GRAPHICS INTERCHANGE FORMAT(sm)

Version 89a

Cover Sheet for the GIF89a Specification

DEFERRED CLEAR CODE IN LZW COMPRESSION

There has been confusion about where clear codes can be found in the data stream. As the specification says, they may appear at anytime. There is not a requirement to send a clear code when the string table is full.

It is the encoder's decision as to when the table should be cleared. When the table is full, the encoder can chose to use the table as is, making no changes to it until the encoder chooses to clear it. The encoder during this time sends out codes that are of the maximum Code Size.

As we can see from the above, when the decoder's table is full, it must not change the table until a clear code is received. The Code Size is that of the maximum Code Size. Processing other than this is done normally.

Because of a large base of decoders that do not handle the decompression in this manner, we ask developers of GIF encoding software to NOT implement this feature until at least January 1991 and later if they see that their particular market is not ready for it. This will give developers of GIF decoding software time to implement this feature and to get it into the hands of their clients before the decoders start "breaking" on the new GIF's. It is not required that encoders change their software to take advantage of the deferred clear code, but it is for decoders.

APPLICATION EXTENSION BLOCK - APPLICATION IDENTIFIER

There will be a Courtesy Directory file located on CompuServe in the PICS forum. This directory will contain Application Identifiers for Application Extension Blocks that have been used by developers of GIF applications. This file is intended to help keep developers that wish to create Application Extension Blocks from using the same Application Identifiers.

This is not an official directory; it is for voluntary participation only and does not guarantee that someone will not use the same identifier.

E-Mail can be sent to Larry Wood (forum manager of PICS) indicating the request for inclusion in this file with an identifier.

CompuServe Incorporated
Document Date : 31 July 1990

Graphics Interchange Format
Programming Reference

Table of Contents

1

1. Disclaimer.

The information provided herein is subject to change without notice. In no event will CompuServe Incorporated be liable for damages, including any loss of revenue, loss of profits or other incidental or consequential damages arising out of the use or inability to use the information; CompuServe Incorporated makes no claim as to the suitability of the information.

2. Foreword.

This document defines the Graphics Interchange Format(sm). The specification given here defines version 89a, which is an extension of version 87a.

The Graphics Interchange Format(sm) as specified here should be considered complete; any deviation from it should be considered invalid, including but not limited to, the use of reserved or undefined fields within control or data blocks, the inclusion of extraneous data within or between blocks, the use of methods or algorithms not specifically listed as part of the format, etc. In general, any and all deviations, extensions or modifications not specified in this document should be considered to be in violation of the format and should be avoided.

3. Licensing.

The Graphics Interchange Format(c) is the copyright property of CompuServe Incorporated. Only CompuServe Incorporated is authorized to define, redefine, enhance, alter, modify or change in any way the definition of the format.

CompuServe Incorporated hereby grants a limited, non-exclusive, royalty-free license for the use of the Graphics Interchange Format(sm) in computer software; computer software utilizing GIF(sm) must acknowledge ownership of the Graphics Interchange Format and its Service Mark by CompuServe Incorporated, in User and Technical Documentation. Computer software utilizing GIF, which is distributed or may be distributed without User or Technical Documentation must display to the screen or printer a message acknowledging ownership of the Graphics Interchange Format and the Service Mark by CompuServe Incorporated; in this case, the acknowledgement may be displayed in an opening screen or leading banner, or a closing screen or trailing banner. A message such as the following may be used:

> "The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated."

For further information, please contact :

> CompuServe Incorporated
> Graphics Technology Department
> 5000 Arlington Center Boulevard

Columbus, Ohio 43220
U. S. A.

CompuServe Incorporated maintains a mailing list with all those individuals and organizations who wish to receive copies of this document when it is corrected

or revised. This service is offered free of charge; please provide us with your mailing address.

4. About the Document.

This document describes in detail the definition of the Graphics Interchange Format. This document is intended as a programming reference; it is recommended that the entire document be read carefully before programming, because of the interdependence of the various parts. There is an individual section for each of the Format blocks. Within each section, the sub-section labeled Required Version refers to the version number that an encoder will have to use if the corresponding block is used in the Data Stream. Within each section, a diagram describes the individual fields in the block; the diagrams are drawn vertically; top bytes in the diagram appear first in the Data Stream. Bits within a byte are drawn most significant on the left end. Multi-byte numeric fields are ordered Least Significant Byte first. Numeric constants are represented as Hexadecimal numbers, preceded by "0x". Bit fields within a byte are described in order from most significant bits to least significant bits.

5. General Description.

The Graphics Interchange Format(sm) defines a protocol intended for the on-line transmission and interchange of raster graphic data in a way that is independent of the hardware used in their creation or display.

The Graphics Interchange Format is defined in terms of blocks and sub-blocks which contain relevant parameters and data used in the reproduction of a graphic. A GIF Data Stream is a sequence of protocol blocks and sub-blocks representing a collection of graphics. In general, the graphics in a Data Stream are assumed to be related to some degree, and to share some control information; it is recommended that encoders attempt to group together related graphics in order to minimize hardware changes during processing and to minimize control information overhead. For the same reason, unrelated graphics or graphics which require resetting hardware parameters should be encoded separately to the extent possible.

A Data Stream may originate locally, as when read from a file, or it may originate remotely, as when transmitted over a data communications line. The Format is defined with the assumption that an error-free Transport Level Protocol is used for communications; the Format makes no provisions for error-detection and error-correction.

The GIF Data Stream must be interpreted in context, that is, the application program must rely on information external to the Data Stream to invoke the decoder process.

6. Version Numbers.

The version number in the Header of a Data Stream is intended to identify the minimum set of capabilities required of a decoder in order to fully process the Data Stream. An encoder should use the earliest possible version number that includes all the blocks used in the Data Stream. Within each block section in this document, there is an entry labeled Required Version which specifies the

earliest version number that includes the corresponding block. The encoder should make every attempt to use the earliest version number covering all the blocks in the Data Stream; the unnecessary use of later version numbers will hinder processing by some decoders.

7. The Encoder.

The Encoder is the program used to create a GIF Data Stream. From raster data and other information, the encoder produces the necessary control and data blocks needed for reproducing the original graphics.

The encoder has the following primary responsibilities.

- Include in the Data Stream all the necessary information to reproduce the graphics.

- Insure that a Data Stream is labeled with the earliest possible Version Number that will cover the definition of all the blocks in it; this is to ensure that the largest number of decoders can process the Data Stream.

- Ensure encoding of the graphics in such a way that the decoding process is optimized. Avoid redundant information as much as possible.

- To the extent possible, avoid grouping graphics which might require resetting hardware parameters during the decoding process.

- Set to zero (off) each of the bits of each and every field designated as reserved. Note that some fields in the Logical Screen Descriptor and the Image Descriptor were reserved under Version 87a, but are used under version 89a.

8. The Decoder.

The Decoder is the program used to process a GIF Data Stream. It processes the Data Stream sequentially, parsing the various blocks and sub-blocks, using the control information to set hardware and process parameters and interpreting the data to render the graphics.

The decoder has the following primary responsibilities.

- Process each graphic in the Data Stream in sequence, without delays other than those specified in the control information.

- Set its hardware parameters to fit, as closely as possible, the control information contained in the Data Stream.

9. Compliance.

An encoder or a decoder is said to comply with a given version of the Graphics Interchange Format if and only if it fully conforms with and correctly implements the definition of the standard associated with that version. An

encoder or a decoder may be compliant with a given version number and not compliant with some subsequent version.

10. About Recommendations.

Each block section in this document contains an entry labeled Recommendation; this section lists a set of recommendations intended to guide and organize the use of the particular blocks. Such recommendations are geared towards making the functions of encoders and decoders more efficient, as well as making optimal use of the communications bandwidth. It is advised that these recommendations be followed.

11. About Color Tables.

The GIF format utilizes color tables to render raster-based graphics. A color table can have one of two different scopes: global or local. A Global Color Table is used by all those graphics in the Data Stream which do not have a Local Color Table associated with them. The scope of the Global Color Table is the entire Data Stream. A Local Color Table is always associated with the graphic that immediately follows it; the scope of a Local Color Table is limited to that single graphic. A Local Color Table supersedes a Global Color Table, that is, if a Data Stream contains a Global Color Table, and an image has a Local Color Table associated with it, the decoder must save the Global Color Table, use the Local Color Table to render the image, and then restore the Global Color Table. Both types of color tables are optional, making it possible for a Data Stream to contain numerous graphics without a color table at all. For this reason, it is recommended that the decoder save the last Global Color Table used until another Global Color Table is encountered. In this way, a Data Stream which does not contain either a Global Color Table or a Local Color Table may be processed using the last Global Color Table saved. If a Global Color Table from a previous Stream is used, that table becomes the Global Color Table of the present Stream. This is intended to reduce the overhead incurred by color tables. In particular, it is recommended that an encoder use only one Global Color Table if all the images in related Data Streams can be rendered with the same table. If no color table is available at all, the decoder is free to use a system color table or a table of its own. In that case, the decoder may use a color table with as many colors as its hardware is able to support; it is recommended that such a table have black and white as its first two entries, so that monochrome images can be rendered adequately.

The Definition of the GIF Format allows for a Data Stream to contain only the Header, the Logical Screen Descriptor, a Global Color Table and the GIF Trailer. Such a Data Stream would be used to load a decoder with a Global Color Table, in preparation for subsequent Data Streams without a color table at all.

