raster image format. Libtiff, a standard ANSI C implementation of the TIFF specification, is free and works on many operating systems. This article discusses some of the pitfalls of TIFF and guides you through use of the libtiff library. The article also shows examples of how to use libtiff for your black-and-white imaging needs.

TIFF (Tag Image File Format) is a raster image format that was originally produced by Adobe. Raster image formats store the picture as a bitmap describing the state of pixels, as opposed to recording the length and locations of primitives such as lines and curves. Libtiff is one of the standard implementations of the TIFF specification and is in wide use today because of its speed, power, and easy source availability.

This article focuses on black-and-white TIFF images; a possible future article will cover color images.

The TIFF challenge

Most file format specifications define some basic rules for the representation of the file. For instance, PNG documents (a competitor to TIFF) are always big endian. TIFF, however, doesn't mandate things like this. Here are some examples of the seemingly basic things that it doesn't define:

- Byte order: big endian or little endian
- Fill order of the bit within the image bytes: most significant or least significant bit first
- Meaning of a given pixel value for black and white: is 0 black or white?

Creating a TIFF file can be very easy, because it is rare to have to do any conversion of the data that you already have. On the other hand, it also means that reading in random TIFFs created by other applications can be very hard -- you have to code for all possible combinations to be reasonably certain of having a reliable product.

So, how do you write an application that can read in all these different possible permutations of the TIFF format? The most important thing to remember is *never make assumptions about the format of the image data you are reading in.*

Writing TIFF files

First I'll show how to write a TIFF file out. Then I'll show how to read a TIFF file back into your program.

Infrastructure for writing

Bitmaps are traditionally represented inside your code by an array of chars. This is because on most operating systems, a char maps well to one byte. In Listing 1, I set up libtiff and create a simple buffer that contains an image I can then write out to disk. You can download this code as write-infrastructure.c (see Resources later in this article).

Listing 1. Setting up the infrastructure (write-infrastructure.c)

```
#include <stdio.h>
#include <tiffio.h>

int
main (int argc, char *argv[])
{
  char buffer[32 * 9];
}
```

The code above is pretty simple. To use libtiff, all you need is to include the tiffio.h header file. To compile this, use the command `gcc foo.c -o foo -ltiff -lm`. The `-ltiff` is a command that includes the library named libtiff, which needs to be in your library path. Once you have started specifying libraries explicitly, you also need to add `-lm`, which is the mathematics library. The char buffer that we have defined here is going to be our black-and-white image, so we should define one of those next.

Writing the image

To make up for that boring example, I am now pleased to present you with what is possibly the worst picture of the Sydney Harbor Bridge ever drawn. In Listing 2, the image is already in the image buffer and all we have to do is save it to the file on disk. The example first opens a TIFF image in write mode and then places the image into that file.

Please note that for clarity I have omitted the actual hex for the image; this is available in the downloadable version of this code, write.c (see Resources), for those who are interested.

Listing 2. The writing code (write.c)

```
#include <stdio.h>
#include <tiffio.h>

int main(int argc, char *argv[]){
  // Define an image
  char buffer[25 * 144] = { /* boring hex omitted */ };
  TIFF *image;

  // Open the TIFF file
  if((image = TIFFOpen("output.tif", "w")) == NULL){
    printf("Could not open output.tif for writing\n");
    exit(42);
  }

  // We need to set some values for basic tags before we can add any data
  TIFFSetField(image, TIFFTAG_IMAGEWIDTH, 25 * 8);
  TIFFSetField(image, TIFFTAG_IMAGELENGTH, 144);
  TIFFSetField(image, TIFFTAG_BITSPERSAMPLE, 1);
  TIFFSetField(image, TIFFTAG_SAMPLESPERPIXEL, 1);
  TIFFSetField(image, TIFFTAG_ROWSPERSTRIP, 144);

  TIFFSetField(image, TIFFTAG_COMPRESSION, COMPRESSION_CCITTFAX4);
  TIFFSetField(image, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_MINISWHITE);
  TIFFSetField(image, TIFFTAG_FILLORDER, FILLORDER_MSB2LSB);
  TIFFSetField(image, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);

  TIFFSetField(image, TIFFTAG_XRESOLUTION, 150.0);
  TIFFSetField(image, TIFFTAG_YRESOLUTION, 150.0);
  TIFFSetField(image, TIFFTAG_RESOLUTIONUNIT, RESUNIT_INCH);

  // Write the information to the file
  TIFFWriteEncodedStrip(image, 0, buffer, 25 * 144);

  // Close the file
  TIFFClose(image);
}
```

