TIFF

Tag Image File Format

Revision 6.0

D R A F T 1

February 14, 1992

Aldus Developers Desk

Aldus Corporation
411 First Avenue South
Seattle, WA 98104-2871

CompuServe:	GO ALDSVC, Message Section #10
Fax:	(206) 343-4240
Applelink:	Aldus Developers Icon

Production Notes

This document was created electronically using Aldus PageMaker® 4.2.

Contents

Revision Notes

This revision replaces TIFF Revision 5.0.

Portions in italics are new or substantially changed in this revision.

Draft 1 Notes

This is the first draft of TIFF 6.0. There will likely be at least one more draft before the final version, which we hope to release in April, 1992.

Reviewers are encouraged to look for technical glitches and unclear wording. It is unlikely that significant technical changes will be made before final release of the TIFF 6.0 specification, since the major changes have all been thoroughly debated and voted on in the TIFF Advisory Committee. So it will very likely not be a good use of your time to send comments on things like the relative merits of JPEG vs other compression scheme, or the relative merits of YCbCr vs other color spaces.

If you do wish to comment on the TIFF 6.0 draft, you can send your comment to the Aldus Developers Desk via one of the following methods:

Internet: tiff-input@aldus.com
Fax: (206) 343-4240

(Please do not use the Internet address for general TIFF questions. They will just be ignored.)

It is likely that the document will be redesigned before final release.

Major New Features

Major enhancements to TIFF 6.0, described in Part 2:

- CMYK image definition
- A revised RGB Colorimetry section.
- YCbCr image definition
- Tiled image definition
- JPEG compression

Clarifications

The LZW compression section was clarified with respect to when to switch the coding bit depth.

The interaction between Compression=2 and PhotometricInterpretation was clarified.

Organizational Changes

- Appendices have been transformed into numbered Sections, to make the organization more consistent and expandable.

- The document was divided into two parts—Baseline and Extensions—to help the developer make better and more consistent implementation choices.
- An index and table of contents were added, to aid in navigation.

Changes in Requirements

- Appendix G, the TIFF Classes appendix, has been integrated into the main body of the specification, to expose developers to what a Baseline TIFF file actually looks like earlier in the document. As part of this integration, the TIFF Classes terminology has been replaced by the more monolithic Baseline TIFF terminology. The intent is to further encourage all mainstream TIFF readers to support the Baseline TIFF requirements, which currently includes bi-level, grayscale, RGB, and palette color images.
- Due to current licensing restrictions, LZW compression support was moved out of the Baseline TIFF section and into the Extensions section.
- Baseline TIFF requirements for palette color images were weakened a bit, in terms of which bit depths are required.

Compatibility

As always, every attempt has been made to add functionality in such a way as to minimize incompatibility problems with respect to files and software that were based on earlier versions of the TIFF specification. The goal is that TIFF files should never become obsolete, and TIFF software should not have to be revised more frequently than absolutely necessary. In particular, Baseline TIFF 6.0 files will generally be readable even by older applications that assume TIFF 5.0 or an earlier version of the specification.

However, TIFF 6.0 files that use one of the major new extensions, such as a new compression scheme or color space, will not be successfully read by older software. Such applications will have to give up in in a case like this, but they will be able to do so gracefully and with a reasonably informative message back to the user, if they have been following the rules.

Part 1: Baseline TIFF

The TIFF specification is divided into two parts.

This part, Part 1, describes *Baseline TIFF*. Baseline TIFF is what we consider to be the core of TIFF, the essentials that all mainstream TIFF developers should support in their products.

Section 1: Introduction

This document describes TIFF, a tag based file format for storing and interchanging raster images.

History

The first version of the TIFF specification was published by the Aldus Corporation in the fall of 1986. Revision 4.0 was released in April, 1987. Revision 5.0 was released in October, 1998.

Scope

TIFF describes image data that, at least in the publishing industry, typically comes from scanners, frame grabbers, and paint and photo retouch programs.

TIFF is not a printer language or page description language. TIFF merely describes and stores a scanned image. The primary design goal was to provide a rich environment within which the exchange of image data between application programs can be accomplished. This richness is required in order to take advantage of the varying capabilities of scanners and other imaging devices. Though TIFF is a rich format, it can easily be used for simple scanners and applications as well, since the number of required fields is quite short.

TIFF will be enhanced on a continuing basis as new imaging needs arise. A high priority has been given to structuring the data in such a way as to minimize the pain of future additions.

Features

TIFF is capable of describing bi-level, grayscale, palette color, and full color image data in several color spaces.

TIFF includes a number of compression schemes, to allow developers to choose the best space/time tradeoff for their applications.

TIFF is not tied to specific scanners, printers, or computer display hardware.

TIFF is portable. It does not depend on particular operating systems, file systems, compilers, or processors.

TIFF was designed to be extensible—to evolve gracefully.

TIFF allows private tags to be defined that are specific to a particular application or organization.

Section 2: Notation

Decimal and Hexadecimal

All numeric values in this document are expressed in decimal unless suffixed by ".H" to denote a hexadecimal value.

Compliance

"Is, shall" indicate mandatory requirements. All compliant writers or readers must meet the specification.

"Should" indicates a recommendation.

"May" indicates an option.

Features designated 'not recommended for general data interchange' shall be considered extensions to TIFF 6.0 proper. Files which use such features shall be designated "Extended TIFF 6.0" files, and the particular extensions used should be documented. A Baseline TIFF 6.0 reader is not required to support any extensions.

Section 3: TIFF Structure

A TIFF file is a sequence of 8-bit bytes, where the bytes are numbered from 0 to N. The largest possible TIFF file is 2**32 bytes in length.

A TIFF file begins with an 8-byte "image file header" that points to an "image file directory (IFD)." An image file directory contains information about the image, as well as pointers into the actual image data.

We will now describe these structures in more detail.

See Figure 1.

Image file header

A TIFF file begins with an 8-byte image file header, containing the following information:

Bytes 0-1: The first word of the file specifies the byte order used within the file. Legal values are:

"II"	(4949.H)
"MM"	(4D4D.H)

In the "II" format, byte order is always from least significant to most significant, for both 16-bit and 32-bit integers This is often called little-endian byte order. In the "MM" format, byte order is always from most significant to least significant, for both 16-bit and 32-bit integers. This is often called big-endian byte order.

In both formats, character strings are stored into sequential byte locations, and are null terminated.

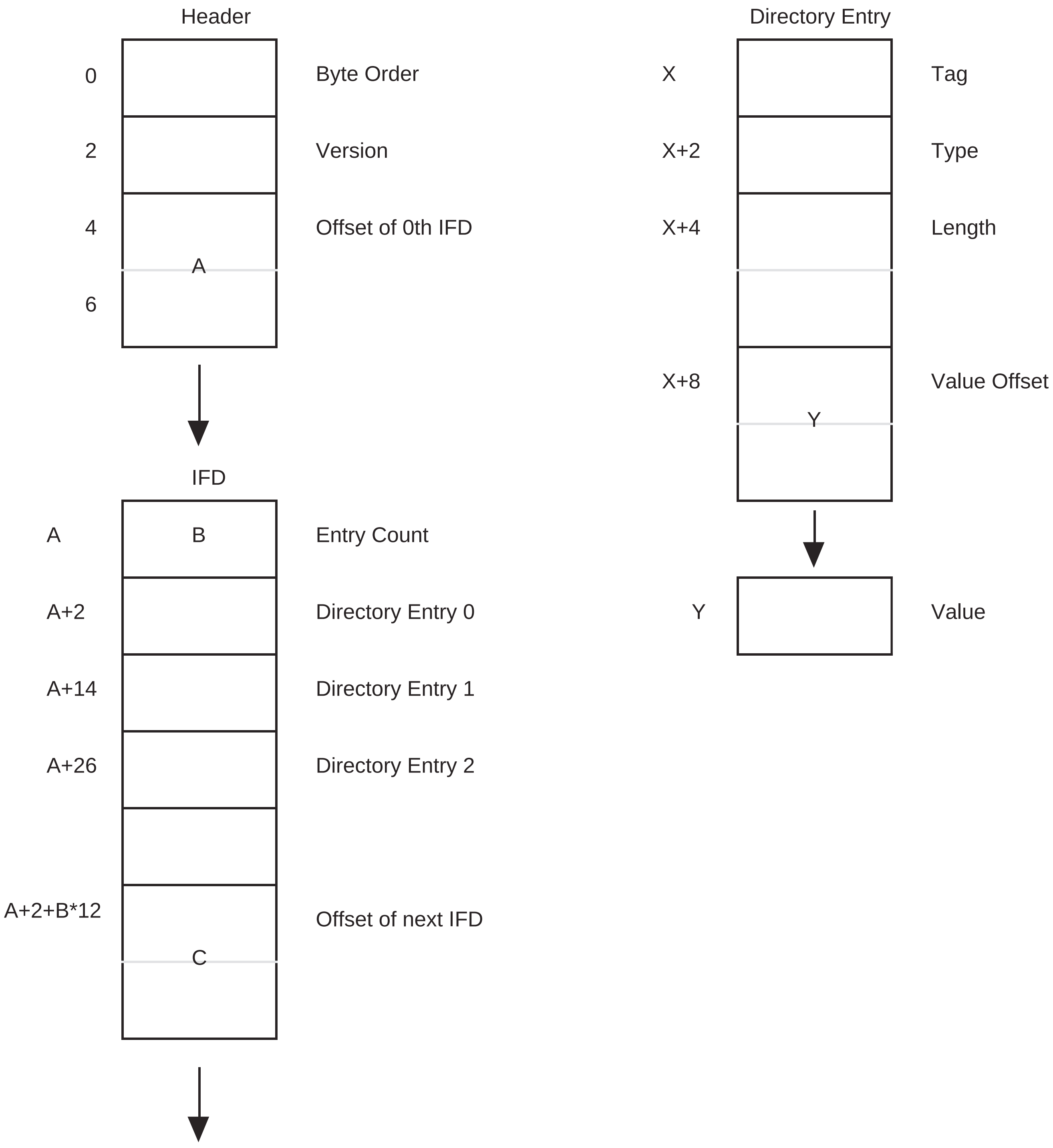
All TIFF readers should support both byte orders.

Bytes 2-3 The second word of the file is the TIFF "version number." This number, 42 (2A.H), should not be confused with the current Revision of the TIFF specification.

The TIFF version number (42) has never changed, and probably never will. If it ever does, it means that TIFF has changed in some way so radical that a TIFF reader should give up immediately. The number 42 was chosen for its deep philosophical significance. It can and should be used as additional verification that this is indeed a TIFF file.

A TIFF file does not contain a real version/revision number. There is no "TIFF X.0" tag. This was a conscious design decision. In many file formats, fields take on different meanings depending on a single version number. The problem is that as the file format "ages," it becomes increasingly difficult to document which fields mean what in a given version, and older software usually has to give up if it encounters a file with a newer version number. We wanted TIFF

Figure 1


Header
0
2
4
6
A
Byte Order
Version
Offset of 0th IFD
IFD
A
A+2
A+14
A+26
A+2+B*12
B
C
Entry Count
Directory Entry 0
Directory Entry 1
Directory Entry 2
Offset of next IFD
Directory Entry
X
X+2
X+4
X+8
Y
Tag
Type
Length
Value Offset
Y
Value

fields to have a permanent and well-defined meaning, so that "older" software can usually read "newer" TIFF files. This seems to result in lower software maintenance costs and more reliable software.

Bytes 4-7	This long word contains the offset (in bytes) of the first Image File Directory. The directory may be at any location in the file after the header but must begin on a word boundary. In particular, an Image File Directory may follow the image data it describes. Readers must simply follow the pointers, wherever they may lead. (The term "byte offset" is always used in this document to refer to a location with respect to the beginning of the file. The first byte of the file has an offset of 0.)

Image file directory

An **Image File Directory (IFD)** consists of a 2-byte count of the number of entries (i.e., the number of fields), followed by a sequence of 12-byte field entries, followed by a 4-byte offset of the next Image File Directory (or 0 if none). (Do not forget to write the 4 bytes of 0 after the last IFD!)

Each 12-byte IFD entry has the following format:

Bytes 0-1	contain the Tag that identifies the field.
Bytes 2-3	contain the field Type.
Bytes 4-7	contain the Length of the field. Length (Count) is the number of values of the indicated Type.
Bytes 8-11	contain the Value Offset, the file offset (in bytes) of the Value for the field. The Value is expected to begin on a word boundary; the corresponding Value Offset will thus be an even number. This file offset may point to anywhere in the file, including after the image data.

Terminology: a *TIFF field* is a logical entity consisting of TIFF tag and its value. This logical concept happens to be implemented as an *IFD Entry*, plus the actual value if it doesn't fit into the last 4 bytes of the IFD Entry. We will often use the terms *TIFF field* and *IFD entry* interchangeably.

The entries in an IFD must be sorted in ascending order by Tag. Note that this is not the order in which the fields are described in this document. The Values to which directory entries point need not be in any particular order in the file.

In order to save time and space, the Value Offset is interpreted to contain the Value instead of pointing to the Value if the Value fits into 4 bytes. If the Value is less than 4 bytes, it is left-justified within the 4-byte Value Offset, i.e., stored in the lower-numbered bytes. Whether or not the Value fits within 4 bytes is determined by looking at the Type and Length of the field.

Note that the Length (Count) is specified in terms of the field type, not the total number of bytes. A single 16-bit word (SHORT) has a Length of 1, not 2, for example.

The field types and their sizes are described below:

1 = BYTE	An 8-bit unsigned integer.

2 = ASCII	An 8-bit byte that contains a 7-bit ASCII code; the last byte must be null.
3 = SHORT	A 16-bit (2-byte) unsigned integer.
4 = LONG	A 32-bit (4-byte) unsigned integer.
5 = RATIONAL	Two LONG's: the first represents the numerator of a fraction, the second the denominator.

The value of the Length part of an ASCII field entry includes the null. If padding is necessary, the Length does not include the pad byte. Note that there is no "count byte," as there is in Pascal-type strings. The Length part of the field takes care of that. The null is not strictly necessary, but may make things slightly simpler for C programmers.

The reader should check the type to ensure that it is what he expects. TIFF currently allows more than 1 valid type for some fields. For example, ImageWidth and ImageLength were specified as having type SHORT. Very large images with more than 64K rows or columns are possible with some devices even now. Rather than add parallel LONG tags for these fields, it is cleaner to allow both SHORT and LONG for ImageColumns and similar fields.

In fact, all modern TIFF readers should be modified if necessary to accept BYTE, SHORT, or LONG values for any unsigned integer field. This allows a single procedure to retrieve any integer value, makes reading more robust, and saves disk space in some situations.

In TIFF 6.0, some additional field types have been defined. TIFF writers are cautioned to avoid using these new field types for at least a year or so following the release of the TIFF 6.0 specification, if at all possible. Even then, most readers will at best ignore fields that use one of the new data types. Before that time, they may well refuse to import files containing one of the new field types, even if the tag is private.

None of the new field types is actually used by a TIFF 6.0 tag. So over the short term, the new field types will be found only in some private tags.

The new field types are:

6 = SBYTE	*An 8-bit signed integer.*
7 = UNDEFINED	*An 8-bit byte that can contain anything at all, depending on the definition of the tag.*
8 = SSHORT	*A 16-bit (2-byte) signed integer.*
9 = SLONG	*A 32-bit (4-byte) signed integer.*
10 = SRATIONAL	*Two SLONG's: the first represents the numerator of a fraction, the second the denominator.*
11 = FLOAT	*Single precision (4-byte) IEEE format.*
12 = DOUBLE	*Double precision (8-byte) IEEE format.*

Warning: it is likely that other TIFF field types will be added in the future. Your reader should skip over fields containing an unexpected field type!

Fields are arrays

Please note that all TIFF fields have a Length (count) associated with them, which means that all TIFF fields are really one-dimensional arrays, even though most TIFF fields contain a single value.

For example, if you need to store some intricate data structure in a private field, you should use the UNDEFINED field type and set the Length to the number of bytes required to contain the data structure.

Multiple images per TIFF file

Note that there may be more than one IFD. Each IFD is said to define a “**subfile**.” One potential use of subsequent subfiles is to describe a related image, such as the next page of a facsimile transmission. A Baseline TIFF reader is not required to read any IFDs beyond the first one, however.

Extensions and Filetypes

The recommended MS-DOS, UNIX, and OS/2 file extension for TIFF files is “.TIF”. The recommended Macintosh Filetype is “TIFF”.

Section 4: Bi-level Images

Now that we have an idea of the overall TIFF structure, we can move on to filling the structure with actual fields (tags and values) that describe raster image data.

To make all of this clearer, we will organize the discussion according to the four Baseline TIFF image types: bi-level, grayscale, palette, and full color images.

A bi-level image contains two colors—black and white. Such data is often called "bi-level" data, since it contains two signal levels.

Color

TIFF allows an application to write out bi-level data in either a white-is-zero or black-is-zero format. The tag that records this information is called PhotometricInterpretation. It is defined like this:

PhotometricInterpretation

Tag = 262 (106.H)
Type = SHORT

Values:

0 = WhiteIsZero. For bi-level and grayscale images: 0 is imaged as white. 2**BitsPerSample-1 is imaged as black. This is the normal value for Compression=2.

1 = BlackIsZero. For bi-level and grayscale images: 0 is imaged as black. 2**BitsPerSample-1 is imaged as white. If this value is specified for Compression=2, the image should display and print reversed.

Bit Depth

Of course, it only takes 1 bit to store a bi-level pixel value. In TIFF, this fact is recorded in the BitsPerSample tag:

BitsPerSample

Tag = 258 (102.H)
Type = SHORT

For bi-level images, the value of this tag is 1.

Compression

The data can be stored either uncompressed or compressed. The tag that records this information is the Compression tag:

Compression

Tag = 259 (103.H)
Type = SHORT

Values:

1 = No compression, but pack data into bytes as tightly as possible, with no unused bits except at the end of a row. The component values are stored as an array of type

BYTE. Each scan line (row) is padded to the next BYTE boundary.

2 = CCITT Group 3 1-Dimensional Modified Huffman run length encoding. See the Modified Huffman Compression section.

32773 = PackBits compression, a simple byte oriented run length scheme. See the PackBits section for details.

Data compression only applies to raster image data. All other TIFF fields are unaffected.

Baseline TIFF readers must handle all three compression schemes.

Pixel Size

An image in TIFF is organized as a rectangular array of pixels, containing M rows of N columns each. These numbers are stored in the following fields:

ImageLength

Tag = 257 (101.H)
Type = SHORT or LONG

The number of rows (sometimes described as ‘scan lines’) in the image.

ImageWidth

Tag = 256 (100.H)
Type = SHORT or LONG

The number of columns in the image, i.e., the number of pixels per scan line.

Physical Size

We often also want to know the size of the picture that is represented by this image. Instead of storing the total image size directly, we store the number of pixels per inch or per centimeter, using the following three fields:

ResolutionUnit

Tag = 296 (128.H)
Type = SHORT

Values:

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio, but no meaningful absolute dimensions.

2 = Inch.

3 = Centimeter.

XResolution

Tag = 282 (11A.H)
Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageColumns direction.

YResolution

Tag = 283 (11B.H)
Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageLength direction.

Location of the Data

Finally, we need to point to the actual compressed or uncompressed data. In TIFF, the data can be stored almost anywhere in the file, to give a TIFF editor maximum flexibility in where to put the data. TIFF also allows and encourages writers to break the image into separate strips, for increased editing flexibility as well as more efficient buffering. The location and size of each strip is stored in these three fields:

RowsPerStrip

Tag	= 278 (116.H)
Type	= SHORT or LONG

The number of rows in each strip (except possibly the last strip.)

For example, if ImageLength is 24, and RowsPerStrip is 10, then there are 3 strips, with 10 rows in the first strip, 10 rows in the second strip, and 4 rows in the third strip. (The data in the last strip is not padded out with 6 extra rows of dummy data.)

StripOffsets

Tag	= 273 (111.H)
Type	= SHORT or LONG

For each strip, the byte offset of that strip. The offset is specified with respect to the beginning of the TIFF file.

StripByteCounts

Tag	= 279 (117.H)
Type	= SHORT or LONG

For each strip, the number of bytes in that strip, *after compression*.

Putting it all together, and adding a couple of less-important fields that we will be discussing later, we look at a sample bilevel image file:

A Sample Bilevel TIFF File

```
Offset                          Value
(hex)   Name                    (mostly hex)
Header:
0000    Byte Order              4D4D
0002    Version                 002A
0004    1st IFD pointer         00000014
IFD:
0014    Entry Count                   000D
0016    NewSubfileType                00FE  0004  00000001   00000000
0022    ImageWidth                    0100  0004  00000001   000007D0
002E    ImageLength                   0101  0004  00000001   00000BB8
003A    Compression                   0103  0003  00000001   8005 0000
0046    PhotometricInterpretation     0106  0003  00000001   0001 0000
0052    StripOffsets                  0111  0004  000000BC   000000B6
005E    RowsPerStrip                  0116  0004  00000001   00000010
006A    StripByteCounts               0117  0003  000000BC   000003A6
0076    XResolution                   011A  0005  00000001   00000696
0082    YResolution                   011B  0005  00000001   0000069E
008E    Software                      0131  0002  0000000E   000006A6
009A    DateTime                      0132  0002  00000014   000006B6
00A6    Next IFD pointer              00000000
Fields pointed to by the tags:
00B6    StripOffsets            Offset0, Offset1, ... Offset187
03A6    StripByteCounts         Count0, Count1, ... Count187
0696    XResolution             0000012C 00000001
069E    YResolution             0000012C 00000001
06A6    Software                “PageMaker 3.0”
06B6    DateTime                “1988:02:18 13:59:59”
Image Data:
00000700                        Compressed data for strip 10
xxxxxxxx                        Compressed data for strip 179
xxxxxxxx                        Compressed data for strip 53
xxxxxxxx                        Compressed data for strip 160
.
.
End of example
```

Comments on the bilevel image example

1. The IFD in our example starts at position hex 14. It could have been anywhere in the file as long

as the position is even and greater than or equal to 8, since the TIFF header is 8 bytes long and must be the first thing in a TIFF file.

2. With 16 rows per strip, we have 188 strips in all.
3. The example uses a number of optional fields, such as DateTime. TIFF readers must safely skip over these fields if they do not want to use the information. And Baseline TIFF readers must not require that such fields be present.
4. To make a point, our example has highly fragmented image data; the strips of our image are not even in sequential order. The point is that strip offsets must not be ignored. Never assume that strip N+1 follows strip N on disk. Incidentally, there is no requirement that the image data must follow the IFD information. As long as you follow the pointers, whether they be IFD pointers, field pointers, or Strip Offsets, you can't go wrong.

Summary list of tags for bilevel images

Here is a list of required tags for bilevel images, in numerical order, which is how they would appear in the IFD. (Note that the example above does not include some of these tags. This is permitted, since the tags that were omitted each have a default, and the default is appropriate for this file.)

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	1
Compression	259	103	SHORT	1 or 2 or 32773
PhotometricInterpretation	262	106	SHORT	0 or 1
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3

Section 5: Grayscale Images

Grayscale images are a generalization of bilevel images. Instead of just black and white, grayscale images can store shades of gray. In order to describe such images, we need to add or change the following tags. The other required tags are the same as for bilevel images.

BitsPerSample = 4 or 8. This gives us either 16 or 256 distinct shades of gray.

Compression = 1 *or 32773 (PackBits)*. In Baseline TIFF, we can store grayscale images either as uncompressed or in PackBits form.

