TIFF File Format Summary

Also Known As: Tag Image File Format

Type	Bitmap
Colors	1- to 24-bit
Compression	Uncompressed, RLE, LZW, CCITT Group 3 and Group 4, JPEG
Maximum Image Size	2^32-1
Multiple Images Per File	Yes
Numerical Format	See article for discussion
Originator	Aldus
Platforms	MS-DOS, Macintosh, UNIX, others
Supporting Applications	Most paint, imaging, and desktop publishing programs
See Also	Chapter 9, *Data Compression* (RLE, LZW, CCITT, and JPEG)

Usage
Used for data storage and interchange. The general nature of TIFF allows it to be used in any operating environment, and it is found on most platforms requiring image data storage.

Comments
The TIFF format is perhaps the most versatile and diverse bitmap format in existence. Its extensible nature and support for numerous data compression schemes allow developers to customize the TIFF format to fit any peculiar data storage needs.

Vendor specifications are available for this format.

Code fragments are available for this format.

Sample images are available for this format.

The TIFF specification was originally released in 1986 by Aldus Corporation as a standard method of storing black-and-white images

created by scanners and desktop publishing applications. This first public release of TIFF was the third major revision of the TIFF format, and although it was not assigned a specific version number, this release may be thought of as TIFF Revision 3.0. The first widely used revision of TIFF, 4.0, was released in April 1987. TIFF 4.0 added support for uncompressed RGB color images and was quickly followed by the release of TIFF Revision 5.0 in August 1988. TIFF 5.0 was the first revision to add the capability of storing palette color images and support for the LZW compression algorithm. (See the sidebar on LZW compression in the section called "Compression" later in this article.) TIFF 6.0 was released in June 1992 and added support for CMYK and YCbCr color images and the JPEG compression method. (See the section called "Color" in Chapter 2, *Computer Graphics Basics*, for a discussion of these color images. See Chapter 9, for a discussion of JPEG compression.)

Contents:

Today, TIFF is a standard file format found in most paint, imaging, and desktop publishing programs and is a format native to the Microsoft Windows GUI. TIFF's extensible nature, allowing storage of multiple bitmap images of any pixel depth, makes it ideal for most image storage needs.

The majority of the description in this chapter covers the current TIFF revision 6.0. Because each successive TIFF revision is built upon the previous revision, most of the information present in this chapter also pertains to TIFF Revision 5.0 as well. And, although more images are currently stored in the TIFF 5.0 format than in any other revision of TIFF, quite a few TIFF 4.0 image files are still in existence. For this reason, information is also included that details the differences between the TIFF 4.0, 5.0, and 6.0 revisions.

TIFF has garnered a reputation for power and flexibility, but it is considered complicated and mysterious as well. In its design, TIFF attempts to be very extensible and provide many features that a programmer might need in a file format. Because TIFF is so extensible and has many capabilities beyond all other image file formats, this format is probably the most confusing format to understand and use.

A common misconception about TIFF is that TIFF files are not very portable between software applications. This is amazing considering that TIFF is widely used as an image data interchange format. Complaints include, "I've downloaded a number of TIFF clip art packages from some BBSs and my paint program or word processor is able to display only some of the TIFF image files, but not all of them," "When I try to display certain TIFF files using my favorite image display program, I get the error message `Unknown Tag Type' or `Unsupported Compression Type'," and "I have a TIFF file created by one application and a second

application on the same machine cannot read or display the image, even though TIFF files created by the second application can be read and displayed by the first application."

These complaints are almost always immediately blamed on the TIFF image files themselves. The files are labeled "bad," because they have been munged during a data file transfer or were exported by software applications that did not know how to properly write a TIFF file. In reality, most TIFF files that do not import or display properly are not bad, and the fault usually lies, instead, with the program that is reading the TIFF file.

If an application only uses black-and-white images, it certainly does not need to support the reading and writing of color and gray-scale TIFF image files. In this case, the application should simply, and politely, refuse to read non-black-and-white TIFF image files and tell you the reason why. By doing this, the application would prevent the user from trying to read unusable image data and would also cut down on the amount of TIFF code the application programmers need to write.

Some applications that read TIFF image files--or any type of image files, for that matter--may just return an ambiguous error code indicating that the file could not be read, leaving the user with the impression that the TIFF file itself is bad (not that the application could not use the image data the TIFF file contained). Such an occurrence is the fault of the application designer in not providing a clearer message informing the user what has happened.

Sometimes, however, you may have an application that should be able to read a TIFF file, and it does not, even though the type of image data contained in the TIFF file is supported by the application. There are numerous reasons why a perfectly good TIFF file cannot be read by an application, and most of them have to do with the application programmer's lack of understanding of the TIFF format itself.

A major source of TIFF reader problems is the inability to read data regardless of byte-ordering scheme. The bytes in a 16-bit and 32-bit word of data are stored in a different order on little-endian architectures (such as the Intel iAPX86), than on big-endian machines (such as the Motorola MC68000A). Reading big-endian data using the little-endian format results in little more than garbage.

Another major source of problems is readers that do not support the encoding algorithm used to compress the image data. Most readers support both raw (uncompressed) and RLE-encoded data but do not support CCITT T.4 and T.6 compression. It is also surprising how many TIFF readers support the reading of color TIFF files, which are either stored as raw or RLE-compressed data, but do not support the decompression of LZW-encoded data.

Most other TIFF reader problems are quite minor, but usually fatal. Such problems include failure to correctly interpret tag data, no support for

color-mapped images, or the inability to read a bitmap scan line that contains an odd number of bytes.

File Organization

TIFF files are organized into three sections: the Image File Header (IFH), the Image File Directory (IFD), and the bitmap data. Of these three sections, only the IFH and IFD are required. It is therefore quite possible to have a TIFF file that contains no bitmapped data at all, although such a file would be highly unusual. A TIFF file that contains multiple images has one IFD and one bitmap per image stored.

TIFF has a reputation for being a complicated format in part because the location of each Image File Directory and the data the IFD points to--including the bitmapped data--may vary. In fact, the only part of a TIFF file that has a fixed location is the Image File Header, which is always the first eight bytes of every TIFF file. All other data in a TIFF file is found by using information found in the IFD. Each IFD and its associated bitmap are known as a TIFF *subfile*. There is no limit to the number of subfiles a TIFF image file may contain.

Each IFD contains one or more data structures called *tags*. Each tag is a 12-byte record that contains a specific piece of information about the bitmapped data. A tag may contain any type of data, and the TIFF specification defines over 70 tags that are used to represent specific information. Tags are always found in contiguous groups within each IFD.