(The output image will not display using the `xview` command on my Linux machine. In fact, I couldn't find an example of a group 4 fax compressed black-and-white image that would display using that program. At the time of publication of this article, I was still unable to find a fix for this.)

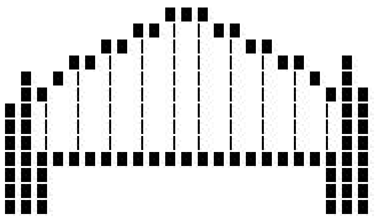
In any case, the sample code shows the basics of using the libtiff API. A few points worth noting:

- The buffers presented to and returned from libtiff each contain 8 pixels in a single byte. Thus, you have to be able to extract the pixels you are interested in. The use of masks, and the right and left shift operators, come in handy here.

- The `TIFFOpen` function is very similar to the `fopen` function you are familiar with.
- We need to set the value for quite a few fields before we can start writing the image out. These fields give libtiff information about the size and shape of the image, as well as the way data will be compressed within the image. These fields need to be set before you can start handing image data to libtiff. There are many more fields for which a value could be set; I have used close to the bare minimum in this example.
- `TIFFWriteEncodedStrip` is the function call that actually inserts the image into the file. This call inserts uncompressed image data into the file. This means that libtiff will compress the image data for you before writing it to the file. If you have already compressed data, then have a look at the `TIFFWriteRawStrip` instead.
- Finally, we close the file with `TIFFClose`.

In case you're curious what the Sydney Harbor Bridge looks like, Figure 1 is a copy of the picture. (I've had to cheat and turn it into a JPEG, because most Web browsers don't support TIFF.)

Figure 1. The Sydney Harbor Bridge, by Michael Still



More information about the libtiff function calls
If you need more information about any of the libtiff function calls mentioned in this article, check out the extensive man pages that come with the library.

Remember that case is important with man pages, so you need to get the case in the function names right. For example, it's `TIFFOpen`, not `tiffopen`.

Reading TIFF files

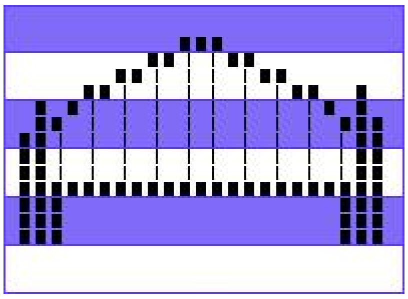
Reading TIFF files reliably is much harder than writing them. Unfortunately, I don't have enough space in this article to discuss all of the important issues, so some of them will need to be left for later articles. There are also plenty of pages on the Web that discuss the issues involved. Some of my favorites are included in the Resources section at the end of this article.

The issue that complicates reading black-and-white TIFF images the most is the several different storage schemes that are possible within the TIFF file itself. Libtiff doesn't hold your hand much with these schemes, so you have to be able to handle them yourself. The three schemes TIFF supports are single-strip images, stripped images, and tiled images:

Single-strip image
This is a special case of a stripped image, as the name suggests. In this case, all of the bitmap is stored in one large block. I have experienced reliability issues on Windows machines with images that are single-strip. The general recommendation is that no one strip should take more than 8 kilobytes uncompressed, which, with black-and-white images limits you to 65,536 pixels in a single strip.

Stripped (or multiple-strip) image
Horizontal blocks of the image are stored together. More than one strip is joined vertically to make the entire bitmap. Figure 2 shows this concept.