PhotometricInterpretation = 0 or 1. The image can either be stored WhiteIsZero or BlackIsZero.

Summary list of tags for grayscale images

The list of required tags for grayscale images, in numerical order:

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	4 or 8
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	0 or 1
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3

Section 6: Palette Color Images

Palette color images are in a sense a generalization of grayscale images. They still have one component per pixel, but the component value is used as an index into a full RGB lookup table. In order to describe such images, we need to add or change the following tags. The other required tags are the same as for grayscale images.

BitsPerSample = 1,2,4, or 8. Note that we can even have 1-bit palette color images. Such images will have two colors, like bilevel images, but those colors need not be black and white.

Compression = 1 *or 32773 (PackBits).*

PhotometricInterpretation = 3 (Palette Color).

ColorMap

Tag	= 320 (140.H)
Type	= SHORT
N	= 3 * (2**BitsPerSample)

This tag defines a Red-Green-Blue color map for palette color images. The palette color pixel value is used to index into all 3 subcurves. For example, a Palette color pixel having a value of 0 would be displayed according to the 0th entry of the Red, Green, and Blue subcurves.

The subcurves are stored sequentially. The Red entries come first, followed by the Green entries, followed by the Blue entries. The length of each subcurve is 2**BitsPerSample. A ColorMap entry for an 8-bit Palette color image would therefore have 3 * 256 entries. The width of each entry is 16 bits, as implied by the type of SHORT. 0 represents the minimum intensity, and 65535 represents the maximum intensity. Black is represented by 0,0,0, and white by 65535, 65535, 65535. The purpose of the color map is to act as a "lookup" table mapping pixel values from 0 to 2**BitsPerSample-1 into RGB triplets.

No default. ColorMap must be included in all palette color images.

Summary list of tags for palette color images

The list of required tags for grayscale images, in numerical order:

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	1, 2, 4 , or 8
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	3
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3
ColorMap	320	140	SHORT	

Section 7: RGB Full Color Images

RGB full color images are in a sense a generalization of palette color images. Like a palette color image, a full color image describes a color picture. However, in the full color case, each pixel is made up of three components—red, green, and blue—and is 24 bits deep. There is no ColorMap.

In order to describe such images, we need to add or change the following tags and values. The other required tags are the same as for palette color images.

SamplesPerPixel

Tag = 277 (115.H)
Type = SHORT

The number of components per pixel. SamplesPerPixel is 1 for bilevel, grayscale, and palette color images. SamplesPerPixel is 3 for RGB images.

Default = 1.

BitsPerSample = 8,8,8. Each component is 8 bits deep.

Compression = 1 *or 32773 (PackBits)*.

PhotometricInterpretation = 2 (RGB).

Summary list of tags for palette color images

The list of required tags for grayscale images, in numerical order:

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	8,8,8 (three values)
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	2
StripOffsets	273	111	SHORT or LONG	
SamplesPerPixel	277	115	SHORT	3
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3

Section 8: Baseline TIFF Requirements

This section describes requirements that are common to all Baseline TIFF images.

General Requirements

The following are required characteristics of all Baseline TIFF files.

Where there are options, TIFF writers can do whichever one they want, but Baseline TIFF readers must be able to handle all of them.

Defaults. TIFF writers may, but are not required, to write out a field that has a default value, if the default value is the one desired. TIFF readers must be prepared to handle either situation.

Other fields. TIFF readers must be prepared to encounter fields other than the required fields in TIFF files. TIFF writers are allowed to write fields such as Make, Model, DateTime, and so on, and TIFF readers can certainly make use of such fields if they exist. TIFF readers must not, however, refuse to read the file if such optional fields do not exist.

‘MM’ and ‘II’ byte order. TIFF readers must be able to handle both byte orders. TIFF writers can do whichever is most convenient or efficient. Images are interchanged between unlike computers with a surprisingly high frequency.

Multiple subfiles. TIFF readers must be prepared for multiple images (i.e., subfiles) per TIFF file, although they are not required to do anything with any image after the first one. TIFF writers must be sure to write a long word of 0 after the last IFD (this is the mandatory way of signalling that this IFD was the last one), as was explained in the TIFF structure discussion.

If a TIFF writer writes multiple subfiles, the first one must be the full resolution image. Subsequent subimages, such as reduced resolution images, may be in any order in the TIFF file. If a reader wants to make use of such subimages, it will have to scan the IFD’s before deciding how to proceed.

TIFF Editors. Editors—applications that modify TIFF files—have a few additional requirements.

TIFF editors must be especially careful about subfiles. If a TIFF editor edits a full-resolution subfile, but does not update an accompanying reduced-resolution subfile, a reader that uses the reduced-resolution subfile for screen display will display the wrong thing. So TIFF editors must either create a new reduced-resolution subfile when they alter a full-resolution subfile, or else they must simply delete any subfiles that they aren’t prepared to deal with.

A similar situation arises with the fields themselves. A TIFF editor need only worry about the TIFF required fields. In particular, it is unnecessary, and probably dangerous, for an editor to copy fields that it does not understand, since the editor may have altered the file in a way that is incompatible with the unknown fields.

No Duplicate Pointers. *Do not point to the same piece of information from more than one place. TIFF readers and editors are under no obligation to detect this condition and handle it properly, and most do not. This wouldn't be a problem if TIFF files were read-only entities, but they are not. This warning covers TIFF tag pointers as well as tags such as StripOffsets and TileOffsets. The small space savings is not worth the trouble it causes.*

And make sure that your strips point to real data. Do not use a stripoffset of 0 to mean something special, for example. There are hundreds of TIFF readers out there, and they will not know what you mean. Also, if you need to record some special information for each strip, use a private tag.

Watch out for extra components. *Some TIFF files may have more components per pixel than you think. We strongly recommend that you skip over them gracefully, using the value of the SamplesPerPixel tag. For example, it is possible that the data will have a PhotometricInterpretation of RGB, but have 4 SamplesPerPixel. See the ExtraSamples tag for further details.*

Watch out for new field types. *Be prepared to handle field types that you are not expecting, such as floating point data. Make sure your reader skips over such fields gracefully. Do not expect that BYTE, ASCII, SHORT, LONG, and RATIONAL will always be a complete list of field types.*

Notes on Required Fields

NewSubfileType. LONG. Recommended but not required.

ImageWidth. SHORT or LONG. (That is, both "SHORT" and "LONG" TIFF field types are allowed, and must be handled properly by readers. TIFF writers can use either.) TIFF readers are not required to read arbitrarily large files however. Some readers will give up if the entire image cannot fit in available memory. (In such cases the reader should inform the user of the nature of the problem.) Others will probably not be able to handle ImageWidth greater than 65535. Recommendation: use LONG, since resolutions seem to keep going up.

ImageLength. SHORT or LONG.

RowsPerStrip. SHORT or LONG. Readers must be able to handle any value between 1 and 2**32-1. However, some readers may try to read an entire strip into memory at one time, so that if the entire image is one strip, the application may run out of memory. Recommendation 1: Set RowsPerStrip such that the size of each strip is about 8K bytes. Do this even for uncompressed data, since it is easy for a writer and makes things simpler for readers. (Note: extremely wide, high-resolution images may have rows larger than 8K bytes; in this case, RowsPerStrip should be 1, and the strip will just have to be larger than 8K.

StripOffsets. SHORT or LONG. As explained in the main part of the specification, the number of StripOffsets depends on RowsPerStrip and ImageLength. (LONG must, of course, be used if the file is more than 64K bytes in length.)

StripByteCounts. SHORT or LONG. *As of TIFF 6.0, we will require StripByteCounts in Baseline TIFF files.*

XResolution, YResolution. RATIONAL. Note that the X and Y resolutions may be unequal. A TIFF reader must be able to handle this case. TIFF pixel-editors will typically not care about the resolution, but applications such as page layout programs will.

ResolutionUnit. SHORT. TIFF readers must be prepared to handle all three values for ResolutionUnit.

Section 9: Comprehensive Description of Baseline TIFF Fields

This section describes the Baseline fields defined in this version of TIFF. More fields may be added in future versions. Whenever possible they will be added in a way that allows old TIFF reader software to read newer TIFF files.

The documentation for each field contains the name of the field (quite arbitrary, but convenient), the numeric Tag value, the field Type, the required Number of Values (N), comments describing the field, and the default, if any. Readers must assume the default value if the field does not exist.

“No default” does not mean that a TIFF writer should not pay attention to the tag. It simply means that there is no default. If the writer has reason to believe that readers will care about the value of this field, the writer should write the field with the appropriate value. TIFF readers can do whatever they want if they encounter a missing “no default” field that they care about.

Many fields described in this part of the document are not required, or are only required for particular types of TIFF files. See the preceding sections for lists of required fields.

Definitions

Before we begin defining the fields, we will define some basic concepts. An Baseline TIFF **image** is defined to be a two-dimensional array of “**pixels**,” each of which consists of one or more “**components**.” With monochromatic data, we have one component per pixel, and “component” and “pixel” can be used interchangeably. RGB color data contains three components per pixel.

Architectural Fields

Basic fields are fields that are fundamental to the pixel architecture of an image and how and where the data is stored in the file.

BitsPerSample

Tag	= 258 (102.H)
Type	= SHORT
N	= SamplesPerPixel

Number of bits per component.

Note that this tag allows a different number of bits per component for each component corresponding to a pixel. For example, RGB color data could use a different number of bits per component for each of the three color planes. Most RGB files will have the same number of BitsPerSample for each component. Even in this case, be sure to include all three entries. Writing “8” when you mean “8,8,8” sets a bad precedent for other fields.

Default = 1. See also SamplesPerPixel.

Compression

Tag	= 259 (103.H)
Type	= SHORT
N	= 1

1 = No compression, but pack data into bytes as tightly as possible, with no unused bits except at the end of a row. The component values are stored as an array of type BYTE, for BitsPerSample <= 8, SHORT if BitsPerSample > 8 and <= 16, and LONG if BitsPerSample > 16 and <= 32. *Each scan line (row) is padded to the next BYTE/SHORT/LONG boundary, depending on the bit depth rules specified in the preceding sentence.* The byte ordering of data >8 bits must be consistent with that specified in the TIFF file header (bytes 0 and 1). “II” format files will therefore have the least significant bytes preceding the most significant bytes while “MM” format files will have the opposite order, *in the SHORT and LONG cases*.

If the number of bits per component is not a power of 2, and you are willing to give up some space for better performance, you may wish to use the next higher power of 2. For example, if your data can be represented in 6 bits, you may wish to specify that it is 8 bits deep, *and then high-order-justify the 6 data bits in each byte.*

Rows are required to begin on byte boundaries. *(SHORT boundaries if BitsPerSample is > 8, LONG boundaries if BitsPerSample is > 16).* The number of bytes per row is therefore (ImageWidth * SamplesPerPixel * BitsPerSample + 7) / 8, assuming integer arithmetic, for PlanarConfiguration=1. *(If BitsPerSample is > 8, replace 7 and 8 with 15 and 16. If BitsPerSample is > 16, replace 7 and 8 with 31 and 32.)* Replace “ImageWidth*SamplesPerPixel” with “ImageWidth” for PlanarConfiguration=2.

Some graphics systems want rows to be word- or double-word-aligned. Uncompressed TIFF rows will need to be copied into word-

or double-word-padded row buffers before being passed to the graphics routines in these environments.

2 = CCITT Group 3 1-Dimensional Modified Huffman run length encoding. See the Modified Huffman compression section. BitsPerSample must be 1, since this type of compression is defined only for bilevel images.

By convention, the normal PhotometricInterpretation for Compression=2 is 0 (WhiteIsZero). When you decompress data that has been compressed by Compression=2, you must translate white runs into 0's and black runs into 1's. If a reader encounters a PhotometricInterpretation of 1 (BlackIsZero) for such an image, the image should be displayed and printed with black and white reversed.

32773 = PackBits compression, a simple byte oriented run length scheme. See the PackBits section for details.

Data compression only applies to raster image data, as pointed to by StripOffsets. All other TIFF information is unaffected.

Default = 1.

ImageWidth

Tag	= 256 (100.H)
Type	= SHORT or LONG
N	= 1

The number of columns in the image, i.e., the number of pixels per scan line. See also ImageLength.

No default.

ImageLength

Tag	= 257 (101.H)
Type	= SHORT or LONG
N	= 1

The number of rows (sometimes described as "scan lines") in the image. See also ImageWidth.

No default.

NewSubfileType

Tag = 254 (FE.H)

Type = LONG

N = 1

Replaces the old SubfileType field, due to limitations in the definition of that field.

A general indication of the kind of data that is contained in this subfile. This field is made up of a set of 32 flag bits. Unused bits are expected to be 0. Bit 0 is the low-order bit.

Currently defined values are:

Bit 0	is 1 if the image is a reduced resolution version of another image in this TIFF file; else the bit is 0.
Bit 1	is 1 if the image is a single page of a multi-page image (see the

PageNumber tag description); else the bit is 0.

Bit 2	is 1 if the image defines a transparency mask for another image in this TIFF file. The PhotometricInterpretation value must be 4, designating a transparency mask.

These values have been defined as bit flags because they are independent of each other.

Default is 0.

PlanarConfiguration

Tag	= 284 (11C.H)
Type	= SHORT
N	= 1

1 = *Chunky format*. The component values for each pixel are stored contiguously, so that there is a single image plane. See PhotometricInterpretation to determine the order of the components within the pixel data. So, for RGB data, the data is stored RGBRGBRGB...and so on.

2 = *Planar format*. The components are stored in separate “component planes.” The values in StripOffsets and StripByteCounts are then arranged as a 2-dimensional array, with SamplesPerPixel rows and StripsPerImage columns. (All of the columns for row 0 are stored first, followed by the columns of row 1, and so on.) PhotometricInterpretation describes the type of data that is stored in each component plane. For example, RGB data is stored with the Red components in one component plane, the Green in another, and the Blue in another.

Planar format is not currently in widespread use, and is not recommended for general interchange. That is, its use should be considered an extension.

If SamplesPerPixel is 1, PlanarConfiguration is irrelevant, and should not be included.

If a row interleave effect is desired, a writer could write out the data as PlanarConfiguration=2—separate sample planes—but break up the planes into multiple strips (one row per strip, perhaps) and interleave the strips.

Default is 1. See also BitsPerSample, SamplesPerPixel.

RowsPerStrip

Tag	= 278 (116.H)
Type	= SHORT or LONG
N	= 1

The number of rows per strip. The image data is organized into strips for fast access to individual rows when the data is compressed—though this field is valid even if the data is not compressed.

RowsPerStrip and ImageLength together tell us the number of strips in the entire image. The equation is

StripsPerImage = (ImageLength + RowsPerStrip - 1) / RowsPerStrip,

assuming integer arithmetic. That is, the number of strips my be computed as the integer part of the quotient of ImageLength + RowsPerStrip - 1 divided by RowsPerStrip.

Yet another way to write the equation would be:

StripsPerImage = ceil (ImageLength/RowsPerStrip)

where "ceil" is a ceiling function, rounding up to the next higher integer.

StripsPerImage is NOT a tag. It is merely a value that a TIFF reader will want to compute, since it gives the number of StripOffsets and StripByteCounts for the image.

Note that either SHORT or LONG values can be used to specify RowsPerStrip. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specification revisions required LONG values and that some software may not expect SHORT values.

Default is 2**32 - 1, which is effectively infinity. That is, the entire image is one strip.

We do not recommend a single strip, however. Choose RowsPerStrip such that each strip is about 8K bytes, even if the data is not compressed, since it makes buffering simpler for readers. The "8K" part is pretty arbitrary, but seems to work well.

See also ImageLength, StripOffsets, StripByteCounts.

SamplesPerPixel

Tag	= 277 (115.H)
Type	= SHORT
N	= 1

The number of components per pixel. SamplesPerPixel is 1 for bilevel, grayscale, and palette color images. SamplesPerPixel is 3 for RGB images.

Default = 1. See also BitsPerSample, PhotometricInterpretation.

StripByteCounts

Tag	= 279 (117.H)
Type	= SHORT or LONG
N	= StripsPerImage for PlanarConfiguration equal to 1.
	= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

For each strip, the number of bytes in that strip, *after compression*.

The existence of this field greatly simplifies the chore of buffering compressed data, if the strip size is reasonable.

No default. See also StripOffsets, RowsPerStrip.

StripOffsets

Tag	= 273 (111.H)
Type	= SHORT or LONG
N	= StripsPerImage for PlanarConfiguration equal to 1.
	= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

For each strip, the byte offset of that strip. The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each strip has a location independent of the locations of other strips. This feature may be useful for editing applications. This field is the only way for a reader to find the image data, and hence is required. *(Unless TileOffsets is used; see TileOffsets.)*

Note that either SHORT or LONG values can be used to specify the strip offsets. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specifications required LONG strip offsets and that some software may not expect SHORT values.

For maximum compatibility with MS-DOS and Windows programs, The StripOffsets array should be less than or equal to 64K bytes in length.

Similarly, the strips themselves, in both compressed and uncompressed forms, should not be larger than 64K bytes.

No default. See also StripByteCounts, RowsPerStrip.

Color Description Fields

ColorMap

Tag	= 320 (140.H)
Type	= SHORT
N	= 3 * (2**BitsPerSample)

This tag defines a Red-Green-Blue color map for palette color images. The palette color pixel value is used to index into all 3 subcurves. For example, a Palette color pixel having a value of 0 would be displayed according to the 0th entry of the Red, Green, and Blue subcurves.

The subcurves are stored sequentially. The Red entries come first, followed by the Green entries, followed by the Blue entries. The length of each subcurve is 2**BitsPerSample. A ColorMap entry for an 8-bit Palette color image would therefore have 3 * 256 entries. The width of each entry is 16 bits, as implied by the type of SHORT. 0 represents the minimum intensity, and 65535 represents the maximum intensity. Black is represented by 0,0,0, and white by 65535, 65535, 65535. The purpose of the color map is to act as a "lookup" table mapping pixel values from 0 to 2**BitsPerSample-1 into RGB triplets.

See also PhotometricInterpretation—palette color.

No default. ColorMap must be included in all palette color images.

PhotometricInterpretation

Tag	= 262 (106.H)
Type	= SHORT
N	= 1

0 = WhiteIsZero. For bilevel and grayscale images: 0 is imaged as white. 2**BitsPerSample-1 is imaged as black. This is the normal value for Compression=2.

1 = BlackIsZero. For bilevel and grayscale images: 0 is imaged as black. 2**BitsPerSample-1 is imaged as white. If this value is specified for Compression=2, the image should display and print reversed.

2 = RGB. In the RGB model, a color is described as a combination of the three primary colors of light (red, green, and blue) in particular concentrations. For each of the three components, 0 represents minimum intensity, and 2**BitsPerSample - 1 represents maximum intensity. Thus an RGB value of (0,0,0) represents black, and (255,255,255) represents white, assuming 8-bit components. For PlanarConfiguration = 1, the components are stored in the indicated order: first Red, then Green, then Blue. For PlanarConfiguration = 2, the StripOffsets for the component planes are stored in the indicated order: first the Red component plane StripOffsets, then the Green plane StripOffsets, then the Blue plane StripOffsets.

3= Palette color. In this mode, a color is described with a single component. The component is used as an index into ColorMap. The component is used to index into each of the red, green and blue curve tables to retrieve an RGB triplet defining an actual color. When this PhotometricInterpretation value is used, ColorMap must also be supplied. SamplesPerPixel must be 1.

4 = Transparency Mask.

This means that the image is used to define an irregularly shaped region of another image in the same TIFF file. SamplesPerPixel and BitsPerSample must be 1. PackBits compression is recommended. The 1-bits define the interior of the region; the 0-bits define the exterior of the region.

A reader application can use the mask to determine which parts of the image to display. Main image pixels that correspond to 1-bits in the transparency mask are imaged to the screen or printer, but main image pixels that correspond to 0-bits in the mask are not displayed or printed.

The image mask is typically at a higher resolution than the main image, if the main image is grayscale or color, so that the edges can be sharp.

There is no default for PhotometricInterpretation, *and it is required*. Do not rely on applications defaulting to what you want!

Resolution Fields

ResolutionUnit

Tag	= 296 (128.H)
Type	= SHORT
N	= 1

To be used with XResolution and YResolution.

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio, but no meaningful absolute dimensions.

The drawback of ResolutionUnit=1 is that different applications will import the image at different sizes. Even if the decision is quite arbitrary, it might be better to use dots per inch or dots per centimeter, and pick XResolution and YResolution such that the aspect ratio is correct and the maximum dimension of the image is about four inches (the “four” is arbitrary.)

2 = Inch.

3 = Centimeter.

Default is 2. See also XResolution, YResolution.

XResolution

Tag	= 282 (11A.H)
Type	= RATIONAL
N	= 1

The number of pixels per ResolutionUnit in the ImageWidth direction.

It is, of course, not mandatory that the image be actually displayed or printed at the size implied by this parameter. It is up to the application to use this information as it wishes.

No default. See also YResolution, ResolutionUnit.

YResolution

Tag	= 283 (11B.H)
Type	= RATIONAL
N	= 1

The number of pixels per ResolutionUnit in the ImageLength direction.

No default. See also XResolution, ResolutionUnit.

Informational Fields

Informational fields are fields that can provide useful information to a user, such as where the image came from. Most are ASCII fields. An application could have some sort of "More Info..." dialog box to display such information.

Artist

Tag = 315 (13B.H)
Type = ASCII

Person who created the image.

If you need to attach a Copyright notice to an image, this is the place to do it. In fact, you may wish to write out the contents of the field immediately after the 8-byte TIFF header. Just make sure your IFD and field pointers are set accordingly, and you're all set.

DateTime

Tag = 306 (132.H)
Type = ASCII
N = 20

Date and time of image creation. Use the format "YYYY:MM:DD HH:MM:SS", with hours on a 24-hour clock, and one space character between the date and the time. The length of the string, including the null, is 20 bytes.

HostComputer

Tag = 316 (13C.H)
Type = ASCII

"ENIAC", or whatever.

See also Make, Model, Software.

ImageDescription

Tag = 270 (10E.H)
Type = ASCII

For example, a user may wish to attach a comment such as "1988 company picnic" to an image.

It has been suggested that this is what the newspaper and magazine industry calls a "slug."

Make

Tag = 271 (10F.H)
Type = ASCII

Manufacturer of the scanner, video digitizer, or whatever.

See also Model, Software.

Model

Tag = 272 (110.H)
Type = ASCII

The model name/number of the scanner, video digitizer, or whatever.

This tag is intended for user information only.

See also Make, Software.

Software

Tag = 305 (131.H)
Type = ASCII

Name and release number of the software package that created the image.

This tag is intended for user information only.

See also Make, Model.

Special Purpose Fields

These fields were defined in earlier versions of the specification, but they are rarely used, for the most part. They have either been superseded by other fields, have been found to have serious drawbacks, or are simply not as useful as once thought.

CellLength

Tag	= 265 (109.H)
Type	= SHORT
N	= 1

The length, in 1-bit components, of the dithering/halftoning matrix. Assumes that Threshholding = 2.

> This field, plus CellWidth and Threshholding, are problematic because they cannot safely be used to reverse-engineer grayscale image data out of dithered/halftoned black-and-white data, which is their only plausible purpose. The only "right" way to do it is to not bother with anything like these fields, and instead write some sophisticated pattern-matching software that can handle screen angles that are not multiples of 45 degrees, and other such challenging dithered/halftoned data.
>
> So we do not recommend trying to convert dithered or halftoned data into grayscale data. Dithered and halftoned data require careful treatment to avoid "stretch marks," but it can be done. If you want grayscale images, get them directly from the scanner or frame grabber or whatever.