12. Blocks, Extensions and Scope.

Blocks can be classified into three groups : Control, Graphic-Rendering and Special Purpose. Control blocks, such as the Header, the Logical Screen Descriptor, the Graphic Control Extension and the Trailer, contain information used to control the process of the Data Stream or information used in setting hardware parameters. Graphic-Rendering blocks such as the Image Descriptor and

the Plain Text Extension contain information and data used to render a graphic on the display device. Special Purpose blocks such as the Comment Extension and the Application Extension are neither used to control the process of the Data Stream nor do they contain information or data used to render a graphic on the display device. With the exception of the Logical Screen Descriptor and the Global Color Table, whose scope is the entire Data Stream, all other Control blocks have a limited scope, restricted to the Graphic-Rendering block that follows them. Special Purpose blocks do not delimit the scope of any Control blocks; Special Purpose blocks are transparent to the decoding process. Graphic-Rendering blocks and extensions are used as scope delimiters for Control blocks and extensions. The labels used to identify labeled blocks fall into three ranges : 0x00-0x7F (0-127) are the Graphic Rendering blocks, excluding the Trailer (0x3B); 0x80-0xF9 (128-249) are the Control blocks; 0xFA-0xFF (250-255) are the Special Purpose blocks. These ranges are defined so that decoders can handle block scope by appropriately identifying block labels, even when the block itself cannot be processed.

13. Block Sizes.

The Block Size field in a block, counts the number of bytes remaining in the block, not counting the Block Size field itself, and not counting the Block Terminator, if one is to follow. Blocks other than Data Blocks are intended to be of fixed length; the Block Size field is provided in order to facilitate skipping them, not to allow their size to change in the future. Data blocks and sub-blocks are of variable length to accommodate the amount of data.

14. Using GIF as an embedded protocol.

As an embedded protocol, GIF may be part of larger application protocols, within which GIF is used to render graphics. In such a case, the application protocol could define a block within which the GIF Data Stream would be contained. The application program would then invoke a GIF decoder upon encountering a block of type GIF. This approach is recommended in favor of using Application Extensions, which become overhead for all other applications that do not process them. Because a GIF Data Stream must be processed in context, the application must rely on some means of identifying the GIF Data Stream outside of the Stream itself.

15. Data Sub-blocks.

a. Description. Data Sub-blocks are units containing data. They do not have a label, these blocks are processed in the context of control blocks, wherever data blocks are specified in the format. The first byte of the Data sub-block indicates the number of data bytes to follow. A

data sub-block may contain from 0 to 255 data bytes. The size of the block does not account for the size byte itself, therefore, the empty sub-block is one whose size field contains 0x00.

b. Required Version. 87a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Block Size	Byte
1			
2			
3			
		Data Values	Byte
up			
to			
255			

i) Block Size - Number of bytes in the Data Sub-block; the size must be within 0 and 255 bytes, inclusive.

ii) Data Values - Any 8-bit value. There must be exactly as many Data Values as specified by the Block Size field.

d. Extensions and Scope. This type of block always occurs as part of a larger unit. It does not have a scope of itself.

e. Recommendation. None.

16. Block Terminator.

a. Description. This zero-length Data Sub-block is used to terminate a sequence of Data Sub-blocks. It contains a single byte in the position of the Block Size field and does not contain data.

b. Required Version. 87a.

c. Syntax.

7 6 5 4 3 2 1 0	Field Name	Type

```
      +---------------+
   0  |               |       Block Size                    Byte
      +---------------+
```

i) Block Size - Number of bytes in the Data Sub-block; this field contains the fixed value 0x00.

ii) Data Values - This block does not contain any data.

d. Extensions and Scope. This block terminates the immediately preceding sequence of Data Sub-blocks. This block cannot be modified by any extension.

e. Recommendation. None.

17. Header.

a. Description. The Header identifies the GIF Data Stream in context. The Signature field marks the beginning of the Data Stream, and the Version field identifies the set of capabilities required of a decoder to fully process the Data Stream. This block is REQUIRED; exactly one Header must be present per Data Stream.

b. Required Version. Not applicable. This block is not subject to a version number. This block must appear at the beginning of every Data Stream.

c. Syntax.

```
       7 6 5 4 3 2 1 0        Field Name                   Type
      +---------------+
   0  |               |       Signature                    3 Bytes
      +-             -+
   1  |               |
      +-             -+
   2  |               |
      +---------------+
   3  |               |       Version                      3 Bytes
      +-             -+
   4  |               |
      +-             -+
   5  |               |
      +---------------+
```

i) Signature - Identifies the GIF Data Stream. This field contains the fixed value 'GIF'.

ii) Version - Version number used to format the data stream. Identifies the minimum set of capabilities necessary to a decoder to fully process the contents of the Data Stream.

Version Numbers as of 10 July 1990 : "87a" - May 1987

Version numbers are ordered numerically increasing on the first two digits starting with 87 (87,88,...,99,00,...,85,86) and alphabetically increasing on the third character (a,...,z).

iii) Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

d. Recommendations.

i) Signature - This field identifies the beginning of the GIF Data Stream; it is not intended to provide a unique signature for the identification of the data. It is recommended that the GIF Data Stream be identified externally by the application. (Refer to Appendix G for on-line identification of the GIF Data Stream.)

ii) Version - ENCODER : An encoder should use the earliest possible version number that defines all the blocks used in the Data Stream. When two or more Data Streams are combined, the latest of the individual version numbers should be used for the resulting Data Stream. DECODER : A decoder should attempt to process the data stream to the best of its ability; if it encounters a version number which it is not capable of processing fully, it should nevertheless, attempt to process the data stream to the best of its ability, perhaps after warning the user that the data may be incomplete.

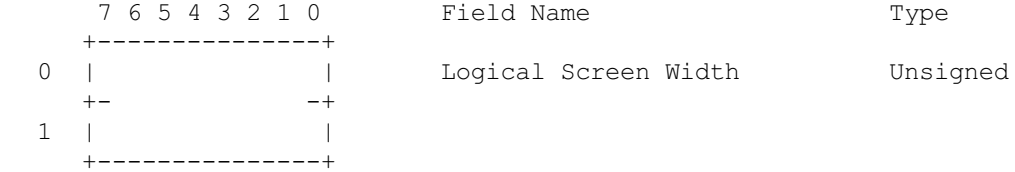
18. Logical Screen Descriptor.

a. Description. The Logical Screen Descriptor contains the parameters necessary to define the area of the display device within which the images will be rendered. The coordinates in this block are given with respect to the top-left corner of the virtual screen; they do not necessarily refer to absolute coordinates on the display device. This implies that they could refer to window coordinates in a window-based environment or printer coordinates when a printer is used.

This block is REQUIRED; exactly one Logical Screen Descriptor must be present per Data Stream.

b. Required Version. Not applicable. This block is not subject to a version number. This block must appear immediately after the Header.

c. Syntax.



2	Logical Screen Height	Unsigned
3		
4	<Packed Fields>	See below
5	Background Color Index	Byte
6	Pixel Aspect Ratio	Byte

<Packed Fields> =	Global Color Table Flag	1 Bit
	Color Resolution	3 Bits
	Sort Flag	1 Bit
	Size of Global Color Table	3 Bits

i) Logical Screen Width - Width, in pixels, of the Logical Screen where the images will be rendered in the displaying device.

ii) Logical Screen Height - Height, in pixels, of the Logical Screen where the images will be rendered in the displaying device.

iii) Global Color Table Flag - Flag indicating the presence of a Global Color Table; if the flag is set, the Global Color Table will immediately follow the Logical Screen Descriptor. This flag also selects the interpretation of the Background Color Index; if the flag is set, the value of the Background Color Index field should be used as the table index of the background color. (This field is the most significant bit of the byte.)

Values : 0 - No Global Color Table follows, the Background Color Index field is meaningless.
1 - A Global Color Table will immediately follow, the Background Color Index field is meaningful.

iv) Color Resolution - Number of bits per primary color available to the original image, minus 1. This value represents the size of the entire palette from which the colors in the graphic were selected, not the number of colors actually used in the graphic. For example, if the value in this field is 3, then the palette of the original image had 4 bits per primary color available to create the image. This value should be set to indicate the richness of the original palette, even if not every color from the whole palette is available on the source machine.

v) Sort Flag - Indicates whether the Global Color Table is sorted. If the flag is set, the Global Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.

Values : 0 - Not ordered.
1 - Ordered by decreasing importance, most important color first.

vi) Size of Global Color Table - If the Global Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Global Color Table. To determine that actual size of the color table, raise 2 to [the value of the field + 1]. Even if there is no Global Color Table specified, set this field according to the above formula so that decoders can choose the best graphics mode to display the stream in. (This field is made up of the 3 least significant bits of the byte.)

vii) Background Color Index - Index into the Global Color Table for

the Background Color. The Background Color is the color used for those pixels on the screen that are not covered by an image. If the Global Color Table Flag is set to (zero), this field should be zero and should be ignored.

viii) Pixel Aspect Ratio - Factor used to compute an approximation of the aspect ratio of the pixel in the original image. If the value of the field is not 0, this approximation of the aspect ratio is computed based on the formula:

Aspect Ratio = (Pixel Aspect Ratio + 15) / 64

The Pixel Aspect Ratio is defined to be the quotient of the pixel's width over its height. The value range in this field allows specification of the widest pixel of 4:1 to the tallest pixel of 1:4 in increments of 1/64th.