Tags that are defined by the TIFF specification are called *public tags* and may not be modified outside of the parameters given in the latest TIFF specification. User-definable tags, called *private tags*, are assigned for proprietary use by software developers through the Aldus Developer's Desk. See the TIFF 6.0 specification for more information on private tags.

Note that the TIFF 6.0 specification has replaced the term *tag* with the term *field*. Field now refers to the entire 12-byte data record, while the term tag has been redefined to refer only to a field's identifying number. Because so many programmers are familiar with the older definition of the term tag, the authors have choosen to continue using tag, rather than field, in this description of TIFF to avoid confusion.

Figure TIFF-1 shows three possible arrangements of the internal data structure of a TIFF file containing three images. In each example, the IFH appears first in the TIFF file. In the first example, each of the IFDs has been written to the file first and the bitmaps last. This arrangement is the most efficient for reading IFD data quickly. In the second example, each IFD is written, followed by its bitmapped data. This is perhaps the most common internal format of a multi-image TIFF file. In the last example, we see that the bitmapped data has been written first, followed by the IFDs. This seemingly unusual arrangement might occur if the bitmapped data is available to be written before the information that appears in the IFDs.

Figure TIFF-1: Three possible physical arrangements of data in a TIFF file

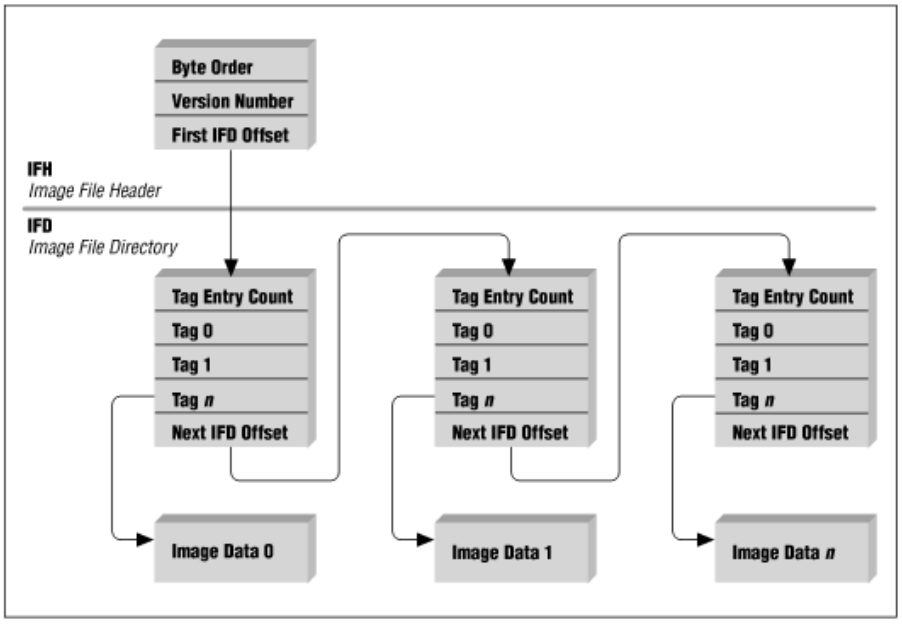
Each IFD is a road map of where all the data associated with a bitmap can be found within a TIFF file. The data is found by reading it directly from within the IFD data structure or by retrieving it from an offset location whose value is stored in the IFD. Because TIFF's internal components are linked together by offset values rather than by fixed positions, as with stream-oriented image file formats, programs that read and write TIFF files are often very complex, thus giving TIFF its reputation.

The offset values used in a TIFF file are found in three locations. The first offset value is found in the last four bytes of the header and indicates the position of the first IFD. The last four bytes of each IFD is an offset value to the next IFD. And the last four bytes of each tag may contain an offset value to the data it represents, or possibly the data itself.

NOTE:
Offsets are always interpreted as a number of bytes from the beginning of the TIFF file.

Figure TIFF-2 shows the way data structures of a TIFF file are linked together.

Figure TIFF-2: Logical organization of a TIFF file



File Details

This section describes the various components of a TIFF file.

Image File Header

TIFF, despite its complexity, has the simplest header of all of the formats described in this book. The TIFF Image File Header (IFH) contains three fields of information and is a total of only eight bytes in length:

```
typedef struct _TiffHeader
{
        WORD  Identifier;  /* Byte-order Identifier */
        WORD  Version;     /* TIFF version number (always 2Ah)
        DWORD IFDOffset;   /* Offset of the first Image File D
} TIFHEAD;
```

Identifier contains either the value 4949h (II) or 4D4Dh (MM). These values indicate whether the data in the TIFF file is written in little-endian (Intel format) or big-endian (Motorola format) order, respectively. All data encountered past the first two bytes in the file obey the byte-ordering scheme indicated by this field. These two values were chosen because they would always be the same, regardless of the byte order of the file.

Version, according to the TIFF specification, contains the version number of the TIFF format. This version number is always 42, regardless of the TIFF revision, so it may be regarded more as an identification number, (or possibly the answer to life, the universe, etc.) than a version number.

A quick way to check whether a file is indeed a TIFF file is to read the first four bytes of the file. If they are:

```
49h 49h 2Ah 00h
```

or:

```
4Dh 4Dh 00h 2Ah
```

then it's a good bet that you have a TIFF file.

IFDOffset is a 32-bit value that is the offset position of the first Image File Directory in the TIFF file. This value may be passed as a parameter to a file seek function to find the start of the image file information. If the Image File Directory occurs immediately after the header, the value of the IFDOffset field is 08h.

Image File Directory

An Image File Directory (IFD) is a collection of information similar to a header, and it is used to describe the bitmapped data to which it is attached. Like a header, it contains information on the height, width, and depth of the image, the number of color planes, and the type of data compression used on the bitmapped data. Unlike a typical fixed header, however, an IFD is dynamic and may not only vary in size, but also may be found anywhere within the TIFF file. There may be more than one IFD contained within any file. The format of an Image File Directory is shown

in Figure TIFF-1.

One of the misconceptions about TIFF is that the information stored in the Image File Directory tags is actually part of the TIFF header. In fact, this information is often referred to as the "TIFF Header Information." While it is true that other formats do store the type of information found in the IFD in the header, the TIFF header does not contain this information. It is possible to think of the IFDs in a TIFF file as extensions of the TIFF file header.

A TIFF file may contain any number of images, from zero on up. Each image is considered to be a separate *subfile* (i.e., a bitmap) and has an IFD describing the bitmapped data. Each TIFF subfile can be written as a separate TIFF file or can be stored with other subfiles in a single TIFF file. Each subfile bitmap and IFD may reside anywhere in the TIFF file after the headers, and there may be only one IFD per image.

