Preface

This memorandum has been prepared jointly by Aldus and Microsoft in conjunction with leading scanner vendors and other interested parties. This document does not represent a commitment on the part of either Microsoft or Aldus to provide support for this file format in any application. When responding to specific issues raised in this memo, or when requesting additional tag or field assignments, please address your correspondence to either:

Developers_ Desk
Aldus Corporation
411 First Ave. South
Suite 200
Seattle, WA 98104
(206) 622-5500

Windows Marketing Group
Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717
(206) 882-8080

Revision Notes

This revision replaces _TIFF Revision 4._ Sections in italics are new or substantially changed in this revision. Also new, but not in italics, are Appendices F, G, and H.

The major enhancements in TIFF 5.0 are:

1. Compression of grayscale and color images, for better disk space utilization. See Appendix F.

2. TIFF Classes_restricted TIFF subsets that can simplify the job of the TIFF implementor. You may wish to scan Appendix G before reading the rest of this document. In fact, you may want to use Appendix G as your main guide, and refer back to the main body of the specification as needed for details concerning TIFF structures and field definitions.

3. Support for _palette color_ images. See the TIFF Class P description in Appendix G, and the new ColorMap field description.

4. Two new tags that can be used to more fully define the characteristics of full color RGB data, and thereby potentially improve the quality of color image reproduction. See Appendix H.

The organization of the document has also changed slightly. In particular, the tags are listed in alphabetical order, within several categories, in the main body of the specification.

As always, every attempt has been made to add functionality in such a way as to minimize incompatibility problems with older TIFF software. In particular, many TIFF 5.0 files will be readable even by older applications that assume TIFF 4.0 or an earlier version of the specification. One exception is with files that use the new TIFF 5.0 LZW compression scheme. Old applications will have to give up in this case, of course, and will do so _gracefully_ if they have been following the rules.

We are grateful to all of the draft reviewers for their suggestions. Especially helpful were Herb Weiner of Kitchen Wisdom Publishing Company, Brad Pillow of TrueVision, and engineers from Hewlett Packard and Quark. Chris Sears of Magenta Graphics provided information which is included as Appendix H.

Abstract

This document describes TIFF, a tag based file format that is designed to promote the interchange of digital image data.

The fields were defined primarily with desktop publishing and related applications in mind, although it is possible that other sorts of imaging applications may find TIFF to be useful.

The general scenario for which TIFF was invented assumes that applications software for scanning or painting creates a TIFF file, which can then be read and incorporated into a _document_ or _publication_ by an application such as a desktop publishing package.

TIFF is not a printer language or page description language, nor is it intended to be a general document interchange standard. 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 similar devices. TIFF is therefore designed to be a superset of existing image file formats for _desktop_ scanners (and paint programs and anything else that produces images with pixels in them) and will be enhanced on a continuing basis as new capabilities arise. A high priority has been given to structuring the data in such a way as to minimize the pain of future additions. TIFF was designed to be an extensible interchange format.

Although TIFF is claimed to be in some sense a rich format, it can easily be used for simple scanners and applications as well, since the application developer need only be concerned with the

capabilities that he requires.

TIFF is intended to be independent of specific operating systems, filing systems, compilers, and processors. The only significant assumption is that the storage medium supports something like a _file,_ defined as 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. Since TIFF uses pointers (byte offsets) quite liberally, a TIFF file is most easily read from a random access device such as a hard disk or flexible diskette, although it should be possible to read and write TIFF files on magnetic tape.

The recommended MS-DOS, UNIX, and OS/2 file extension for TIFF files is _.TIF._ The recommended Macintosh filetype is _TIFF_. Suggestions for conventions in other computing environments are welcome.

1) Structure

In TIFF, individual fields are identified with a unique tag. This allows particular fields to be present or absent from the file as required by the application. For an explanation of the rationale behind using a tag structure format, see Appendix A.

A TIFF file begins with an 8-byte _image file header_ that points to one or more _image file directories._ The image file directories contain information about the images, as well as pointers to the actual image data.

See Figure 1.

We will now describe these structures in more detail.

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_ (hex 4949)
_MM_ (hex 4D4D)

In the _II_ format, byte order is always from least significant to most significant, for both 16-bit and 32-bit

integers. In the _MM_ format, byte order is always from most significant to least significant, for both 16-bit and 32-bit integers. In both formats, character strings are stored into sequential byte locations.

All TIFF readers should support both byte orders_see Appendix G.

Bytes 2-3 The second word of the file is the TIFF _version number._ This number, 42 (2A in hex), is not to be equated with the current Revision of the TIFF specification. In fact, 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 have a real version/revision number. This was an explicit, 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 fields to have a permanent and well-defined meaning, so that _older_ software can usually read _newer_ TIFF files. The bottom line is lower software release 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 for the field.
Bytes 2-3 contain the field Type.
Bytes 4-7 contain the Length (_Count_ might have been a better term) of the field.
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.

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. For a numerically ordered list of tags, see

Appendix E. 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.

The Length is specified in terms of the data type, not the total number of bytes. A single 16-bit word (SHORT) has a Length of 1, not 2, for example. The data types and their lengths are described below:

1 = BYTE An 8-bit unsigned integer.
2 = ASCII 8-bit bytes that store ASCII codes; 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 ImageWidth and similar fields. See Appendix G for specific recommendations.

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 _sub-image_ that is somehow related to the main image, such as a reduced resolution version of the image.

If you have not already done so, you may wish to turn to Appendix G to study the sample TIFF images.

2) Definitions

Note that the TIFF structure as described in the previous section is not specific to imaging applications in any way. It is only

the definitions of the fields themselves that jointly describe an image.

Before we begin defining the fields, we will define some basic concepts. An image is defined to be a rectangular array of _pixels,_ each of which consists of one or more _samples._ With monochromatic data, we have one sample per pixel, and _sample_ and _pixel_ can be used interchangeably. RGB color data contains three samples per pixel.

3) The Fields

This section describes the fields defined in this version of TIFF. More fields may be added in future versions_if possible they will be added in such a way so as not to break old software that encounters a newer TIFF file.

The documentation for each field contains the name of the field (quite arbitrary, but convenient), the Tag value, the field Type, the Number of Values (N) expected, 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, short of refusing to import the file. For example, if a writer does not write out a PhotometricInterpretation field, some applications will interpret the image _correctly,_ and others will display the image inverted. This is not a good situation, and writers should be careful not to let it happen.