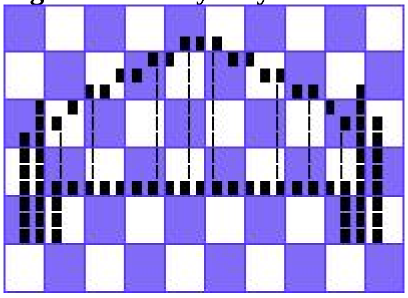
Figure 2. The Sydney Harbor Bridge, in strips



Tiled image

Like your bathroom wall, it is composed of tiles. This representation is shown in Figure 3, and is useful for extremely large images. Tiles are especially useful when you want to manipulate only a small portion of the image at any one time.

Figure 3. The Sydney Harbor Bridge, in tiles



Tiled images are comparatively uncommon, so I will focus on stripped images in this article. Remember that the single-strip case is merely a subset of multiple-strip images.

Infrastructure for reading

The most important thing to remember when reading in TIFF images is to be flexible. The reading example (Listing 3 below) has the same basic concepts as the writing example (Listing 2 above), with the major difference being that the reading example needs to deal with many possible input images. Apart from stripping and tiling, the most important thing to be flexible about is photometric interpretation. Luckily, with black-and-white images there are only two photometric interpretations to worry about; with color, and to a certain extent grayscale images, there are many more.

What is *photometric interpretation*? Well, the representation of the image in the buffer is really a very arbitrary thing. I might code my bitmaps so that 0 means black (`TIFFTAG_MINISBLACK`), whereas you might prefer that 1 means black (`TIFFTAG_MINISWHITE`). TIFF allows both, so our code has to be able to handle both cases. In the example below, I have assumed that the internal buffers need to be in `MINISWHITE`, so we will convert images that are in `MINISBLACK`.

The other big thing to bear in mind is *fill order*, that is, whether the first bit in the byte is the highest value or the lowest. Listing 3 also handles both of these correctly. I have assumed that the buffer should have the most significant bit first. TIFF images can be either big endian or little endian, but libtiff handles this for us. Thankfully, libtiff also supports the various compression algorithms without you having to worry about those. These are by far the scariest area of TIFF, so it is still worth your time to use libtiff. Listing 3 is downloadable as read.c (see Resources).

Listing 3. The reading code (read.c)

```
#include <stdio.h>
#include <tiffio.h>

int main(int argc, char *argv[]){
  TIFF *image;
  uint16 photo, bps, spp, fillorder;
  uint32 width;
  tsize_t stripSize;
```

```
unsigned long imageOffset, result;
int stripMax, stripCount;
char *buffer, tempbyte;
unsigned long bufferSize, count;

// Open the TIFF image
if((image = TIFFOpen(argv[1], "r")) == NULL){
  fprintf(stderr, "Could not open incoming image\n");
  exit(42);
}

// Check that it is of a type that we support
if((TIFFGetField(image, TIFFTAG_BITSPERSAMPLE, &bps) == 0) || (bps != 1)){
  fprintf(stderr, "Either undefined or unsupported number of bits per sample\n");
  exit(42);
}

if((TIFFGetField(image, TIFFTAG_SAMPLESPERPIXEL, &spp) == 0) || (spp != 1)){
  fprintf(stderr, "Either undefined or unsupported number of samples per pixel\n");
  exit(42);
}

// Read in the possibly multiple strips
stripSize = TIFFStripSize (image);
stripMax = TIFFNumberOfStrips (image);
imageOffset = 0;

bufferSize = TIFFNumberOfStrips (image) * stripSize;
if((buffer = (char *) malloc(bufferSize)) == NULL){
  fprintf(stderr, "Could not allocate enough memory for the uncompressed image\n");
  exit(42);
}

for (stripCount = 0; stripCount < stripMax; stripCount++){
  if((result = TIFFReadEncodedStrip (image, stripCount,
                                     buffer + imageOffset,
                                     stripSize)) == -1){
    fprintf(stderr, "Read error on input strip number %d\n", stripCount);
    exit(42);
  }

  imageOffset += result;
}

// Deal with photometric interpretations
if(TIFFGetField(image, TIFFTAG_PHOTOMETRIC, &photo) == 0){
  fprintf(stderr, "Image has an undefined photometric interpretation\n");
  exit(42);
}

if(photo != PHOTOMETRIC_MINISWHITE){
  // Flip bits
  printf("Fixing the photometric interpretation\n");

  for(count = 0; count < bufferSize; count++)
    buffer[count] = ~buffer[count];
}

// Deal with fillorder
if(TIFFGetField(image, TIFFTAG_FILLORDER, &fillorder) == 0){
  fprintf(stderr, "Image has an undefined fillorder\n");
  exit(42);
}

if(fillorder != FILLORDER_MSB2LSB){
  // We need to swap bits -- ABCDEFGH becomes HGFEDCBA
  printf("Fixing the fillorder\n");

  for(count = 0; count < bufferSize; count++){
    tempbyte = 0;
    if(buffer[count] & 128) tempbyte += 1;
    if(buffer[count] & 64) tempbyte += 2;
    if(buffer[count] & 32) tempbyte += 4;
    if(buffer[count] & 16) tempbyte += 8;
    if(buffer[count] & 8) tempbyte += 16;
    if(buffer[count] & 4) tempbyte += 32;
    if(buffer[count] & 2) tempbyte += 64;
    if(buffer[count] & 1) tempbyte += 128;
    buffer[count] = tempbyte;
  }
}

// Do whatever it is we do with the buffer -- we dump it in hex
if(TIFFGetField(image, TIFFTAG_IMAGEWIDTH, &width) == 0){
  fprintf(stderr, "Image does not define its width\n");
  exit(42);
}
```

```
  for(count = 0; count < bufferSize; count++){
    printf("%02x", (unsigned char) buffer[count]);
    if((count + 1) % (width / 8) == 0) printf("\n");
    else printf(" ");
  }

  TIFFClose(image);
}
```