This may sound confusing, but it's not really. We have seen that the TIFF header contains an offset value that points to the location of the first IFD in the TIFF file. To find the first IFD, all we need do is seek to this offset and start reading the IFD information. The last field of every IFD contains an offset value to the next IFD, if any. If the offset value of any IFD is 00h, then there are no more images left to read in the TIFF file.

An IFD may vary in size, because it may contain a variable number of data records, called *tags*. Each tag contains a unique piece of information, just as fields do within a header. However, there is a difference. Tags may be added and deleted from an IFD much the same way that notebook paper may be added to or removed from a three-ring binder. The fields of a conventional header, on the other hand, are fixed and unmovable, much like the pages of this book. Also, the number of tags found in an IFD may vary, while the number of fields in a type header is fixed.

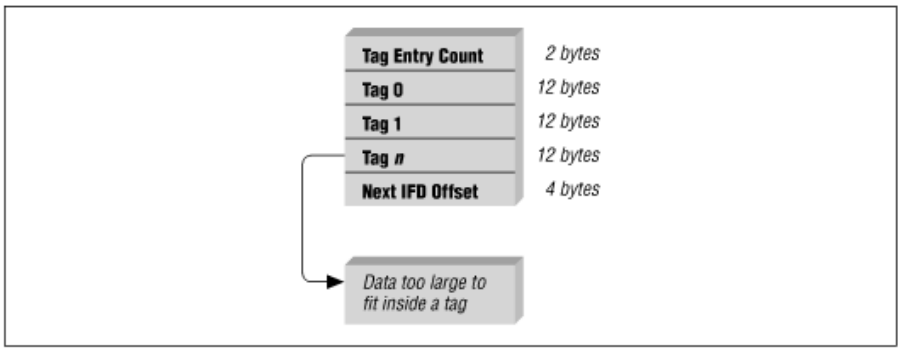
The format of an Image File Directory is shown in the following structure:

```
typedef struct _TifIfd
{
        WORD    NumDirEntries;    /* Number of Tags in IFD  */
        TIFTAG  TagList[];        /* Array of Tags  */
        DWORD   NextIFDOffset;    /* Offset to next IFD  */
} TIFIFD;
```

NumDirEntries is a 2-byte value indicating the number of tags found in the IFD. Following this field is a series of tags; the number of tags corresponds to the value of the NumDirEntries field. Each tag structure is 12 bytes in size and, in the sample code above, is represented by an array of structures of the data type definition TIFTAG. (See the next section for more information on TIFF tags.) The number of tags per IFD is limited to 65,535.

NextIFDOffset contains the offset position of the beginning of the next IFD. If there are no more IFDs, then the value of this field is 00h.

Figure TIFF-3: Format of an Image File Directory



Tags

As mentioned in the previous section, a tag can be thought of as a data field in a file header. However, whereas a header data field may only contain data of a fixed size and is normally located only at a fixed position within a file header, a tag may contain, or point to, data that is any number of bytes in size and is located anywhere within an IFD.

The versatility of the TIFF tag pays a price in its size. A header field used to hold a byte of data need only be a byte in size. A tag containing one byte of information, however, must always be twelve bytes in size.

A TIFF tag has the following 12-byte structure:

```
typedef struct _TifTag
{
        WORD   TagId;       /* The tag identifier  */
        WORD   DataType;    /* The scalar type of the data ite
        DWORD  DataCount;   /* The number of items in the tag
        DWORD  DataOffset;  /* The byte offset to the data ite
} TIFTAG;
```

TagId is a numeric value identifying the type of information the tag contains. More specifically, the TagId indicates what the tag information represents. Typical information found in every TIFF file includes the height and width of the image, the depth of each pixel, and the type of data encoding used to compress the bitmap. Tags are normally identified by their TagId value and should always be written to an IFD in ascending order of the values found in the TagId field.

DataType contains a value indicating the scalar data type of the information found in the tag. The following values are supported:

1 BYTE 8-bit unsigned integer

2 ASCII 8-bit, NULL-terminated string

3 SHORT 16-bit unsigned integer

4 LONG 32-bit unsigned integer

5 RATIONAL Two 32-bit unsigned integers

The BYTE, SHORT, and LONG data types correspond to the BYTE, WORD, and DWORD data types used throughout this book. The ASCII

data type contains strings of 7-bit ASCII character data, which are always NULL-terminated and may be padded out to an even length if necessary. The RATIONAL data type is actually two LONG values and is used to store the two components of a fractional value. The first value stores the numerator, and the second value stores the denominator.

The TIFF 6.0 revision added the following new data types:

6	SBYTE	8-bit signed integer
7	UNDEFINE	8-bit byte
8	SSHORT	16-bit signed integer
9	SLONG	32-bit signed integer
10	SRATIONAL	Two 32-bit signed integers
11	FLOAT	4-byte single-precision IEEE floating-point value
12	DOUBLE	8-byte double-precision IEEE floating-point value

The SBYTE, SSHORT, SLONG, and SRATIONAL data types are used to store signed values. The FLOAT and DOUBLE data types are used specifically to store IEEE-format single- and double-precision values. The UNDEFINE data type is an 8-bit byte that may contain untyped or opaque data and is typically used in private tags. An example of the use of this data type is to store an entire data structure within a private tag specifying the DataType as UNDEFINE (value of 7) and a DataCount equal to the number of bytes in the structure.

With the exception of the SMinSampleValue and SMaxSampleValue tags (which may use any data type), none of these newer data types is used by any TIFF 6.0 tags. They are therefore found only in private tags.

DataCount indicates the number of items referenced by the tag and doesn't show the actual size of the data itself. Therefore, a DataCount of 08h does not necessarily indicate that eight bytes of data exist in the tag. This value indicates that eight items exist for the data type specified by this tag. For example, a DataCount value of 08h and a DataType of 03h indicate that the tag data is eight contiguous 16-bit unsigned integers, a total of 32 bytes in size. A DataCount of 28h and a DataType of 02h indicate an ASCII character string 40 bytes in length, including the NULL-terminator character, but not any padding if present. And a DataCount of 01h and a DataType of 05h indicate a single RATIONAL value a total of eight bytes in size.

DataOffset is a 4-byte field that contains the offset location of the actual tag data within the TIFF file. If the tag data is four bytes or less in size, the data may be found in this field. If the tag data is greater than four bytes in size, then this field contains an offset to the position of the data in the TIFF file. Packing data within the DataOffset field is an optimization within the TIFF specification and is not required to be

performed. Most data is typically stored outside the tag, occurring before or after the IFD (see Figure TIFF-3).