The fields are grouped into several categories: basic, informational, facsimile, document storage and retrieval, and no longer recommended. A future version of the specification may pull some of these categories into separate companion documents.

Many fields are described in this document, but most are not _required._ See Appendix G for a list of required fields, as well as examples of how to combine fields into valid and useful TIFF files.

Basic Fields

Basic fields are fields that are fundamental to the pixel architecture or visual characteristics of an image.

BitsPerSample
Tag = 258 (102)
Type = SHORT

N = SamplesPerPixel

Number of bits per sample. Note that this tag allows a different number of bits per sample for each sample corresponding to a pixel. For example, RGB color data could use a different number of bits per sample for each of the three color planes. Most RGB files will have the same number of BitsPerSample for each sample. 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.

ColorMap
Tag = 320 (140)
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.

The ColorResponseCurves field may be used in conjunction with ColorMap to further refine the meaning of the RGB triplets in the ColorMap. However, the ColorResponseCurves default should be sufficient in most cases.

See also PhotometricInterpretation_palette color.

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

ColorResponseCurves
Tag = 301 (12D)
Type = SHORT
N = 3 * (2**BitsPerSample)

This tag defines three color response curves, one each for Red, Green and Blue color information. The Red entries come first, followed by the Green entries, followed by the Blue entries. The

length of each subcurve is 2**BitsPerSample, using the BitsPerSample value corresponding to the respective primary. 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. Therefore, a ColorResponseCurve entry for RGB data where each of the samples is 8 bits deep would have 3 * 256 entries, each consisting of a SHORT.

The purpose of the color response curves is to refine the content of RGB color images.

See Appendix H, section VII, for further information.

Default: curves based on the NTSC recommended gamma of 2.2.

Compression
Tag = 259 (103)
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 bytes are stored as an array of type BYTE, for BitsPerSample <= 8, SHORT if BitsPerSample > 8 and <= 16, and LONG if BitsPerSample > 16 and <= 32. 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 have the least significant bytes preceeding the most significant bytes while _MM_ format files will have the opposite.

If the number of bits per sample 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.

Rows are required to begin on byte boundaries. The number of bytes per row is therefore (ImageWidth * SamplesPerPixel * BitsPerSample + 7) / 8, assuming integer arithmetic, for PlanarConfiguration=1. Bytes per row is (ImageWidth * BitsPerSample + 7) / 8 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 Appendix B: _Data Compression -- Scheme 2._ BitsPerSample must be 1, since this type of compression is defined only for bilevel images.

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. Therefore, the normal PhotometricInterpretation for those compression types is 0 (WhiteIsZero). 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.

5 = LZW Compression, for grayscale, mapped color, and full color images. See Appendix F.

32773 = PackBits compression, a simple byte oriented run length scheme for 1-bit images. See Appendix C.

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

Default = 1.

GrayResponseCurve
Tag = 291 (123)
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 doesnÆt 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 permissable 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 Appendix G.

If both GrayResponseCurve and PhotometricInterpretation fields exist in the IFD, GrayResponseCurve values override the PhotometricInterpretation value. But it is a good idea to write out both, since some applications do not yet pay attention to the GrayResponseCurve.

Writers may wish to purchase a Kodak Reflection Density Guide, catalog number 146 5947, available for $10 or so at prepress supply houses, to help them figure out reasonable density values for their scanner or frame grabber. If that sounds like too much work, we recommend a curve that is linear in intensity/reflectance. To compute reflectance from density: R = 1 / pow(10,D). To compute density from reflectance: D = log10 (1/R). A typical 4-bit GrayResponseCurve may look therefore something like: 2000, 1177, 875, 699, 574, 477, 398, 331, 273, 222, 176, 135, 97, 62, 30, 0, assuming GrayResponseUnit=3. Such a curve would be consistent with PhotometricInterpretation=1.

See also GrayResponseUnit, PhotometricInterpretation, ColorMap.

GrayResponseUnit
Tag = 290 (122)
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.

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

The image_s length (height) in pixels (Y: vertical). The number of rows (sometimes described as _scan lines") in the image. See also ImageWidth.

No default.

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

The image_s width, in pixels (X: horizontal). The number of columns in the image. See also ImageLength.

No default.

NewSubfileType
Tag = 254 (FE)
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 pretty much independent of each other. For example, it be useful to have four images in a single TIFF file: a full resolution image, a reduced resolution image, a transparency mask for the

full resolution image, and a transparency mask for the reduced resolution image. Each of the four images would have a different value for the NewSubfileType field.

Default is 0.

PhotometricInterpretation
Tag = 262 (106)
Type = SHORT
N = 1

0 = For bilevel and grayscale images: 0 is imaged as white. 2**BitsPerSample-1 is imaged as black. If GrayResponseCurve exists, it overrides the PhotometricInterpretation value, although it is safer to make them match, since some old applications may still be ignoring GrayResponseCurve. This is the normal value for Compression=2.

1 = For bilevel and grayscale images: 0 is imaged as black. 2**BitsPerSample-1 is imaged as white. If GrayResponseCurve exists, it overrides the PhotometricInterpretation value, although it is safer to make them match, since some old applications may still be ignoring GrayResponseCurve. 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 samples, 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 samples. For PlanarConfiguration = 1, the samples are stored in the indicated order: first Red, then Green, then Blue. For PlanarConfiguration = 2, the StripOffsets for the sample planes are stored in the indicated order: first the Red sample plane StripOffsets, then the Green plane StripOffsets, then the Blue plane StripOffsets.

The ColorResponseCurves field may be used to globally refine or alter the color balance of an RGB image without having to change the values of the pixels themselves.

3="Palette color._ In this mode, a color is described with a single sample. The sample is used as an index into ColorMap. The sample 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, the color response curves 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. The Transparency Mask must have the same ImageLength and ImageWidth as the main image.

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.

It is possible to generalize the notion of a transparency mask to include partial transparency, but it is not clear that such information would be useful to a desktop publishing program.

No default. That means that if you care if your image is displayed and printed as _normal_ vs _inverted,_ you must write out this field. Do not rely on applications defaulting to what you want! PhotometricInterpretation = 1 is recommended for bilevel (except for Compression=2) and grayscale images, due to popular user interfaces for changing the brightness and contrast of images.

PlanarConfiguration
Tag = 284 (11C)
Type = SHORT
N = 1

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

2 = The samples are stored in separate _sample 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 sample plane. For example, RGB data is stored with the Red samples in one sample plane, the Green in another, and the Blue in another.