This code works by first opening the image and checking that it is one that we can handle. It then reads in all of the strips for the image and appends them together into one large memory block. If required, it also flips bits until the photometric interpretation is the one we can handle, and deals with having to swap bits if the fill order is wrong. Finally, our sample outputs the image as a series of lines composed of hex values. Remember that each of the values represents 8 pixels in the actual image.

Conclusion

In this article I have shown you how to write and read some simple black-and-white images using libtiff, and introduced some key issues to be aware of. Before you start coding with libtiff, remember to put some thought into what compression algorithm you should be using for your images -- group 4 fax is great for black-and-white, but what you use for color really depends on your needs.

Resources

- Download the complete source files mentioned in this article: write-infrastructure.c, write.c, and read.c.

- The libtiff Web site is a good place to download the libtiff source. You may even find a binary package for your chosen operating system.

- If all else fails, the Adobe TIFF specification can be useful.

- Xloadimage lets you view a large number of different types of images, including TIFF, GIF, CALS, PC Paintbrush, and many others.

- The Cooper Union for the Advancement of Science and Art has some class notes from a course dealing with libtiff, "The TIFF Image File Format".

- In Looking through wxWindows (*developerWorks*, February 2001), software developer Markus Neifer introduces the wxWindows C++ and Python GUI toolkit.

- Check out GNOMEnclature: Livening things up (*developerWorks*, January 2000) for an introduction to GNOME Canvas, a powerful graphics tool for presenting data, building games, and more.

- Looking for Java graphics? In Creating images in Java applications (*developerWorks*, February 2001), Ivor Horton shows how to draw and render simple graphic images while maintaining performance.

- If you need a printer driver, take a look at the IBM Linux Technology Center's Omni printer driver, which provides support for over 400 printers.

- Browse more Linux resources on *developerWorks*.

- Browse more Open source resources on *developerWorks*.

About the author

Michael Still has been working in the image processing field for several years, including a couple of years managing and developing large image databases for an Australian government department. He currently works for Tower Software, which manufactures a world-leading EDMS and records management package called TRIM. Michael is also the developer of Panda, an open source PDF generation API, as well as the maintainer of the comp.text.pdf USENET frequently asked questions document. You can contact Michael at mikal@stillhq.com.