Table TIFF-1 lists all of the public tags included in the TIFF 4.0, 5.0, and 6.0 specifications. Note that some tags have become obsolete and are not found in the current revision of TIFF; however, we provide them because the TIFF 4.0 and TIFF 5.0 specs are still in some use. Also, note that several tags may support more than one data type.

In the table below, an asterisk (*) means that the tag is defined, a hyphen (-) means that the tag is not defined, and an "x" means that the tag is obsolete.

Tag Name	Tag ID	Tag Type	4.0	5.0	6.0
Artist	315	ASCII	-	*	*
BadFaxLines[1]	326	SHORT or LONG	-	-	-
BitsPerSample	258	SHORT	*	*	*
CellLength	265	SHORT	*	*	*
CellWidth	264	SHORT	*	*	*
CleanFaxData[1]	327	SHORT	-	-	-
ColorMap	320	SHORT	-	*	*
ColorResponseCurve	301	SHORT	*	*	x
ColorResponseUnit	300	SHORT	*	x	x
Compression	259	SHORT	*	*	*
Uncompressed	1		*	*	*
CCITT 1D	2		*	*	*
CCITT Group 3	3		*	*	*
CCITT Group 4	4		*	*	*
LZW	5		-	*	*
JPEG	6		-	-	*
Uncompressed	32771		*	x	x
Packbits	32773		*	*	*
ConsecutiveBadFaxLines[1]	328	LONG or SHORT	-	-	-
Copyright	33432	ASCII	-	-	*

Table TIFF-1: TIFF Tag Types Listed Alphabetically by Name

DateTime	306	ASCII	-	*	*
DocumentName	269	ASCII	*	*	*
DotRange	336	BYTE or SHORT	-	-	*
ExtraSamples	338	BYTE	-	-	*
FillOrder	266	SHORT	*	*	*
FreeByteCounts	289	LONG	*	*	*
FreeOffsets	288	LONG	*	*	*
GrayResponseCurve	291	SHORT	*	*	*
GrayResponseUnit	290	SHORT	*	*	*
HalftoneHints	321	SHORT	-	-	*
HostComputer	316	ASCII	-	*	*
ImageDescription	270	ASCII	*	*	*
ImageHeight	257	SHORT or LONG	*	*	*
ImageWidth	256	SHORT or LONG	*	*	*
InkNames	333	ASCII	-	-	*
InkSet	332	SHORT	-	-	*
JPEGACTTables	521	LONG	-	-	*
JPEGDCTTables	520	LONG	-	-	*
JPEGInterchangeFormat	513	LONG	-	-	*
JPEGInterchangeFormatLength	514	LONG	-	-	*
JPEGLosslessPredictors	517	SHORT	-	-	*
JPEGPointTransforms	518	SHORT	-	-	*
JPEGProc	512	SHORT	-	-	*
JPEGRestartInterval	515	SHORT	-	-	*
JPEGQTables	519	LONG	-	-	*
Make	271	ASCII	*	*	*
MaxSampleValue	281	SHORT	*	*	*
MinSampleValue	280	SHORT	*	*	*

Model	272	ASCII	*	*	*
NewSubFileType	254	LONG	-	*	*
NumberOfInks	334	SHORT	-	-	*
Orientation	274	SHORT	*	*	*
PageName	285	ASCII	*	*	*
PageNumber	297	SHORT	*	*	*
PhotometricInterpretation	262	SHORT	*	*	*
WhiteIsZero	0		*	*	*
BlackIsZero	1		*	*	*
RGB	2		*	*	*
RGB Palette	3		-	*	*
Tranparency Mask	4		-	-	*
CMYK	5		-	-	*
YCbCr	6		-	-	*
CIELab	8		-	-	*
PlanarConfiguration	284	SHORT	*	*	*
Predictor	317	SHORT	-	*	*
PrimaryChromaticities	319	RATIONAL	-	*	*
ReferenceBlackWhite	532	LONG	-	-	*
ResolutionUnit	296	SHORT	*	*	*
RowsPerStrip	278	SHORT or LONG	*	*	*
SampleFormat	339	SHORT	-	-	*
SamplesPerPixel	277	SHORT	*	*	*
SMaxSampleValue	341	Any	-	-	*
SMinSampleValue	340	Any	-	-	*
Software	305	ASCII	-	*	*
StripByteCounts	279	LONG or SHORT	*	*	*
StripOffsets	273	SHORT or LONG	*	*	*

SubFileType	255	SHORT	*	x	x
T4Options[2]	292	LONG	*	*	*
T6Options[3]	293	LONG	*	*	*
TargetPrinter	337	ASCII	-	-	*
Thresholding	263	SHORT	*	*	*
TileByteCounts	325	SHORT or LONG	-	-	*
TileLength	323	SHORT or LONG	-	-	*
TileOffsets	324	LONG	-	-	*
TileWidth	322	SHORT or LONG	-	-	*
TransferFunction[4]	301	SHORT	-	-	*
TransferRange	342	SHORT	-	-	*
XPosition	286	RATIONAL	*	*	*
XResolution	282	RATIONAL	*	*	*
YCbCrCoefficients	529	RATIONAL	-	-	*
YCbCrPositioning	531	SHORT	-	-	*
YCbCrSubSampling	530	SHORT	-	-	*
YPosition	287	RATIONAL	*	*	*
YResolution	283	RATIONAL	*	*	*
WhitePoint	318	RATIONAL	-	*	*

Table Footnotes:

[1] Tags BadFaxLines, CleanFaxData, and ConsecutiveBadFaxLines are part of TIFF Class F now maintained by Aldus and are not actually defined by the TIFF 6.0 specification.

[2] Tag 292 was renamed from Group3Options to T4Options in TIFF 6.0.

[3] Tag 293 was renamed from Group3Options to T6Options by TIFF 6.0.

[4] Tag 301 was renamed from ColorResponseCurve to TransferFunction by TIFF 6.0.

Organization of TIFF Tag Data

To keep developers from having to guess which tags should be written to a TIFF file and what tags are important to read, the TIFF specification

defines the concept of baseline TIFF images. These baselines are defined by the type of image data they store: bi-level, gray-scale, palette-color, and full-color. Each baseline has a minimum set of tags, which are required to appear in each type of TIFF file.

In the TIFF 5.0 specification, these baselines were referred to as TIFF classes. Each TIFF file consisted of a common baseline (Class X) and was modified by an additional class depending upon the type of image data stored. The classes were Class B (bi-level), Class G (gray-scale), Class P (palette-color), and Class R (full-color RGB).