If SamplesPerPixel is 1, PlanarConfiguration is irrelevant, and should not be included.
Default is 1. See also BitsPerSample, SamplesPerPixel.

Predictor
Tag = 317 (13D)
Type = SHORT

N = 1

To be used when Compression=5 (LZW). See Appendix F.

1 = No prediction scheme used before coding.

Default is 1.

ResolutionUnit
Tag = 296 (128)
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 quite arbitrary.)
2 = Inch.
3 = Centimeter.

Default is 2. See also XResolution, YResolution.

RowsPerStrip
Tag = 278 (116)
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.

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. See Appendix G for further recommendations.

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)
Type = SHORT
N = 1

The number of samples 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)
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. 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)
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 must exist.

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. See Appendix G for further recommendations.

No default. See also StripByteCounts, RowsPerStrip.

XResolution
Tag = 282 (11A)
Type = RATIONAL
N = 1

The number of pixels per ResolutionUnit in the X direction, i.e., in the ImageWidth direction. It is, of course, not mandatory that the image be actually 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)
Type = RATIONAL
N = 1

The number of pixels per ResolutionUnit in the Y direction, i.e., 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)
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)
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)
Type = ASCII

ENIAC, or whatever.

See also Make, Model, Software.

ImageDescription
Tag = 270 (10E)
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)
Type = ASCII

Manufacturer of the scanner, video digitizer, or whatever.

See also Model, Software.

Model
Tag = 272 (110)
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)
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.

Facsimile Fields

Facsimile fields may be useful if you are using TIFF to store facsimile messages in _raw_ form. They are not recommended for use in interchange with desktop publishing applications.

Compression (a basic tag)
Tag = 259 (103)
Type = SHORT
N   = 1

3 = Facsimile-compatible CCITT Group 3, exactly as specified in _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. Each strip must begin on a byte boundary. (But recall that an image can be a single strip.) Rows that are not the first row of a strip are not required to begin on a byte boundary. The data is stored as bytes, not words_byte-reversal is not allowed. See the Group3Options field for Group 3 options such as 1D vs 2D coding.

4 = Facsimile-compatible CCITT Group 4, exactly as specified in _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. Each strip must begin on a byte boundary. Rows that are not the first row of a strip are not required to begin on a byte boundary. The data is stored as bytes, not words. See the Group4Options field for Group 4 options.

Group3Options
Tag = 292 (124)
Type = LONG
N   = 1

See Compression=3. 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. It is probably not safe to try to read the file if any bit of this field is set that you don_t know the meaning of.

Bit 0 is 1 for 2-dimensional coding (else 1-dimensional is assumed). For 2-D coding, if more than one strip is specified, each strip must begin with a 1-dimensionally coded line. That

is, RowsPerStrip should be a multiple of _Parameter K_ as documented in the CCITT specification.

Bit 1 is 1 if uncompressed mode is used.

Bit 2 is 1 if fill bits have been added as necessary before

EOL codes such that EOL always ends on a byte boundary, thus ensuring an eol-sequence of a 1 byte preceded by a zero nibble: xxxx-0000 0000-0001.

Default is 0, for basic 1-dimensional coding. See also Compression.

Group4Options
Tag = 293 (125)
Type = LONG
N = 1

See Compression=4. 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. It is probably not safe to try to read the file if any bit of this field is set that you don_t know the meaning of. Gray scale and color coding schemes are under study, and will be added when finalized.

For 2-D coding, each strip is encoded as if it were a separate image. In particular, each strip begins on a byte boundary; and the coding for the first row of a strip is encoded independently of the previous row, using horizontal codes, as if the previous row is entirely white. Each strip ends with the 24-bit end-of-facsimile block (EOFB).

Bit 0 is unused.
Bit 1 is 1 if uncompressed mode is used.

Default is 0, for basic 2-dimensional binary compression. See also Compression.

Document Storage and Retrieval Fields

These fields may be useful for document storage and retrieval applications. They are not recommended for use in interchange with desktop publishing applications.

DocumentName
Tag = 269 (10D)
Type = ASCII

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

See also PageName.

PageName
Tag = 285 (11D)
Type = ASCII

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

See also DocumentName.

No default.

PageNumber
Tag = 297 (129)
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)
Type = RATIONAL

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)
Type = RATIONAL

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.

No Longer Recommended

These fields are not recommended except perhaps for local use. They should not be used for image interchange. They have either been superseded by other fields, have been found to have serious drawbacks, or are simply not as useful as once thought. They may be dropped entirely from a future revision of the specification.

CellLength
Tag = 265 (109)
Type = SHORT
N = 1

The length, in 1-bit samples, 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)
Type = SHORT
N = 1

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

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

FillOrder
Tag = 266 (10A)
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 FillOrder=2 for private (non-interchange) use only.

Default is FillOrder = 1.

FreeByteCounts
Tag = 289 (121)
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)
Type = LONG

For each _free block_ in the file, its byte offset.

See also FreeByteCounts.

MaxSampleValue
Tag = 281 (119)
Type = SHORT
N = SamplesPerPixel

The maximum used sample 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)
Type = SHORT
N = SamplesPerPixel

The minimum used sample 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)
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)
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. It is extremely costly for most readers to perform image rotation _on the fly,_ i.e., when importing and printing; and users of most desktop publishing applications do not expect a file imported by the application to be altered permanently in any way.

Threshholding
Tag = 263 (107)
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.
4) Private Fields

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 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. If that is not enough, feel free to ask for more. 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 a block of 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 problems some day.

5) Image File Format Issues

In the quest to give users no reason NOT to buy a product, some scanning and image editing applications overwhelm users with an incredible number of _Save As..._ options. Let_s get rid of as many of these as we possibly can. For example, a single TIFF choice should suffice once most major readers are supporting the three TIFF compression schemes; then writers can always compress. And given TIFF_s flexibility, including private tag and image editing support features, there does not seem to be any legitimate reason for continuing to write image files using proprietary formats.

Along the same lines, there is no excuse for making a user have to know the file format of a file that is to be read by an application program. TIFF files, as well as most other file formats, contain sufficient information to enable software to automatically and reliably distinguish one type of file from another.

6) For Further Information

Contact the Aldus Developers_ Desk for sample TIFF files, source code fragments, and a list of features that are currently supported in Aldus products. The Aldus Developers_ Desk is the current _TIFF administrator._

Various TIFF related aids are found in Microsoft_s Windows Developers Tookit for developers writing Windows applications.