No default. See also Threshholding.

CellWidth

Tag	= 264 (108.H)
Type	= SHORT
N	= 1

The width, in 1-bit components, of the dithering/halftoning matrix.

No default. See also Threshholding. See the comments for CellLength.

FillOrder

Tag	= 266 (10A.H)
Type	= SHORT
N	= 1

The order of data values within a byte.

1 = most significant bits of the byte are filled first. That is, data values (or code words) are ordered from high order bit to low order bit within a byte.

2 = least significant bits are filled first. Since little interest has been expressed in least-significant fill order to date, and since it is easy and inexpensive for writers to reverse bit order (use a 256-byte lookup table), we recommend that FillOrder=2 be used only in special-purpose applications.

Note that FillOrder is completely separate from whether the file is an "II" or "MM" style file. FillOrder affects only the order of bits within a byte. "II" and "MM" affect only the order of bytes within a word (SHORT) or double word (LONG).

Default is FillOrder = 1.

FreeByteCounts

Tag	= 289 (121.H)
Type	= LONG

For each “free block” in the file, the number of bytes in the block.

TIFF readers can ignore FreeOffsets and FreeByteCounts if present.

FreeOffsets and FreeByteCounts do not constitute a remapping of the logical address space of the file.

Since this information can be generated by scanning the IFDs, StripOffsets, and StripByteCounts, FreeByteCounts and FreeOffsets are not needed.

In addition, it is not clear what should happen if FreeByteCounts and FreeOffsets exist in more than one IFD.

See also FreeOffsets.

FreeOffsets

Tag	= 288 (120.H)
Type	= LONG

For each “free block” in the file, its byte offset.

See also FreeByteCounts.

GrayResponseCurve

Tag	= 291 (123.H)
Type	= SHORT
N	= 2**BitsPerSample

The purpose of the gray response curve and the gray units is to provide more exact photometric interpretation information for gray scale image data, in terms of optical density.

The GrayScaleResponseUnits specifies the accuracy of the information contained in the curve. Since optical density is specified in terms of fractional numbers, this tag is necessary to know how to interpret the stored integer information. For example, if GrayScaleResponseUnits is set to 4 (ten-thousandths of a unit), and a GrayScaleResponseCurve number for gray level 4 is 3455, then the resulting actual value is 0.3455. Optical densitometers typically measure densities within the range of 0.0 to 2.0.

If the gray scale response curve is known for the data in the TIFF file, and if the gray scale response of the output device is known, then an intelligent conversion can be made between the input data and the output device. For example, the output can be made to look just like the input. In addition, if the input image lacks contrast (as can be seen from the response curve), then appropriate contrast enhancements can be made.

The purpose of the gray scale response curve is to act as a “lookup” table mapping values from 0 to 2**BitsPerSample-1 into specific density values. The 0th element of the GrayResponseCurve array is used to define the gray value for all pixels having a value of 0, the 1st element of the GrayResponseCurve array is used to define the gray value for all pixels having a value of 1, and so on, up to 2**BitsPerSample-1. If your data is “really,” say, 7-bit data, but you are adding a 1-bit pad to each pixel to turn it into 8-bit data, everything still works: If the data is high-order justified, half of your GrayResponseCurve entries (the odd ones, probably) will never be used, but that doesn’t hurt anything. If the data is low-order justified, your pixel values will be between 0 and 127, so make your GrayResponseCurve accordingly. What your curve does from 128 to 255 doesn’t matter. Note that low-order justification is probably not a good idea, however, since not all applications look at GrayResponseCurve. Note also that LZW compression yields the same compression ratio regardless of whether the data is high-order or low-order justified.

It is permissible to have a GrayResponseCurve even for bilevel (1-bit) images. The GrayResponseCurve will have 2 values. It should be noted, however, that TIFF B readers are not required to pay attention to GrayResponseCurves in TIFF B files.

See also GrayResponseUnit, PhotometricInterpretation.

GrayResponseUnit

Tag	= 290 (122.H)
Type	= SHORT
N	= 1

1 = Number represents tenths of a unit.

2 = Number represents hundredths of a unit.

3 = Number represents thousandths of a unit.

4 = Number represents ten-thousandths of a unit.

5 = Number represents hundred-thousandths of a unit.

Modifies GrayResponseCurve.

See also GrayResponseCurve.

For historical reasons, the default is 2. However, for greater accuracy, we recommend using 3.

MaxSampleValue

Tag	= 281 (119.H)
Type	= SHORT
N	= SamplesPerPixel

The maximum used component value. For example, if the image consists of 6-bit data low-order-justified into 8-bit bytes, MaxSampleValue will be no greater than 63. This is field is not to be used to affect the visual appearance of the image when displayed. Nor should the values of this field affect the interpretation of any other field. Use it for statistical purposes only.

Default is 2**(BitsPerSample) - 1.

MinSampleValue

Tag	= 280 (118.H)
Type	= SHORT
N	= SamplesPerPixel

The minimum used component value. This field is not to be used to affect the visual appearance of the image when displayed. See the comments for MaxSampleValue.

Default is 0.

SubfileType

Tag	= 255 (FF.H)
Type	= SHORT
N	= 1

A general indication of the kind of data that is contained in this subfile. Currently defined values are:

1 = full resolution image data—ImageWidth, ImageLength, and StripOffsets are required fields; and

2 = reduced resolution image data—ImageWidth, ImageLength, and StripOffsets are required fields. It is further assumed that a reduced resolution image is a reduced version of the entire extent of the corresponding full resolution data.

3 = single page of a multi-page image (see the PageNumber tag description).

Note that several image types can be found in a single TIFF file, with each subfile described by its own IFD.

No default.

Continued use of this field is not recommended. Writers should instead use the new and more general NewSubfileType field.

Orientation

Tag	= 274 (112.H)
Type	= SHORT
N	= 1

1 = The 0th row represents the visual top of the image, and the 0th column represents the visual left hand side.

2 = The 0th row represents the visual top of the image, and the 0th column represents the visual right hand side.

3 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual right hand side.

4 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual left hand side.

5 = The 0th row represents the visual left hand side of the image, and the 0th column represents the visual top.

6 = The 0th row represents the visual right hand side of the image, and the 0th column represents the visual top.

7 = The 0th row represents the visual right hand side of the image, and the 0th column represents the visual bottom.

8 = The 0th row represents the visual left hand side of the image, and the 0th column represents the visual bottom.

Default is 1.

This field is recommended for private (non-interchange) use only. Most images are scanned and stored in the Orientation=1 format, and most TIFF readers can only handle this case.

Threshholding

Tag	= 263 (107.H)
Type	= SHORT
N	= 1

1 = a bilevel “line art” scan. BitsPerSample must be 1.

2 = a “dithered” scan, usually of continuous tone data such as photographs. BitsPerSample must be 1.

3 = Error Diffused.

Default is Threshholding = 1. See also CellWidth, CellLength.

Section 10: PackBits Compression

Abstract

This document describes a simple compression scheme for bilevel scanned and paint type files.

Motivation

The TIFF specification defines a number of compression schemes. Compression type 1 is really no compression, other than basic pixel packing. Compression type 2, based on CCITT 1D compression, is powerful, but not trivial to implement. LZW compression is typically quite good at compression most bilevel images, as well as many deeper images such as palette color and grayscale images, but is also not trivial to implement. PackBits is a simple but often effective alternative.

Description

Several good schemes were already in use in various settings. We somewhat arbitrarily picked the Apple Macintosh PackBits scheme. It is byte oriented, so there is no problem with word alignment. And it has a good worst case behavior (at most 1 extra byte for every 128 input bytes). For Macintosh users, there are toolbox utilities PackBits and UnPackBits that will do the work for you, but it is easy to implement your own routines.

A pseudo code fragment to unpack might look like this:

```
Loop until you get the number of unpacked bytes you are expecting:
    Read the next source byte into n.
    If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
    Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
    Else if n is -128, noop.
Endloop
```

In the inverse routine, it’s best to encode a 2-byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it’s best to merge the three into one literal run. Always encode 3-byte repeats as replicate runs.

So that’s the algorithm. Here are some other rules:

- Each row must be packed separately. Do not compress across row boundaries.
- The number of uncompressed bytes per row is defined to be (ImageWidth + 7) / 8. If the uncompressed bitmap is required to have an even number of bytes per row, decompress into word-aligned buffers.
- If a run is larger than 128 bytes, simply encode the remainder of the run as one or more additional replicate runs.

When PackBits data is uncompressed, the result should be interpreted as per compression type 1 (no compression).

Section 11: Modified Huffman Compression

Abstract

This document describes a method for compressing bilevel data that is based on the CCITT Group 3 1D facsimile compression scheme.

References

1. “Standardization of Group 3 facsimile apparatus for document transmission,” Recommendation T.4, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 16 through 31.
2. “Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus,” Recommendation T.6, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 40 through 48.

We do not believe that these documents are necessary in order to implement Compression=2. We have included (verbatim in most places) all the pertinent information in this section. However, if you wish to order the documents, you can write to ANSI, Attention: Sales, 1430 Broadway, New York, N.Y., 10018. Ask for the publication listed above—it contains both Recommendation T.4 and T.6.

Relationship to the CCITT Specifications

The CCITT Group 3 and Group 4 specifications describe communications protocols for a particular class of devices. They are not by themselves sufficient to describe a disk data format. Fortunately, however, the CCITT coding schemes can be readily adapted to this different environment. The following is one such adaptation. Most of the language is copied directly from the CCITT specifications.

Coding Scheme

A line (row) of data is composed of a series of variable length code words. Each code word represents a run length of either all white or all black. (Actually, more than one code word may be required to code a given run, in a manner described below.) White runs and black runs alternate.

In order to ensure that the receiver (decompressor) maintains color synchronization, all data lines will begin with a white run length code word set. If the actual scan line begins with a black run, a white run length of zero will be sent (written). Black or white run lengths are defined by the code words in Tables 1 and 2. The code words are of two types: Terminating code words and Make-up code words. Each run length is represented by zero or more Make-up code words followed by exactly one Terminating code word.

Run lengths in the range of 0 to 63 pels (pixels) are encoded with their appropriate Terminating code word. Note that there is a different list of code words for black and white run lengths.

Run lengths in the range of 64 to 2623 (2560+63) pels are encoded first by the Make-up code word representing the run length that is nearest to, not longer than, that required. This is then followed by the Terminating code word representing the difference between the required run length and the run length represented by the Make-up code.

Run lengths in the range of lengths longer than or equal to 2624 pels are coded first by the Make-up code of 2560. If the remaining part of the run (after the first Make-up code of 2560) is 2560 pels or greater, additional Make-up code(s) of 2560 are issued until the remaining part of the run becomes less than 2560 pels. Then the remaining part of the run is encoded by Terminating code or by Make-up code plus Terminating code, according to the range mentioned above.

It is considered an unrecoverable error if the sum of the run lengths for a line does not equal the value of the ImageWidth field.

New rows always begin on the next available byte boundary.

No EOL code words are used. No fill bits are used, except for the ignored bits at the end of the last byte of a row. RTC is not used.

By convention, the normal PhotometricInterpretation for Compression=2 is 0 (WhiteIsZero). A PhotometricInterpretation value of 1 (BlackIsZero) shall have the effect of reversing black and white in the image.

Table 1/T.4 Terminating codes

White run length	Code word	Black run length	Code word
0	00110101	0	0000110111
1	000111	1	010
2	0111	2	11
3	1000	3	10
4	1011	4	011
5	1100	5	0011
6	1110	6	0010
7	1111	7	00011
8	10011	8	000101
9	10100	9	000100
10	00111	10	0000100
11	01000	11	0000101
12	001000	12	0000111
13	000011	13	00000100
14	110100	14	00000111
15	110101	15	000011000
16	101010	16	0000010111
17	101011	17	0000011000
18	0100111	18	0000001000
19	0001100	19	00001100111
20	0001000	20	00001101000
21	0010111	21	00001101100
22	0000011	22	00000110111
23	0000100	23	00000101000
24	0101000	24	00000010111
25	0101011	25	00000011000
26	0010011	26	000011001010
27	0100100	27	000011001011
28	0011000	28	000011001100
29	00000010	29	000011001101
30	00000011	30	000001101000
31	00011010	31	000001101001
32	00011011	32	000001101010
33	00010010	33	000001101011
34	00010011	34	000011010010
35	00010100	35	000011010011
36	00010101	36	000011010100
37	00010110	37	000011010101
38	00010111	38	000011010110
39	00101000	39	000011010111
40	00101001	40	000001101100
41	00101010	41	000001101101
42	00101011	42	000011011010
43	00101100	43	000011011011
44	00101101	44	000001010100
45	00000100	45	000001010101
46	00000101	46	000001010110
47	00001010	47	000001010111

48	00001011	48	000001100100
49	01010010	49	000001100101
50	01010011	50	000001010010
51	01010100	51	000001010011
52	01010101	52	000000100100
53	00100100	53	000000110111
54	00100101	54	000000111000
55	01011000	55	000000100111
56	01011001	56	000000101000
57	01011010	57	000001011000
58	01011011	58	000001011001
59	01001010	59	000000101011
60	01001011	60	000000101100
61	00110010	61	000001011010
62	00110011	62	000001100110
63	00110100	63	000001100111

Table 2/T.4 Make-up codes

White run length	Code word	Black run length	Code word
64	11011	64	0000001111
128	10010	128	000011001000
192	010111	192	000011001001
256	0110111	256	000001011011
320	00110110	320	000000110011
384	00110111	384	000000110100
448	01100100	448	000000110101
512	01100101	512	0000001101100
576	01101000	576	0000001101101
640	01100111	640	0000001001010
704	011001100	704	0000001001011
768	011001101	768	0000001001100
832	011010010	832	0000001001101
896	011010011	896	0000001110010
960	011010100	960	0000001110011
1024	011010101	1024	0000001110100
1088	011010110	1088	0000001110101
1152	011010111	1152	0000001110110
1216	011011000	1216	0000001110111
1280	011011001	1280	0000001010010
1344	011011010	1344	0000001010011
1408	011011011	1408	0000001010100
1472	010011000	1472	0000001010101
1536	010011001	1536	0000001011010
1600	010011010	1600	0000001011011
1664	011000	1664	0000001100100
1728	010011011	1728	0000001100101
EOL	000000000001	EOL	000000000001

Additional make-up codes

White and Black run length	Make-up code word
1792	00000001000
1856	00000001100
1920	00000001101
1984	000000010010
2048	000000010011
2112	000000010100
2176	000000010101
2240	000000010110
2304	000000010111
2368	000000011100
2432	000000011101
2496	000000011110
2560	000000011111

Section 12: TIFF Administration

For Further Information

On-line files: A version of the TIFF specification in PostScript format can be found on CompuServe ("Go Aldus") and on AppleLink (Aldus Developers Icon). Sample files from the TIFF Developer Kit (see below) can also be found at these locations.

By contacting the Aldus Developers Desk (see contact information on the first page), you can:

- order a TIFF Developer Kit, containing a hard copy of the specification, sample TIFF-parsing and decompression code, and sample TIFF files. All files are in both DOS and Apple Macintosh formats. There is a fee.
- become a member of the Aldus Developers Association
- ask questions relating to the use of TIFF in Aldus products, via the CompuServe or AppleLink forums. By using one of these forums, an Aldus representative or another developer may be able to assist you. Please include your e-mail and fax addresses.
- reserve private tags or values (see below).

Because of the tremendous growth in the usage of TIFF, Aldus is unfortunately not able to provide a general consulting service for TIFF implementors. TIFF developers are encouraged to study sample TIFF files and sample source code, read TIFF documentation thoroughly, and work with other developers of other products that are important to you in terms of image data interchange. A number of other vendors are providing various TIFF services, especially in relationship to their own products. Contact the appropriate product manager or developer support service group.

We do, however, do our best to answer questions relating to the use of TIFF in Aldus products, especially if you are a member of the Aldus Developers Association.

If you are an experienced TIFF developer that is interested in contract programming for other developers, please let us know, so that we can give your name to others that may need your services.

Private Fields and Values

An organization may wish to store information that is meaningful to only that organization in a TIFF file. Tags numbered 32768 or higher are reserved for that purpose. Upon request, the administrator (the Aldus Developers Desk) will allocate and register a block of private tags for an organization, to avoid possible conflicts with other organizations. Tags are normally allocated in blocks of five or less. You do not need to tell the TIFF administrator or anyone else what you are going to use them for.

Private enumerated values can be accommodated in a similar fashion. For example, you may wish to experiment with a new compression scheme within TIFF. Enumeration constants numbered 32768 or higher are reserved for private usage. Upon request, the administrator will allocate and register one or more enumerated values for a particular field (Compression, in our example), to avoid possible conflicts.

Tags and values which are allocated in the private number range are not prohibited from being included in a future revision of this specification. Several such instances can be found in the TIFF specification.

Do not choose your own tag numbers. If you do, it could cause serious compatibility problems some day.

If you need more than 5 or 10 tags, we suggest that you reserve a single private tag, define it as a LONG, and use its value as a pointer (offset) to a private IFD or other data structure of your choosing. Within that IFD, you can use whatever tags you want, since no one else will even know that it is an IFD unless you tell them. This gives you some 65,000 private tags.

Submitting a Proposal

Any person or group that wishes to propose a change or addition to the TIFF specification should prepare a proposal that includes the following information:

- Name of the person or group making the request, and your affiliation.
- The reason for the request.
- A list of changes exactly as you propose that they appear in the specification. Use inserts, callouts, or other obvious editorial techniques to indicate areas of change, and number each change.
- Discussion of the potential impact on the installed base.
- A list of contacts outside your company who may support your position. Include their affiliation.

Please send your proposal to Internet address: tiff-input@aldus.com. (From AppleLink, you can send to: tiff-input@aldus.com@internet#. From CompuServe, you can send to: >INTERNET:tiff-input@aldus.com.) Do not send TIFF implementation questions to this address; see above for Aldus Developers Desk TIFF support policies.

The TIFF Advisory Committee

The TIFF Advisory Committee is a working group of TIFF experts from a number of hardware and software manufacturers. It was formed in the spring of 1991, for the purpose of providing a forum to debate and refine proposals for the TIFF 6.0 release of the specification. It is not yet clear whether this will be an ongoing group, or whether it will go into a period of hibernation until momentum builds for another major release of the TIFF specification.

If you are a TIFF expert and are interested in spending significant time and effort to work on this committee, contact the Aldus Developers Desk for further information. For the TIFF 6.0 release, the group met every two or three months, typically somewhere on the west coast of the U.S. Accessibility via e-mail (typically Internet or AppleLink) is a requirement for membership, since that has proven to be an invaluable means for getting work done between meetings.

Other TIFF Extensions

The Aldus TIFF sections on CompuServe and AppleLink will contain proposed extensions from Aldus and other companies that are not yet approved by the TIFF Advisory Committee.

Many of these proposals will never be approved or even considered by the TIFF Advisory Committee, especially if they represent specialized uses of TIFF that do not fall within the domain of publishing or general graphics/picture interchange. Use them at your own risk; it is unlikely that these features will be widely supported. And if you do write files that incorporate these extensions, be sure to either not call them TIFF files, or in some other way mark them so that they will not easily be confused with mainstream TIFF files.

Aldus will provide a place on Compuserve and Applelink for storing such documents. Contact the Aldus Developers Desk for instructions. We recommend that all submissions be in the form of either simple text or in a portable PostScript form that can be downloaded to any PostScript printer in any computing environment.

If a non-Aldus contact name is listed, please use that contact rather than Aldus for submitting requests for future enhancements to that extension.

Part 2: TIFF Extensions

Part 2 contains extensions to Baseline TIFF, as well as several pieces of auxiliary documentation that may be useful to developers.

The features described in this part were either contained in earlier versions of the specification, or have been approved by the TIFF Advisory Committee.

TIFF Extensions are TIFF features that may not be supported by all TIFF readers. TIFF creators who use these features will have to work closely with TIFF readers in their part of the industry to ensure successful interchange.

Some of the extensions were written by Aldus, and some were written by other developers. If a non-Aldus contact name is listed, please use that contact rather than Aldus for submitting requests for future enhancements to that extension.

Section 13: Tags vs Fixed Fields

If you like reading about file format philosophy, this section is for you. It contains absolutely nothing that you need to know in order to read or write TIFF files.

A file format is defined by both form (structure) and content. The content of TIFF consists of definitions of individual fields. It is therefore the content that we are ultimately interested in. The structure merely tells us how to find the fields. Yet the structure deserves serious consideration for a number of reasons that are not at all obvious at first glance. Since the structure described herein departs significantly from several other approaches, it may be useful to discuss the rationale behind it.

The simplest, most straightforward structure for something like an image file is a positional format. In a positional scheme, the location of the data defines what the data means. For example, the field for “number of rows” might begin at byte offset 30 in the image file.

This approach is simple and easy to implement and is perfect for static environments. But if a significant amount of ongoing change must be accommodated, subtle problems begin to appear. For example, suppose that a field must be superseded by a new, more general field. You could bump a version number to flag the change. Then new software has no problem doing something sensible with old data, and all old software will reject the new data, even software that didn’t care about the old field. This may seem like no more than a minor annoyance at first glance, but causing old software to break more often than it would really need to can be very costly and, inevitably, causes much gnashing of teeth among customers.

Furthermore, it can be avoided. One approach is to store a “valid” flag bit for each field. Now you don’t have to bump the version number, as long as you can put the new field somewhere that doesn’t disturb any of the old fields. Old software that didn’t care about that old field anyway can continue to function. (Old software that did care will of course have to give up, but this is an unavoidable price to be paid for the sake of progress, barring total omniscience.)

Another problem that crops up frequently is that certain fields are likely to make sense only if other fields have certain values. This is not such a serious problem in practice; it just makes things more confusing. Nevertheless, we note that the “valid” flag bits described in the previous paragraph can help to clarify the situation.

Field-dumping programs can be very helpful for diagnostic purposes. A desirable characteristic of such a program is that it doesn’t have to know much about what it is dumping. In particular, it would be nice if the program could dump ASCII data in ASCII format, integer data in integer format, and so on, without having to teach the program about new fields all the time. So maybe we should add a “data type” component to our fields, plus information on how long the field is, so that our dump program can walk through the fields without knowing what the fields “mean.”

But note that if we add one more component to each field, namely a tag that tells what the field means, we can dispense with the “valid” flag bits, and we can also avoid wasting space on the non-valid fields in the file. Simple image creation applications can write out several fields and be done.

We have now derived the essentials of a tag-based image file format.

Finally, a caveat. A tag based scheme cannot guarantee painless growth. But is does provide a useful tool to assist in the process.

Section 14: New General Purpose Tags

ExtraSamples

Tag	= 338 (152.H)
Type	= SHORT
N	= m

This tag specifies that each pixel has *m* extra components, whose interpretation is defined by the one of the values listed below. When this tag is used, the SamplesPerPixel tag has a value that is greater than the PhotometricInterpretation would suggest.

For example, full-color RGB data normally has SamplesPerPixel=3. If SamplesPerPixel is greater than 3, then the ExtraSamples tag describes the meaning of the extra samples. E.g., if SamplesPerPixel is 5, then ExtraSamples will contain 2 values, one for each extra sample.

ExtraSamples is typically used when including non-color information, such as opacity, in an image.