The TIFF 6.0 specification redefines these classes into four separate baseline TIFF file configurations. Class X is combined with each of the other four classes to form the bi-level, gray-scale, color-palette, and full-color baselines. Although TIFF 6.0 largely does away with the concept of TIFF classes, it is likely that because more TIFF 5.0 format files exist than any other, TIFF files will be referred to by these class designations for many years to come.

It is worth mentioning that a de facto TIFF class, Class F, exists specifically for the storage of facsimile images using the TIFF format. This class of TIFF file, created by Cygnet Technologies, is used by Everex products to store facsimile data, and is also known as the Everex Fax File Format. Although Cygnet Technologies is no longer in business, TIFF Class F remains in use and is considered by some to be an excellent storage format for facsimile data.

Table TIFF-2 lists the minimum required tags that must appear in the IFD of each TIFF 6.0 baseline. Note that several of these tags have default values that are used if the tag does not actually appear in a TIFF file:

- Bi-level (formerly Class B) and gray-scale (formerly Class G) TIFF files must contain the thirteen tags listed. These tags must appear in all revision 5.0 and 6.0 TIFF files regardless of the type of image data stored.
- Palette-color (formerly Class P) TIFF files add a fourteenth required tag that describes the type of palette information found within the TIFF image file.
- RGB (formerly Class R) TIFF files contain the same tags as bi-level TIFF files and add a fourteenth required tag, which describes the format of the bitmapped data in the image.
- YCbCr TIFF files add four additional tags to the baseline.
- Class F TIFF files add three tags.

Class Name	Tag Type	Tag Name
Bi-level and	254	NewSubfileType

Table TIFF-2: Minimum Required Tags for Each TIFF Class

Gray-scale	256	ImageWidth
	257	ImageLength
	258	BitsPerSample
	259	Compression
	262	PhotometricInterpretation
	273	StripOffsets
	277	SamplesPerPixel
	278	RowsPerStrip
	279	StripByteCounts
	282	XResolution
	283	YResolution
	296	ResolutionUnit

The following classes contain the above 13 tags plus the following tags:

Palette-color	320	ColorMap
RGB	284	PlanarConfiguration
YCbCr	529	YCbCrCoefficients
	530	YCbCrSubSampling
	531	YCbCrPositioning
	532	ReferenceBlackWhite
Class F	326	BadFaxLines
	327	CleanFaxData
	328	ConsecutiveBadFaxLines

All other tags found in the TIFF specification are available to meet developer's needs. While a TIFF reader must be able to support the parsing and interpretation of all tags it considers necessary, it is certainly not necessary for a TIFF writer to include as many tags as possible in every TIFF file written.

Image Data

TIFF files contain only bitmap data, although adding a few tags to

support vector- or text-based images would not be a hard thing to do. As we have seen, the bitmapped data in a TIFF file does not always appear immediately after the image header, as with most other formats. Instead, it may appear almost anywhere within the TIFF file. And, because the majority of the work performed by a TIFF reader and writer is the creation and manipulation of the image data, a thorough understanding of how the image data is stored within a TIFF file is necessary, starting with the concept of *strips*.

NOTE:
TIFF 6.0 images may contain tiles rather than strips.

Strips

It is always amusing to come across a TIFF reader or viewer whose author posts the caveat in the source code, "This TIFF reader does not support stripped images." A large proportion of TIFF readers who fail to read certain TIFF image files do so because the author of the reader did not quite understand the concept of how image data can be organized within a TIFF file. In this case, not only did the author of the reader fail to understand how to write code to read strips, he or she also failed to realize that every TIFF 5.0 (and earlier) image contains strips.

A *strip* is an individual collection of one or more contiguous rows of bitmapped image data. Dividing the image data into strips makes buffering, random access, and interleaving of the image data much easier. This concept exists in several other image file formats, and is given names such as *blocks*, *bands*, and *chunks*. TIFF strips differ from other such concepts in several important ways due to the structure of the TIFF format.

Three tags are necessary to define the strips of bitmapped data within a TIFF file. These three tags are RowsPerStrip, StripOffsets, and StripByteCounts.

The first required tag, RowsPerStrip, indicates the number of rows of compressed bitmapped data found in each strip. The default value for RowsPerStrip is 2^32-1, which is the maximum possible size of a TIFF image. All of the strips in a TIFF subfile must use the same compression scheme and have the same bit sex, color sex, pixel depth, and so on. To find the number of strips in a non-YCbCr subfile image, we would use the RowsPerStrip tag, the ImageLength tag, and the following calculation:

```
StripsInImage =
          floor((ImageLength * (RowsPerStrip - 1)) / RowsPerS
```

The second required tag, StripOffsets, is important because without it a TIFF reader has absolutely no hope of locating the image data within a TIFF file. This tag contains an array of offset values, one per strip, which indicate the position of the first byte of each strip within the TIFF file. The first value in the array is for the first strip, the second value for the second strip, and so on. If the image data is separated into planes (PlanarConfiguration = 2), StripOffsets contains a 2D array of values,

which is SamplesPerPixel in width. All of the columns for color component (plane) 0 are stored first, followed by all the columns for color component (plane) 1, and so forth. The strips of planar image data may be written to the TIFF file in any order but are typically written by plane (RRRRGGGGBBBB) or by color component (RGBRGBRGBRGB). StripOffsets values are always interpreted from the beginning of the file.

The StripOffsets tag allows each strip in a TIFF file to have a location that is completely independent from all other strips in the same subfile. This means that strips may occur in any order and be found anywhere within the TIFF file. Many "quick and dirty" TIFF readers find the beginning of the first strip and then attempt to read in the image data as one large chunk without checking the StripOffsets array for the position of each additional strip. This technique works if all the strips in the TIFF file are contiguously written to the TIFF file and are in the same consecutive order as the original rows in the bitmap. If the strips are stored out of sequence, perhaps in a planar or interlaced fashion, or are aligned on paragraph or page boundaries, the image data read will not be in its proper order, and the image will appear sliced up and rearranged on the display. If the strips are stored in a fairly random fashion, a large part of the data read might not be part of the image, or even the TIFF file itself. In this case, anything that is displayed would be mostly garbage.

The value of the RowsPerStrip tag and the size of each element in the array of the StripOffsets tag is usually a LONG (32-bit) value. TIFF 5.0 added the ability of this tag to use SHORT (16-bit) values instead. Very old TIFF readers may expect the values in this tag to always be LONG and will therefore read the offset values improperly if they are SHORT. This modification was made primarily for TIFF readers that read the StripOffsets values into an array in memory before using them. The TIFF 6.0 specification suggests that the offset values should not require such an array to be larger than 64K in size.