Finally, a number of scanner vendors are providing various TIFF services, such as helping to distribute the TIFF specification and answering TIFF questions. Contact the appropriate product manager or developer support service group.

Appendix A: Tag Structure Rationale

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.

Appendix B: Data Compression_Scheme 2

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 Appendix. 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.

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
```

Appendix C: Data Compression_Scheme 32773_ _PackBits_

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. Compression type 5 is typically very effective for 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 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).

Appendix D

Appendix D has been deleted. It formerly contained guidelines for passing TIFF files on the Microsoft Windows Clipboard. This was judged to not be a good idea, in light of the ever-increasing size of scanned images. Applications are instead encouraged to employ file-based mechanisms to exchange TIFF data. Aldus_ PageMaker, for example, implements a _File Place_ command to allow TIFF files to be imported.

Appendix E: Numerical List of TIFF Tags

NewSubfileType
Tag = 254 (FE)
Type = LONG
N = 1

SubfileType
Tag = 255 (FF)
Type = SHORT
N = 1

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

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

BitsPerSample

```
Tag  = 258  (102)
Type = SHORT
N    = SamplesPerPixel

Compression
Tag  = 259  (103)
Type = SHORT
N    = 1

PhotometricInterpretation
Tag  = 262  (106)
Type = SHORT
N    = 1


Threshholding
Tag  = 263  (107)
Type = SHORT
N    = 1

CellWidth
Tag  = 264  (108)
Type = SHORT
N    = 1

CellLength
Tag  = 265  (109)
Type = SHORT
N    = 1
```



```
FillOrder
Tag  = 266  (10A)
Type = SHORT
N    = 1

DocumentName
Tag  = 269  (10D)
Type = ASCII

ImageDescription
Tag  = 270 (10E)
Type = ASCII

Make
Tag  = 271  (10F)
Type = ASCII

Model
```

Tag = 272 (110)
Type = ASCII

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

Orientation
Tag = 274 (112)
Type = SHORT
N   = 1

SamplesPerPixel
Tag = 277 (115)
Type = SHORT
N   = 1

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

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

MinSampleValue
Tag = 280 (118)
Type = SHORT
N   = SamplesPerPixel

MaxSampleValue
Tag = 281 (119)
Type = SHORT
N   = SamplesPerPixel

XResolution
Tag = 282 (11A)
Type = RATIONAL
N   = 1

YResolution

```
Tag  = 283  (11B)
Type = RATIONAL
N    = 1

PlanarConfiguration
Tag  = 284  (11C)
Type = SHORT
N    = 1

PageName
Tag  = 285  (11D)
Type = ASCII


XPosition
Tag  = 286  (11E)
Type = RATIONAL

YPosition
Tag  = 287  (11F)
Type = RATIONAL

FreeOffsets
Tag  = 288  (120)
Type = LONG

FreeByteCounts
Tag  = 289  (121)
Type = LONG

GrayResponseUnit
Tag  = 290 (122)
Type = SHORT
N    = 1

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

Group3Options
Tag  = 292  (124)
```



```
Type = LONG
N    = 1

Group4Options
Tag  =  293  (125)
Type = LONG
```

```
N    = 1

ResolutionUnit
Tag  = 296 (128)
Type = SHORT
N    = 1

PageNumber
Tag  = 297  (129)
Type = SHORT
N    = 2

ColorResponseCurves
Tag  = 301 (12D)
Type = SHORT
N    = 3 * (2**BitsPerSample)