The following table gives the possible values for each item in the tag's value:

Value	Description
0	Undefined data
1	Associated Alpha data (with pre-multiplied color)
2	Unassociated Alpha data

Associated Alpha data is opacity information; it is fully described in ALPHA data section. Unassociated Alpha data is transparency information that logically exists independent of an image; it is commonly called (collectively) a soft matte. Note that including both Unassociated and Associated Alpha is undefined because Associated Alpha specifies that color components are pre-multiplied by the alpha component, while Unassociated Alpha specifies the opposite.

By convention, extra components that are present must be stored as the “last components” in each pixel. For example, if SamplesPerPixel is 4 and there is 1 extra component, then it is located in the last component location (SamplesPerPixel-1) in each pixel.

Comments

Components designated as “extra” are just like other components in a pixel. In particular, the size of such components is defined by the value of the BitsPerSample tag.

Note that, with the introduction of this tag, TIFF readers must not assume a particular SamplesPerPixel value based on the value of the PhotometricInterpretation tag. For example, if it is an RGB file, SamplesPerPixel may be greater than 3.

Section 15: Document Storage and Retrieval

These fields may be useful for document storage and retrieval applications. They will very likely be ignored by other applications.

DocumentName

Tag	= 269 (10D.H)
Type	= ASCII

The name of the document from which this image was scanned.

See also PageName.

PageName

Tag	= 285 (11D.H)
Type	= ASCII

The name of the page from which this image was scanned.

See also DocumentName.

No default.

PageNumber

Tag	= 297 (129.H)
Type	= SHORT
N	= 2

This tag is used to specify page numbers of a multiple page (e.g. facsimile) document. Two SHORT values are specified. The first value is the page number; the second value is the total number of pages in the document.

Note that pages need not appear in numerical order. The first page is 0 (zero).

No default.

XPosition

Tag	= 286 (11E.H)
Type	= RATIONAL
N	= 1

The X offset of the left side of the image, with respect to the left side of the page, in ResolutionUnits.

No default. See also YPosition.

YPosition

Tag	= 287 (11F.H)
Type	= RATIONAL
N	= 1

The Y offset of the top of the image, with respect to the top of the page, in

ResolutionUnits. In the TIFF coordinate scheme, the positive Y direction is down, so that YPosition is always positive.

No default. See also XPosition.

Section 16: LZW Compression

Abstract

This document describes an adaptive compression scheme for raster images.

Restrictions

We have been informed by UNISYS Corporation that UNISYS believes that one or more of their patents covers any and all usages of the LZW (Ziv-Lempel) compression technique, including the use of LZW compressed data within a TIFF file. At this time, UNISYS has chosen not to grant royalty-free licensing for developers who write software that reads and writes TIFF files. The same is true for any other image file format that incorporates LZW compression.

The Aldus Corporation was unaware of any potential patent infringement with respect to use of LZW compression. The LZW research papers at our disposal made no mention of patent restrictions, and LZW compression was in widespread use long before its inclusion in TIFF, both in commercial and public domain software packages.

We therefore do not encourage the use of LZW within TIFF files. The financial and administrative burdens resulting from licensing would pose an undue burden especially for smaller developers.

We regret any hardhip that this issue will cause for the desktop computer developer and user communities.

If you wish to discuss the issue further with UNISYS or discuss licensing arrangements, the current contact at UNISYS is Rob Pressman, (215) 986-4111.

The main UNISYS LZW patent is U.S. Patent #4,558,302.

Reference

Terry A. Welch, “A Technique for High Performance Data Compression”, IEEE Computer, vol. 17 no. 6 (June 1984). Describes the basic Lempel-Ziv & Welch (LZW) algorithm. The author’s goal in the article is to describe a hardware-based compressor that could be built into a disk controller or database engine, and used on all types of data. There is no specific discussion of raster images. We intend to give sufficient information in this Section so that the article is not required reading.

Requirements

A compression scheme with the following characteristics should work well in a desktop publishing environment:

- Must work well for images of any bit depth, including images deeper than 8 bits per component.

- Must be effective: an average compression ratio of at least 2:1 or better. And it must have a reasonable worst-case behavior, in case something really strange is thrown at it.
- Should not depend on small variations between pixels. Palette color images tend to contain abrupt changes in index values, due to common patterning and dithering techniques. These abrupt changes do tend to be repetitive, however, and the scheme should make use of this fact.
- For images generated by paint programs, the scheme should not depend on a particular pattern width. 8x8 pixel patterns are common now, but we should not assume that this situation will not change.
- Must be fast. It should not take more than 5 seconds to decompress a 100K byte grayscale image on a 68020- or 386-based computer. Compression can be slower, but probably not by more than a factor of 2 or 3.
- The level of implementation complexity must be reasonable. We would like something that can be implemented in no more than a couple of weeks by a competent software engineer with some experience in image processing. The compiled code for compression and decompression combined should be no more than about 10K.
- Does not require floating point software or hardware.

The following sections describe an algorithm based on the “LZW” (Lempel-Ziv & Welch) technique that meets the above requirements. In addition meeting our requirements, LZW has the following characteristics:

- LZW is fully reversible. All information is preserved. But if noise or information is removed from an image, perhaps by smoothing or zeroing some low-order bitplanes, LZW compresses images to a smaller size. Thus, 5-bit, 6-bit, or 7-bit data masquerading as 8-bit data compresses better than true 8-bit data. Smooth images also compress better than noisy images, and simple images compress better than complex images.

° On a 68082- or 386-based computer, LZW software can be written to compress at between 30K and 80K bytes per second, depending on image characteristics. LZW decompression speeds are typically about 50K bytes per second.

° LZW works well on bilevel images, too. It always beats PackBits, and generally ties CCITT 1D (Modified Huffman) compression, on our test images. Tying CCITT 1D is impressive in that LZW seems to be considerably faster than CCITT 1D, at least in our implementation.

° Our implementation is written in C, and compiles to about 2K bytes of object code each for the compressor and decompressor.

° One of the nice things about LZW is that it is used quite widely in other applications such as archival programs, and is therefore more of a known quantity.

The Algorithm

Each strip is compressed independently. We strongly recommend that RowsPerStrip be chosen such that each strip contains about 8K bytes before compression. We want to keep the strips small enough so that the compressed and uncompressed versions of the strip can be kept entirely in memory even on small machines, but large enough to maintain nearly optimal compression ratios.

The LZW algorithm is based on a translation table, or string table, that maps strings of input characters into codes. The TIFF implementation uses variable-length codes, with a maximum code length of 12 bits. This string table is different for every strip, and, remarkably, does not need to be kept around for the decompressor. The trick is to make the decompressor automatically build the same table as is built when compressing the data. We use a C-like pseudocode to describe the coding scheme:

```
InitializeStringTable();
WriteCode(ClearCode);
Ω = the empty string;
for each character in the strip {
    K = GetNextCharacter();
    if Ω+K is in the string table {
        Ω = Ω+K;     /* string concatenation */
    } else {
        WriteCode (CodeFromString(Ω));
        AddTableEntry(Ω+K);
        Ω = K;
    }
} /* end of for loop */
WriteCode (CodeFromString(Ω));
WriteCode (EndOfInformation);
```

That’s it. The scheme is simple, although it is fairly challenging to implement efficiently. But we need a few explanations before we go on to decompression.

The “characters” that make up the LZW strings are bytes containing TIFF uncompressed (Compression=1) image data, in our implementation. For example, if BitsPerSample is 4, each 8-bit LZW character will contain two 4-bit pixels. If BitsPerSample is 16, each 16-bit pixel will span two 8-bit LZW characters.

(It is also possible to implement a version of LZW where the LZW character depth equals BitsPerSample, as was described by Draft 2 of Revision 5.0. But there is a major problem with this approach. If BitsPerSample is greater than 11, we can not use 12-bit-maximum codes, so that the resulting LZW table is unacceptably large. Fortunately, due to the adaptive nature of LZW, we do not pay a significant compression ratio penalty for combining several pixels into one byte before compressing. For example, our 4-bit sample images compressed about 3 percent worse, and our 1-bit images compressed about 5 percent better. And it is easier to write an LZW compressor that always uses the same character depth than it is to write one which can handle varying depths.)

We can now describe some of the routine and variable references in our pseudocode:

InitializeStringTable() initializes the string table to contain all possible single-character strings. There are 256 of them, numbered 0 through 255, since our characters are bytes.

WriteCode() writes a code to the output stream. The first code written is a Clear code, which is defined to be code #256.

Ω is our “prefix string.”

GetNextCharacter() retrieves the next character value from the input stream. This will be number between 0 and 255, since our characters are bytes.

The “+” signs indicate string concatenation.

AddTableEntry() adds a table entry. (InitializeStringTable() has already put 256 entries in our table. Each entry consists of a single-character string, and its associated code value, which is, in our application, identical to the character itself. That is, the 0th entry in our table consists of the string <0>, with corresponding code value of <0>, the 1st entry in the table consists of the string <1>, with corresponding code value of <1>, ..., and the 255th entry in our table consists of the string <255>, with corresponding code value of <255>.) So the first entry that we add to our string table will be at position 256, right? Well, not quite, since we will reserve code #256 for a special “Clear” code, and code #257 for a special “EndOfInformation” code that we will write out at the end of the strip. So the first multiple-character entry added to the string table will be at position 258.

Let’s try an example. Suppose we have input data that looks like:

Pixel 0: <7>

Pixel 1: <7>

Pixel 2: <7>

Pixel 3: <8>

Pixel 4: <8>

Pixel 5: <7>

Pixel 6: <7>

Pixel 7: <6>

Pixel 8: <6>

First, we read Pixel 0 into K. ΩK is then simply <7>, since Ω is the empty string at this point. Is the string <7> already in the string table? Of course, since all single character strings were put in the table by InitializeStringTable(). So set Ω equal to <7>, and go to the top of the loop.

Read Pixel 1 into K. Does ΩK (<7><7>) exist in the string table? No, so we get to do some real work. We write the code associated with Ω to output (write <7> to output), and add ΩK (<7><7>) to the table as entry 258. Store K (<7>) into Ω. Note that although we have added the string consisting of Pixel 0 and Pixel 1 to the table, we “re-use” Pixel 1 as the beginning of the next string.

Back at the top of the loop. We read Pixel 2 into K. Does ΩK (<7><7>) exist in the string table? Yes, the entry we just added, entry 258, contains exactly <7><7>. So we just add K onto the end of Ω, so that Ω is now <7><7>.

Back at the top of the loop. We read Pixel 3 into K. Does ΩK (<7><7><8>) exist in the string table? No, so write the code associated with Ω (<258>) to output, and add ΩK to the table as entry 259. Store K (<8>) into Ω.

Back at the top of the loop. We read Pixel 4 into K. Does ΩK (<8><8>) exist in the string table? No, so write the code associated with Ω (<8>) to output, and add ΩK to the table as entry 260. Store K (<8>) into Ω.

Continuing, we get the following results:

After reading:	We write to output:	And add table entry:
Pixel 0		
Pixel 1	<7>	258: <7><7>
Pixel 2		
Pixel 3	<258>	259: <7><7><8>
Pixel 4	<8>	260: <8><8>
Pixel 5	<8>	261: <8><7>
Pixel 6		
Pixel 7	<258>	262: <7><7><6>
Pixel 8	<6>	263: <6><6>

WriteCode() also requires some explanation. The output code stream, <7><258><8><8><258><6>... in our example, should be written using as few bits as possible. When we are just starting out, we can use 9-bit codes, since our new string table entries are greater than 255 but less than 512. *After adding table entry 511, switch to 10-bit codes (i.e., entry 512 should be a 10-bit code.) Likewise, switch to 11-bit codes after table entry 1023, and 12-bit codes after table entry 2047.* We will somewhat arbitrarily limit ourselves to 12-bit codes, so that our table can have at most 4096 entries. If we push it any farther, tables tend to get too large.

Whenever you add a code to the output stream, it "counts" toward the decision about bumping the code bit length. This is important when writing the last code word before an EOI code or ClearCode, to avoid code length errors.

What happens if we run out of room in our string table? This is where the afore-mentioned Clear code comes in. As soon as we use entry 4094, we write out a (12-bit) Clear code. (If we wait any longer to write the Clear code, the decompressor might try to interpret the Clear code as a 13-bit code.) At this point, the compressor re-initializes the string table and starts writing out 9-bit codes again.

Note that whenever you write a code and add a table entry, Ω is not left empty. It contains exactly one character. Be careful not to lose it when you write an end-of-table Clear code. You can either write it out as a 12-bit code before writing the Clear code, in which case you will want to do it right after adding table entry 4093, or after the clear code as a 9-bit code. Decompression gives the same result in either case.

To make things a little simpler for the decompressor, we will require that each strip begins with a Clear code, and ends with an EndOfInformation code.

Every LZW-compressed strip must begin on a byte boundary. It need not begin on a word boundary. LZW compression codes are stored into bytes in high-to-low-order fashion, i.e., FillOrder is assumed to be 1. The compressed codes are written as bytes, not words, so that the compressed data will be identical regardless of whether it is an 'II' or 'MM' file.

Note that the LZW string table is a continuously updated history of the strings that have been encountered in the data. It thus reflects the characteristics of the data, providing a high degree of adaptability.

LZW Decoding

The procedure for decompression is a little more complicated, but still not too bad:

```
while ((Code = GetNextCode()) != EoiCode) {
    if (Code == ClearCode) {
        InitializeTable();
        Code = GetNextCode();
        if (Code == EoiCode)
          break;
        WriteString(StringFromCode(Code));
        OldCode = Code;
    }  /* end of ClearCode case */

    else {
        if (IsInTable(Code)) {
            WriteString(StringFromCode(Code));
            AddStringToTable(StringFromCode(OldCode
)+FirstChar(StringFromCode(Code)));
            OldCode = Code;
        } else {
            OutString = StringFromCode(OldCode) +
FirstChar(StringFromCode(OldCode));
            WriteString(OutString);
            AddStringToTable(OutString);
            OldCode = Code;
        }
    } /* end of not-ClearCode case */
} /* end of while loop */
```

The function GetNextCode() retrieves the next code from the LZW-coded data. It must keep track of bit boundaries. It knows that the first code that it gets will be a 9-bit code. We add a table entry each time we get a code, so GetNextCode() must switch over to 10-bit codes as soon as string *#510* is stored into the table. *Similarly, the switch is made to 11-bit codes after #1022, and to 12-bit codes after #2046.*

The function StringFromCode() gets the string associated with a particular code from the string table.

The function AddStringToTable() adds a string to the string table. The “+” sign joining the two parts of the argument to AddStringToTable indicate string concatenation.

StringFromCode() looks up the string associated with a given code.

WriteString() adds a string to the output stream.

When SamplesPerPixel Is Greater Than 1

We have so far described the compression scheme as if SamplesPerPixel were always 1, as will be the case with palette color and grayscale images. But what do we do with RGB image data?

Tests on our sample images indicate that the LZW compression ratio is nearly identical regardless of whether PlanarConfiguration=1 or PlanarConfiguration=2, for RGB images. So use whichever configuration you prefer, and simply compress the bytes in the strip.

It is worth cautioning that compression ratios on our test RGB images were disappointing low: somewhere between 1.1 to 1 and 1.5 to 1, depending on the image. Vendors are urged to do what they can to remove as much noise from their images as possible. Preliminary tests indicate that significantly better compression ratios are possible with less noisy images. Even something as simple as zeroing out one or two least-significant bitplanes may be quite effective, with little or no perceptible image degradation.

Implementation

The exact structure of the string table and the method used to determine if a string is already in the table are probably the most significant design decisions in the implementation of a LZW compressor and decompressor. Hashing has been suggested as a useful technique for the compressor. We have chosen a tree based approach, with good results. The decompressor is actually more straightforward, as well as faster, since no search is involved—strings can be accessed directly by code value.

Performance

Many people do not realize that the performance of any compression scheme depends greatly on the type of data to which it is applied. A scheme that works well on one data set may do poorly on the next.

But since we do not want to burden the world with too many compression schemes, an adaptive scheme such as LZW that performs quite well on a wide range of images is very desirable. LZW may not always give optimal compression ratios, but its adaptive nature and relative simplicity seem to make it a good choice.

Experiments thus far indicate that we can expect compression ratios of between 1.5 and 3.0 to 1 from LZW, with no loss of data, on continuous tone grayscale scanned images. If we zero the least significant one or two bitplanes of 8-bit data, higher ratios can be achieved. These bitplanes often consist chiefly of noise, in which case little or no loss in image quality will be perceived. Palette color images created in a paint program generally compress much better than continuous tone scanned images, since paint images tend to be more repetitive. It is not unusual to achieve compression ratios of 10 to 1 or better when using LZW on palette color paint images.

By way of comparison, PackBits, used in TIFF for black and white bilevel images, does not do well on color paint images, much less continuous tone grayscale and color images. 1.2 to 1 seemed to be about average for 4-bit images, and 8-bit images are worse.

It has been suggested that the CCITT 1D scheme could be used for continuous tone images, by compressing each bitplane separately. No doubt some compression could be achieved, but it seems unlikely that a scheme based on a fixed table that is optimized for short black runs separated by longer white runs would be a very good choice on any of the bitplanes. It would do quite well on the high-order bitplanes (but so would a simpler scheme like PackBits), and would do quite poorly on the low-order bitplanes. We believe that the compression ratios would generally not be very impressive, and the process would in addition be quite slow. Splitting the pixels into bitplanes and putting them back together is somewhat expensive, and the coding is also fairly slow when implemented in software.

Another approach that has been suggested uses a 2D differencing step following by coding the differences using a fixed table of variable-length codes. This type of scheme works quite well on many 8-bit grayscale images, and is probably simpler to implement than LZW. But it has a number of disadvantages when used on a wide variety of images. First, it is not adaptive. This makes a big difference when compressing data such as 8-bit images that have been “sharpened” using one of the standard techniques. Such images tend to get larger instead of smaller when compressed. Another disadvantage of these schemes is that they do not do well with a wide range of bit depths. The built-in code table has to be optimized for a particular bit depth in order to be effective.

Finally, we should mention “lossy” compression schemes. Extensive research has been done in the area of lossy, or non-information-preserving image compression. These techniques generally yield much higher compression ratios than can be achieved by fully-reversible, information-preserving image compression techniques such as PackBits and LZW. Some disadvantages: many of the lossy techniques are so computationally expensive that hardware assists are required. Others are so complicated that most microcomputer software vendors could not afford either the expense of implementation or the increase in application object code size. Yet others sacrifice enough image quality to make them unsuitable for publishing use.

In spite of these difficulties, we believe that there will one day be a standardized lossy compression scheme for full color images that will be usable for publishing applications on microcomputers. An International Standards Organization group, ISO/IEC/JTC1/SC2/WG8, in cooperation with CCITT Study Group VIII, is hard at work on a scheme that might be appropriate. We expect that a future revision of TIFF will incorporate this scheme once it is finalized, if it turns out to satisfy the needs of desktop publishers and others in the microcomputer community. This will augment, not replace, LZW as an approved TIFF compression scheme. LZW will very likely remain the scheme of choice for Palette color images, and perhaps 4-bit grayscale images, and may well overtake CCITT 1D and PackBits for bilevel images.

Future LZW Extensions

Some images compress better using LZW coding if they are first subjected to a process wherein each pixel value is replaced by the difference between the pixel and the preceding pixel. Performing this differencing in two dimensions helps some images even more. However, many images do not compress better with this extra preprocessing, and for a significant number of images, the compression ratio is actually worse. We are therefore not making differencing an integral part of the TIFF LZW compression scheme.

But all TIFF readers that read LZW files must pay attention to the Predictor tag. If it is 1, which is the default case, LZW decompression may proceed safely. If it is not 1, and the reader does not recognize the specified prediction scheme, the reader should give up. *See the Differencing Predictor section.*

Acknowledgments

The original LZW reference has already been given. The use of ClearCode as a technique to handle overflow was borrowed from the compression scheme used by the Graphics Interchange Format (GIF), a small-color-paint-image-file format used by CompuServe that also is an adaptation of the LZW technique.

Section 17: Differencing Predictor

We now define a Predictor that greatly improves compression ratios for some images.

Predictor

Tag	= 317 (13D.H)
Type	= SHORT
N	= 1

A predictor is a mathematical operator that is applied to the image data before the encoding scheme is applied. Currently this tag is used only with LZW (Compression=5) encoding, since LZW is probably the only TIFF encoding scheme that benefits significantly from a predictor step. See the LZW Compression section.

The possible values are:

1 = No prediction scheme used before coding.

2 = Horizontal differencing.

Default is 1.

The algorithm

The idea is to make use of the fact that many continuous tone images rarely vary much in pixel value from one pixel to the next. In such images, if we replace the pixel values by differences between consecutive pixels, many of the differences should be 0, plus or minus 1, and so on. This reduces the apparent information content, and thus allows LZW to encode the data more compactly.

Assuming 8-bit grayscale pixels for the moment, a basic C implementation might look something like this:

```
char        image[ ][ ];
int         row, col;

/* take horizontal differences:
 */
for (row = 0; row < nrows; row++)
  for (col = ncols - 1; col >= 1; col--)
            image[row][col] -= image[row][col-1];
```

If we don’t have 8-bit components, we need to work a little harder, so that we can make better use of the architecture of most CPUs. Suppose we have 4-bit components, packed two to a byte, in normal TIFF uncompressed (i.e., Compression=1) fashion. In order to find differences, we want to first expand each 4-bit component into an 8-bit byte, so that we have one component per byte, low-order justified. We then perform the above horizontal differencing. Once the differencing has been completed, we then repack the 4-

bit differences two to a byte, in normal TIFF uncompressed fashion.

If the components are greater than 8 bits deep, expanding the components into 16-bit words instead of 8-bit bytes seems like the best way to perform the subtraction on most computers.

Note that we have not lost any data up to this point, nor will we lose any data later on. It might at first seem that our differencing might turn 8-bit components into 9-bit differences, 4-bit components into 5-bit differences, and so on. But it turns out that we can completely ignore the "overflow" bits caused by subtracting a larger number from a smaller number and still reverse the process without error. Normal twos complement arithmetic does just what we want. Try an example by hand if you need more convincing.

Up to this point we have implicitly assumed that we are compressing bilevel or grayscale images. An additional consideration arises in the case of color images.

If PlanarConfiguration is 2, there is no problem. Differencing proceeds the same way as it would for grayscale data.

If PlanarConfiguration is 1, however, things get a little trickier. If we didn't do anything special, we would be subtracting red component values from green component values, green component values from blue component values, and blue component values from red component values, which would not give the LZW coding stage much redundancy to work with. So we will do our horizontal differences with an offset of SamplesPerPixel (3, in the RGB case). In other words, we will subtract red from red, green from green, and blue from blue. The LZW coding stage is identical to the SamplesPerPixel=1 case. We require that BitsPerSample be the same for all 3 components.

Results and guidelines

LZW without differencing works well for 1-bit images, 4-bit grayscale images, and synthetic color images. But natural 24-bit color images and some 8-bit grayscale images do much better with differencing. For example, our 24-bit natural test images hardly compressed at all using "plain" LZW: the average compression ratio was 1.04 to 1. The average compression ratio with horizontal differencing was 1.40 to 1. (A compression ratio of 1.40 to 1 means that if the uncompressed image is 1.40MB in size, the compressed version is 1MB in size.)