The third tag, StripByteCounts, maintains an array of values that indicates the size of each strip in bytes. And, like the StripOffsets tag, this tag is also an array of values, one per strip, 1D for chunky format and 2D for planar format, each of which is calculated from the number of bytes of compressed bitmapped data stored in each strip.

This tag is necessary because there are several cases in which the strips in an image may each contain a different number of bytes. The first case occurs when using compressed bitmapped image data. As we have said, the StripBytesCounts value is the size of the image data *after* it is compressed. Although there is a fixed number of bytes in an uncompressed row, the size of a compressed row varies depending upon the data it contains. Because we are always storing a fixed number of rows, not bytes, per strip, it is likely that most strips will be of different lengths because each compressed row will vary in size. Only when the bitmap data is not compressed will each strip be the same size.

Well, almost... The last strip in an image is sometimes an exception. TIFF writers typically attempt to create strips so that each strip in a TIFF

image has the same number of rows. For example, a bitmap with 2200 rows can be divided into 22 strips, each containing 100 rows of bitmapped data. However, it is not always possible to divide the number of rows equally among the desired number of strips. For example, if we needed to divide a bitmap containing 482 rows into strips containing five rows each, we would end up with a total of 97 strips, 96 strips containing five rows of data and the 97th strip containing the remaining two rows. The RowsPerStrip tag value of 5 would be correct for all strip lengths except for the last strip.

The truth is that a TIFF reader does not need to know the number of rows in each strip to read the image data, only the number of bytes. Otherwise, the TIFF specification would require that every "short" strip be padded with additional rows of dummy data, but it doesn't. Instead, we simply read the last StripByteCounts value to determine how many bytes to read for the last strip. What the TIFF specification doesn't make clear is that the RowsPerStrip value specifies the *maximum* value, and not the required value, of the number of rows per strip. Many TIFF files, in fact, store a single strip of data and specify an arbitrarily large RowsPerStrip value.

There are several advantages to organizing bitmap data in strips.

First, not all applications can read an entire file into memory. Many desktop machines still have only one megabyte or less of memory available to them. And even if a system has gobs of memory, there is no guarantee that a TIFF reader will be able to use it. Such a TIFF reader can allocate the largest buffer it can manage and then read in the bitmapped data one strip at a time until the buffer is filled. If the image is panned or scrolled, data in the buffer can be discarded and more strips read in. If the entire image can fit in memory, all the strips in the TIFF file would then be read.

Because compressed strips can vary in size, the StripByteCounts values cannot be accurately used by an application to dynamically allocate a buffer in memory (unless every value is read and the largest value is used to allocate the buffer). Therefore, it is recommended that each strip be limited to about 8K in size. If a TIFF reader can allocate a much larger buffer than 8K, then multiple strips may be read into the buffer. Although TIFF strips can be larger, perhaps to support an image where each compressed or uncompressed row is greater than 8K in size, the size of a strip should never exceed 64K. Allocating a buffer greater than 64K can be a bit tricky when using certain system architectures.

Second, having access to a table of strip offsets makes random access of the bitmapped data easier. If a TIFF reader needed to display the last 100 rows of a 480-row image, and the bitmaps were divided into 48 strips of 10 rows each, the reader would skip over the first 380 rows and read in the strips stored at the last 10 offsets in the StripOffsets tag array. If no strip offsets were present, the entire image would need to be read to find the starting position of the last 100 rows.

And while it is possible that the bitmap in a TIFF file may be written out as one long strip--and many TIFF files are written this way--it is not a good idea to do so. These so-called unstripped images often fail because an application must attempt to allocate enough memory to hold the entire image. For large images, or small systems, enough memory may not be available. One can only hope that a TIFF file reader would exit from such a situation gracefully.

Tiles

Strips are not the only possible way to organize bitmapped data. TIFF 6.0 introduced the concept of *tiled*, rather than stripped, bitmapped data. A strip is a 1D object that only has a length. A tile can be thought of as a 2D strip that has both width and length, very similar to a bitmap. In fact, you can think of each tile in an image as a small bitmap containing a piece of a larger bitmap. All you need to do is fit the tiles together in their proper locations to get the image. (This concept only serves to remind me that I must replace the linoleum in my bathroom one day.)

Dividing an image into rectangular tiles rather than horizontal strips has the greatest benefit on very large high-resolution images. Many electronic document imaging (EDI) applications cannot manipulate images larger than E size (6848 pixels wide by 8800 pixels long) because of the large amount of memory required to buffer, decompress, and manipulate even a few hundred rows of image data. Even if you just wanted to display the upper-left corner of an image you would still be forced to decompress the entire strip and maintain it in memory. If the image data were tiled, however, you would only decompress the tiles that contained the image data for the part of the image you wanted to display.

Many compression algorithms, such as JPEG, compress data not as 1D strips, but instead as 2D tiles. Storing the compressed data as tiles optimizes the decompression of the data quite a bit. In fact, the support of 2D compression algorithms is perhaps the primary reason why the capability of tiling image data was added to TIFF 6.0.

When tiles are used instead of strips, the three strip tags, RowsPerStrip, StripByteCounts, and StripOffsets, are replaced by the tags TileWidth, TileLength, TileOffsets, and TileByteCounts. As you might have guessed, the tile tags are used in much the same way that the strips tags are. And, like strips, tiles are either all uncompressed, or all compressed using the same scheme. Also, TIFF images are either striped or tiled, but never both.

TileWidth and TileLength describe the size of the tiles storing the image data. The values of TileWidth and TileLength must be a multiple of 16, and all tiles in a subfile are always the same size. These are important compatibility considerations for some applications, especially when using the tile-oriented JPEG compression scheme. The TIFF 6.0 specification recommends that tiles should contain 4K to 32K of image data before compression. Finally, tiles need not be square. Rectangular

tiles compress just as well.

The TileWidth and TileLength tag values can be used to determine the number of tiles in non-YCbCr image subfiles:

```
TilesAcross = (ImageWidth + (TileWidth - 1)) / TileWidth;
TilesDown = (ImageLength + (TileLength - 1)) / TileLength;
TilesInImage = TilesAcross * TilesDown;
```

If the image is separated into planes (PlanarConfiguration = 2), the number of tiles is calculated like this:

```
TilesInImage = TilesAcross * TilesDown * SamplesPerPixel;
```

The TileOffsets tag contains an array of offsets to the first byte of each tile. Tiles are not necessarily stored in a contiguous sequence in a subfile. Each tile has a separate location offset value and is independent of all other tiles in the subfile. The offsets in this tag are ordered left-to-right and top-to-bottom. If the image data is separated into planes, all of the offsets for the first plane are stored first, followed by the offsets for the second plane, and so on. The number of offset values in this tag are equal to the number of tiles in the image (PlanarConfiguration = 1) or the number of tiles multiplied by the SamplesPerPixel tag value (PlanarConfiguration = 2). All offset values are interpreted from the beginning of the TIFF file.