Software
Tag  = 305  (131)
Type = ASCII

DateTime
Tag  = 306  (132)
Type = ASCII
N    = 20

Artist
Tag  = 315  (13B)
Type = ASCII

HostComputer
Tag  = 316  (13C)
Type = ASCII

Predictor
Tag  = 317 (13D)
Type = SHORT
N    = 1

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

PrimaryChromaticities
Tag  = 319 (13F)
Type = RATIONAL
N    = 6
```

```
ColorMap
Tag  = 320 (140)
Type = SHORT
N    = 3 * (2**BitsPerSample)
```

Appendix F: Data Compression_Scheme 5_LZW Compression

Abstract

This document describes an adaptive compression scheme for raster images.

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 Appendix 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 sample.
- 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);
_ = the empty string;
for each character in the strip {
     K = GetNextCharacter();
     if _+K is in the string table {
```



```
          _ = _+K;  /* string concatenation */
     } else {
          WriteCode (CodeFromString(_));
          AddTableEntry(_+K);
          _ = K;
     }
} /* end of for loop */
WriteCode (CodeFromString(_));
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.

_ 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. _K is then simply <7>, since _ 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 _ equal to <7>, and go to the top of the loop.

Read Pixel 1 into K. Does _K (<7><7>) exist in the string table? No, so we get to do some real work. We write the code associated with _ to output (write <7> to output), and add _K (<7><7>) to

the table as entry 258. Store K (<7>) into _. 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 _K (<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 _, so that _ is now <7><7>.

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

Back at the top of the loop. We read Pixel 4 into K. Does _K (<8><8>) exist in the string table? No, so write the code

associated with _ (<8>) to output, and add _K to the table as entry 260. Store K (<8>) into _.

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. But when we add table entry 512, we must switch to 10-bit codes. Likewise, we switch to 11-bit codes at 1024, and 12-bit codes at 2048. 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.

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, _ 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)+Firs
tChar(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 #511 is stored into the table.

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 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 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.

However, it is possible that a _prediction_ stage like differencing may exist which is effective over a broad range of images. If such a scheme is found, it may be incorporated in the next major TIFF revision. If so, a new value will be defined for the new _Predictor_ TIFF tag. Therefore, 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.

Acknowledgements

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. Joff Morgan and Eric Robinson of Aldus were each instrumental in their own way in getting LZW off the ground.

Appendix G: TIFF Classes

Rationale

TIFF was designed to make life easier for scanner vendors, desktop publishing software developers, and users of these two classes of products, by reducing the proliferation of proprietary scanned image formats. It has succeeded far beyond our expectations in this respect. But we had expected that TIFF would be of interest to only a dozen or so scanner vendors (there weren_t any more than that in 1985), and another dozen or so desktop publishing software vendors. This turned out to be a gross underestimate. The only problem with this sort of success is that TIFF was designed to be powerful and flexible, at the expense of simplicity. It takes a fair amount of effort to handle all the options currently defined in this specification (probably no application does a complete job), and that is currently the only way you can be sure that you will be able to import any TIFF image, since there are so many image-generating applications out there now.

So here is an attempt to channel some of the flexibility of TIFF into more restrictive paths, using what we have learned in the past two years about which options are the most useful. Such an undertaking is of course filled with fairly arbitrary decisions. But the result is that writers can more easily write files that will be successfully read by a wide variety of applications, and readers can know when they can stop adding TIFF features.

The price we pay for TIFF Classes is some loss in the ability to adapt. Once we establish the requirements for a TIFF Class, we can never add new requirements, since old software would not know about these new requirements. (The best we can do at that point is establish new TIFF Classes. But the problem with that is that we could quickly have too many TIFF Classes.) So we must believe that we know what we are doing in establishing these Classes. If we do not, any mistakes will be expensive.

Overview

Four TIFF Classes have been defined:

- Class B for bilevel (1-bit) images
- Class G for grayscale images
- Class P for palette color images
- Class R for RGB full color images

To save time and space, we will usually say _TIFF B_, _TIFF G_, _TIFF P,_ and _TIFF R._ If we are talking about all four types, we may write _TIFF X._

(Note to fax people: if you are interested in a fax TIFF F Class, please get together and decide what should be in TIFF Class F files. Let us know if we can help in any way. When you have decided, send us your results, so that we can include the information here.)

Core Requirements

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

General Requirements

The following are required characteristics of all TIFF Class X files.

Where there are options, TIFF X writers can do whichever one they want, though we will often recommend a particular choice, but TIFF X readers must be able to handle all of them. Please pay close attention to the recommendations. It is possible that at some point in the future, new and even-simpler TIFF classes will be defined that include only recommended features.

You will need to read at least the first three sections of the main specification in order to fully understand the following discussion.

Defaults. TIFF X 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 X readers must be prepared to handle either situation.

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

_MM_ and æIIÆ byte order. TIFF X readers must be able to handle both byte orders. TIFF writers can do whichever is most convenient or efficient. Images are crossing the IBM PC/Macintosh boundary (and others as well) with a surprisingly high frequency. We could force writers to all use the same byte order, but preliminary evidence indicates that this will cause problems when we start seeing greater-than-8-bit images. Reversing bytes while scanning could well slow down the scanning process enough to cause the scanning mechanism to stop, which tends to create image quality problems.

Multiple subfiles. TIFF X 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

X writers must be sure to write a long word of 0 after the last IFD (this is the standard way of signalling that this IFD was the last one) as indicated in the TIFF structure discussion.

If a TIFF X writer writes multiple subfiles, the first one must be the full resolution image. Subsequent subimages, such as reduced resolution images and transparency masks, 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 IFDÆs before deciding how to proceed.

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

Required Fields

NewSubfileType. LONG. Recommended but not required.

ImageWidth. SHORT or LONG. (That is, both _SHORT_ and _LONG_ TIFF data types are allowed, and must be handled properly by readers. TIFF writers can use either.) TIFF X 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. Recommendation: use 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. Recommendation 2: use LONG.

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

StripByteCounts. SHORT or LONG. Many existing TIFF images do not contain StripByteCounts, because, in a strict sense, they are unnecessary. It is possible to write an efficient TIFF reader that does not need to know in advance the exact size of a compressed strip. But it does make things considerably more complicated, so we will require StripByteCounts in TIFF X files. Recommendation: use SHORT, since strips are not supposed to be very large.

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

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

TIFF Class B - Bilevel

Required (in addition to the above core requirements)

The following fields and values are required for TIFF B files, in addition to the fields required for all TIFF X images (see above).

SamplesPerPixel = 1. SHORT. (Since this is the default, the field need not be present. The same thing holds for other required TIFF X fields that have defaults.)

BitsPerSample = 1. SHORT.

Compression = 1, 2 (CCITT 1D), or 32773 (PackBits). SHORT. TIFF B readers must handle all three. Recommendation: use PackBits. It is simple, effective, fast, and has a good worst-case behavior. CCITT 1D is definitely more effective in some situations, such as scanning a page of body text, but is tough to implement and test, fairly slow, and has a poor worst-case behavior. Besides, scanning a page of 12 point text is not very useful for publishing applications, unless the image data is turned into ASCII text via OCR software, which is outside the scope of TIFF.

PhotometricInterpretation = 0 or 1. SHORT.

A Sample TIFF B Image

```
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 TIFF B 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 X readers must safely skip over these fields if they do not want to use the information. And TIFF X readers must not require that such fields be present.

4. Just for fun, 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. Incidentally, there is no requirement that the image data follows the IFD information. Just the follow the pointers, whether they be IFD pointers, field pointers, or Strip Offsets.

TIFF Class G - Grayscale

Required (in addition to the above core requirements)

SamplesPerPixel = 1. SHORT.

BitsPerSample = 4, 8. SHORT. There seems to be little justification for working with grayscale images shallower than 4 bits, and 5-bit , 6-bit, and 7-bit images can easily be stored as 8-bit images, as long as you can compress the _unused_ bit planes without penalty. And we can do just that with LZW (Compression = 5.)

Compression = 1 or 5 (LZW). SHORT. Recommendation: use 5, since LZW decompression is turning out to be quite fast.

PhotometricInterpretation = 0 or 1. SHORT. Recommendation: use 1, due to popular user interfaces for adjusting brightness and contrast.

TIFF Class P - Palette Color

Required (in addition to the above core requirements)

SamplesPerPixel = 1. SHORT. We use each pixel value as an index into all three color tables in ColorMap.

BitsPerSample = 1,2,3,4,5,6,7, or 8. SHORT. 1,2,3,4, and 8 are probably the most common, but as long as we are doing that, the rest come pretty much for free.

Compression = 1 or 5. SHORT.

PhotometricInterpretation = 3 (Palette Color). SHORT.

ColorMap. SHORT.

Note that bilevel and grayscale images can be represented as special cases of palette color images. As soon as enough major applications support palette color images, we may want to start getting rid of distinctions between bilevel, grayscale, and palette color images.

TIFF Class R - RGB Full Color

Required (in addition to the above Core Requirements)

SamplesPerPixel = 3. SHORT. One sample each for Red, Green, and Blue.

BitsPerSample = 8,8,8. SHORT. Shallower samples can easily be stored as 8-bit samples with no penalty if the data is compressed with LZW. And evidence to date indicates that images deeper than 8 bits per sample are not worth the extra work, even in the most demanding publishing applications.