Values : 0 - No aspect ratio information is given.
1..255 - Value used in the computation.

d. Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

e. Recommendations. None.

19. Global Color Table.

a. Description. This block contains a color table, which is a sequence of bytes representing red-green-blue color triplets. The Global Color Table is used by images without a Local Color Table and by Plain Text Extensions. Its presence is marked by the Global Color Table Flag being set to 1 in the Logical Screen Descriptor; if present, it immediately follows the Logical Screen Descriptor and contains a number of bytes equal to

3 x 2^(Size of Global Color Table+1).

This block is OPTIONAL; at most one Global Color Table may be present per Data Stream.

b. Required Version. 87a

11

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Red 0	Byte
1		Green 0	Byte
2		Blue 0	Byte
3		Red 1	Byte
		Green 1	Byte
up			
to	. . . .	...	
		Green 255	Byte
767		Blue 255	Byte

d. Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

e. Recommendation. None.

20. Image Descriptor.

a. Description. Each image in the Data Stream is composed of an Image Descriptor, an optional Local Color Table, and the image data. Each image must fit within the boundaries of the Logical Screen, as defined in the Logical Screen Descriptor.

The Image Descriptor contains the parameters necessary to process a table based image. The coordinates given in this block refer to coordinates within the Logical Screen, and are given in pixels. This block is a Graphic-Rendering Block, optionally preceded by one or more Control

blocks such as the Graphic Control Extension, and may be optionally followed by a Local Color Table; the Image Descriptor is always followed by the image data.

This block is REQUIRED for an image. Exactly one Image Descriptor must be present per image in the Data Stream. An unlimited number of images may be present per Data Stream.

b. Required Version. 87a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Image Separator	Byte
1		Image Left Position	Unsigned
2			
3		Image Top Position	Unsigned
4			
5		Image Width	Unsigned
6			
7		Image Height	Unsigned
8			
9	| | | | |	<Packed Fields>	See below

<Packed Fields> =		
	Local Color Table Flag	1 Bit
	Interlace Flag	1 Bit
	Sort Flag	1 Bit
	Reserved	2 Bits
	Size of Local Color Table	3 Bits

i) Image Separator - Identifies the beginning of an Image Descriptor. This field contains the fixed value 0x2C.

ii) Image Left Position - Column number, in pixels, of the left edge of the image, with respect to the left edge of the Logical Screen. Leftmost column of the Logical Screen is 0.

iii) Image Top Position - Row number, in pixels, of the top edge of the image with respect to the top edge of the Logical Screen. Top

row of the Logical Screen is 0.

iv) Image Width - Width of the image in pixels.

v) Image Height - Height of the image in pixels.

vi) Local Color Table Flag - Indicates the presence of a Local Color Table immediately following this Image Descriptor. (This field is the most significant bit of the byte.)

Values : 0 - Local Color Table is not present. Use Global Color Table if available.
1 - Local Color Table present, and to follow immediately after this Image Descriptor.

vii) Interlace Flag - Indicates if the image is interlaced. An image is interlaced in a four-pass interlace pattern; see Appendix E for details.

Values : 0 - Image is not interlaced.
1 - Image is interlaced.

viii) Sort Flag - Indicates whether the Local Color Table is sorted. If the flag is set, the Local Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.

Values : 0 - Not ordered.
1 - Ordered by decreasing importance, most important color first.

ix) Size of Local Color Table - If the Local Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Local Color Table. To determine that actual size of the color table, raise 2 to the value of the field + 1. This value should be 0 if there is no Local Color Table specified. (This field is made up of the 3 least significant bits of the byte.)

d. Extensions and Scope. The scope of this block is the Table-based Image Data Block that follows it. This block may be modified by the Graphic Control Extension.

e. Recommendation. None.

21. Local Color Table.

a. Description. This block contains a color table, which is a sequence of bytes representing red-green-blue color triplets. The Local Color Table is used by the image that immediately follows. Its presence is marked by

the Local Color Table Flag being set to 1 in the Image Descriptor; if present, the Local Color Table immediately follows the Image Descriptor and contains a number of bytes equal to

3x2^(Size of Local Color Table+1).

If present, this color table temporarily becomes the active color table and the following image should be processed using it. This block is OPTIONAL; at most one Local Color Table may be present per Image Descriptor and its scope is the single image associated with the Image Descriptor that precedes it.

b. Required Version. 87a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Red 0	Byte
1		Green 0	Byte
2		Blue 0	Byte
3		Red 1	Byte
		Green 1	Byte
up			
to	. . . .	...	
		Green 255	Byte
767		Blue 255	Byte

d. Extensions and Scope. The scope of this block is the Table-based Image Data Block that immediately follows it. This block cannot be modified by any extension.

e. Recommendations. None.

22. Table Based Image Data.

a. Description. The image data for a table based image consists of a sequence of sub-blocks, of size at most 255 bytes each, containing an index into the active color table, for each pixel in the image. Pixel indices are in order of left to right and from top to bottom. Each index must be within the range of the size of the active color table, starting

at 0. The sequence of indices is encoded using the LZW Algorithm with variable-length code, as described in Appendix F

b. Required Version. 87a.

c. Syntax. The image data format is as follows:

7 6 5 4 3 2 1 0	Field Name	Type
	LZW Minimum Code Size	Byte
/	Image Data	Data Sub-blocks

i) LZW Minimum Code Size. This byte determines the initial number of bits used for LZW codes in the image data, as described in Appendix F.

d. Extensions and Scope. This block has no scope, it contains raster data. Extensions intended to modify a Table-based image must appear before the corresponding Image Descriptor.

e. Recommendations. None.

23. Graphic Control Extension.

a. Description. The Graphic Control Extension contains parameters used when processing a graphic rendering block. The scope of this extension is the first graphic rendering block to follow. The extension contains only one data sub-block.

This block is OPTIONAL; at most one Graphic Control Extension may precede a graphic rendering block. This is the only limit to the number of Graphic Control Extensions that may be contained in a Data Stream.

b. Required Version. 89a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Extension Introducer	Byte
1		Graphic Control Label	Byte
0		Block Size	Byte
1		<Packed Fields>	See below

```
    +---------------+
2   |               |       Delay Time                    Unsigned
    +-             -+
3   |               |
    +---------------+
4   |               |       Transparent Color Index       Byte
    +---------------+

    +---------------+
0   |               |       Block Terminator              Byte
    +---------------+
```

```
 <Packed Fields>  =     Reserved                     3 Bits
                        Disposal Method              3 Bits
                        User Input Flag              1 Bit
                        Transparent Color Flag       1 Bit
```

i) Extension Introducer - Identifies the beginning of an extension

block. This field contains the fixed value 0x21.

ii) Graphic Control Label - Identifies the current block as a Graphic Control Extension. This field contains the fixed value 0xF9.

iii) Block Size - Number of bytes in the block, after the Block Size field and up to but not including the Block Terminator. This field contains the fixed value 4.

iv) Disposal Method - Indicates the way in which the graphic is to be treated after being displayed.

```
Values :    0 -   No disposal specified. The decoder is
                  not required to take any action.
            1 -   Do not dispose. The graphic is to be left
                  in place.
            2 -   Restore to background color. The area used by the
                  graphic must be restored to the background color.
            3 -   Restore to previous. The decoder is required to
                  restore the area overwritten by the graphic with
                  what was there prior to rendering the graphic.
         4-7 -    To be defined.
```

v) User Input Flag - Indicates whether or not user input is expected before continuing. If the flag is set, processing will continue when user input is entered. The nature of the User input is determined by the application (Carriage Return, Mouse Button Click, etc.).

```
Values :    0 -   User input is not expected.
            1 -   User input is expected.
```

When a Delay Time is used and the User Input Flag is set, processing will continue when user input is received or when the

delay time expires, whichever occurs first.

vi) Transparency Flag - Indicates whether a transparency index is given in the Transparent Index field. (This field is the least significant bit of the byte.)

Values : 0 - Transparent Index is not given.
1 - Transparent Index is given.

vii) Delay Time - If not 0, this field specifies the number of hundredths (1/100) of a second to wait before continuing with the processing of the Data Stream. The clock starts ticking immediately after the graphic is rendered. This field may be used in conjunction with the User Input Flag field.

viii) Transparency Index - The Transparency Index is such that when encountered, the corresponding pixel of the display device is not modified and processing goes on to the next pixel. The index is present if and only if the Transparency Flag is set to 1.

ix) Block Terminator - This zero-length data block marks the end of

the Graphic Control Extension.

d. Extensions and Scope. The scope of this Extension is the graphic rendering block that follows it; it is possible for other extensions to be present between this block and its target. This block can modify the Image Descriptor Block and the Plain Text Extension.

e. Recommendations.

i) Disposal Method - The mode Restore To Previous is intended to be used in small sections of the graphic; the use of this mode imposes severe demands on the decoder to store the section of the graphic that needs to be saved. For this reason, this mode should be used sparingly. This mode is not intended to save an entire graphic or large areas of a graphic; when this is the case, the encoder should make every attempt to make the sections of the graphic to be restored be separate graphics in the data stream. In the case where a decoder is not capable of saving an area of a graphic marked as Restore To Previous, it is recommended that a decoder restore to the background color.

ii) User Input Flag - When the flag is set, indicating that user input is expected, the decoder may sound the bell (0x07) to alert the user that input is being expected. In the absence of a specified Delay Time, the decoder should wait for user input indefinitely. It is recommended that the encoder not set the User Input Flag without a Delay Time specified.