The final tag, TileByteCounts, contains the number of bytes in each compressed tile. The number of values in this tag is also equal to the number of tags in the image, and the values are ordered the same way as the values in the TileOffsets tag.

Normally, a tile size is chosen that fits an image exactly. An image 6400 pixels wide by 4800 pixels long may be divided evenly into 150 tiles, each 640 pixels wide by 320 pixels long. However, not all image dimensions are divisible by 16. An image 2200 pixels wide by 2850 long cannot be evenly divided into tiles whose size must be multiples of 16. The solution is to choose a "best-fit" tile size and fill out the image data with padding.

To find a best-fit tile size, we must choose a tile size that minimally overlaps the size of the image. In this example, we want to use tiles that are 256 pixels wide by 320 pixels long, roughly the same aspect ratio as the image. Using tiles this size requires that 104 extra pixels of padding be added to each row and that 30 additional rows be added to the image length. The size of the image data plus padding is now 2304 pixels wide by 2880 pixels long and can be evenly divided among 81 of our 256 by 320 pixel tiles.

In this example, you may notice that the total amount of padding added to the image before tiling is 365,520 pixels. For a 1-bit image, this equals an extra 45,690 bytes of image data. No appreciable gain in compression results from tiling small images. Also, avoid using tiles that are excessively large and require excessive amounts of padding.

Compression

TIFF supports perhaps more types of data compression than any other image file format. It is also quite possible to use an unsupported compression scheme just by adding the needed user-defined tags. TIFF 4.0 supported only Run-Length Encoding (RLE) and CCITT T.4 and T.6 compression. These compression schemes are typically only for use with 8-bit color, and gray-scale and 1-bit black-and-white images, respectively. TIFF 5.0 added the LZW compression scheme, typically for color images, and TIFF 6.0 added the JPEG compression method for use with continuous-tone color and gray-scale images. (All of these data compression schemes, including a variety of RLE algorithms, are discussed in Chapter 9.)

LZW Is Not Free

If you are creating or modifying software that implements the LZW algorithm, be aware that under certain circumstances, you will need to pay a licensing fee for the use of LZW.

Unisys Corporation owns the patent for the LZW codec (encoding/decoding algorithm) and requires that a licensing fee be paid for each software program which implements the LZW algorithm.

Many people have concluded that the Unisys licensing claim applies only to LZW encoders (software that creates LZW data) and not to LZW decoders (software that only reads LZW data). However, Unisys believes that its patent covers the full LZW codec and requires a licensing fee even for software that reads, but does not write, LZW data.

For more information about the entire issue of LZW licensing, refer to the section called "LZW Legal Issues" in Chapter 9. For a popular alternative to graphics file formats that use LZW, consider using the Portable Network Graphics (PNG) file format.

TIFF uses the PackBits RLE compression scheme found in the Macintosh toolbox. PackBits is a simple and effective algorithm for compressing data and is easy to implement. The name "PackBits" would lead a programmer to believe that it is a bit-wise RLE, packing runs of bits. However, PackBits is a byte-wise RLE and is most efficient at encoding runs of bytes.

PackBits actually has three types of data packets that may be written to the encoded data stream. The first is a two-byte *encoded run packet*. The first byte (the run-count byte) indicates the number of bytes in the run, and the second byte stores the value of each byte in the run. The actual run-count value stored is in the range 0 to 127 and represents the values 1 to 128 (run count + 1).

Another type of packet, the *literal run packet*, stores 12 to 128 bytes literally in the encoded data stream without compressing them. Literal run packets are used to store data with very few runs, as found in very complex or noisy images. The literal run packet's run count is in the range of -127 to -1, indicating that 2 to 128 run values (-(run count) + 1) follow the run count byte.

The last type of packet is the no-op packet. No-ops are one byte in length and have a value of -128. The no-op packet has no use in the PackBits compression scheme and is therefore never found in PackBits-encoded data.

Decompressing PackBits-encoded data is a simple matter of reading a packet of encoded data and converting it to the appropriate byte run. Once again, the run-count byte value is stored one less than the actual number of bytes in the run. It is therefore necessary to add one to the run-count value before using it.

Refer to the TIFF 6.0 specification for more information on PackBits compression, and to Chapter 9, for more information on RLE algorithms.

NOTE: Problems with TIFF 6.0 JPEG
Commentary by Dr. Tom Lane of the Independent JPEG Group, a member of the TIFF Advisory Committee

TIFF 6.0 added JPEG to the list of TIFF compression schemes. Unfortunately, the approach taken in the 6.0 specification is a very poor design. A new design has been developed by the TIFF Advisory Committee. If you are considering implementing JPEG in TIFF, I strongly urge you to follow the revised design described in TIFF Tech Note #2 rather than that of the 6.0 spec.

The fundamental problem with the TIFF 6.0 JPEG design is that JPEG's various tables and parameters are broken out as separate fields, which the TIFF control logic must manage. This is bad software engineering--that information should be private to the JPEG compressor/decompressor. Worse, the fields themselves are specified without thought for future extension and without regard to well-established TIFF conventions. Here are some of the more significant problems:

- **The JPEG table fields use a highly nonstandard layout: rather than containing data directly in the field structure, the fields hold pointers to information elsewhere in the file. This requires special-purpose code to be added to *every* TIFF-manipulating application. Even a trivial TIFF editor (for example, a program to add an ImageDescription field to a TIFF file) must be explicitly aware of the internal structure of the JPEG-related tables, or it will probably break the file. Every other auxiliary field in TIFF follows the normal TIFF rules and can be copied or relocated by standard code.**
- **The specification requires the TIFF control logic to know a great deal about JPEG details--for example, such arcana as how to compute the length of a Huffman code table. The length is not supplied in the field structure and can be found only by inspecting the table contents.**
- **The design specifies separate Huffman tables for each color component. This neglects the fact that baseline JPEG codecs may**

support only two sets of Huffman tables. Thus, an encoder must either waste space (by storing duplicate Huffman tables) or violate the TIFF convention that prohibits duplicate pointers. Furthermore, baseline decoders must test to see which tables are identical--a waste of time and code space.