PlanarConfiguration = 1 or 2. SHORT. Recommendation: use 1.

Compression = 1 or 5. SHORT.

PhotometricInterpretation = 2 (RGB). SHORT.

Recommended

Recommended for TIFF Class R, but not required, are the new (as of Revision 5.0) colorimetric information tags. See Appendix H.

Conformance and User Interface

Applications that write valid TIFF X files should include _TIFF B_ and/or _TIFF G_ and/or _TIFF P_ and/or _TIFF R_ and/or in their product spec sheets, if they can write the respective TIFF Class X files. If your application writes all four of these types, you may wish to write it as _TIFF B,G,P,R._ Of course, a term like _TIFF B,_ while fine for communicating with other vendors, will not convey much information to a typical user. In this case, a phrase such as _Standard TIFF Black-and-White Scanned Images_ might be better.

The same terminology guidelines apply to applications that read TIFF Class X files.

If your application reads more kinds of files than it writes, or vice versa, it would be a good idea to make that clear to the buyer. For example, if your application reads TIFF B and TIFF G

files, but writes only TIFF G files, you should write it that way in the spec sheet.

Appendix H: Image Colorimetry Information

Chris Sears
210 Lake Street
San Francisco, CA 94118

June 4, 1988
Revised August 8, 1988

I. Introduction

Our goal is to accurately reproduce a color image using different devices. Accuracy requires techniques of measurement and a standard of comparison. Different devices imply device independence. Colorimetry provides the framework to solve these problems. When an image has a complete colorimetric description, in principle it can be reproduced identically on different monitors and using different media, such as offset lithography.

The colorimetry data is specified when the image is created or changed. A scanned image has colorimetry data derived from the filters and light sources of the scanner and a synthetic image has colorimetry data corresponding to the monitor used to create it or the monitor model of the rendering environment. This data is used to map an input image to the markings or colors of a particular output device.

Section II describes various standards organizations and their work in color.
Section III describes our motivation for seeking these tags.
Section IV describes our goals of reproduction.
Sections V, VI and VII introduce the colorimetry tags.
Section VIII specifies the tags themselves.
Section IX describes the defaults.
Section X discusses the limitations and some of the other issues.
Section XI provides a few references.

II. Related Standards

TIFF is a general standard for describing image data. It would be foolish for us to change TIFF in a way that did not match existing industry and international standards. Therefore, we have taken pains to note in the discussion below the efforts of various standards organizations and select defaults from the work of these organizations.

CIE (Commission Internationale de lÆEclairage) The basis of the colorimetry information is the CIE 1931 Standard Observer [3]. While other color models could be supported [1] [4], CIE 1931 XYZ is the international standard accepted across industries for specifying color and CIE xyY is the chromaticity diagram associated with CIE 1931 XYZ tristimulus values.

NTSC (National Television System Committee) NTSC is of interest primarily for historical reasons and its use in encoding television data. Manufacturing standards for monitors have for some time drifted significantly from the 1953 NTSC colorimetry specification.

SMPTE (Society of Motion Picture and Television Engineers) Most of the work by NTSC has been largely subsumed by SMPTE. This organization has a set of standards called "Recommended Practices" that apply to various technical aspects of film and television production [5] [6].

ISO (International Standards Organization) ISO has become involved in color standards through work on a color addendum to "Office Document Architecture (ODA) and Interchange Format" [7].

ANSI (American National Standards Institute) ANSI is the American representative to ISO .

III. Motivation

Our motivation for defining these tags stems from our research and development in color separation technology. With the information described here and the RGB pixel data, we have all of the information necessary for generating high-quality color separations. We could supply the colorimetry information outside of the image file. But since TIFF provides a convenient mechanism for bundling all of the relevant information in a single place, tags are defined to describe this information in color TIFF files.

A color image rendered with incorrect colorimetry information looks different from the original. One of our early test images has an artifact in it where the image was scanned with one set of primaries and color ramps were overlaid on top of it with different primaries. The blue ramp looked purple when we printed it. Using incorrect gamma tables or white points can also lead to distorted images. The best way to avoid these kinds of errors is to allow the creator of an image to supply the colorimetry information along with the RGB values [1] [2].

The purpose of the colorimetry data is to allow a projective transformation from the primaries and white point of the image to the primaries and white point of the rendering media. Gamma reflects the non-linear transfer gradient of real media.

IV. Colorimetric Color Reproduction

Earlier we said that given the proper colorimetric data an image could be rendered identically using two different calibrated devices. By identical, we mean colorimetric reproduction [9].

Specifically, the chromaticities match and the luminance is scaled to correspond to the luminance range of the output device. Because of this, we only need the chromaticity coordinates of the white point and primaries. The absolute luminance is arbitrary and unnecessary.

V. White Point

In TIFF 4.0, the white point was specified as D65. This appendix allocates a separate tag for describing the white point and D65 is the logical default since it is the SMPTE standard [6].

The white point is defined colorimetrically in the CIE xyY chromaticity diagram. While it is rare for monitors to differ from D65, scanned images often have different white points. Rendered images can have arbitrary white points. The graphic arts use D50 as the standard viewing light source [8].

VI. Primary Chromaticities

In TIFF 4.0, the primary color chromaticities matched the NTSC specification. With the wide variety of color scanners, monitors and renderers, TIFF needs a mechanism for accurately describing the chromaticities of the primary colors. We use SMPTE as the default chromaticity since conventional monitors are closer to SMPTE and some monitors (Conrac 6545) are manufactured to the SMPTE specifications. We donÆt use the NTSC chromaticities and white point because present day monitors donÆt use them and must be _matrixed_ to approximate them.

As an example, the primary color chromaticities used by the Sony Trinatron differ from those recommended by SMPTE. In general, since real monitors vary from the industry standards, the chromaticities of primaries are described in the CIE xyY system. This allows a reproduction system to compensate for the differences.

VII. Color Response Curves

This tag defines three color response curves, one each for red, green, and blue color information. The width of each entry is 16 bits, as implied by the type SHORT. The minimum intensity is represented by 0 and the maximum by 65535. For example, black is represented by 0,0,0 and white by 65535, 65535, 65535. The length of each curve is 2**BitsPerSample. A ColorResponseCurves field for RGB data where each of the samples is 8 bits deep would have 3*256 entries. The 256 red entries would come first, followed by 256 green entries, followed by 256 blue entries.