Although the combination of LZW coding with horizontal differencing does not result in any loss of data, it may be worthwhile in some situations to give up some information by removing as much noise as possible from the image data before doing the differencing, especially with 8-bit components. The simplest way to get rid of noise is to mask off one or two low-order bits of each 8-bit component. On our 24-bit test images, LZW with horizontal differencing yielded an average compression ratio of 1.4 to 1. When the low-order bit was masked from each component, the compression ratio climbed to 1.8 to 1; the compression ratio was 2.4 to 1 when masking two bits, and 3.4 to 1 when masking three bits. Of course, the more you mask, the more you risk losing useful information along with the noise. We encourage you to experiment to find the best compromise for your device. For some applications it may be useful to let the user make the final decision.

Interestingly, most of our RGB images compressed slightly better using

PlanarConfiguration=1. One might think that compressing the red, green, and blue difference planes separately (PlanarConfiguration=2) might give better compression results than mixing the differences together before compressing (PlanarConfiguration=1), but this does not appear to be the case.

Incidentally, we tried taking both horizontal and vertical differences, but the extra complexity of two-dimensional differencing did not appear to pay off for most of our test images. About one third of the images compressed slightly better with two-dimensional differencing, about one third compressed slightly worse, and the rest were about the same.

Section 18: TIFF Tiles

Introduction

Motivation

This document describes how to organize your image into tiles instead of strips.

For low to medium resolution images, the current TIFF method of breaking the image into strips is adequate. But larger images can be accessed more efficiently—and compression tends to work better—if the image is broken into roughly square tiles instead of horizontally wide but vertically narrow strips.

Relationship to existing tags

When the new tiling tags are used, they replace StripOffsets, StripByteCounts, and RowsPerStrip. Use of tiles will therefore cause older TIFF readers to give up, since they will have no way of knowing where the image data is, nor how it is organized. **Do not** use both strip-oriented and tile-oriented tags in the same TIFF file, in a futile quest for backward compatibility. It’s a nice thought, and could almost work with uncompressed data; but even in this case, the potential for disaster makes it a bad choice.

We do not anticipate that StripOffsets, StripByteCounts, and RowsPerStrip will become obsolete anytime soon, if ever. Tiling is an optional TIFF feature that will likely be used only by some applications that create very large images.

Padding

The tile size is defined by TileWidth and TileLength. All of the tiles in an image are the same size; that is, they have the same pixel dimensions.

Boundary tiles are padded out to the tile boundaries. For example, if the TileWidth is 64 and the ImageWidth is 129, then the image is 3 tiles wide, and 63 pixels of padding must be added to fill out the rightmost column of tiles. The same holds for TileLength and ImageLength. It doesn’t really matter what value is used for padding, since good TIFF readers display only the pixels defined by ImageWidth and ImageLength, and ignore any padded pixels. Some compression schemes work best if the padding is accomplished by replicating the last column and last row, instead of padding with 0’s.

The price for padding the image out to tile boundaries is that some space is wasted. But compression generally shrinks the padded areas to almost nothing. Even if you don’t compress, remember that tiling is intended for large images. Large images have lots of comparatively small tiles, so that the percentage of wasted space will be very small, generally on the order of a few percent or less.

The advantages of padding the image out to tile boundaries are that implementations can be simpler and faster, and that it is more compatible with certain tile-oriented compression schemes, such as the emerging JPEG standard.

Tiles are compressed individually, just like strips. *That is, each row of data in a tile is treated as a separate “scanline” when compressing.* Compression includes any padded areas of the rightmost and bottom tiles, so that all the tiles in an image are the same size when uncompressed.

All of the following fields are required for tiled images:

New Fields

TileWidth

Tag	= 322 (142.H)
Type	= SHORT or LONG
N	= 1

The tile width, in pixels. That is, the number of columns in each tile.

Assuming integer arithmetic, three computed values that are useful in the following field descriptions are:

TILESACROSS = (ImageWidth + TileWidth - 1) / TileWidth

TILESDOWN = (ImageLength + TileLength - 1) / TileLength

TILESPERIMAGE = TILESACROSS * TILESDOWN

These computed values are not TIFF fields; they are simply values which are determined by the ImageWidth, TileWidth, ImageLength, and TileLength fields.

Thus, TileWidth and ImageWidth together tell us the number of tiles that span the width of the image (TILESACROSS). TileLength and ImageLength together tell us the number of tiles that span the length of the image (TILESDOWN).

We recommend choosing TileWidth and TileLength such that the resulting tiles are about 4K to 32K bytes, before compression. This seems to be a reasonable value for most applications and compression schemes.

TileWidth must be a multiple of 16. This restriction improves performance in some graphics environments, and enhances compatibility with certain compression schemes such as the emerging JPEG standard.

Tiles need not be square.

Note that ImageWidth can be less than TileWidth, although this means that either your tiles are too large or that you are using tiling on really small images, neither of which is recommended. The same observation holds for ImageLength and TileLength.

No default. See also TileLength, TileOffsets, TileByteCounts.

TileLength

Tag	= 323 (143.H)
Type	= SHORT or LONG
N	= 1

The tile length (height) in pixels. That is, the number of rows (sometimes described as “scan lines”) in each tile. TileLength must be a multiple of 16, which enhances compatibility with certain compression schemes such as the emerging JPEG standard.

No default. See also TileWidth, TileOffsets, TileByteCounts.

TileOffsets

Tag	= 324 (144.H)

Type = LONG
N = TilesPerImage for PlanarConfiguration = 1
= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the byte offset of that tile, as compressed and stored on disk. The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each tile has a location independent of the locations of other tiles.

The offsets are ordered left-to-right, top-to-bottom. For PlanarConfiguration = 2, the offsets for the first component plane are stored first, followed by all the offsets for the second component plane, and so on.

No default. See also TileWidth, TileLength, TileByteCounts.

TileByteCounts

Tag = 325 (145.H)
Type = SHORT or LONG
N = TilesPerImage for PlanarConfiguration = 1
= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the number of (compressed) bytes in that tile.

See TileOffsets for a description of how the byte counts are ordered.

No default. See also TileWidth, TileLength, TileOffsets.

Section 19: CMYK Images

Motivation

This document describes how to store separated (usually CMYK) image data in a TIFF file.

In a separated image, each pixel consists of N components or components. Each component represents the amount of a particular ink that is to be used to represent the image at that location, typically using a halftoning technique.

For example, in a CMYK image, each pixel consists of 4 components. Each component represents the amount of cyan, magenta, yellow, or black process ink that is to be used to represent the image at that location.

The fields described in this appendix can be used for more than simple 4-color process (CMYK) printing. They can also be used for describing an image made up of more than 4 inks, such as a cyan, magenta, yellow, red, green, blue, and black ink image. Such an image is sometimes called a high-fidelity image, and has the advantage of a wider printed color gamut.

Since separated images are quite device-specific, and restricted to color prepress usage, they should <u>not</u> be used for general image data interchange. Separated images are to be used only for prepress applications in which the imagesetter, paper, ink, and printing press characteristics are known by the creator of the separated image.

Note: there is no single method of converting RGB data to CMYK data and back. In a perfect world, something close to cyan = 255-red, magenta = 255-green, and yellow = 255-blue should work; but characteristics of printing inks and printing presses, economics, and the fact the meaning of RGB itself depends on other parameters combine to spoil this inherent simplicity.

Requirements

In addition to the normal Baseline TIFF requirements, a separated TIFF file must have the following characteristics:

SamplesPerPixel = N. SHORT. The number of inks. (For example, N=4 for CMYK, since we have one component each for cyan, magenta, yellow, and black.

BitsPerSample = 8,8,8,8 (for CMYK). SHORT. For now, only 8-bit components are recommended. The value “8” is repeated SamplesPerPixel times.

PlanarConfiguration = 1 or 2. SHORT.

Compression = 1. SHORT. ‘1’ means no compression (see the Compression tag discussion in the main body of the TIFF specification).

PhotometricInterpretation = 5 (Separated). SHORT. The components represent desired percent dot coverage of each ink, where the larger component values represent a higher percentage of ink dot coverage, and smaller values represent less ink.

And we have some new tags, all of which are optional. But note that InkSet has a default of 1, which is also the only currently defined value. The reader must check to see that the value for InkSet is indeed what he is expecting, since we may add other types of separated images some day.

InkSet

Tag	= 332 (14C.H)
Type	= SHORT
N	= 1

The set of inks that are used in a separated (PhotometricInterpretation=5) image.

1 = CMYK. For each pixel, each of the components represents the quantity for the respective ink. The order of the components is cyan, magenta, yellow, black.

2 = not CMYK. See the InkNames tag for a description of the inks to be used.

Default is 1 (CMYK).

InkNames

Tag	= 333 (14D.H)
Type	= ASCII
N	= total number of characters in all the ink name strings, including the zeros

The name of each ink that is used in a separated (PhotometricInterpretation=5) image, written as a list of concatenated, zero-terminated ASCII strings. The number of strings must be equal to SamplesPerPixel.

See also InkSet, above.

No default.

DotRange

Tag	= 336 (150.H)
Type	= BYTE or SHORT
N	= 2, or 2*SamplesPerPixel

The component values that are to correspond to a 0% dot and 100% dot. DotRange[0] corresponds to a 0% dot, and DotRange[1] corresponds to a 100% dot.

If a DotRange pair is included for each component, the values for each component are stored together, so that the pair for Cyan would be first, followed by the pair for Magenta, and so on. *Use of multiple dot ranges is, however, strongly discouraged for now, in the interests of simplicity and compatibility with ANSI IT8 standards.*

A number of prepress systems like to keep some “headroom” and “footroom” for various purposes on both ends of the range. What exactly to do with components that are less than the 0% aim point or greater than the 100% aim point is not specified, and is application-dependent.

It is strongly recommended that the writer not attempt to use this field to reverse the sense of the pixel values. That is, DotRange[0] should be less than DotRange[1]. And both are, of course, within the range [0, (2**BitsPerSample) - 1]. There is probably no good technical reason for this recommendation, as long as software engineers are careful in their implementations; but if something CAN go wrong, it usually will.

Note that a writer can include a different range for each separation if desired, in which case N is 2*SamplesPerPixel, instead of N=2.

Default: a component value of 0 corresponds to a 0% dot, and a component value of 255 (assuming 8-bit pixels) corresponds to a 100% dot. That is, DotRange[0] = 0 and DotRange[1] = (2**BitsPerSample) - 1.

TargetPrinter

Tag	= 337 (151.H)
Type	= ASCII
N	= any

An arbitrary text string containing a description of the printing environment for which this separation was intended.

History

This Section has been expanded from earlier drafts, with the addition of the **InkSet, InkNames, DotRange**, **TargetPrinter**, and a new compression scheme, but is nevertheless backward-compatible with the earlier draft versions.

Future enhancements: definition of the characterization information, so that the CMYK data can be retargeted to a different printing environment, and so that display on a CRT or proofing device can more accurately represent the color. ANSI IT8 is working on such a proposal.

This completes the separated image section.

Section 20: The HalftoneHints Tag

Please direct comments to:
Ed Beeman
Hewlett Packard
esb@hpgrla.gr.hp.com

Importance of Highlight & Shadow placement

The single most easily recognized failing of continuous tone images is incorrect placement of the highlight and shadow. It is critical that a halftone process be capable of printing the lightest areas of the image as the smallest halftone spot capable of the output device at the specified printer resolution and screen ruling. Specular highlights (small ultra-white areas) should be printable as paper only. Similarly so for the shadow areas.

Consistency in highlight and shadow placement allows the user to obtain predictable results on a wide variety of halftone output devices. Proper implementation of this tag will provide a significant step toward device independent imaging, such that low cost printers may to be used as effective proofing devices for images which will later be halftoned on a high resolution imagesetter.

The HalftoneHints Tag

HalftoneHints

Tag	= 321 (141.H)
Type	= SHORT
N	= 2

The purpose of the HalftoneHints tag is to convey to the halftone function the range of gray levels within a colorimetrically specified image which should retain tonal detail. The tag contains two values of sixteen bits each, and therefore is contained wholly within the tag itself; no offset is required. The first word specifies the highlight gray level which should be halftoned at the lightest printable tint of the final output device. The second word specifies the shadow gray level which should be halftoned at the darkest printable tint of the final output device. Portions of the image which are whiter than the highlight gray level will quickly, if not immediately, fade to specular highlights. There is no default value specified, since the highlight and shadow gray levels are a function of the subject matter of a particular image.

Appropriate values may be derived algorithmically, or may be specified by the user, either directly or indirectly.

The HalftoneHints tag as defined here defines an achromatic function. It can be used just as effectively with color images as with monochrome images. When used with opponent color spaces such as CIE L*a*b* or YCbCr, it refers to the achromatic component only; L* in the case of CIELab, and Y in the case of YCbCr. When used with tri-stimulus spaces such as RGB, it suggests to retain tonal detail for all colors with an NTSC gray component within the bounds of the R=G=B=Highlight to R=G=B=Shadow range.

Comments for TIFF writers:

TIFF writers are encouraged to include the HalftoneHints tag in all color or grayscale images where BitsPerSample >1. Although no default value is specified, prior to the introduction of this tag it has been common practice to implicitly specify the highlight and shadow gray levels as 1 and 2**BitsperSample-2 and manipulate the image data to this definition. There are some disadvantages to this technique, and it is not feasible for a fixed gamut colorimetric image type. Appropriate values may be derived algorithmically, or may be specified by the user, either directly or indirectly. Automatic algorithms exist for analyzing the histogram of the achromatic intensity of an image and defining the minimum and maximum values as the highlight and shadow settings such that tonal detail is retained throughout the image. This kind of algorithm may try to impose a highlight or shadow where none really exists in the image, which may require user controls to override the automatic setting.

It should be noted that the choice of the highlight and shadow values is some what output dependent. For instance, in situations where the dynamic range of the output medium is very limited (as in newsprint and to a lesser degree in laser output), it may be desirable for the user to clip some of the lightest or darkest tones in order to avoid the reduced contrast resulting from compressing the tone of the entire image. Different settings might be chosen for 150 line halftone printed on coated stock. Keep in mind these values may be adjusted later (which is likely to not be possible unless the image is stored as a colorimetric fixed full gamut image), and that more sophisticated page-layout applications may be capable of presenting a user interface to re-make these decisions at a point where the halftone process is well understood.

It should be noted that although CCDs are linear intensity detectors, TIFF writers may choose to manipulate the image to store gamma compensated data. Gamma compensated data is more efficient at encoding an image than linear intensity data because it requires fewer BitsPerPixel to eliminate banding in the darker tones. It also has the advantage of being closer to the tone response of the display or printer, and therefore less likely to produce poor results from applications which are not rigorous about their treatment of images. Be aware that the PhotometricInterpretation value of 0 or 1 (grayscale) implies linear data, since no gamma is specified. The PhotometricInterpretation value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If data is written as something other than the default, then a GrayResponseCurve tag or a TransferFunction tag must be present to define the deviation. For grayscale data, be sure that the densities in the GrayResponseCurve are consistent with the PhotometricInterpretation tag and the HalftoneHints tag.

Comments for TIFF readers:

TIFF readers which will be sending a grayscale image to a halftone output device, whether it is a binary laser printer, or a PostScript imagesetter, should make an effort to maintain the highlight and shadow placement. This requires two steps. First, determine the highlight and shadow gray level of a particular image. Second, communicate that information to the halftone engine.

To determine the highlight and shadow gray levels, begin by looking for a HalftoneHints tag. If it exists, it takes precedence, and the first word represents the gray level of the highlight, and the second word represents the gray level of the shadow. If the image is a colorimetric image (i.e. it has a GrayResponseCurve tag or a TransferFunction tag), but does not contain a HalftoneHints tag, then the gamut mapping techniques described earlier should be used to determine the highlight and shadow values. If neither of these conditions are true, then the file should be treated as if a HalftoneHints tag had indicated a highlight at gray level 1, and a shadow at gray level 2**BitsPerPixel-2 (or vice-versa depending on the PhotometricInterpretation tag). Once the highlight and shadow gray levels have been determined, the next step is to communicate this information to the halftone module. The halftone module may exist within the same application as the TIFF reader, it may exist within a separate printer driver, or it may exist within the Raster Image Processor (RIP) of the printer itself. Whether the halftone process is a simple dither pattern, or a general purpose spot function, it has some gray level at which the lightest printable tint will be rendered. The HalftoneHint concept is best implemented in an environment where this lightest printable tint is easily and consistently specified.

There are several ways in which an application can communicate the highlight and shadow to the halftone function. Some environments may allow the application to pass the highlight and shadow to the halftone module explicitly along with the image. This is the best approach, but many environments do not yet provide this capability. Other environments may provide fixed gray levels at which the highlight and shadow will be rendered. For these cases, the application should build a tone map which matches the highlight and shadow specified in the image to the highlight and shadow gray level of the halftone module. This approach requires more work by the application software, but will still provide excellent results. Some environments will not have any consistent concept of highlight and shadow at all. In these environments, the best an application can do is characterize each of the supported printers and save the observed highlight and shadow gray levels. The application can then use these values to achieve the desired results, provided that the environment doesn't change.

Once the highlight and shadow areas are selected, care should be taken to appropriately map intermediate gray levels to those expected by the halftone engine, which may or may not be linear Reflectance. It should be noted that although CCDs are linear intensity detectors, and many TIFF files are stored as linear intensity, most output devices require significant tone compensation (sometimes called gamma correction) to correctly display or print linear data. Be aware that the PhotometricInterpretation value of 0, 1 implies linear data, since no gamma is specified. The PhotometricInterpretation value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If a GrayResponseCurve tag or a TransferFunction tag is present it may define something other than the default.

Some background on the halftone process:

In order to obtain the best results when printing a continuous tone raster image, it is seldom desirable to simply reproduce the tones of the original on the printed page. Most often there is some gamut mapping required. Often this is because the tonal range of the original extends beyond the tonal range of the output medium. In some cases the tone range of the original is within the gamut of the output medium , but it may be more pleasing to expand the tone of the image to fill the range of the output. Given that the tone of the original is to be adjusted, there is a whole range of possibilities for the level of sophistication which may be undertaken by a software application.

Printing monochrome output is far less sophisticated than printing color output. For monochrome output the first priority is to control the placement of the highlight and the shadow. Ideally, a quality halftone will have sufficient levels of gray such that a standard observer is not able to distinguish the interface between any two adjacent levels of gray. In practice, however, there is often a significant step between the tone of the paper and the tone of the lightest printable tint. Although usually less severe, the problem is similar between solid ink and the darkest printable tint. Since the dynamic range between the lightest printable tint and the darkest printable tint is usually less than one would like, it is common to maximize the tone of the image within these bounds. Not all images will have a highlight (an area of the image which is desirable to print as light as possible while still retaining tonal detail), but if one exists, it should be carefully controlled to print at the lightest printable tint of the output medium. Similarly the darkest areas of the image which should retain tonal detail should be printed as the darkest printable tint of the output medium. Tones lighter or darker than these may be clipped at the limits of the paper and ink. Satisfactory results may be obtained in monochrome work by doing nothing more than a perceptually linear mapping of the image between these rigorously controlled endpoints. This level of sophistication is sufficient for many mid-range applications, although the results will often appear somewhat flatter, i.e. lower in contrast, than may be desired.

The next step is to increase contrast slightly in the tonal range of the image which contains the most important subject matter. To perform this step well requires considerably more information about the image and about the press. In order to know where to add contrast, the algorithm must have access to first the keyness of the image; the tone range which the user considers most important. In order to know how much contrast to add, the algorithm must have access to the absolute tone of the original, and the dynamic range of the output device so that it may calculate the amount of tone compression to which the image is actually subjected.

Most images are called normal key. The important subject areas of a normal key image are in the midtones. These images do well when a so-called “sympathetic curve” is applied, which increases the contrast in midtones slightly, at the expense of the contrast in the lighter and darker tones. White china on a white tablecloth is an example of a high key image. High key images benefit from higher contrast in lighter tones, with less contrast needed in the midtones and darker tones. Low key images have important subject matter in the darker tones, and benefit from increasing the contrast in the darker tones. Specifying the keyness of an image might be attempted by automatic techniques, but will likely be failure prone without user input. For example, a photo of a bride in a white wedding dress may be a high key image if you are selling wedding dresses, but may be a normal key image if you are the parents of the bride, and are more interested in her smile.

Sophisticated color reproduction makes use of all of these principles, and then applies them in three dimensions. The mapping of the highlight and shadow becomes only one small, albeit critical, portion of the total issue of mapping colors which are too saturated for the output medium. Here again automatic techniques may be employed as a first pass, with the user becoming involved in the clip/compress mapping decision. The HalftoneHints tag is still useful in communicating which portions of the intensity of the image must be retained, and which may be clipped. Again, a sophisticated application may override these settings if later user input is received.

Section 21: Associated Alpha Handling

Please direct comments to:
Sam Leffler
Silicon Graphics
sam@sgi.com

Introduction.

A common technique in computer graphics is to assemble an image from one or more elements that are rendered separately. When elements are combined using compositing techniques, matte or coverage information must be present for each pixel in order to create a properly anti-aliased accumulation of the full image [Porter84]. This matting information is an example of additional per-pixel data that must be maintained with an image. This document describes how to use the tag to store the requisite matting information, commonly called the associated alpha, or just alpha. This scheme enables efficient manipulation of image data during compositing operations.

Images with matting information are stored in their natural format, but with an additional component per pixel. The tag, is included with the image to indicate that an extra component of each pixel contains associated alpha data. In addition, when associated alpha data are included with RGB data, the RGB components must be stored pre-multiplied by the associated alpha component and component values in the range [0,2**BitsPerSample-1] are implicitly mapped onto the interval [0,1]. That is, for each pixel (r,g,b) and opacity A, where r, g, b, and A are in the range [0,1], (A*r,A*g,A*b,A) must be stored in the file. If A is zero, then the color components should be interpreted as zero. Storing data in this pre-multiplied format, allows compositing operations to be implemented most efficiently. In addition, storing pre-multiplied data makes it possible to specify colors with components outside the normal [0,1] interval. The latter is useful for defining certain operations that effect only the luminescence [Porter84].

Tags

ExtraSamples

Tag	= 338 (152.H)
Type	= SHORT
N	= 1

This tag must have a value of 1 (Associated Alpha data with pre-multiplied color components). The associated alpha data stored in SamplesPerPixel-1 of each pixel contains the opacity of that pixel and the color information is pre-multiplied by alpha.

Comments

Associated alpha data is just another component added to each pixel. Thus, for example, its size is defined by the value of the BitsPerSample tag.

Note that since data is stored with RGB components already multiplied by alpha, naive applications that want to display an RGBA image on a display can do so simply by displaying the RGB component values. This works because it is effectively the same as merging the image over a black background. That is, to merge one image over another, the color of resultant pixels are calculated as:

$$C_r = C_{over} * A_{over} + C_{under} * (1 - A_{over})$$

Since the “under image” is a black background, this equation reduces to

$$C_r = C_{over} * A_{over}$$

which is exactly the pre-multiplied color; i.e. what is stored in the image.

On the other hand, to print a RGBA image, one must composite the image over a suitable background page color. For a white background, this is easily done by adding 1 - A to each color component. For an arbitrary background color C_{back}, the *printed color* of each pixel is

$$C_{print} = C_{image} + C_{back} * (1 - A_{image})$$

(since C_{image} is pre-multiplied).

Since the tag is independent of other tags, this scheme permits alpha information to be stored in whatever organization is appropriate. In particular, components can be stored packed (PlanarConfiguration=1); this is important for good I/O performance and for good memory access performance on machines that are sensitive to data locality. Note, however, that if this scheme is used, that TIFF readers must not derive the SamplesPerPixel from the value of the PhotometricInterpretation tag; e.g. if RGB, that SamplesPerPixel is 3.