- **The JPEGInterchangeFormat field again violates the proscription against duplicate pointers; it envisions having the normal strip/tile pointers pointing into the larger data area pointed to by JPEGInterchangeFormat. TIFF editing applications must be specifically aware of this relationship, because they must maintain it or, if they can't, must delete the JPEGInterchangeFormat field.**
- **The JPEGQTables field is fixed at a byte per table entry; there is no way to support 16-bit quantization values. This is a serious impediment to extending TIFF to use 12-bit JPEG.**
- **The design cannot support using different quantization tables in different strips/tiles of an image (so as to encode some areas at higher quality than others). Furthermore, because quantization tables are tied one-for-one to color components, the design cannot support table switching options that are likely to be added in future JPEG extensions.**

In addition to these major design errors, the TIFF 6.0 JPEG specification is seriously ambiguous. In particular, several incompatible interpretations are possible for its handling of JPEG restart markers, and Section 22, "JPEG Compression," actually contradicts Section 15, "Tiled Images," about the restrictions on tile sizes.

Finally, the 6.0 design creates problems for implementations that need to keep the JPEG codec separate from the TIFF control logic--consider using a JPEG chip that was not designed specifically for TIFF. JPEG codecs generally want to produce or consume a standard JPEG datastream, not just raw data. (If they do handle raw data, a separate out-of-band mechanism must be provided to load tables into the codec.) With such a codec, the TIFF control logic must be prepared to parse JPEG markers to create the TIFF table fields (when encoding) or to synthesize JPEG markers from the fields (when decoding). Of course, this means that the control logic must know a great deal more about JPEG than we would like. The parsing and reconstruction of the markers also represents a fair amount of unnecessary work.

Due to all these problems, the TIFF Advisory Committee has developed a replacement JPEG-in-TIFF scheme. The rough outline is as follows:

1. **Each image segment (strip or tile) in a JPEG-compressed TIFF image contains a legal JPEG datastream, complete with all markers. This data forms an independent image of the proper dimensions for the strip or tile.**

2. **To avoid duplicate tables in a multi-segment file, segments may use the JPEG "abbreviated image data" datastream structure, in which DQT and DHT tables are omitted. The common tables are to be supplied in a JPEG "abbreviated table specification" datastream, which is contained in a newly defined "JPEGTables" TIFF field. Because the tables in question typically amount to 550 bytes or so, the savings are worthwhile.**

3. **All the field definitions in the existing Section 22, "JPEG Compression," are deleted. (In practice, those field tag values will remain reserved indefinitely, and this scheme will use a new Compression code, Compression = 7. Implementations that have TIFF 6.0-style files to contend with may continue to read them, using whatever interpretation of 6.0 they used before.)**

This replacement design is described in TIFF Tech Note #2. The Tech Note is currently being distributed in draft form, because Adobe has not yet formally accepted it. However, I expect that the Note will be accepted as is, and that the design it describes will replace the existing Section 22 when version 7.0 of the TIFF spec is released.

If you are considering implementing JPEG in TIFF, please use the design of the Tech Note. The 6.0 JPEG design has not been widely implemented, and with any luck it will stay that way.

For Further Information

For further information about the TIFF format, see the specification.

TIFF was formerly maintained by the Aldus Developer's Association. Aldus has recently merged with Adobe Systems, which now holds the copyright to the TIFF specification and administers and maintains the TIFF format.

All information on the TIFF format may now be obtained through the Adobe Developer Support group. However, this group supplies only general TIFF information and does not provide any TIFF tutoring, sample TIFF source code, or sample TIFF files. Contact the Adobe Developer Support group, at *devsup-person@adobe.com*

Questions about the the Adobe Developer's Association should be directed to:

Adobe Developer's Association
1585 Charleston Road
P.O. Box 7900
Mountain View CA 94039-7900
Voice: 415-961-4111
FAX: 415-967-9231
FTP: *ftp://ftp.adobe.com*
WWW: *http://www.adobe.com/Support/ADA.html*
BBS: 206-623-6984

Adobe distributes the TIFF 6.0 specifiction in PDF format in the "Technical Notes for Developers" section on the Adobe homepage, at:

http://www.adobe.com/Support/TechNotes.html#tiff
http://www.adobe.com/PDFs/TN/TIFF6.pdf

or on the Adobe FTP server:

ftp://ftp.adobe.com/pub/adobe/DeveloperSupport/TechNotes/PDFfiles/TIFF6.pdf

or in paper form for $25US by calling 1-800-831-6395.

TIFF support in Europe may be obtained via email or from Adobe's BBS in Edinburgh, Scotland:

Email: *eurosupport@adobe.com*
BBS: +44 131 458 4666

The Adobe Acrobat reader for PDF files may be obtained for free from:

ftp://ftp.adobe.com/pub/adobe/Applications/Acrobat/

Technical information on Aldus products, including the TIFF Class F specification, is available from Adobe's Automated Technical Support for Aldus Products FAXback service in which information may be requested automatically via a FAX machine. This service may be reached at 800-288-6832 (toll-free), or 206-628-5728.

You will also find useful TIFF information and tools at:

ftp://ftp.sgi.com/textonly/tiff/ (maintained by Sam Leffler)
http://www-mipl.jpl.nasa.gov/~ndr/tiff/textonly.html (maintained by Niles Ritter)

See the following references for more information about TIFF:

Aldus Corporation. *TIFF Developer's Toolkit*, Revision 5.0. Seattle, WA, November 1988.

Aldus Corporation. *TIFF Developer's Toolkit*, Revision 6.0, Seattle, WA, June 1992.

Aldus Corporation. *Aldus Developer News*, Seattle, WA.

Campbell, Joe. *The Spirit of TIFF Class F*, Cygnet Technologies, Berkeley, CA. April 1990.

Hewlett-Packard Company. *HP TIFF Developer's Manual*, Greeley, CO, November 1988.

Katz, Alan, and Danny Cohen, "RFC 1314: A File Format for the Exchange of Images in the Internet," USC Information Sciences Institute, Marina Del Rey, CA, April 1992.

This document outlines the use of TIFF as a standard format for

exchanging facsimile images within the Internet.

Murray, James. "TIFF File Format," *C Gazette*, Winter 1990-91, pp. 27-42.

RFC 804, "Standardization of Group 3 Facsimile Apparatus for Document Transmission."

This document is a draft of the CCITT Recommendation T.4 explaining the CCITT Group 3 encoding scheme used by TIFF.

More Resources

Terms of Service | Privacy Policy | Contact Info