24. Comment Extension.

a. Description. The Comment Extension contains textual information which is not part of the actual graphics in the GIF Data Stream. It is suitable for including comments about the graphics, credits, descriptions or any

other type of non-control and non-graphic data. The Comment Extension may be ignored by the decoder, or it may be saved for later processing; under no circumstances should a Comment Extension disrupt or interfere with the processing of the Data Stream.

This block is OPTIONAL; any number of them may appear in the Data Stream.

b. Required Version. 89a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Extension Introducer	Byte
1		Comment Label	Byte
N		Comment Data	Data Sub-blocks
0		Block Terminator	Byte

i) Extension Introducer - Identifies the beginning of an extension block. This field contains the fixed value 0x21.

ii) Comment Label - Identifies the block as a Comment Extension. This field contains the fixed value 0xFE.

iii) Comment Data - Sequence of sub-blocks, each of size at most 255 bytes and at least 1 byte, with the size in a byte preceding the data. The end of the sequence is marked by the Block Terminator.

iv) Block Terminator - This zero-length data block marks the end of the Comment Extension.

d. Extensions and Scope. This block does not have scope. This block cannot be modified by any extension.

e. Recommendations.

i) Data - This block is intended for humans. It should contain text using the 7-bit ASCII character set. This block should not be used to store control information for custom processing.

ii) Position - This block may appear at any point in the Data Stream at which a block can begin; however, it is recommended that Comment Extensions do not interfere with Control or Data blocks; they should be located at the beginning or at the end of the Data Stream to the extent possible.

25. Plain Text Extension.

a. Description. The Plain Text Extension contains textual data and the parameters necessary to render that data as a graphic, in a simple form. The textual data will be encoded with the 7-bit printable ASCII characters. Text data are rendered using a grid of character cells

defined by the parameters in the block fields. Each character is rendered in an individual cell. The textual data in this block is to be rendered as mono-spaced characters, one character per cell, with a best fitting font and size. For further information, see the section on Recommendations below. The data characters are taken sequentially from the data portion of the block and rendered within a cell, starting with the upper left cell in the grid and proceeding from left to right and from top to bottom. Text data is rendered until the end of data is reached or the character grid is filled. The Character Grid contains an integral number of cells; in the case that the cell dimensions do not allow for an integral number, fractional cells must be discarded; an encoder must be careful to specify the grid dimensions accurately so that this does not happen. This block requires a Global Color Table to be available; the colors used by this block reference the Global Color Table in the Stream if there is one, or the Global Color Table from a previous Stream, if one was saved. This block is a graphic rendering block, therefore it may be modified by a Graphic Control Extension. This block is OPTIONAL; any number of them may appear in the Data Stream.

b. Required Version. 89a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Extension Introducer	Byte
1		Plain Text Label	Byte
0		Block Size	Byte
1		Text Grid Left Position	Unsigned
2			
3		Text Grid Top Position	Unsigned
4			
5		Text Grid Width	Unsigned
6			
7		Text Grid Height	Unsigned
8			
9		Character Cell Width	Byte
10		Character Cell Height	Byte
11		Text Foreground Color Index	Byte

12		Text Background Color Index	Byte
N		Plain Text Data	Data Sub-blocks
0		Block Terminator	Byte

i) Extension Introducer - Identifies the beginning of an extension block. This field contains the fixed value 0x21.

ii) Plain Text Label - Identifies the current block as a Plain Text Extension. This field contains the fixed value 0x01.

iii) Block Size - Number of bytes in the extension, after the Block Size field and up to but not including the beginning of the data portion. This field contains the fixed value 12.

iv) Text Grid Left Position - Column number, in pixels, of the left edge of the text grid, with respect to the left edge of the Logical Screen.

v) Text Grid Top Position - Row number, in pixels, of the top edge of the text grid, with respect to the top edge of the Logical Screen.

vi) Image Grid Width - Width of the text grid in pixels.

vii) Image Grid Height - Height of the text grid in pixels.

viii) Character Cell Width - Width, in pixels, of each cell in the grid.

ix) Character Cell Height - Height, in pixels, of each cell in the grid.

x) Text Foreground Color Index - Index into the Global Color Table to be used to render the text foreground.

xi) Text Background Color Index - Index into the Global Color Table to be used to render the text background.

xii) Plain Text Data - Sequence of sub-blocks, each of size at most 255 bytes and at least 1 byte, with the size in a byte preceding the data. The end of the sequence is marked by the Block Terminator.

xiii) Block Terminator - This zero-length data block marks the end of the Plain Text Data Blocks.

d. Extensions and Scope. The scope of this block is the Plain Text Data

Block contained in it. This block may be modified by the Graphic Control Extension.

e. Recommendations. The data in the Plain Text Extension is assumed to be preformatted. The selection of font and size is left to the discretion of the decoder. If characters less than 0x20 or greater than 0xf7 are encountered, it is recommended that the decoder display a Space character (0x20). The encoder should use grid and cell dimensions such that an integral number of cells fit in the grid both horizontally as well as vertically. For broadest compatibility, character cell dimensions should be around 8x8 or 8x16 (width x height); consider an image for unusual sized text.

26. Application Extension.

a. Description. The Application Extension contains application-specific information; it conforms with the extension block syntax, as described below, and its block label is 0xFF.

b. Required Version. 89a.

c. Syntax.

	7 6 5 4 3 2 1 0	Field Name	Type
0		Extension Introducer	Byte
1		Extension Label	Byte
0		Block Size	Byte
1			
2			
3		Application Identifier	8 Bytes
4			
5			
6			
7			
8			
9			
10		Appl. Authentication Code	3 Bytes

```
      +-              -+
   11 |                |
      +----------------+

      +================+
      |                |
      |                |       Application Data              Data Sub-blocks
      |                |
      |                |
      +================+

      +----------------+
    0 |                |       Block Terminator              Byte
      +----------------+
```

i) Extension Introducer - Defines this block as an extension. This field contains the fixed value 0x21.

ii) Application Extension Label - Identifies the block as an Application Extension. This field contains the fixed value 0xFF.

iii) Block Size - Number of bytes in this extension block, following the Block Size field, up to but not including the beginning of the Application Data. This field contains the fixed value 11.

iv) Application Identifier - Sequence of eight printable ASCII characters used to identify the application owning the Application Extension.

v) Application Authentication Code - Sequence of three bytes used to authenticate the Application Identifier. An Application program may use an algorithm to compute a binary code that uniquely identifies it as the application owning the Application Extension.

d. Extensions and Scope. This block does not have scope. This block cannot be modified by any extension.

e. Recommendation. None.

27. Trailer.

a. Description. This block is a single-field block indicating the end of the GIF Data Stream. It contains the fixed value 0x3B.

b. Required Version. 87a.

c. Syntax.

```
       7 6 5 4 3 2 1 0        Field Name                   Type
      +----------------+
    0 |                |       GIF Trailer                  Byte
      +----------------+
```

d. Extensions and Scope. This block does not have scope, it terminates the GIF Data Stream. This block may not be modified by any extension.

e. Recommendations. None.

Appendix
A. Quick Reference Table.

Block Name	Required	Label	Ext.	Vers.
Application Extension	Opt. (*)	0xFF (255)	yes	89a
Comment Extension	Opt. (*)	0xFE (254)	yes	89a
Global Color Table	Opt. (1)	none	no	87a
Graphic Control Extension	Opt. (*)	0xF9 (249)	yes	89a
Header	Req. (1)	none	no	N/A
Image Descriptor	Opt. (*)	0x2C (044)	no	87a (89a)
Local Color Table	Opt. (*)	none	no	87a
Logical Screen Descriptor	Req. (1)	none	no	87a (89a)
Plain Text Extension	Opt. (*)	0x01 (001)	yes	89a
Trailer	Req. (1)	0x3B (059)	no	87a
Unlabeled Blocks				
Header	Req. (1)	none	no	N/A
Logical Screen Descriptor	Req. (1)	none	no	87a (89a)
Global Color Table	Opt. (1)	none	no	87a
Local Color Table	Opt. (*)	none	no	87a
Graphic-Rendering Blocks				
Plain Text Extension	Opt. (*)	0x01 (001)	yes	89a
Image Descriptor	Opt. (*)	0x2C (044)	no	87a (89a)
Control Blocks				
Graphic Control Extension	Opt. (*)	0xF9 (249)	yes	89a
Special Purpose Blocks				
Trailer	Req. (1)	0x3B (059)	no	87a
Comment Extension	Opt. (*)	0xFE (254)	yes	89a

Application Extension	Opt. (*)	0xFF (255)	yes	89a

legend:	(1)	if present, at most one occurrence
	(*)	zero or more occurrences
	(+)	one or more occurrences

Notes : The Header is not subject to Version Numbers.
(89a) The Logical Screen Descriptor and the Image Descriptor retained their syntax from version 87a to version 89a, but some fields reserved under version 87a are used under version 89a.

Appendix
B. GIF Grammar.