In addition to being independent of data storage-related tags, the tag is also independent of the PhotometricInterpretation tag. This means, for example, that it is easy to use this tag to specify greyscale data and associated matte information. Note that a Palette image with associated alpha will not have the colormap indices pre-multiplied, but rather the RGB colormap values will be pre-multiplied.

Unassociated Alpha and Transparency Masks

Some image manipulation applications support notions of transparency masks and soft-edge masks. The associated alpha information described in this Section is different from this *unassociated alpha* information in many ways, most importantly:

1. Associated alpha describes opacity or coverage at each pixel, while clipping-related alpha information describes a boolean relationship. That is, associated alpha can specify fractional coverage at a pixel, while masks specify either 0 or 100 percent coverage.
2. Once defined, associated alpha is not intended to be removed or edited, except as a result of compositing the image; it is an integral part of an image. Unassociated alpha, on the other hand, is designed as an ancillary piece of information.

References

[Porter84] “Compositing Digital Images”. Thomas Porter, Tom Duff; Lucasfilm Ltd. ACM SIGGRAPH Proceedings Volume 18, Number 3. July, 1984.

Section 22: Data Sample Format

Please direct comments to:
Nancy Cam
nance@sgi.com

Introduction

TIFF implicitly types all data samples as unsigned integer values. Certain applications, however, require the ability to store image related data in other formats such as floating point. This appendix presents a scheme for describing a variety of data sample formats.

New Tags

SampleFormat

Tag	= 339 (153.H)
Type	= SHORT
N	= SamplesPerPixel

This tag specifies how to interpret each data sample in a pixel. Possible values are:

1 unsigned integer data

2 two’s complement signed integer data

3 IEEE floating point data [IEEE]

4 undefined data format

Note that the SampleFormat tag does not specify the size of data samples; this is still done by the BitsPerSample tag.

A tag value of “undefined” is a statement by the writer that it did not know how to interpret the data samples; for example, if it was copying an existing image. A reader would typically treat an image with “undefined” data as if the tag were not present; i.e. as unsigned integer data.

Default is 1, unsigned integer data.

SMinSampleValue

Tag	= 340 (154.H)
Type	= the field type that best matches the sample data
N	= SamplesPerPixel

This new tag specifies the minimum sample value. Note that a value should be given for each data sample. That is, if the image has 3 SamplesPerPixel, 3 values must be specified.

The default for SMinSampleValue and SMaxSampleValue is the full range of the data type.

SMaxSampleValue

Tag	= 341 (155.H)
Type	= the field type that best matches the sample data
N	= SamplesPerPixel

This new tag specifies the maximum sample value.

Comments

The introduction of the SampleFormat tag allows for more general imaging (such as image processing) applications to employ TIFF as a valid file format. These tags are not part of the Baseline TIFF requirements. They are defined in this specification as a means for providing support for a broader set of imaging applications, specifically, those involving typed image data.

The importance of tags such as SMinSampleValue and SMaxSampleValue become more meaningful when image data is typed. The presence of these tags makes it possible for readers to assume that data samples are bound to the range [SMinSampleValue, SMaxSampleValue] without scanning the image data.

References

[IEEE] “IEEE Standard 754 for Binary Floating-point Arithmetic”.

Section 23: RGB Image Colorimetry Information

Please direct comments to:
Dan or Chris Sears
sears@netcom.com

Introduction

Color printers, displays and scanners continue to improvein quality and availability while they drop in price. Now the problem is to display color images so thatthey appear to be identical on different hardware.

The key to reproducing the same color on different devicesis to use the CIE 1931 XYZ color matching functions,the international standard for color comparison. UsingCIE XYZ, an image's colorimetry information can fullydescribe its color interpretation. The approach takenhere is essentially calibrated RGB.

Using the tags described below tags, an application candisplay the image on different hardware and achieve colorimetricallyidentical results. The process of using this colorimetryinformation for displaying an image is straightforwardon a color monitor but it is more complex for colorprinters. Results will be limited by the color gamutand other characteristics of the display or printingdevice.

The following tags describe the image colorimetry informationof a TIFF image:

WhitePoint	chromaticity of thewhite point of the image
PrimaryChromaticities	chromaticities ofthe primaries of the image
TransferFunction	transfer function forthe pixel data
ReferenceBlackWhite	pixel component headroomand footroom parameters

Both the TransferFunction and ReferenceBlackWhite tagshave defaults based on industry standards. An imagehas a colorimetric interpretation if and only if boththe WhitePoint and PrimaryChromaticities tags are present. An image without these colorimetry tags will be displayedin an application and hardware dependent manner.

Note: in the definitions below, BitsPerSample is usedas if it were a single number when in fact it isan array of SamplesPerPixel numbers. The elements ofthis array may not always be equal, for example: 5/6/516-bit pixels. In the case of unequal BitsPerSamplevalues, the definitions below can be extended in a straightforwardmanner.

The colorimetry information tag definitions have the followingdifferences with TIFF 5.0:

- removed the use of image colorimetry defaults
- renamed the ColorResponseCurves tag as TransferFunction
- optionally allowed a single TransferFunctiontable to describe all three channels
- described the use of the TransferFunctiontag for YCbCr, Palette, MinIsWhite and

MinIsBlackPhotometricInterpretation types
- added the ReferenceBlackWhite tag
- tag sizes don't depend on SamplesPerPixel

Tag Definitions

WhitePoint

Tag	= 318 (13E hex)
Type	= RATIONAL
N	= 2

The chromaticityof the white point of the image. This is the chromaticitywhen each of the primaries has its ReferenceWhite value. The value is described using the 1931 CIE xy chromaticitydiagram and only the chromaticity is specified. Thisvalue can correspond to the chromaticity of the alignmentwhite of a monitor, the filter set and light sourcecombination of a scanner or the imaging model of arendering package. The ordering is white[x], white[y].

For example, the CIE Standard Illuminant D65 used byCCIR Recommendation 709 and Kodak PhotoYCC is:

3127/10000,3290/10000

No default.

PrimaryChromaticities

Tag	=319 (13F hex)
Type	= RATIONAL
N	= 6

The chromaticities of the primaries of the image. Thisis the chromaticity for each of the primaries when ithas its ReferenceWhite value and the other primarieshave their ReferenceBlack values. These values are describedusing the 1931 CIE xy chromaticity diagram and onlythe chromaticities are specified. These values can correspondto the chromaticities of the phosphors of a monitor,the filter set and light source combination of a scanneror the imaging model of a rendering package. The orderingis red[x], red[y], green[x], green[y], blue[x], blue[y].

For example the CCIR Recommendation 709 primaries, alsoused by PhotoYCC, are:

640/1000,330/1000,

300/1000, 600/1000,

150/1000, 60/1000

No default.

TransferFunction

Tag	=301 (12D hex)
Type	= SHORT

N = {1 or 3} * (1 << BitsPerSample)

Describes a transferfunction for the image in tabular style. Pixel componentscan be gamma compensated, companded, non-uniformly quantizedor coded in some other way. The TransferFunction mapsthe pixel components from a non-linear BitsPerSample (e.g.8-bit) form into a 16-bit linear form without a perceptibleloss of accuracy.

If N = 1 << BitsPerSample, the transfer function isthe same for each channel and all channels share asingle table. Of course, this assumes that each channelhas the same BitsPerChannel value.

If N = 3 * (1 << BitsPerSample), there are three tablesand the ordering is the same as for pixel componentsof the PhotometricInterpretation tag. These tables areseparate and not interleaved. For example, with RGBimages all red entries come first, followed by all greenentries, followed by all blue entries.

The length of each component table is 1 << BitsPerSample. The width of each entry is 16 bits as implied bythe type SHORT. The value 0 represents the minimumintensity and 65535 represents the maximum intensity. The values [0, 0, 0] represent black and [65535, 65535,65535] represent white.

The TransferFunction can be applied to images with aPhotometricInterpretation value of RGB, Palette, YCbCr,MinIsWhite and MinIsBlack. The TransferFunction is notused with other PhotometricInterpretation types.

For RGB PhotometricInterpretation, ReferenceBlackWhite expandsthe coding range and the TransferFunction tables decompandthe RGB value. The WhitePoint and PrimaryChromaticitiesfurther describe the RGB colorimetry.

For Palette PhotometricInterpretation, the Colormap mapsthe pixel into three 16-bit values that when scaledto BitsPerSample-bits serve as indices into the TransferFunctiontables which decompand the RGB value. The WhitePointand PrimaryChromaticities further describe the underlyingRGB colorimetry.

A Palette value can be scaled into a TransferFunctionindex by:

```
index= (value * ((1 << BitsPerSample) - 1)) / 65535L;
```

A TransferFunction index can be scaled into a Palettevalue by:

```
value= (index * 65535L) / ((1 << BitsPerSample) - 1);
```

For YCbCr PhotometricInterpretation, ReferenceBlackWhiteexpands the coding range, the YCbCrCoefficients describethe decoding matrix to transform YCbCr into RGB andthe TransferFunction tables decompand the RGB value.The WhitePoint and PrimaryChromaticities tags provide furtherdescription of the underlying RGB colorim-etry.

After coding range expansion by ReferenceBlackWhite, RGBvalues may be outside the domain of the TransferFunction. Also, the display device matrix can transform RGB valuesinto display device RGB values outside the domain ofthe device. These values are handled in an applicationdependent manner.

For RGB images with non-default ReferenceBlackWhite codingrange expansion and for YCbCr images, the resolutionof the TransferFunction may be insufficient. As anillustration, after the YCbCr transformation matrix, thedecoded RGB values must be rounded to index into theTransferFunction tables. Applications needing the extraaccuracy should interpolate between the elements of theTransferFunction tables. Cubic spline interpolation isrecommended.

For MinIsWhite and MinIsBlack PhotometricInterpretation,the TransferFunction decompands the grayscale pixel valueto a linear 16-bit form. Note that a TransferFunctionvalue of 0 represents black and 65535 represents whiteregardless of whether a grayscale image is MinIsWhiteor MinIsBlack. For example, the zeroth element of aMinIsWhite TransferFunction table will likely be 65535. This extensionof the TransferFunction tag for grayscale images is intendedto replace the GrayResponseCurve tag.

The TransferFunction does not describe a transfer characteristicoutside of the range for ReferenceBlackWhite. For example,the single PhotoYCC TransferFunction table can be generatedby:

```
   for (i = 0; (i / 255.0) < 0.018;i++)
      table[i]= floor(((4.5 * i) / 255.0) * 65535.0 + 0.5);
    for (; i < 256; i++)
      table[i]= floor((1.099 * pow(i / 255.0, 0.45)) * 65535
+ 0.5);
```

An application needing finer precision or the negativelobe that PhotoYCC uses can identify this table as aspecial case. It should look at the WhitePoint, thePrimaryChromaticities, the TransferFunction and the ReferenceBlackWhitetags. If they match the PhotoYCC values, the PhotoYCC TransferFunction can be evaluated analytically.

Default is a single table corresponding to the NTSCstandard gamma value of 2.2. This table is used foreach channel. It can be generated by:

```
    NValues = 1 << BitsPerSample;
    for (table[0]= 0, i = 1; i < NValues; i++)
      table[i]= floor(pow(i / (NValues - 1.0), 2.2) * 65535
+ 0.5);
```

ReferenceBlackWhite

Tag	=532 (214 hex)
Type	= RATIONAL
N	= 6

Specifies a pairof headroom and footroom codes for each pixel component. The first component code within a pair is associatedwith ReferenceBlack and the second is associated withReferenceWhite. The ordering of pairs is the same asfor pixel components of the PhotometricInterpretation type. ReferenceBlackWhite can be applied to images with aPhotometricInterpretation value of RGB or YCbCr. ReferenceBlackWhiteis not used with other PhotometricInterpretation values.

Computer graphics commonly places black and white atthe extremities of the binary representation of imagedata, for example black at code 0 and white at code255. In other disciplines such as printing, film andvideo there are practical reasons to provide footroomcodes below ReferenceBlack and headroom codes above ReferenceWhite.

In film applications, they correspond to the densitiesDmax and Dmin. In video applica-tions, ReferenceBlackcorresponds to 7.5 IRE and 0 IRE in systems with andwithout setup respectively and ReferenceWhite correspondsto 100 IRE units.

Using YCbCr and the CCIR Recommendation 601.1 video standardas an example, code 16 represents ReferenceBlack andcode 235 represents ReferenceWhite for the luminance component,Y. For the chrominance components, Cb and Cr, code128 represents ReferenceBlack and code 240 representsReferenceWhite. With Cb and Cr the ReferenceWhite valueis used to code reference blue and reference red respectively.

The full range component value is converted from thecode by:

```
value= (code - ReferenceBlack) * CodingRange
   /(ReferenceWhite - ReferenceBlack);
```

The code is convertedfrom the full range component value by:

```
code = (value * (ReferenceWhite - ReferenceBlack) / CodingRange)
   + ReferenceBlack;
```

For RGB imagesand for the Y component of YCbCr images CodingRangeis defined as:

```
CodingRange = (1 << BitsPerSample) - 1;
```

For the Cb and Cr components of YCbCr images CodingRangeis defined as:

```
CodingRange = 127;
```

For RGB images, in the default special case of no headroomor footroom this conversion can be skipped because thescaling multiplier equals 1.0 and the value equals thecode.

For YCbCr images, in the case of no headroom or footroomthe conversion for Y can be skipped since the valueequals the code. For Cb and Cr, ReferenceBlack muststill be subtracted from the code. In the generalcase, the scaling multiplication for the Cb and Cr componentcodes can be factored into the YCbCr transform matrix.

Useful ReferenceBlackWhite values for YCbCr images are:

[0/1, 255/1,128/1, 255/1, 128/1, 255/1]
  no headroom/footroom

[15/1, 235/1, 128/1, 240/1, 128/1, 240/1]
  CCIR Recommendation 601.1 headroom/footroom

[0/1, 181883/1000, 156/1, 255/1, 137/1, 232/1]
  Kodak PhotoYCC

Useful ReferenceBlackWhitevalues for BitsPerSample = 8,8,8 Class R images are:

[0/1, 255/1,0/1, 255/1, 0/1, 255/1]
  no headroom/footroom

[16/1, 235/1, 16/1, 235/1, 16/1, 235/1]
  CCIR Recommendation 601.1 headroom/footroom

Default is [0,NV, 0, NV, 0, NV] where NV = (1 << BitsPerSample)- 1.

Section 24: YC_bC_r Images

Please send any comments to:
Eric Hamilton
C-Cube Microsystems
eric@c3.pla.ca.us

Introduction

Digitizers of video sources that create RGB data are becoming more capable and less expensive. Class R RGB is adequate for this purpose (see Appendices G and H). However, for both digital video and image compression applications a color difference color space is needed. The television industry depends on YC_bC_r for digital video. For image compression, subsampling the chrominance components allows for greater compression. Class Y supports these images and applications.

Class Y is based on CCIR Recommendation 601-1, “Encoding Parameters of Digital Television for Studios.” Class Y also has parameters that allow the description of related standards such as CCIR Recommendation 709 and technological variations such as component sample positioning.

YC_bC_r is a distinct PhotometricInterpretation type. RGB pixels are converted to and from YC_bC_r for storage and display.

Class Y defines the following tags:

YC_bC_rCoefficients	transformation from RGB to YC_bC_r
YC_bC_rSubSampling	subsampling of the chrominance components
YC_bC_rPositioning	positioning of chrominance component samples relative to the luminance samples

In addition, ReferenceBlackWhite, which specifies coding range expansion, is required by Class Y (see the RGB Colorimetry section).

Class Y YC_bC_r images have three components: Y, the luminance component, and C_b and C_r, two chrominance components. Class Y uses the international standard notation YC_bC_r for color difference component coding. This is often incorrectly called YUV, which properly applies only to composite coding.

The transformations between YC_bC_r and RGB are linear transformations of un-interpreted RGB sample data, typically gamma-corrected values. The YC_bC_rCoefficients tag describes the parameters of this transformation.

Another feature of Class Y comes from subsampling the chrominance components. A Class Y image can be compressed by reducing the spatial resolution of chrominance components. This takes advantage of the relative insensitivity of the human visual system to chrominance detail. The YC_bC_rSubSampling tag describes the degree of subsampling which has taken place.

When a Class Y image is subsampled, each C_b and C_r sample is associated with a group of luminance samples. The YC_bC_rPositioning tag describes the position of the chrominance component samples relative to the group of luminance samples: centered or cosited.

Class Y requires use of the ReferenceBlackWhite tag. This tag expands the coding range by describing the reference black and white values for the different components that allow headroom and footroom for digital video images. Since the default for ReferenceBlackWhite is inappropriate for Class Y, it must be used explicitly.

At first sight it may seem that the information conveyed by Class Y and the RGB Colorimetry section are redundant. However, decoding YC_bC_r to RGB primaries requires the YC_bC_r tags and interpretation of the resulting RGB primaries requires the colorimetry and transfer function information. See the RGB Colorimetry section for details.

Extensions to Existing Tags

Class Y images use a distinct PhotometricInterpretation Tag value:

PhotometricInterpretation

Tag	= 262 (106.H)
Type	= SHORT
N	= 1

This Tag indicates the color space of the image. The new value is:

6 = YC_bC_r

A value of 6 indicates that the image data is in the YC_bC_r color space. TIFF uses the international standard notation YC_bC_r for color difference sample coding. Y is the luminance component. C_b and C_r are the two chrominance components. RGB pixels are converted to and from YC_bC_r form for storage and display.

Tags defined in Class Y

YC_bC_rCoefficients

Tag	= 529 (211.H)
Type	= RATIONAL
N	= 3

The transformation from RGB to YC_bC_r image data. The transformation is specified as three rational values which represent the coefficients used to compute luminance, Y.

The three rational coefficient values, *LumaRed*, *LumaGreen* and *LumaBlue*, are the proportions of red, green, and blue respectively in luminance, Y.

Y, C_b, and C_r may be computed from RGB using the luminance coefficients specified by this tag as follows:

$$Y = (\mathit{LumaRed} * R + \mathit{LumaGreen} * G + \mathit{LumaBlue} * B)$$

$$C_b = (B - Y) / (2 - 2 * \mathit{LumaBlue})$$

$$C_r = (R - Y) / (2 - 2 * \mathit{LumaRed})$$

R, G, and B may be computed from YC_bC_r as follows:

$R = C_r * (2 - 2 * \mathit{LumaRed}) + Y$

$G = (Y - \mathit{LumaBlue} * B - \mathit{LumaRed} * R) / \mathit{LumaGreen}$

$B = C_b * (2 - 2 * \mathit{LumaBlue}) + Y$

In disciplines such as printing, film and video there are practical reasons to provide footroom codes below the ReferenceBlack code and headroom codes above ReferenceWhite code. In such cases the values of the transformation matrix used to convert from YC_bC_r to RGB must be multiplied by a scale factor in order to produce full range RGB values. These scale factors depend on the reference ranges specified by the ReferenceBlackWhite tag. See the ReferenceBlackWhite and TransferFunction tags for more details.

The values coded by this tag will typically reflect the transformation specified by a standard for YC_bC_r encoding. The following table contains examples of commonly used values.

Standard	*LumaRed*	*LumaGreen*	*LumaBlue*
CCIR Recommendation 601-1	299 / 1000	587 / 1000	114 / 1000
CCIR Recommendation 709	2125 / 10000	7154 / 10000	721 / 10000

The default values for this tag are those defined by CCIR Recommendation 601-1: 299/1000, 587/1000 and 114/1000, for *LumaRed*, *LumaGreen* and *LumaBlue*, respectively.

YC_bC_rSubSampling

Tag	= 530 (212.H)
Type	= SHORT
N	= 2

Specifies the subsampling factors used for the chrominance components of a YC_bC_r image. The two fields of this tag, *YC_bC_rSubsampleHoriz* and *YC_bC_rSubsampleVert*, specify the horizontal and vertical subsampling factors respectively.

The two fields of this tag are defined as follows:

Short 0: *YC_bC_rSubsampleHoriz*:

1 =	ImageWidth of this chroma image is equal to the ImageWidth of the associated luma image.
2 =	ImageWidth of this chroma image is half the ImageWidth of the associated luma image.
4 =	ImageWidth of this chroma image is one quarter the ImageWidth of the associated luma image.

Short 1: *YC_bC_rSubsampleVert*:

1 =	ImageLength (height) of this chroma image is equal to the ImageLength of the associated luma image.
2 =	ImageLength (height) of this chroma image is half the ImageLength of the associated luma image.
4 =	ImageLength (height) of this chroma image is one quarter the

ImageLength of the associated luma image.

Both C_b and C_r have the same subsampling ratio. Also, *YC_bC_rSubsampleVert* shall always be less than or equal to *YC_bC_rSubsampleHoriz*.

ImageWidth and ImageLength are constrained to be integer multiples of *YC_bC_rSubsampleHoriz* and *YC_bC_rSubsampleVert* respectively. TileWidth and TileLength have the same constraints. RowsPerStrip must be an integer multiple of *YC_bC_rSubsampleVert*.

The default values of this tag are [2, 2].

YC_bC_rPositioning

Tag	= 531 (213.H)
Type	= SHORT
N	= 1

Specifies the positioning of subsampled chrominance components relative to luminance samples.

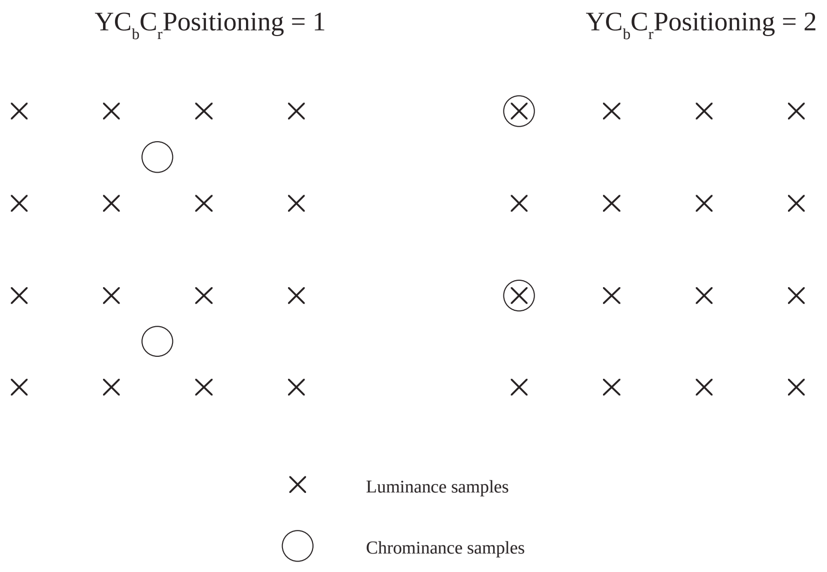
Specification of the spatial positioning of pixel samples relative to the other samples is necessary for proper image post processing and accurate image presentation. In Class Y files, the position of the subsampled chrominance components are defined with respect to the luminance component. Since components must be sampled orthogonally (along rows and columns), the spatial position of the samples in a given subsampled component may be determined by specifying the horizontal and vertical offsets of the first sample, i.e. the sample in the upper left corner, with respect to the luminance component. The horizontal and vertical offsets of the first chrominance sample are denoted Xoffset[0,0] and Yoffset[0,0] respectively. Xoffset[0,0] and Yoffset[0,0] are defined in terms of the number of samples in the luminance component.