The purpose of the ColorResponseCurves field is to act as a lookup table mapping sample values to specific intensity values,

so that an image created on one system can be displayed on another with minimal loss of color fidelity. The ColorResponseCurves field thus describes the _gamma_ of an image, so that a TIFF reader on another system can compensate for both the image gamma and the gamma of the reading system.

Gamma is a term that relates to the typically nonlinear response of most display devices, including monitors. In most display systems, the voltage applied to the CRT is directly proportional to the value of the red, green, or blue sample. However, the resulting luminance emitted by the phosphor is not directly proportional to the voltage. This relationship is approximated in most displays by

```
luminance = voltage ** gamma
```

The NTSC standard gamma of 2.2 adequately describes most common video systems. The standard gamma of 2.2 implies a dim viewing surround. (We know of no SMPTE recommended practice for gamma.) The following example uses an 8 bit sample with value of 127.

```
voltage = 127 / 255 = 0.4980
luminance = 0.4980 ** 2.2 = 0.2157
```

In the examples below, we only consider a single primary and therefore only a single curve. The same analysis applies to each of the red, green, and blue primaries and curves. Also, and without loss of generality, we assume that there is no hardware color map, so that we must alter the pixel values themselves. If there is a color map, the manipulations can be done on the map instead of on the pixels.

If no ColorResponseCurves field exists in a color image, the reader should assume a gamma of 2.2 for each of the primaries. This default curve can be generated with the following C code:

```
ValuesPerSample = 1 << BitsPerSample;
for (curve[0] = 0, i = 1; i < ValuesPerSample; i++)
     curve[i] = floor (pow (i / (ValuesPerSample - 1.0),
2.2) * 65535.0 + .5);
```

The displaying or rendering application can know its own gamma, which we will call the _destination gamma._ (An uncalibrated system can usually assume that its gamma is 2.2 without going too far wrong.) Using this information the application can compensate for the gamma of the image, as we shall see below.

If the source and destination systems are both adequately described by a gamma of 2.2, the writer would omit the ColorResponseCurves field, and the reader can simply read the image directly into the frame buffer. If a writer writes out the ColorResponseCurves field, then a reader must assume that the gammas differ. A reader must then perform the following computation on each sample in the image:

```
     NewSampleValue = floor (pow (curve[OldSampleValue] /
65535.0, 1.0 / DestinationGamma) *
       (ValuesPerSample - 1.0) + .5);
```

Of course, if the _gamma_ of the destination system is not well-approximated with an exponential function, an arbitrary table lookup may be used in place of raising the value to 1.0 / DestinationGamma.

Leave out ColorResponseCurves if using the default gamma. This saves about 1.5K in the most common case, and, after all, omission is the better part of compression.

Do not use this field to store frame buffer color maps. Use instead the ColorMap field. Note, however, that ColorResponseCurves may be used to refine the information in a ColorMap if desired.

The above examples assume that a single parameter gamma system adequately approximates the response characteristics of the image source and destination systems. This will usually be true, but our use of a table instead of a single gamma parameter gives the flexibility to describe more complex relationships, without requiring additional computation or complexity.

VIII. New Tags and Changes

The following tags should be placed in the "Basic Fields" section of
the TIFF specification:

White Point
Tag = 318 (13E)
Type = RATIONAL
N = 2

The white point of the image. Note that this value is described using the 1931 CIE xyY chromaticity diagram and only the chromaticity is specified. The luminance component is arbitrary and not specified. This can correspond to the white point of a monitor that the image was painted on, the filter set/light source combination of a scanner, or to the white point of the illumination model of a rendering package.

Default is the SMPTE white point, D65: x = 0.313, y = 0.329.

The ordering is x, y.

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

The primary color chromaticities. Note that these values are described using the 1931 CIE xyY chromaticity diagram and only the chromaticities are specified. For paint images, these represent the chromaticities of the monitor and for scanned images they are derived from the filter set/light source combination of a scanner.

Default is the SMPTE primary color chromaticities:

Red: x = 0.635 y = 0.340
Green: x = 0.305 y = 0.595
Blue: x = 0.155 y = 0.070

The ordering is red x, red y, green x, green y, blue x, blue y.

Color Response Curves

Default for ColorResponseCurves represents curves corresponding to the NTSC standard gamma of 2.2.

IX. Defaults

The defaults used by TIFF reflect industry standards. Both the WhitePoint and PrimaryChromaticities tags have defaults that are promoted by SMPTE . In addition, the default for the ColorResponseCurves tag matches the NTSC specification of a gamma of 2.2.

The purpose of these defaults is to allow reasonable results in the absence of accurate colorimetry data. An uncalibrated scanner or paint system produces an image that be displayed identically, though probably incorrectly on two different but calibrated systems. This is better then the uncertain situation where the image might be rendered differently on two different but calibrated systems.

X. Limitations and Issues

This section discusses several of the limitations and issues

involved in colorimetric reproduction.

Scope of Usefulness

For many purposes the data recommended here is unnecessary and can be omitted. For presentation graphics where there are only a few colors, being able to tell red from green is probably good enough. In this case the tags can be ignored and there is no overhead. In more demanding color reproduction environments, this data allows images to be described device independently and at small cost.

User Burdens

The data we recommend isnÆt a user burden; it is really a systems issue. It allows a systems solution but doesnÆt require user intercession. Calibration however is a separate issue. It is likely to involve the user.

Resolution Versus Fidelity

Some manufacturers supply greater than 24 bits of resolution for color specification. The purpose of this is either to avoid artifacts such as contouring in the shadows or in some cases to be more specific or device independent about the color. Both reasons can be misguided. Other, less expensive techniques can be used to prevent artifacts, such as deeper color maps. As for accuracy, fidelity is more important than precision.

Colorimetric Color Reproduction

There are other choices for objectives of color reproduction [9]. Spectral color reproduction is a stronger condition and most are weaker, such as preferred color reproduction. While device independent spectral color reproduction is impossible, device independent colorimetric reproduction is possible, within a tolerance and within the limits of the gamuts of the devices. By choosing a strong criteria we allow the important objectives of weaker criteria, such as preferred color reproduction, to be part of design packages.

Metamerism

If two patches of color are identical under one light and different under another, they are said to be metameric pairs. Colorimetric color reproduction is a weaker condition than spectral color reproduction and hence allows metamerism problems. By standardizing the viewing conditions we can largely finesse

the metamerism problem for print. Because television is self-luminous and doesnÆt use spectral absorption, metamerism isnÆt so much a problem.

Color Models - xyY Versus Luv, etc.