A Grammar is a form of notation to represent the sequence in which certain objects form larger objects. A grammar is also used to represent the number of objects that can occur at a given position. The grammar given here represents the sequence of blocks that form the GIF Data Stream. A grammar is given by listing its rules. Each rule consists of the left-hand side, followed by some form of equals sign, followed by the right-hand side. In a rule, the right-hand side describes how the left-hand side is defined. The right-hand side consists of a sequence of entities, with the possible presence of special symbols. The following legend defines the symbols used in this grammar for GIF.

Legend:	<>	grammar word
	::=	defines symbol
	*	zero or more occurrences
	+	one or more occurrences
	|	alternate element
	[]	optional element

Example:

<GIF Data Stream> ::= Header <Logical Screen> <Data>* Trailer

This rule defines the entity <GIF Data Stream> as follows. It must begin with a Header. The Header is followed by an entity called Logical Screen, which is defined below by another rule. The Logical Screen is followed by the entity Data, which is also defined below by another rule. Finally, the entity Data is followed by the Trailer. Since there is no rule defining the Header or the Trailer, this means that these blocks are defined in the document. The entity Data has a special symbol (*) following it which means that, at this position,

the entity Data may be repeated any number of times, including 0 times. For further reading on this subject, refer to a standard text on Programming Languages.

The Grammar.

```
<GIF Data Stream> ::=     Header <Logical Screen> <Data>* Trailer

<Logical Screen> ::=      Logical Screen Descriptor [Global Color Table]

<Data> ::=                <Graphic Block>  |
                          <Special-Purpose Block>

<Graphic Block> ::=       [Graphic Control Extension] <Graphic-Rendering Block>

<Graphic-Rendering Block> ::=  <Table-Based Image>  |
                               Plain Text Extension

<Table-Based Image> ::=   Image Descriptor [Local Color Table] Image Data

<Special-Purpose Block> ::=    Application Extension  |
                               Comment Extension
```


NOTE : The grammar indicates that it is possible for a GIF Data Stream to contain the Header, the Logical Screen Descriptor, a Global Color Table and the GIF Trailer. This special case is used to load a GIF decoder with a Global Color Table, in preparation for subsequent Data Streams without color tables at all.

Appendix
C. Glossary.

Active Color Table - Color table used to render the next graphic. If the next graphic is an image which has a Local Color Table associated with it, the active color table becomes the Local Color Table associated with that image. If the next graphic is an image without a Local Color Table, or a Plain Text Extension, the active color table is the Global Color Table associated with the Data Stream, if there is one; if there is no Global Color Table in the Data Stream, the active color table is a color table saved from a previous Data Stream, or one supplied by the decoder.

Block - Collection of bytes forming a protocol unit. In general, the term includes labeled and unlabeled blocks, as well as Extensions.

Data Stream - The GIF Data Stream is composed of blocks and sub-blocks representing images and graphics, together with control information to render them on a display device. All control and data blocks in the Data Stream must follow the Header and must precede the Trailer.

Decoder - A program capable of processing a GIF Data Stream to render the images and graphics contained in it.

Encoder - A program capable of capturing and formatting image and graphic raster data, following the definitions of the Graphics Interchange Format.

Extension - A protocol block labeled by the Extension Introducer 0x21.

Extension Introducer - Label (0x21) defining an Extension.

Graphic - Data which can be rendered on the screen by virtue of some algorithm. The term graphic is more general than the term image; in addition to images, the term graphic also includes data such as text, which is rendered using character bit-maps.

Image - Data representing a picture or a drawing; an image is represented by an array of pixels called the raster of the image.

Raster - Array of pixel values representing an image.

Appendix
D. Conventions.

Animation - The Graphics Interchange Format is not intended as a platform for animation, even though it can be done in a limited way.

Byte Ordering - Unless otherwise stated, multi-byte numeric fields are ordered with the Least Significant Byte first.

Color Indices - Color indices always refer to the active color table, either the Global Color Table or the Local Color Table.

Color Order - Unless otherwise stated, all triple-component RGB color values are specified in Red-Green-Blue order.

Color Tables - Both color tables, the Global and the Local, are optional; if present, the Global Color Table is to be used with every image in the Data Stream for which a Local Color Table is not given; if present, a Local Color Table overrides the Global Color Table. However, if neither color table is present, the application program is free to use an arbitrary color table. If the graphics in several Data Streams are related and all use the same color table, an encoder could place the color table as the Global Color Table in the first Data Stream and leave subsequent Data Streams without a Global Color Table or any Local Color Tables; in this way, the overhead for the table is eliminated. It is recommended that the decoder save the previous Global Color Table to be used with the Data Stream that follows, in case it does not contain either a Global Color Table or any Local Color Tables. In general, this allows the application program to use past color tables, significantly reducing transmission overhead.

Extension Blocks - Extensions are defined using the Extension Introducer code to mark the beginning of the block, followed by a block label, identifying the type of extension. Extension Codes are numbers in the range from 0x00 to 0xFF, inclusive. Special purpose extensions are transparent to the decoder and may be omitted when transmitting the Data Stream on-line. The GIF capabilities

dialogue makes the provision for the receiver to request the transmission of all blocks; the default state in this regard is no transmission of Special purpose blocks.

Reserved Fields - All Reserved Fields are expected to have each bit set to zero (off).

Appendix
E. Interlaced Images.

The rows of an Interlaced images are arranged in the following order:

Group 1 : Every 8th. row, starting with row 0.	(Pass 1)
Group 2 : Every 8th. row, starting with row 4.	(Pass 2)
Group 3 : Every 4th. row, starting with row 2.	(Pass 3)
Group 4 : Every 2nd. row, starting with row 1.	(Pass 4)

The Following example illustrates how the rows of an interlaced image are ordered.

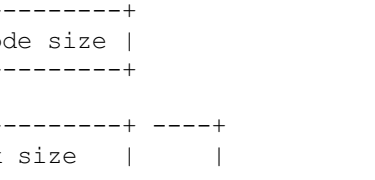
Row Number	Interlace Pass
0	1
1	4
2	3
3	4
4	2
5	4
6	3
7	4
8	1
9	4
10	3
11	4
12	2
13	4
14	3
15	4
16	1
17	4
18	3
19	4

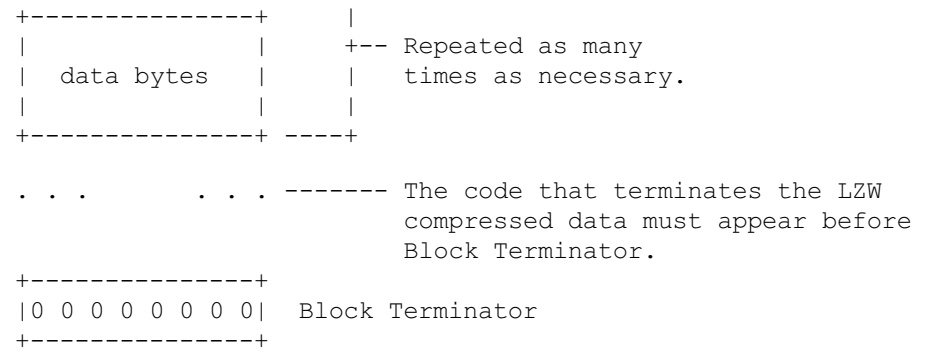
Appendix
F. Variable-Length-Code LZW Compression.

The Variable-Length-Code LZW Compression is a variation of the Lempel-Ziv Compression algorithm in which variable-length codes are used to replace patterns detected in the original data. The algorithm uses a code or translation table constructed from the patterns encountered in the original data; each new pattern is entered into the table and its index is used to replace it in the compressed stream.

The compressor takes the data from the input stream and builds a code or translation table with the patterns as it encounters them; each new pattern is entered into the code table and its index is added to the output stream; when a pattern is encountered which had been detected since the last code table refresh, its index from the code table is put on the output stream, thus achieving the data compression. The expander takes input from the compressed data stream and builds the code or translation table from it; as the compressed data stream is processed, codes are used to index into the code table and the corresponding data is put on the decompressed output stream, thus achieving data decompression. The details of the algorithm are explained below. The Variable-Length-Code aspect of the algorithm is based on an initial code size (LZW-initial code size), which specifies the initial number of bits used for the compression codes. When the number of patterns detected by the compressor in the input stream exceeds the number of patterns encodable with the current number of bits, the number of bits per LZW code is increased by one.

The Raster Data stream that represents the actual output image can be represented as:





The conversion of the image from a series of pixel values to a transmitted or stored character stream involves several steps. In brief these steps are:

1. Establish the Code Size - Define the number of bits needed to represent the actual data.

2. Compress the Data - Compress the series of image pixels to a series of

compression codes.

3. Build a Series of Bytes - Take the set of compression codes and convert to a string of 8-bit bytes.

4. Package the Bytes - Package sets of bytes into blocks preceded by character counts and output.

ESTABLISH CODE SIZE

The first byte of the Compressed Data stream is a value indicating the minimum number of bits required to represent the set of actual pixel values. Normally this will be the same as the number of color bits. Because of some algorithmic constraints however, black & white images which have one color bit must be indicated as having a code size of 2.
This code size value also implies that the compression codes must start out one bit longer.

COMPRESSION

The LZW algorithm converts a series of data values into a series of codes which may be raw values or a code designating a series of values. Using text characters as an analogy, the output code consists of a character or a code representing a string of characters.