The values for this tag are defined as follows:

Tag value	YC_bC_r Positioning	X and Y offsets of first chrominance sample
1	centered	Xoffset[0,0] = *ChromaSubsampleHoriz* / 2 - 0.5 Yoffset[0,0] = *ChromaSubsampleVert* / 2 - 0.5
2	cosited	Xoffset[0,0] = 0 Yoffset[0,0] = 0

Tag value 1 (centered) must be specified for compatibility with industry standards such as Postcript Level 2 and QuickTime. Tag value 2 (cosited) must be specified for compatibility with most digital video standards, such as CCIR Recommendation 601-1.

As an example, for *ChromaSubsampleHoriz* = 4 and *ChromaSubsampleVert* = 2, the centers of the samples are positioned as illustrated below:



Proper subsampling of the chrominance components incorporates an anti-aliasing filter which reduces the spectral bandwidth of the full resolution samples. The type of filter used for subsampling determines the value of the YC_bC_rPositioning tag.

For YC_bC_rPositioning = 1 (centered), subsampling of the chrominance components can easily be accomplished using a symmetrical digital filter with an even number of taps (coefficients). A commonly used filter for 2:1 subsampling utilizes two taps (1/2,1/2).

For YC_bC_rPositioning = 2 (cosited), subsampling of the chrominance components can easily be accomplished using a symmetrical digital filter with an odd number of taps. A commonly used filter for 2:1 subsampling utilizes three taps (1/4,1/2,1/4).

The default value of this tag is 1.

ReferenceBlackWhite

Tag	= 532 (214.H)
Type	= LONG
N	= 2 x SamplesPerPixel

This tag specifies a pair of image data values (codes) for each component of the image data. The first value conveys the image data value associated with ReferenceBlack and the second value conveys the image data value associated with ReferenceWhite.

Computer graphics commonly places black and white at the extremities of the binary representation of image data, for example black at code 0 and white at code 255. In other disciplines such as printing, film and video there are practical reasons to provide footroom codes below ReferenceBlack and headroom codes above ReferenceWhite.

In print applications, the reference values typically correspond to 100% and 0% ink coverage. In film applications, the reference values typically correspond to the densities Dmax and Dmin. In video applications, the reference black value corresponds to 7.5 IRE and 0 IRE in systems with and without setup respectively and the ReferenceWhite corresponds to 100 IRE units.

Using Class Y and the CCIR Recommendation 601-1 video standard as an example, code 16 represents ReferenceBlack and code 235 represents ReferenceWhite for the luminance (Y) component. For the chrominance components (Cb and Cr), code 128 represents ReferenceBlack and code 240 represents ReferenceWhite. With Cb and Cr the ReferenceWhite value is used to code reference blue and reference red respectively.

The full range component value, FullRangeValue, is converted from the code by:

```
FullRangeValue = (code - ReferenceBlack) * CodingRange
    / (ReferenceWhite - ReferenceBlack);
```

The code is converted from the full range component value by:

```
code = (FullRangeValue * (ReferenceWhite - ReferenceBlack) / CodingRange)
    + ReferenceBlack;
```

For RGB images, and for the Y component of YCbCr images, the CodingRange is 2 ** BitsPerSample - 1. For the Cb and Cr components of YCbCr images, the CodingRange is 127.

For RGB images, in the default special case of no headroom or footroom this conversion can be omitted because the scaling multiplier is one and the value equals the code.

For YCbCr images, in the case of no headroom or footroom the conversion for Y can be omitted since the value equals the code. For Cb and Cr, ReferenceBlack must still be subtracted from the code. In the general case, the scaling multiplication for each component code can be factored into the YCbCr transform matrix.

Useful ReferenceBlackWhite values for Class Y images are:

[0, 255, 128, 255, 128, 255]	no headroom/footroom
[16, 235, 128, 240, 128, 240]	CCIR Rec. 601-1 headroom/footroom

Useful ReferenceBlackWhite values for Class R images are:

[0, 255, 0, 255, 0, 255]	no headroom/footroom (default)
[16, 235, 16, 235, 16, 235]	CCIR Rec. 601-1 headroom/footroom

Default is [0, (2 ** BitsPerSample) - 1] for each component in a pixel.

Ordering of Component Samples

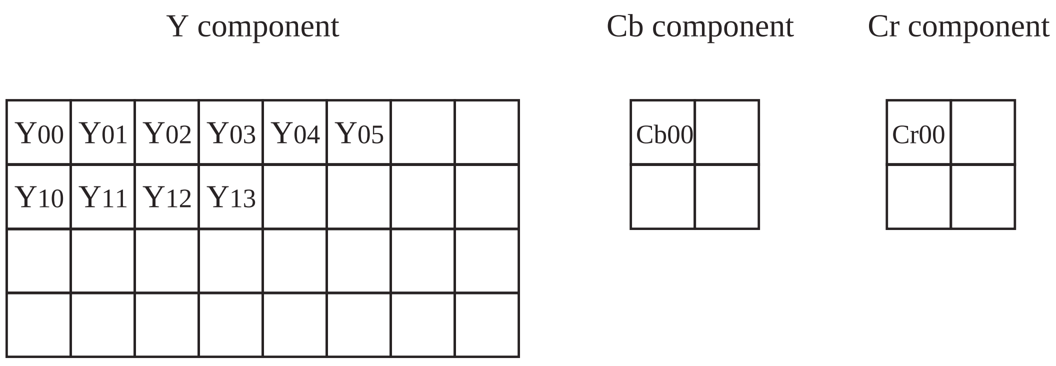
This section defines the ordering convention used for Y, C_b, and C_r component samples when the PlanarConfiguration tag value = 1 (interleaving). For PlanarConfiguration = 2, component samples are stored as 3 separate planes and the ordering is the same as for other PhotometricInterpretation tag values.

For PlanarConfiguration = 1, the component sample order is based on the subsampling factors, *ChromaSubsampleHoriz* and *ChromaSubsampleVert*, defined by the YC_bC_rSubSampling tag. The image data within a TIFF file is comprised of one or more “data units”, where a data unit is defined to be a sequence of samples:

- one or more Y samples
- a C_b sample
- a C_r sample

The Y samples within a data unit are specified as a two dimensional array having *ChromaSubsampleVert* rows of *ChromaSubsampleHoriz* samples.

Expanding on the example in the previous section, consider a YC_bC_r image having *ChromaSubsampleHoriz* = 4 and *ChromaSubsampleVert* = 2:



For PlanarConfiguration = 1, the sample order is:

Y_{00}, Y_{01}, Y_{02}, Y_{03}, Y_{10}, Y_{11}, Y_{12}, Y_{13}, Cb_{00}, Cr_{00}, Y_{04}, Y_{05} ...

Minimum Requirements for YCbCr Images

In addition to the general Baseline TIFF requirements, a YCbCr file must have the following characteristics:

SamplesPerPixel = 3. SHORT. Three components representing Y, Cb and Cr.

BitsPerSample = 8,8,8. SHORT.

Compression = none (1), LZW (5) or JPEG (6). SHORT.

PhotometricInterpretation = YC_bC_r (6). SHORT.

ReferenceBlackWhite = 6 LONGS. Specify the reference values for black and white.

If the conversion from RGB is not according to CCIR Recommendation 601-1, code YC_bC_rCoefficients.

Section 25: JPEG Compression

Send any comments to:
Eric Hamilton
C-Cube Microsystems
eric@c3.pla.ca.us

Introduction

Image compression reduces the storage requirements of pictorial data. In addition, it reduces the time necessary for access, communication and display of images. In order to address the standardization of compression techniques an international standards group was formed: the Joint Photographic Experts Group (JPEG). JPEG has as its objective to create a joint ISO/CCITT standard for continuous tone image compression (color and grayscale).

JPEG decided that because of the broad scope of the standard, no one algorithmic procedure was able to satisfy the requirements of all applications. It was decided to specify different algorithmic processes where each process is targeted to satisfy the requirements of a class of applications. Thus the JPEG standard became a “toolkit” whereby the particular algorithmic “tools” are selected according to the needs of the application environment.

The algorithmic processes fundamentally fall into two classes: lossy and lossless. Those based on the Discrete Cosine Transform (DCT) are lossy, and typically provide for substantial compression without significant degradation of the reconstructed image with respect to the source image.

The simplest DCT-based coding process is the baseline process. It provides a capability which is sufficient for most applications. There are additional DCT-based processes which extend the baseline process to a broader range of applications.

The second class of coding processes is targeted for those applications requiring lossless compression. The lossless processes are not DCT-based and are utilized independently of any of the DCT-based processes.

This Section describes the JPEG baseline and the JPEG lossless processes and the extensions to TIFF defined to support JPEG compression.

JPEG Baseline Process

The baseline process is a DCT-based algorithm which compresses images having 8 bits per component. The baseline process operates only in sequential mode. In sequential mode, the image is processed from left to right and top to bottom in a single pass by compressing the first row of data followed by the second row and continuing until the end of image is reached. Sequential operation has minimal buffering requirements and thus permits inexpensive implementations.

The JPEG baseline process is an algorithm which inherently introduces error into the reconstructed image and cannot be utilized for lossless compression. The algorithm accepts as input only those images having 8 bits per component. Images with fewer than 8 bits per component may be compressed using the baseline process algorithm by left justifying each input component within a byte before compression.

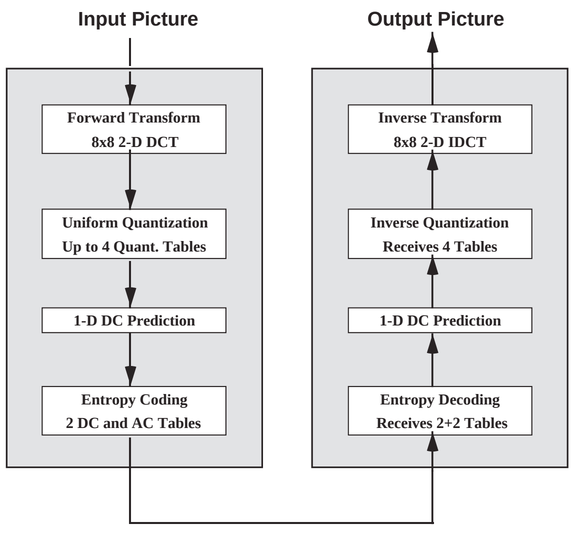

Input Picture
Output Picture
Forward Transform 8x8 2-D DCT
Uniform Quantization Up to 4 Quant. Tables
1-D DC Prediction
Entropy Coding 2 DC and AC Tables
Inverse Transform 8x8 2-D IDCT
Inverse Quantization Receives 4 Tables
1-D DC Prediction
Entropy Decoding Receives 2+2 Tables

Figure 1. Baseline Process Encoder and Decoder

A functional block diagram of the Baseline encoding and decoding processes is contained in Figure 1. Encoder operation consists of dividing each component of the input image into 8x8 blocks, performing the two-dimensional DCT on each block, quantizing each DCT coefficient uniformly, subtracting the quantized DC coefficient from the corresponding term in the previous block, and finally entropy coding the quantized coefficients using variable length codes (VLCs). Decoding is performed by inverting each of the encoder operations in the reverse order.

The DCT

Before performing the foward DCT input pixels are level-shifted so that they range from -128 to +127. Blocks of 8x8 pixels are transformed with the two-dimensional 8x8 DCT:

$$F(u,v) = \frac{1}{4} C(u)C(v) \sum\sum f(x,y) \cos \frac{\pi(2x+1)u}{16} \cos\frac{\pi(2y+1)v}{16}$$

and blocks are inverse transformed by the decoder with the Inverse DCT:

$$f(x,y) = \frac{1}{4} \sum \sum C(u)C(v)\, F(u,v) \cos\frac{\pi(2x+1)u}{16} \cos\frac{\pi(2y+1)v}{16}$$

with $u, v, x, y = 0, 1, 2, \ldots 7$

where x, y = spatial coordinates in the pel domain

u, v = coordinates in the transform domain

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u, v = 0 \\ 1 & \text{otherwise} \end{cases}$$

Although the exact method for computation of the DCT and IDCT is not subject to standardization and will not be specified by JPEG, it is probable that JPEG will adopt DCT conformance specifications which designate the accuracy to which the DCT must be computed. The DCT conformance specifications will assure that any two JPEG implementations will produce visually similar reconstructed images.

Quantization

The coefficients of the DCT are quantized to reduce their magnitude and increase the number of zero-value coefficients. The DCT coefficients are independently quantized by uniform quantizers. A uniform quantizer divides the real number line into steps of equal size as shown in Figure 2. The quantization step-size applied to each coefficient is determined from the contents of a 64-element quantization table.

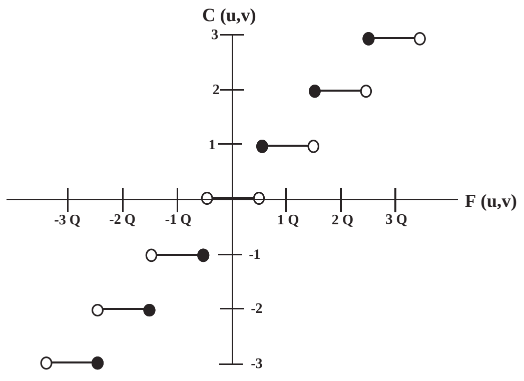


Figure 2. Uniform Quantization

The baseline process provides for up to 4 different quantization tables to be defined and assigned to separate interleaved components within a single scan of the input image. Although the values of each quantization table should ideally be determined through rigorous subjective testing which estimates the human psycho-visual thresholds for each DCT coefficient and for each color component of the input image, JPEG has developed quantization tables which work well for CCIR 601 resolution images and has published these in the informational section of the proposed standard.

DC Prediction

The DCT coefficient located in the upper-left hand corner of the transformed block represents the average spatial intensity of the block and is referred to as the “DC coefficient”. After the DCT coefficients are quantized but before they are entropy coded, DC prediction is performed. DC prediction simply means that the DC term of the previous block is subtracted from the DC term of the current block prior to encoding.

Zig-Zag Scan

Prior to entropy coding, the DCT coefficients are ordered into a one-dimensional sequence according to a “zig-zag” scan. The DC coefficient is coded first followed by AC coefficient coding proceeding in the order illustrated in Figure 3.

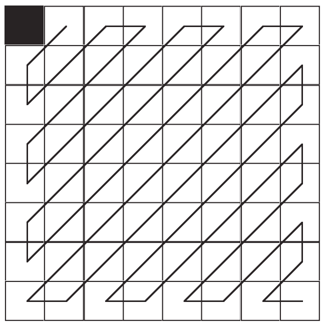


Figure 3. Zig-Zag Scan of DCT Coefficients

Entropy Coding

The quantized DCT coefficients are further compressed using entropy coding. The baseline process performs entropy coding using variable length codes (VLCs) and variable length integers (VLIs).

VLCs, commonly known as Huffman codes, compress data symbols by creating shorter codes to represent frequently occurring symbols and longer codes for occasionally occurring symbols. One reason for using VLCs is that they are easily implemented by means of lookup tables.

Separate code tables are provided for the coding of DC and AC coefficients. The following sections describe the respective coding methods used for coding DC and AC coefficients.

DC Coefficient Coding

DC prediction produces a “differential DC coefficient” which is typically small in magnitude due to the high correlation of neighboring DC coefficients. Each differential DC coefficient is encoded by a VLC which represents the number of significant bits in the DC term followed by a VLI representing the value itself. The VLC is coded by first determining the number of significant bits, SSSS, in the differential DC coefficient through the following table:

SSSS	Differential DC Value
0	0
1	-1, 1
2	-3,-2, 2,3
3	-7..-4, 4..7
4	-15..-8, 8..15
5	-31..-16, 16..31
6	-63..-32, 32..63
7	-127..-64, 64..127
8	-255..-128, 128..255
9	-511..-256, 256..511
10	-1023..-512, 512..1023
11	-2047..-1024, 1024..2047
12	-4095..-2048, 2048..4095

SSSS is then coded from the selected DC VLC table. The VLC is followed by a VLI having SSSS bits which represents the value of the differential DC coefficient itself. If the coefficient is positive, the VLI is simply the low order bits of the coefficient. If the coefficient is negative, then the VLI is the low order bits of the coefficient-1.

AC Coefficient Coding

In a similar fashion, AC coefficients are coded with alternating VLC and VLI codes. The VLC table, however, is a two-dimensional table which is indexed by a composite 8-bit value. The lower 4 bits of the 8-bit value, i.e. the column index, is the number of significant bits, SSSS, of a non-zero AC coefficient. SSSS is computed through the same table as that used for coding the DC coefficient. The higher order 4 bits, the row index, is the number of zero coefficients, NNNN, which precede the non-zero AC coefficient. The first column of the two-dimensional coding table contains codes which represent control functions. Figure 4 illustrates the general structure of the AC coding table.

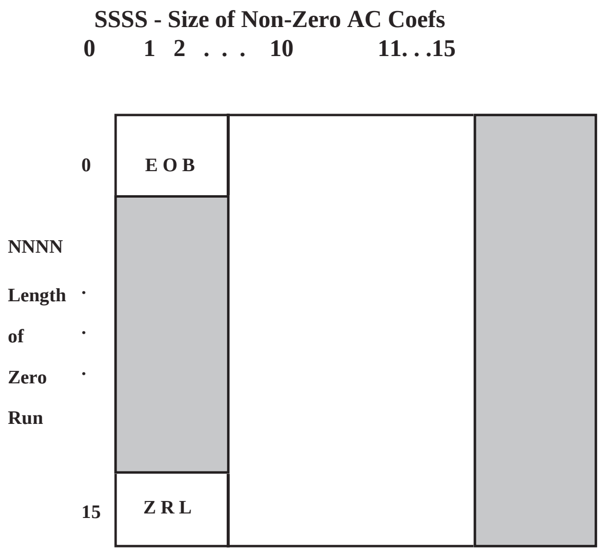


Figure 4. 2-D Run-Size Value Array for AC Coefs
The shaded portions are undefined in the baseline process

The flow chart in Figure 5 specifies the AC coefficient coding procedure. AC coefficients are coded by traversing the block in the zig-zag sequence and counting the number of zero coefficients until a non-zero AC coefficient is encountered. If the count of consecutive zero coefficients exceeds 15, then a ZRL code is coded and the zero run-length count is reset. When a non-zero AC coefficient is found, the number of significant bits in the non-zero coefficient, SSSS, is combined with the zero run-length which precedes that coefficient, NNNN, to form an index into the two-dimensional VLC table. The selected VLC is then coded. The VLC is followed by a VLI which represents the value of the AC coefficient. This process is repeated until the end of block is reached. If the last AC coefficient is zero, then an End of Block (EOB) VLC is encoded.

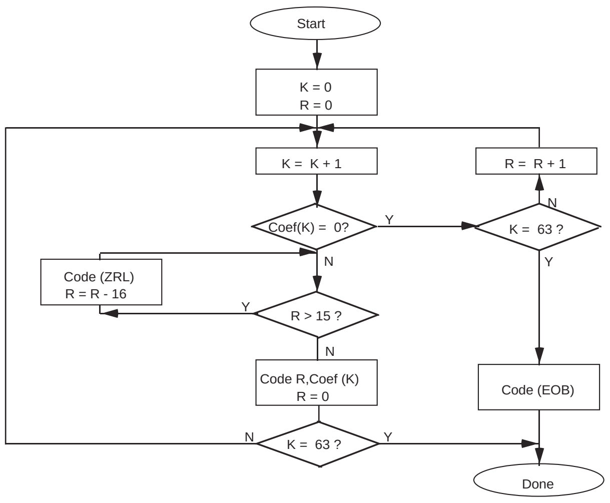


Figure 5. Encoding Procedure for AC Coefs

JPEG Lossless Processes

The JPEG lossless coding processes utilize a spatial prediction algorithm based upon a two-dimensional Differential Pulse Code Modulation (DPCM) technique. They are compatible with a wider range of input pixel precision than the DCT-based algorithms (2 to 16 bits per component). Although the primary motivation for specifying a spatial algorithm is to provide a method for lossless compression, JPEG allows for quantization of the input data, resulting in lossy compression and higher compression rates.

Although JPEG provides for use of either the Huffman or Arithmetic entropy coding models by the processes for lossless coding, only the Huffman coding model is supported by this version of TIFF. The following is a brief overview of the lossless process with Huffman coding.

Control Structure

Much of the control structure developed for the sequential DCT procedures is also used for sequential lossless coding. Either interleaved or non-interleaved data ordering may be used.

Coding Model

The coding model developed for coding the DC coefficients of the DCT is extended to allow a number of one-dimensional and two-dimensional predictors for the lossless coding function. Each component uses an independent predictor.

Prediction

Figure 6 shows the relationship between the neighboring values used for prediction and the sample being coded.

```
  |   |   |   |   |
--+---+---+---+---+--
  |   | C | B |   |
--+---+---+---+---+--
  |   | A | Y |   |
--+---+---+---+---+--
  |   |   |   |   |
```

Figure 6. Relationship between sample and prediction samples

Y is the sample to be coded and A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above.

The allowed predictors are listed in the following table.

Selection-value	Prediction
0	no prediction (differential coding)
1	A
2	B
3	C
4	A+B-C
5	A+((B-C)/2)
6	B+((A-C)/2)
7	(A+B)/2

Selection-value 0 shall only be used for differential coding in the hierarchical mode. Selections 1, 2 and 3 are one dimensional predictors and selections 4, 5, 6, and 7 are two dimensional predictors. The divide by 2 in the prediction equations is done by a arithmetic-right-shift of the integer values.

The difference between the prediction value and the input is calculated modulo 2**16. Therefore, the prediction can also be treated as a modulo 2**16 value. In the decoder the difference is decoded and added, modulo 2**16, to the prediction.

Huffman Coding of the Prediction Error

The Huffman coding procedures defined for coding the DC coefficients are used to code the modulo 2**16 differences. The table for DC coding is extended to 17 entries which allows for coding of the modulo 2**16 differences.

Point Transformation Prior to Lossless Coding

For the lossless processes only, the input image data may optionally be scaled (quantized) prior to coding by specifying a nonzero value in the point transformation parameter. Point transformation is defined to be division by a power of 2.

If the point transformation field is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by 2**Pt, where Pt is the value of the point transform signaling field. The output of the decoder is rescaled to the input range by multiplying by 2**Pt. Note that the scaling of input and output can be performed by arithmetic shifts.

Overview of the JPEG extension to TIFF

In extending the TIFF definition to include JPEG compressed data, it is necessary to note the following:

- JPEG is effective only on continuous-tone color spaces:

  Grayscale (Photometric Interpretation = 1)
  RGB (Photometric Interpretation = 2)
  CMYK (Photometric Interpretation = 5) (See the CMYK Images section.)
  YC_bC_r (Photometric Interpretation = 6) (See the YCbCr images section.)

- Color conversion to YC_bC_r is often used as part of the compression process because the chrominance components can be subsampled and compressed to a greater degree without significant visual loss of quality. Tags are defined to describe how this conversion has taken place and the degree of subsampling employed (see the YCbCr Images section).

- New tags have been defined to specify the JPEG parameters used for compression and to allow quantization tables and Huffman code tables to be incorporated into the TIFF file.

- TIFF is compatible with compressed image data which conforms to the syntax of the JPEG interchange format for compressed image data. Tags are defined which may be utilized to facilitate conversion from TIFF to interchange format.

- The PlanarConfiguration Tag is used to specify whether or not the compressed data is interleaved as defined by JPEG. For any of the JPEG DCT-based processes, the interleaved data units are coded 8x8 blocks rather than component samples.