We choose xyY over Luv [1] because XYZ is the international standard for color specification and xyY is the chromaticity diagram associated with XYZ. Luv is meant for color difference measurement.

Ambient Environment And Viewing Surrounds

The viewing environment affects how the eye perceives color. The eye adapts to a dark room and it adapts to a colored surround. While these problems can be compensated for within the colorimetric framework [4], it is much better to finesse them by standardizing. The design environment should match the intended

viewing environment. Specifically it should not be a pitch dark room and, on average, it should be of a neutral color. For print, ANSI recommends a Munsell N-8 surface [8].

XI. References

In particular, we would like to mention the work of Stuart Ring of the Copy Products Division of the Eastman Kodak Company. He and his colleagues are promoting a color data interchange paradigm. They are working closely with the ISO 8613 Working Group [7].

[1] Color Data Interchange Paradigm, Eastman Kodak, Rochester, New York, 7 December 1987.

[2] Color Reproduction and Illumination Models, Roy Hall, International Summer Institute: State of the Art in Computer Graphics, 1986.

[3] CIE Colorimetry: Official Recommendations of the International Commission on Illumination, Publication 15-2, 1986.

[4] Color Science: Concepts and Methods, Quantitative Data and Formulae, Gunter Wyszecki, W.S. Stiles, John Wiley and Sons, Inc., New York, New York, 1982.

[5] Color Monitor Colorimetry, SMPTE Recommended Practice RP 145-1987.

[6] Color Temperature for Color Television Studio Monitors, SMPTE Recommended Practice RP 37.

[7] Office Document Architecture (ODA) and Interchange Format_Addendum on Colour, ISO 8613 Working Draft.

[8] ANSI Standard PH 2.30-1985.

[9] The Reproduction of Colour in Photography, Printing and Television, R. W. G. Hunt, Fountain Press, Tolworth, England, 1987.

[10] Raster Graphics Handbook, The Conrac Corporation, Van Nostrand Reinhold Company, New York, New York, 1985. Good description of gamma.

Appendix I: Horizontal Differencing Predictor

This appendix, written after the release of Revision 5.0 of the TIFF specification, is still in draft form. Please send any comments to the Aldus Developers Desk.

Revision 5.0 of the TIFF specification defined a new tag called _Predictor_ that describes techniques that may be used in conjuction with TIFF compression schemes. We now define a Predictor that greatly improves compression ratios for some images.

The horizontal differencing predictor is assigned the tag value Predictor = 2:

Predictor
Tag = 317 (13D)
Type = SHORT
N = 1

A predictor a mathematical operator that is applied to the image data before the encoding scheme is applied. Currently (as of

revision 5.0) 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 Appendix F.

1 = No prediction scheme used before coding.
2 = Horizontal differencing. See Appendix I.

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 samples, 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 samples, 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 sample into an 8-bit byte, so that we have one sample 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 samples are greater than 8 bits deep, expanding the samples 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 samples into 9-bit differences, 4-bit samples 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 didnÆt do anything special, we would be subtracting red sample values from green sample values, green sample values from blue sample values, and blue sample values from red sample 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 samples.

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 samples. The simplest way to get rid of noise is to mask off one or two low-

order bits of each 8-bit sample. 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 sample, 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.

Appendix J: Palette Color

This appendix, written after the release of Revision 5.0 of the TIFF specification, is still in draft form. Please send any comments to the Aldus Developers Desk.

Revision 5.0 of the TIFF specification defined a new PhotometricInterpretation value called _palette color._ We have been wondering lately if this additional complexity is worth the implementation expense. If not, let_s drop it before too many people start creating palette color images.

The Proposal

Instead of a separate palette color image type, there seems to be no compelling reason why palette (mapped) color images should not be stored as _full color_ (usually 24-bit) RGB images.

Objections

The most obvious objection is the amount of space required. But if you care about how much space the image takes up on disk, you should use LZW compression, which is ideally suited to most palette color images. (LZW compresses most paint-type palette color images 5:1 or more.) And if you use LZW compression, it turns out that palette color images stored as full color images compress to almost exactly the same size as palette color images stored as palette color images. That is, with LZW compression, there is no penalty for storing palette color images as full color RGB images. The resulting file may be a few percent larger, but it will not be three times as large. See Appendix F for more information on how LZW works.

Another objection might be that an application might want to process the image differently if it is _really_ a palette color image. But we can easily add auxiliary information that can help a TIFF reader to quickly categorize color images if it wants to. See the _New tags_ section below.

Benefits

It may be obvious, but it is probably worth discussing why we want to abolish palette color images as a distinct classification.

The main problem is that palette color as a separate type makes life more hazardous and confusing for users. The confusion factor is aggravated because users already have to be somewhat aware of distinctions between bilevel, grayscale, and color

images. Having two main types of color images is hard for many users to grasp, and it is probably not possible to totally hide this complexity from the user in certain situations. The hazard level goes up because some applications may accept palette color but not full color images, or full color but not palette color images, or may accept 8-bit palette color images but not 4-bit or 3-bit palette color images.

The second problem is that writing and maintaining code to deal with an additional image type is somewhat expensive for TIFF readers. The cost of supporting palette color images will depend on the application, but we believe that, in general, the cost will be substantial. It seems to make more sense to put the burden on TIFF writers to convert palette color images into full color image form than to make TIFF readers handle an additional color image type, since there are more TIFF readers than writers at this point.

New tags

Here are some proposed new tags that can help to classify color images, and make up for not having a separate palette color class. They are not required for TIFF Class R , but are strongly recommended for color TIFF images created by palette-type color paint programs.

ColorImageType
Tag = 318 (13E)
Type = SHORT
N = 1

Gives TIFF color image readers a better idea of what kind of color image it is. There will be borderline cases.

1 = Continuous tone, natural image.
2 = Synthetic image, using a greatly restricted range of colors. Such images are produced by most color paint programs. See ColorList for a list of colors used in this image.

Default is 1.

ColorList
Tag = 319 (13F)
Type = BYTE or SHORT
N = the number of colors that are used in this image, times SamplesPerPixel

A list of colors that are used in this image. Use of this field is only practical for images containing a greatly restricted (usually less than or equal to 256) range of colors. ColorImageType should be 2. See ColorImageType.

The list is organized as an array of RGB triplets, with no pad. The RGB triplets are not guaranteed to be in any particular order. Note that the red, green, and blue components can either be a BYTE or a SHORT in length. BYTE should be sufficient for most applications.

No default.