- Alhough JPEG codes consecutive image blocks in a single contiguous bitstream, it is extremely useful to employ the concept of tiles in an image. The TIFF Tiles section defines some new tags for tiles, which should be used in place of the older tags for strips. The concept of tiling an image in both dimensions is important because JPEG hardware may be limited in the size of each block that is handled.

- Note that the nomenclature used in the TIFF specification is different from the JPEG Draft International Standarditte Draft (ISO DIS 10918-1) in some respects. The following terms should be equated when reading this Section:

TIFF name	JPEG DIS name
ImageWidth	Number of Pixels
ImageLength	Number of Lines
SamplesPerPixel	Number of Components
JPEGQTable	Quantization Table
JPEGDCTable	Huffman Table for DC coefficients
JPEGACTable	Huffman Table for AC coefficients

Strips and Tiles

The JPEG extension to TIFF has been designed to be consistent with the existing TIFF strip and tile structures and to allow quick conversion to and from the stream-oriented compressed image format defined by JPEG.

Compressed images conforming to the syntax of the JPEG interchange format can be converted to TIFF simply by defining a single strip or tile for the entire image and then concatenating the TIFF image description fields to the JPEG compressed image data. The strip or tile offset tag points directly to the start of the entropy coded data (not to a JPEG marker).

Multiple strips or tiles are supported in JPEG compressed images using restart markers. Restart markers, inserted periodically into the compressed image data, delineate image segments known as restart intervals. At the start of each restart interval, coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

In order to maximize interchangeability of TIFF files with other formats, a restriction is placed on tile height for files containing JPEG compressed image data conforming to the JPEG interchange format syntax. The restriction, imposed only when the tile width is defined to be shorter than the image width and when the JPEGInterchangeFormat Tag is present and non-zero, states that the tile height must be equal to the height of one JPEG Minimum Coded Unit (MCU). This restriction ensures that TIFF files may be converted to JPEG interchange format without undergoing decompression.

Extensions to Existing Tags

Compression

Tag = 259 (103.H)
Type = SHORT
N = 1

This Tag indicates the type of compression used. The new value is:

6 = JPEG

New Tags

JPEGProc

Tag = 512 (200.H)
Type = SHORT
N = 1

This Tag indicates the JPEG process used to produce the compressed data. The values for this tag are defined to be consistent with the numbering convention used in ISO DIS 10918-2. Two values are defined at this time.

1	= Baseline sequential process
14	= Lossless process with Huffman coding

When the lossless process with Huffman coding is selected by this Tag, the Huffman tables used to encode the image are specified by the JPEGDCTables tag, and the JPEGACTables tag is not used.

Values indicating JPEG processes other than those specified above will be defined in the future.

Not all of the tags described in this Section are relevant to the JPEG process selected by this Tag. The following table specifies the tags which are applicable to each value defined by this Tag.

Tag Name	JPEGProc =1	JPEGProc =14
JPEGInterchangeFormat	X	X
JPEGInterchangeFormatLength	X	X
JPEGRestart Interval	X	X
JPEGLosslessPredictors		X
JPEGPointTransforms		X
JPEGQTables	X	
JPEGDCTables	X	X
JPEGACTables	X	

This Tag is mandatory whenever the Compression Tag is JPEG (no default).

JPEGInterchangeFormat

Tag = 513 (201.H)

Type	= LONG
N	= 1

This Tag indicates whether or not a JPEG interchange format bitstream is present in the TIFF file. If a JPEG interchange format bitstream is present then this Tag points to the Start of Image (SOI) marker code.

If this Tag is zero or not present, a JPEG interchange format bitstream is not present.

JPEGInterchangeFormatLength

Tag	= 514 (202.H)
Type	= LONG
N	= 1

This Tag indicates the length in bytes of the JPEG interchange format bitstream. This Tag is useful for extracting the JPEG interchange format bitstream without parsing the bitstream.

This Tag is relevant only if the JPEGInterchangeFormat Tag is present and is non-zero.

JPEGRestartInterval

Tag	= 515 (203.H)
Type	= SHORT
N	= 1

This Tag indicates the length of the restart interval used in the compressed image data. The restart interval is defined as the number of Minimum Coded Units (MCUs) between restart markers.

Restart intervals are used in JPEG compressed images to provide support for multiple strips or tiles. At the start of each restart interval, coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more information about the restart interval and restart markers.

If this Tag is zero or is not present, the compressed data does not contain restart markers.

JPEGLosslessPredictors

Tag	= 517 (205.H)
Type	= SHORT
N	= SamplesPerPixel

This Tag points to a list of lossless predictor selection-values, one per component.

The allowed predictors are listed in the following table.

Selection-value	Prediction
1	A
2	B
3	C

4	A+B-C
5	A+((B-C)/2)
6	B+((A-C)/2)
7	(A+B)/2

A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above the sample to be coded respectively.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

This Tag is mandatory whenever the JPEGProc Tag specifies one of the lossless processes (no default).

JPEGPointTransforms

Tag = 518 (206.H)
Type = SHORT
N = SamplesPerPixel

This Tag points to a list of point transform values, one per component. This Tag is relevant only for lossless processes.

If the point transformation value is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by 2**Pt, where Pt is the point transform value. The output of the decoder is rescaled to the input range by multiplying by 2**Pt. Note that the scaling of input and output can be performed by arithmetic shifts.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

The default value of this Tag is 0 for each component (no scaling).

JPEGQTables

Tag = 519 (207.H)
Type = LONG
N = SamplesPerPixel

This Tag points to a list of offsets to the quantization tables, one per component.

Each table consists of 64 BYTES (one for each DCT coefficient in the 8x8 block). The quantization tables are stored in zigzag order.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables.

This Tag is mandatory whenever the JPEGProc Tag specifies a DCT-based process (no default).

JPEGDCTables

Tag = 520 (208.H)
Type = LONG
N = SamplesPerPixel

This Tag points to a list of offsets to the DC Huffman tables or the lossless Huffman tables, one per component.

The format of each table is as follows:

16 BYTES of “BITS”, indicating the number of codes of lengths 1 to 16;

Up to 17 BYTES of “VALUES”, indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables.

This Tag is mandatory for all JPEG processes (no default).

JPEGACTables

Tag	= 521 (209.H)
Type	= LONG
N	= SamplesPerPixel

This Tag points to a list of offsets to the Huffman AC tables, one per component.

The format of each table is as follows:

16 BYTES of “BITS”, indicating the number of codes of lengths 1 to 16;

Up to 256 BYTES of “VALUES”, indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables.

This Tag is mandatory whenever the JPEGProc Tag specifies a DCT-based process (no default).

Minimum Requirements for TIFF with JPEG Compression

The following table shows the minimum requirements of a TIFF file using tiling and containing JPEG data compressed with the Baseline process.

Tag = NewSubFileType (254)
Type = Long
Length = 1
Value = 0
	Single image

Tag = ImageWidth (256)
Type = Long
Length = 1
Value = ?	

Tag = ImageLength (257)
Type = Long
Length = 1
Value = ?	

Tag = BitsPerSample (258)
Type = Short
Length = SamplesPerPixel
Value = ?
	8 : Monochrome
	8,8,8 : RGB
	8,8,8 : YCbCr
	8,8,8,8 : CMYK

Tag = Compression (259)
Type = Long
Length = 1
Value = 6
	6 : JPEG compression

Tag = PhotometricInterpretation (262)
Type = Short
Length = 1
Value = ?
	0,1 : Monochrome
	2 : RGB
	5 : CMYK
	6 : YCbCr

Tag = SamplesPerPixel (277)
Type = Short
Length = 1
Value = ?
	1 : Monochrome
	3 : RGB
	3 : YCbCr
	4 : CMYK

Tag = XResolution (282)
Type = Rational
Length = 1
Value = ?	

Tag = YResolution (283)
Type = Rational
Length = 1
Value = ?	

Tag = PlanarConfiguration (284)
Type = Short
Length = 1
Value = ?
	1 : Block Interleaved
	2 : Not interleaved

Tag = ResolutionUnit (296)
Type = Short
Length = 1
Value = ?	

Tag = TileWidth (322)
Type = Short
Length = 1
Value = ?
	Multiple of 8

Tag = TileLength (323)
Type = Short
Length = 1
Value = ?
	Multiple of 8

Tag = TileOffsets (324)
Type = Long
Length = Number of tiles
Value = ?	

Tag = TileByteCounts (325)
Type = Long
Length = Number of tiles
Value = ?	

Tag = JPEGProc (512)
Type = Short
Length = 1
Value = ?
	1 : Baseline process

Tag = JPEGQTables (519)
Type = Long
Length = SamplesPerPixel
Value = ?
	Offsets to tables

Tag = JPEGDCTables (520)
Type = Long
Length = SamplesPerPixel
Value = ?
	Offsets to tables

Tag = JPEGACTables (521)
Type = Long
Length = SamplesPerPixel
Value = ?
	Offsets to tables

References

[1] Wallace, G., “Overview of the JPEG Still Picture Compression Algorithm”, Electronic Imaging East ’90.

[2] ISO/IEC DIS 10918-1, “Digital Compression and Coding of Continuous-tone Still Images”, Sept. 1991.

Section 26: CIELAB Images

What is CIELAB?

CIELAB is a color space which is colorimetric, has separate lightness and chroma channels, and is approximately perceptually uniform. It has excellent applicability for device independent manipulation of continuous tone images. These attributes make it an excellent choice for many image editing functions.

1976 CIEL*a*b* is represented as a Euclidean space with the following three quantities plotted along axes at right angles: *L** representing lightness; *a** representing the red/green axis, and *b** representing the yellow/blue axis.

The formulas for 1976 CIE *L*a*b** follow:

$$L^*=116(Y/Y_n)^{1/3}-16 \qquad \text{for } Y/Y_n > 0.008856$$
$$L^*=903.3(Y/Y_n) \qquad \text{for } Y/Y_n <= 0.008856 \qquad \text{*see note below.}$$
$$a^*=500\left[(X/X_n)^{1/3}-(Y/Y_n)^{1/3}\right]$$
$$b^*=200\left[(Y/Y_n)^{1/3}-(Z/Z_n)^{1/3}\right].$$

where X_n,Y_n, and Z_n are the CIE *X*, *Y*, and *Z* tristimulus values of an *appropriate* reference white. Also, if any of the ratios X/X_n, Y/Y_n, or Z/Z_n is equal to or less than 0.008856, it is replaced in the formulas with

$$7.787F + 16/116,$$

where *F* is X/X_n, Y/Y_n, or Z/Z_n, as appropriate (note: these low-light conditions are of no relevance for most document imaging applications). Tiff is defined such that each quantity be encoded with 8 bits. This provides 256 levels of *L** lightness; 256 levels (+/- 127) of *a**, and 256 levels (+/- 127) of *b**. Dividing the 0-100 range of *L** into 256 levels provides lightness steps that are less than half the size of a “just noticeable difference”. This eliminates banding, even under conditions of substantial tonal manipulation. Limiting the theoretically unbounded *a** and *b** ranges to +/- 127 allows encoding in 8 bits without eliminating any but the most saturated self-luminous colors. It is anticipated that the rare specialized applications requiring support of these extreme cases would be unlikely to use CIELAB anyway. All object colors, in fact all colors within the theoretical MacAdam limits, fall within the +/- 127 *a*/b** range.

The TIFF CIELAB Tags.

PhotometricInterpretation

Tag	= 262 (106)
Type	= SHORT

N = 1

8 = 1976 CIE $L^*a^*b^*$

Usage of other Tags.

BitsPerSample: 8

SamplesPerPixel - ExtraSamples: 3 for $L^*a^*b^*$, 1 implies L^* only, for monochrome data.

Compression: same as other multi-bit formats. JPEG compression applies.

PlanarConfiguration: both chunky and planar data could be supported.

WhitePoint: does not apply

PrimaryChromaticities: does not apply.

TransferFunction: does not apply

Alpha Channel information will follow the lead of other data types.

The reference white for this data type is the *perfect reflecting diffuser* (100% diffuse reflectance at all visible wavelengths). The L^* range is from 0 (perfect absorbing black) to 100 (perfect reflecting diffuse white). The a^* and b^* ranges will be represented as signed 8 bit values having the range -127 to +127.

Converting between RGB and CIELAB, a Caveat

The above CIELAB formulae are derived from CIE *XYZ*. Converting from CIELAB to *RGB* requires an additional set of formulae for converting between *RGB* and *XYZ*. For standard NTSC primaries these are:

$$\begin{array}{lclllcl} \mathbf{R} & & \mathbf{0.6070} & \mathbf{0.1740} & \mathbf{0.2000} & & \mathbf{X} \\ \mathbf{G} & * & \mathbf{0.2990} & \mathbf{0.5870} & \mathbf{0.1140} & = & \mathbf{Y} \\ \mathbf{B} & & \mathbf{0.0000} & \mathbf{0.0660} & \mathbf{1.1110} & & \mathbf{Z} \end{array}$$

Generally, D65 illumination is used and a perfect reflecting diffuser is used for the reference white.

Since CIELAB is not a directly displayable format, some conversion to RGB will be required. While look up table accelerated CIELAB to RGB conversion is certainly possible and fast, TIFF writers may choose to include a low resolution RGB subfile as an integral part of TIFF CIELAB.

Color Difference Measurements in CIELAB

The differences between two colors in L^*, a^*, and b^* are denoted by DL^*, Da^*, and Db^*, respectively, with the total (3-dimensional) color difference represented as:

$$\Delta E^*_{ab} = \left[(\Delta E^*)^2+(\Delta a^*)^2+(\Delta b^*)^2\right]^{1/2}.$$

This color difference can also be expressed in terms of L^*, C^*, and a measure of hue. In this case, h_{ab} is *not* used because it is an angular measure and cannot be combined with L^* and C^* directly. A linear-distance form of hue is used instead:

CIE 1976 *a,b hue-difference,* ΔH^*_{ab}

$$\Delta H^*_{ab} = \left[(\Delta E^*)^2 - (\Delta L^*)^2 - (\Delta C^*)^2\right]^{1/2}.$$

where D*C** is the chroma difference between the two colors. The total color difference expression using this hue-difference is:

$$\Delta E^*_{\mathrm{ab}} = \left[(\Delta L^*)^2 + (\Delta H^*)^2 + (\Delta b^*)^2\right]^{1/2}.$$

It is important to remember that color difference is 3-dimensional: much more can be learned from a DL*a*b* triplet than from a single DE value. The DL*C*H* form is often the most useful since it gives the error information in a form which has more familiar perception correlates. Caution is in order, however, when using DH* for large hue differences since it is a straight-line approximation of a curved hue distance.

The merits of CIELAB

Colorimetric.

First and foremost, CIELAB is colorimetric. It is traceable to the internationally-recognized standard CIE 1931 Standard Observer. This insures that it encodes color in a manner which is accurately modeled after the human vision system. Colors seen as matching are encoded identically, and colors seen as not matching are encoded differently. CIELAB provides an unambiguous definition of color without the necessity of additional information such as with RGB (primary chromaticities, white point, and gamma curves).

Device Independent.

Unlike RGB spaces which associate closely with physical phosphor colors, CIELAB contains no device association. CIELAB is not taylored for one device or device type at the expence of all others.

Full Color Gamut.

Any one image or imaging device usually encounters a very limited subset of the entire range of humanly perceptible color. Collectively, however, these images and devices span a much larger gamut of color. A truly versatile exchange color space should encompass all of these colors, ideally providing support for all visible color. RGB, PhotoYCC, YCbCr, and other display spaces suffer from gamut limitations which exclude significant regions of easily printable colors. CIELAB is defined for all visible color.

Efficiency

A good exchange space will maximize accuracy of translations between itself and other spaces. It will represent colors compactly for a given accuracy. These attributes are provided through visual uniformity. One of the greatest disadvantages of the classic CIE system (and RGB systems as well) is that colors within it are far from equally visually spaced. Encoding full-color images in a linear-intensity space such as the typical RGB space or, especially the XYZ space, requires a very large range (greater than 8-bits/primary) to eliminate banding artifacts. Adopting a *non*-linear RGB space improves the efficiency but not nearly to the extent as with a perceptually uniform space where these problems are nearly eliminated. A uniform space is also more efficiently compressed (see below).

Public Domain / Single Standard

CIELAB maintains no preferential attachments to any private organization. Its existence as a single standard leaves no room for ambiguity. Since 1976, CIELAB has continually gained popularity as a widely accepted and heavily used standard.

Luminance/Chrominance Separation.

The advantages for image size compression made possible by having a separate lightness or luminance channel are immense. Many such spaces exist. The degree to which the luminance information is fully isolated into a single channel is an important consideration. Recent studies (Kasson and Plouffe of IBM) support CIELAB as a leading candidate placing it above CIELUV, YIQ, YUV, Ycc, and XYZ.

Other advantages support a separate lightness or luminance channel. Tone and contrast editing and detail enhancement are most easily accomplished with such a channel. Conversion to a black and white representation is also easiest with this type of space.

When the chrominance channels are encoded as opponents as with CIELAB, there are other compression, image manipulation, and white point handling advantages.

Compressibility (Data).

Opponent spaces such as CIELAB are inherently more compressible than tristimulus spaces such as RGB. The chroma content of an image can be compressed to a greater extent, without objectionable loss, than can the lightness content. The opponent arrangement of CIELAB allows for spatial subsampling and efficent compression using JPEG.

Compressibility (Gamut).

Adjusting the color range of an image to match the capabilities of the intended output device is a critical function within computational color reproduction. Luminance/chrominance separation, especially when provided in a polar form, is very desirable for facilitating gamut compression. Accurate gamut compression in a tri-linear color space is very difficult.

CIELAB has a polar form (*metric hue angle*, and *metric chroma*, described below) which serves compression needs fairly well. Because CIELAB is not perfectly uniform, problems can arise when compressing along constant hue lines. Noticeable hue errors are sometimes introduced. This problem is no less severe with other contending color spaces.

This polar form also provides advantages for local color editing of images. The polar form is not proposed as part of the TIFF addition.

Getting the most from CIELAB

Image Editors

The advantages of image editing within a perceptually uniform polar color space are tremendous. A detailed description of these advantages extends well beyond the scope of this document. As previously mentioned, many common tonal manipulation tasks are most efficiently performed when only a single channel is affected. Edge enhancement, contrast adjustment, and general tone curve manipulation all ideally affect only the lightness component of an image.

A perceptual polar space works excellently for specifying a color range for masking purposes. As an example, a red shirt can be quickly changed to a green shirt without the tedious task of drawing an outline mask. The operation can be performed with a loosely, and quickly, drawn mask region combined with a hue (and perhaps chroma) range which encompasses the shirt's colors. The hue component of the shirt can then be adjusted leaving the lightness and chroma detail in place.

Color cast adjustment is easily realized by shifting either or both of the chroma channels over the entire image or blended in over the region of interest.

Converting from CIELAB to a device specific space

For fast conversion to an RGB display, CIELAB can be decoded using 3x3 matrixing followed by gamma correction. The computational complexity required for accurate CRT display is the same with CIELAB as with extended luminance-chrominance spaces.

Converting CIELAB for accurate printing on CMYK devices requires computational complexity no greater than with *accurate* conversion from any other colorimetric space. Gamut compression becomes one of the more significant tasks for any such conversion.

Section 27: TIFF Tags Sorted by Number

TagName	Decimal	Hex	Type	Number of values
NewSubfileType	254	FE	LONG	1
SubfileType	255	FF	SHORT	1
ImageWidth	256	100	SHORT or LONG	1
ImageLength	257	101	SHORT or LONG	1
BitsPerSample	258	102	SHORT	SamplesPerPixel
Compression	259	103	SHORT	1
Uncompressed	1			
CCITT 1D	2			
Group 3 Fax	3			
Group 4 Fax	4			
LZW	5			
JPEG	6			
PackBits	32773			
PhotometricInterpretation	262	106	SHORT	1
WhiteIsZero	0			
BlackIsZero	1			
RGB	2			
RGB Palette	3			
Transparency mask	4			
CMYK	5			
YCbCr	6			
CIELab	8			
Threshholding	263	107	SHORT	1
CellWidth	264	108	SHORT	1
CellLength	265	109	SHORT	1
FillOrder	266	10A	SHORT	1
DocumentName	269	10D	ASCII	
ImageDescription	270	10E	ASCII	
Make	271	10F	ASCII	
Model	272	110	ASCII	
StripOffsets	273	111	SHORT or LONG	StripsPerImage
Orientation	274	112	SHORT	1
SamplesPerPixel	277	115	SHORT	1
RowsPerStrip	278	116	SHORT or LONG	1
StripByteCounts	279	117	LONG or SHORT	StripsPerImage
MinSampleValue	280	118	SHORT	SamplesPerPixel
MaxSampleValue	281	119	SHORT	SamplesPerPixel
XResolution	282	11A	RATIONAL	1
YResolution	283	11B	RATIONAL	1
PlanarConfiguration	284	11C	SHORT	1
PageName	285	11D	ASCII	
XPosition	286	11E	RATIONAL	
YPosition	287	11F	RATIONAL	
FreeOffsets	288	120	LONG	
FreeByteCounts	289	121	LONG	
GrayResponseUnit	290	122	SHORT	1
GrayResponseCurve	291	123	SHORT	2**BitsPerSample
Group3Options	292	124	LONG	1
Group4Options	293	125	LONG	1
ResolutionUnit	296	128	SHORT	1

PageNumber	297	129	SHORT	2
TransferFunction	301	12D	SHORT	{1 or SamplesPerPixel} * (2 ** BitsPerSample)
Software	305	131	ASCII	
DateTime	306	132	ASCII	20
Artist	315	13B	ASCII	
HostComputer	316	13C	ASCII	
Predictor	317	13D	SHORT	1
WhitePoint	318	13E	RATIONAL	2
PrimaryChromaticities	319	13F	RATIONAL	6
ColorMap	320	140	SHORT	3 * (2**BitsPerSample)
HalftoneHints	321	141	SHORT	2
TileWidth	322	142	SHORT or LONG	1
TileLength	323	143	SHORT or LONG	1
TileOffsets	324	144	LONG	TilesPerImage
TileByteCounts	325	145	SHORT or LONG	TilesPerImage
InkSet	332	14C	SHORT	1
InkNames	333	14D	ASCII	total number of characters in all the ink name strings, including the zeros
DotRange	336	150	BYTE or SHORT	2, or 2*SamplesPerPixel
TargetPrinter	337	151	ASCII	any
ExtraSamples	338	152	BYTE	number of extra components per pixel
SampleFormat	339	153	SHORT	1
SMinSampleValue	340	154	Any	SamplesPerPixel
SMaxSampleValue	341	155	Any	SamplesPerPixel
JPEGProc	512	200	SHORT	1
JPEGInterchangeFormat	513	201	LONG	1
JPEGInterchangeFormatLength	514	202	LONG	1
JPEGRestartInterval	515	203	SHORT	1
JPEGLosslessPredictors	517	205	SHORT	SamplesPerPixel
JPEGPointTransforms	518	206	SHORT	SamplesPerPixel
JPEGQTables	519	207	LONG	SamplesPerPixel
JPEGDCTables	520	208	LONG	SamplesPerPixel
JPEGACTables	521	209	LONG	SamplesPerPixel
YCbCrCoefficients	529	211	RATIONAL	3
YCbCrSubSampling	530	212	SHORT	2
YCbCrPositioning	531	213	SHORT	1
ReferenceBlackWhite	532	214	LONG	2 * SamplesPerPixel

Index