The LZW algorithm used in GIF matches algorithmically with the standard LZW algorithm with the following differences:

1. A special Clear code is defined which resets all compression/decompression parameters and tables to a start-up state. The value of this code is 2**<code size>. For example if the code size indicated was 4 (image was 4 bits/pixel) the Clear code value would be 16 (10000 binary). The Clear code can appear at any point in the image data stream and therefore requires the LZW algorithm to process succeeding codes as if a new data stream was starting. Encoders should output a Clear code as the first code of each image data stream.

2. An End of Information code is defined that explicitly indicates the end of the image data stream. LZW processing terminates when this code is encountered. It must be the last code output by the encoder for an image. The value of this code is <Clear code>+1.

3. The first available compression code value is <Clear code>+2.

4. The output codes are of variable length, starting at <code size>+1 bits per code, up to 12 bits per code. This defines a maximum code value of 4095 (0xFFF). Whenever the LZW code value would exceed the current code length, the code length is increased by one. The packing/unpacking of these codes must then be altered to reflect the new code length.

BUILD 8-BIT BYTES

Because the LZW compression used for GIF creates a series of variable length codes, of between 3 and 12 bits each, these codes must be reformed into a series of 8-bit bytes that will be the characters actually stored or transmitted. This provides additional compression of the image. The codes are formed into a stream of bits as if they were packed right to left and then

picked off 8 bits at a time to be output.

Assuming a character array of 8 bits per character and using 5 bit codes to be packed, an example layout would be similar to:

```
    +---------------+
  0 |               |    bbbaaaaa
    +---------------+
  1 |               |    dcccccbb
    +---------------+
  2 |               |    eeeedddd
    +---------------+
  3 |               |    ggfffffe
    +---------------+
  4 |               |    hhhhhggg
    +---------------+
          . . .
    +---------------+
  N |               |
    +---------------+
```

Note that the physical packing arrangement will change as the number of bits per compression code change but the concept remains the same.

PACKAGE THE BYTES

Once the bytes have been created, they are grouped into blocks for output by preceding each block of 0 to 255 bytes with a character count byte. A block with a zero byte count terminates the Raster Data stream for a given image. These blocks are what are actually output for the GIF image. This block format has the side effect of allowing a decoding program the ability to read past the actual image data if necessary by reading block counts and then skipping over

the data.

FURTHER READING

[1] Ziv, J. and Lempel, A. : "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, May 1977.
[2] Welch, T. : "A Technique for High-Performance Data Compression", Computer, June 1984.
[3] Nelson, M.R. : "LZW Data Compression", Dr. Dobb's Journal, October 1989.

Appendix
G. On-line Capabilities Dialogue.

NOTE : This section is currently (10 July 1990) under revision; the information provided here should be used as general guidelines. Code written based on this information should be designed in a flexible way to accommodate any changes resulting from the revisions.

The following sequences are defined for use in mediating control between a GIF sender and GIF receiver over an interactive communications line. These sequences do not apply to applications that involve downloading of static GIF files and are not considered part of a GIF file.

GIF CAPABILITIES ENQUIRY

The GIF Capabilities Enquiry sequence is issued from a host and requests an interactive GIF decoder to return a response message that defines the graphics parameters for the decoder. This involves returning information about available screen sizes, number of bits/color supported and the amount of color detail supported. The escape sequence for the GIF Capabilities Enquiry is defined as:

ESC[>0g 0x1B 0x5B 0x3E 0x30 0x67

GIF CAPABILITIES RESPONSE

The GIF Capabilities Response message is returned by an interactive GIF decoder and defines the decoder's display capabilities for all graphics modes that are supported by the software. Note that this can also include graphics printers as well as a monitor screen. The general format of this message is:

#version;protocol{;dev, width, height, color-bits, color-res}...<CR>

'#' GIF Capabilities Response identifier character.

version	GIF format version number; initially '87a'.
protocol='0'	No end-to-end protocol supported by decoder Transfer as direct 8-bit data stream.
protocol='1'	Can use CIS B+ error correction protocol to transfer GIF data interactively from the host directly to the display.
dev = '0'	Screen parameter set follows.
dev = '1'	Printer parameter set follows.
width	Maximum supported display width in pixels.
height	Maximum supported display height in pixels.
color-bits	Number of bits per pixel supported. The number of supported colors is therefore 2**color-bits.
color-res	Number of bits per color component supported in the hardware color palette. If color-res is '0' then no hardware palette table is available.

Note that all values in the GIF Capabilities Response are returned as ASCII decimal numbers and the message is terminated by a Carriage Return character.

The following GIF Capabilities Response message describes three standard IBM PC Enhanced Graphics Adapter configurations with no printer; the GIF data stream

can be processed within an error correcting protocol:

#87a;1;0,320,200,4,0;0,640,200,2,2;0,640,350,4,2<CR>

ENTER GIF GRAPHICS MODE

Two sequences are currently defined to invoke an interactive GIF decoder into action. The only difference between them is that different output media are selected. These sequences are:

ESC[>1g Display GIF image on screen

0x1B 0x5B 0x3E 0x31 0x67

ESC[>2g Display image directly to an attached graphics printer. The image may optionally be displayed on the screen as well.

0x1B 0x5B 0x3E 0x32 0x67

Note that the 'g' character terminating each sequence is in lowercase.

INTERACTIVE ENVIRONMENT

The assumed environment for the transmission of GIF image data from an interactive application is a full 8-bit data stream from host to micro. All 256 character codes must be transferrable. The establishing of an 8-bit data path for communications will normally be taken care of by the host application programs. It is however up to the receiving communications programs supporting GIF to be able to receive and pass on all 256 8-bit codes to the GIF decoder software.

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 doesnt 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		Black	
run length	Code word	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:

o Each row must be packed separately. Do not compress across row boundaries.

o 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.
o 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.
o 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.
o 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:

o 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.
o 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.
o 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.
o Our implementation is written in C, and compiles to about 2K bytes of object code each for the compressor and decompressor.
o 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);
Omega = the empty string;
for each character in the strip {
```

```
          K = GetNextCharacter();
          if Omega+K is in the string table {
               Omega = Omega+K;  /* string concatenation */
          } else {
               WriteCode (CodeFromString(Omega));
               AddTableEntry(Omega+K);
               Omega = K;
          }
     } /* end of for loop */
     WriteCode (CodeFromString(Omega));
     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.

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

Read Pixel 1 into K. Does OmegaK (<7><7>) exist in the string table? No, so we get to do some real work. We write the code associated with Omega to output (write <7> to output), and add OmegaK (<7><7>) to the table as entry 258. Store K (<7>) into Omega. 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 OmegaK (<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 Omega, so that Omega is now <7><7>.

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

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

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

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

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

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

LZW Decoding

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

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

          else {
               if (IsInTable(Code)) {
```

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

The function GetNextCode() retrieves the next code from the LZW-coded data. It must keep track of bit boundaries. It knows that the first code that it gets will be a 9-bit code. We add a table entry each time we get a code, so GetNextCode() must switch over to 10-bit codes as soon as string #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 "II" 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 IFDs 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 lEclairage) 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 dont use the NTSC chromaticities and white point because present day monitors dont 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 isnt a user burden; it is really a systems issue. It allows a systems solution but doesnt 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 doesnt use spectral absorption, metamerism isnt 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 didnt 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.

LZW and GIF explained

Steve Blackstock

I hope this little document will help enlighten those of you out there who want to know more about the Lempel-Ziv Welch compression algorithm, and, specifically, the implementation that GIF uses. Before we start, here's a little terminology, for the purposes of this document:

"character" a fundamental data element. In normal text files, this is just a single byte. In raster images, which is what we're interested in, it's an index that specifies the color of a given pixel. I'll refer to an arbitray character as "K".

"charstream" a stream of characters, as in a data file.

"string" a number of continuous characters, anywhere from one to very many characters in length. I can specify an arbitrary string as "[...]K".

"prefix" almost the same as a string, but with the implication that a prefix immediately precedes a character, and a prefix can have a length of zero. So, a prefix and a character make up a string. I will refer to an arbitrary prefix as "[...]".

"root" a single-character string. For most purposes, this is a character, but we may occasionally make a distinction. It is [...]K, where [...] is empty.

"code" a number, specified by a known number of bits, which maps to a string.

"codestream" the output stream of codes, as in the "raster data".

"entry" a code and its string.

"string table" a list of entries; usually, but not necessarily, unique.

LZW is a way of compressing data that takes advantage of repetition of strings in the data. Since raster data usually contains a lot of this repetition, LZW is a good way of compressing and decompressing it. For the moment, lets consider normal LZW encoding and decoding. GIF's variation on the concept is just an extension from there.

LZW manipulates three objects in both compression and decompression: the charstream, the codestream, and the string table. In compression, the charstream is the input and the codestream is the output. In decompression, the codestream is the input and the charstream is the output. The string table is a product of both compression and decompression, but is never passed from one to the other.

The first thing we do in LZW compression is initialize our string table. To do this, we need to choose a code size (how many bits) and know how many values our characters can possibly take. Let's say our code size is 12 bits, meaning we can store 0->FFF, or 4096 entries in our string table. Lets also say that we have 32 possible different characters. (This corresponds to, say, a picture in which there are 32

different colors possible for each pixel.) To initialize the table, we set code#0 to character#0, code #1 to character#1, and so on, until code#31 to character#31. Actually, we are specifying that each code from 0 to 31 maps to a root. There will be no more entries in the table that have this property.

Now we start compressing data. Let's first define something called the "current prefix". It's just a prefix that we'll store things in and compare things to now and then. I will refer to it as "[.c.]". Initially, the current prefix has nothing in it. Let's also define a "current string", which will be the current prefix plus the next character in the charstream. I will refer to the current string as "[.c.]K", where K is some character. OK, look at the first character in the charstream. Call it P. Make [.c.]P the current string. (At this point, of course, it's just the root P.) Now search through the string table to see if [.c.]P appears in it. Of course, it does now, because our string table is initialized to have all roots. So we don't do anything. Now make [.c.]P the current prefix. Look at the next character in the charstream. Call it Q. Add it to the current prefix to form [.c.]Q, the current string. Now search through the string table to see if [.c.]Q appears in it. In this case, of course, it doesn't. Aha! Now we get to do something. Add [.c.]Q (which is PQ in this case) to the string table for code#32, and output the code for [.c.] to the codestream. Now start over again with the current prefix being just the root P. Keep adding characters to [.c.] to form [.c.]K, until you can't find [.c.]K in the string table. Then output the code for [.c.] and add [.c.]K to the string table. In pseudo-code, the algorithm goes something like this:

```
[1] Initialize string table;
[2] [.c.] <- empty;
[3] K <- next character in charstream;
[4] Is [.c.]K in string table?
 (yes: [.c.] <- [.c.]K;
       go to [3];
 )
 (no: add [.c.]K to the string table;
      output the code for [.c.] to the codestream;
      [.c.] <- K;
      go to [3];
 )
```

It's as simple as that! Of course, when you get to step [3] and there aren't any more characters left, you just output the code for [.c.] and throw the table away. You're done.

Wanna do an example? Let's pretend we have a four-character alphabet: A,B,C,D. The charstream looks like ABACABA. Let's compress it. First, we initialize our string table to: #0=A, #1=B, #2=C, #3=D. The first character is A, which is in the string table, so [.c.] becomes A. Next we get AB, which is not in the table, so we output code #0 (for [.c.]), and add AB to the string table as code #4. [.c.] becomes B. Next we get [.c.]A = BA, which is not in the string table, so output code #1, and add BA to the string table as code #5. [.c.] becomes A. Next we get AC, which is not in the string table. Output code #0, and add AC to the string table as code #6. Now [.c.] becomes C. Next we get [.c.]A = CA, which is not in the table. Output #2 for C, and add CA to table as code#7. Now [.c.] becomes A. Next we get AB, which IS in the string table, so [.c.] gets AB, and we look at ABA, which is not in the string table, so output the code for AB, which is #4, and add ABA to the string table as code #8. [.c.] becomes A. We can't get any more characters, so we just output #0 for the code for A, and

we're done. So, the codestream is #0#1#0#2#4#0.

A few words (four) should be said here about efficiency: use a hashing strategy. The search through the string table can be computationally intensive, and some hashing is well worth the effort. Also, note that "straight LZW" compression runs the risk of overflowing the string table - getting to a code which can't be represented in the number of bits you've set aside for codes. There are several ways of dealing with this problem, and GIF implements a very clever one, but we'll get to that.

An important thing to notice is that, at any point during the compression, if [...]K is in the string table, [...] is there also. This fact suggests an efficient method for storing strings in the table. Rather than store the entire string of K's in the table, realize that any string can be expressed as a prefix plus a character: [...]K. If we're about to store [...]K in the table, we know that [...] is already there, so we can just store the code for [...] plus the final character K.

That takes care of compression. Decompression is perhaps more difficult conceptually, but it is really easier to program. We again have to start with an initialized string table. This table comes from what knowledge we have about the charstream that we will eventually get, like what possible values the characters can take. In GIF files, this information is in the header as the number of possible pixel values. The beauty of LZW, though, is that this is all we need to know. We will build the rest of the string table as we decompress the codestream. The compression is done in such a way that we will never encounter a code in the codestream that we can't translate into a string.

We need to define something called a "current code", which I will refer to as "<code>", and an "old-code", which I will refer to as "<old>". To start things off, look at the first code. This is now <code>. This code will be in the intialized string table as the code for a root. Output the root to the charstream. Make this code the old-code <old>. *Now look at the next code, and make it <code>. It is possible that this code will not be in the string table, but let's assume for now that it is. Output the string corresponding to <code> to the codestream. Now find the first character in the string you just translated. Call this K. Add this to the prefix [...] generated by <old> to form a new string [...]K. Add this string [...]K to the string table, and set the old-code <old> to the current code <code>. Repeat from where I typed the asterisk, and you're all set. Read this paragraph again if you just skimmed it!!!

Now let's consider the possibility that <code> is not in the string table. Think back to compression, and try to understand what happens when you have a string like P[...]P[...]PQ appear in the charstream. Suppose P[...] is already in the string table, but P[...]P is not. The compressor will parse out P[...], and find that P[...]P is not in the string table. It will output the code for P[...], and add P[...]P to the string table. Then it will get up to P[...]P for the next string, and find that P[...]P is in the table, as the code just added. So it will output the code for P[...]P if it finds that P[...]PQ is not in the table. The decompressor is always "one step behind" the compressor. When the decompressor sees the code for P[...]P, it will not have added that code to it's string table yet because it needed the beginning character of P[...]P to add to the string for the last code, P[...], to form the code for P[...]P. However, when a decompressor finds a code that it doesn't know yet, it will always be

the very next one to be added to the string table. So it can guess at what the string for the code should be, and, in fact, it will always be correct. If I am a decompressor, and I see code#124, and yet my string table has entries only up to code#123, I can figure out what code#124 must be, add it to my string table, and output the string. If code#123 generated the string, which I will refer to here as a prefix, [...], then code#124, in this special case, will be [...] plus the first character of [...]. So just add the first character of [...] to the end of itself. Not too bad. As an example (and a very common one) of this special case, let's assume we have a raster image in which the first three pixels have the same color value. That is, my charstream looks like: QQQ....

For the sake of argument, let's say we have 32 colors, and Q is the color#12. The compressor will generate the code sequence 12,32,.... (if you don't know why, take a minute to understand it.) Remember that #32 is not in the initial table, which goes from #0 to #31. The decompressor will see #12 and translate it just fine as color Q. Then it will see #32 and not yet know what that means. But if it thinks about it long enough, it can figure out that QQ should be entry#32 in the table and QQ should be the next string output. So the decompression pseudo-code goes something like:

```
[1] Initialize string table;
[2] get first code: <code>;
[3] output the string for <code> to the charstream;
[4] <old> = <code>;
[5] <code> <- next code in codestream;
[6] does <code> exist in the string table?
 (yes: output the string for <code> to the charstream;
       [...] <- translation for <old>;
       K <- first character of translation for <code>;
       add [...]K to the string table;        <old> <- <code>;  )
 (no: [...] <- translation for <old>;
      K <- first character of [...];
      output [...]K to charstream and add it to string table;
      <old> <- <code>
 )
[7] go to [5];
```

Again, when you get to step [5] and there are no more codes, you're finished. Outputting of strings, and finding of initial characters in strings are efficiency problems all to themselves, but I'm not going to suggest ways to do them here. Half the fun of programming is figuring these things out!

Now for the GIF variations on the theme. In part of the header of a GIF file, there is a field, in the Raster Data stream, called "code size". This is a very misleading name for the field, but we have to live with it. What it is really is the "root size". The actual size, in bits, of the compression codes actually changes during compression/decompression, and I will refer to that size here as the "compression size". The initial table is just the codes for all the roots, as usual, but two special codes are added on top of those. Suppose you have a "code size", which is usually the number of bits per pixel in the image, of N. If the number of bits/pixel is one, then N must be 2: the roots take up slots #0 and #1 in the initial table, and the two special codes will take up slots #4 and #5. In any other case, N is the number of bits per pixel, and the roots take up slots #0 through #(2**N-1), and the special codes are (2**N) and (2**N + 1). The initial compression size will be N+1 bits per code. If you're encoding, you output the codes (N+1) bits at a time to start with, and

if you're decoding, you grab (N+1) bits from the codestream at a time. As for the special codes: <CC> or the clear code, is (2**N), and <EOI>, or end-of-information, is (2**N + 1). <CC> tells the compressor to re- initialize the string table, and to reset the compression size to (N+1). <EOI> means there's no more in the codestream. If you're encoding or decoding, you should start adding things to the string table at <CC> + 2. If you're encoding, you should output <CC> as the very first code, and then whenever after that you reach code #4095 (hex FFF), because GIF does not allow compression sizes to be greater than 12 bits. If you're decoding, you should reinitialize your string table when you observe <CC>. The variable compression sizes are really no big deal. If you're encoding, you start with a compression size of (N+1) bits, and, whenever you output the code (2**(compression size)-1), you bump the compression size up one bit. So the next code you output will be one bit longer. Remember that the largest compression size is 12 bits, corresponding to a code of 4095. If you get that far, you must output <CC> as the next code, and start over. If you're decoding, you must increase your compression size AS SOON AS YOU write entry #(2**(compression size) - 1) to the string table. The next code you READ will be one bit longer. Don't make the mistake of waiting until you need to add the code (2**compression size) to the table. You'll have already missed a bit from the last code. The packaging of codes into a bitsream for the raster data is also a potential stumbling block for the novice encoder or decoder. The lowest order bit in the code should coincide with the lowest available bit in the first available byte in the codestream. For example, if you're starting with 5-bit compression codes, and your first three codes are, say, <abcde>, <fghij>, <klmno>, where e, j, and o are bit#0, then your codestream will start off like:

```
byte#0: hijabcde
byte#1: .klmnofg
```

so the differences between straight LZW and GIF LZW are: two additional special codes and variable compression sizes. If you understand LZW, and you understand those variations, you understand it all!

Just as sort of a P.S., you may have noticed that a compressor has a little bit of flexibility at compression time. I specified a "greedy" approach to the compression, grabbing as many characters as possible before outputting codes. This is, in fact, the standard LZW way of doing things, and it will yield the best compression ratio. But there's no rule saying you can't stop anywhere along the line and just output the code for the current prefix, whether it's already in the table or not, and add that string plus the next character to the string table. There are various reasons for wanting to do this, especially if the strings get extremely long and make hashing difficult. If you need to, do it.

JPEG File Interchange Format
Version 1.02

September 1, 1992

Eric Hamilton
C-Cube Microsystems
1778 McCarthy Blvd.
Milpitas, CA 95035

+1 408 944-6300
Fax: +1 408 944-6314
E-mail: eric@c3.pla.ca.us

JPEG File Interchange Format
Version 1.02

Why a File Interchange Format

JPEG File Interchange Format is a minimal file format which enables JPEG bitstreams to be exchanged between a wide variety of platforms and applications. This minimal format does not include any of the advanced features found in the TIFF JPEG specification or any application specific file format. Nor should it, for the only purpose of this simplified format is to allow the exchange of JPEG compressed images.

JPEG File Interchange Format features

- Uses JPEG compression
- Uses JPEG interchange format compressed image representation
- PC or Mac or Unix workstation compatible
- Standard color space: one or three components. For three components, YCbCr (CCIR 601-256 levels)
- APP0 marker used to specify Units, X pixel density, Y pixel density, thumbnail
- APP0 marker also used to specify JFIF extensions
- APP0 marker also used to specify application-specific information

JPEG Compression

Although any JPEG process is supported by the syntax of the JPEG File Interchange Format (JFIF) it is strongly recommended that the JPEG baseline process be used for the purposes of file interchange. This ensures maximum compatibility with all applications supporting JPEG. JFIF conforms to the JPEG Draft International Standard (ISO DIS 10918-1).

The JPEG File Interchange Format is entirely compatible with the standard JPEG

interchange format; the only additional requirement is the mandatory presence of the APP0 marker right after the SOI marker. Note that JPEG interchange format requires (as does JFIF) that all table specifications used in the encoding process be coded in the bitstream prior to their use.

Compatible across platforms

The JPEG File Interchange Format is compatible across platforms: for example, it does not use any resource forks, supported by the Macintosh but not by PCs or workstations.

Standard color space

The color space to be used is YCbCr as defined by CCIR 601 (256 levels). The RGB components calculated by linear conversion from YCbCr shall not be gamma corrected (gamma = 1.0). If only one component is used, that component shall be Y.

APP0 marker used to identify JPEG FIF

The APP0 marker is used to identify a JPEG FIF file. The JPEG FIF APP0 marker is mandatory right after the SOI marker.

The JFIF APP0 marker is identified by a zero terminated string: "JFIF". The APP0 can be used for any other purpose by the application provided it can be distinguished from the JFIF APP0.

The JFIF APP0 marker provides information which is missing from the JPEG stream: version number, X and Y pixel density (dots per inch or dots per cm), pixel aspect ratio (derived from X and Y pixel density), thumbnail.

APP0 marker used to specify JFIF extensions

Additional APP0 marker segment(s) can optionally be used to specify JFIF extensions. If used, these segment(s) must immediately follow the JFIF APP0 marker. Decoders should skip any unsupported JFIF extension segments and continue decoding.

The JFIF extension APP0 marker is identified by a zero terminated string: "JFXX". The JFIF extension APP0 marker segment contains a 1-byte code which identifies the extension. This version, version 1.02, has only one extension defined: an extension for defining thumbnails stored in formats other than 24-bit RGB.

APP0 marker used for application-specific information

Additional APP0 marker segments can be used to hold application-specific information which does not affect the decodability or displayability of the JFIF file. Application-specific APP0 marker segments must appear after the JFIF APP0 and any JFXX APP0 segments. Decoders should skip any unrecognized application-specific APP0 segments.

Application-specific APP0 marker segments are identified by a zero terminated string which identifies the application (not "JFIF" or "JFXX"). This string should be an organization name or company trademark. Generic strings such as dog, cat, tree, etc. should not be used.

Conversion to and from RGB

Y, Cb, and Cr are converted from R, G, and B as defined in CCIR Recommendation 601 but are normalized so as to occupy the full 256 levels of a 8-bit binary encoding. More precisely:

```
Y   = 256 * E'y
Cb  = 256 * [ E'Cb ] + 128
Cr  = 256 * [ E'Cr ] + 128
```

where the E'y, E'Cb and E'Cb are defined as in CCIR 601. Since values of E'y have a range of 0 to 1.0 and those for E'Cb and E'Cr have a range of -0.5 to +0.5, Y, Cb, and Cr must be clamped to 255 when they are maximum value.

RGB to YCbCr Conversion

YCbCr (256 levels) can be computed directly from 8-bit RGB as follows:

```
Y  =     0.299  R + 0.587  G + 0.114  B
Cb =   - 0.1687 R - 0.3313 G + 0.5    B + 128
Cr =     0.5    R - 0.4187 G - 0.0813 B + 128
```

NOTE - Not all image file formats store image samples in the order R0, G0, B0, ... Rn, Gn, Bn. Be sure to verify the sample order before converting an RGB file to JFIF.

YCbCr to RGB Conversion

RGB can be computed directly from YCbCr (256 levels) as follows:

```
R = Y                     + 1.402   (Cr-128)
G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)
B = Y + 1.772   (Cb-128)
```

Image Orientation

In JFIF files, the image orientation is always top-down. This means that the first image samples encoded in a JFIF file are located in the upper left hand corner of the image and encoding proceeds from left to right and top to bottom. Top-down orientation is used for both the full resolution image and the thumbnail image.

The process of converting an image file having bottom-up orientation to JFIF must include inverting the order of all image lines before JPEG encoding

Spatial Relationship of Components

Specification of the spatial positioning of pixel samples within components relative to the samples of other components is necessary for proper image post processing and accurate image presentation. In JFIF files, the position of the pixels in subsampled components are defined with respect to the highest resolution component. Since components must be sampled orthogonally (along rows and columns), the spatial position of the samples in a given subsampled component may be determined by specifying the horizontal and vertical offsets of the first sample, i.e. the sample in the upper left corner, with respect to the highest resolution component.

The horizontal and vertical offsets of the first sample in a subsampled component, Xoffseti[0,0] and Yoffseti[0,0], is defined to be

```
Xoffseti[0,0] = ( Nsamplesref / Nsamplesi  ) / 2 - 0.5
Yoffseti[0,0] = ( Nlinesref / Nlinesi  ) / 2 - 0.5
```

where
Nsamplesref is the number of samples per line in the largest component,
Nsamplesi is the number of samples per line in the ith component,
Nlinesref is the number of lines in the largest component,
Nlinesi is the number of lines in the ith component.

Proper subsampling of components incorporates an anti-aliasing filter which reduces the spectral bandwidth of the full resolution components. Subsampling can easily be accomplished using a symmetrical digital filter with an even number of taps (coefficients). A commonly used filter for 2:1 subsampling utilizes two taps (1/2,1/2).

NOTE - This definition is compatible with industry standards such as Postcript Level 2 and QuickTime. This defintition is not compatible with the conventions used by CCIR Recommendation 601-1 and other digital video formats. For these formats, pre-processing of the chrominance components is necessary prior to compression in order to ensure accurate reconstruction of the compressed image.

JPEG File Interchange Format Specification

The syntax of a JFIF file conforms to the syntax for interchange format defined in Annex B of ISO DIS 10918-1. In addition, a JFIF file uses APP0 marker segments and constrains certain parameters in the frame header as defined below.

```
    X'FF', SOI
         X'FF', APP0, length, identifier, version, units, Xdensity, Ydensity, Xthumbnail,
          Ythumbnail, (RGB)n
              length     (2 bytes)  Total APP0 field byte count, including the byte
                                    count value (2 bytes), but excluding the APP0
                                    marker itself
              identifier (5 bytes)  = X'4A', X'46', X'49', X'46', X'00'
                                    This zero terminated string ("JFIF") uniquely
                                    identifies this APP0 marker.  This string shall
                                    have zero parity (bit 7=0).
              version    (2 bytes)  = X'0102'
                                    The most significant byte is used for major
                                    revisions, the least significant byte for minor
                                    revisions. Version 1.02 is the current released
                                    revision.
              units      (1 byte)   Units for the X and Y densities.
                                    units = 0:  no units, X and Y specify the pixel
                                         aspect ratio
                                    units = 1:  X and Y are dots per inch
                                    units = 2:  X and Y are dots per cm
              Xdensity   (2 bytes)  Horizontal pixel density
              Ydensity   (2 bytes)  Vertical pixel density
              Xthumbnail (1 byte)   Thumbnail horizontal pixel count
              Ythumbnail (1 byte)   Thumbnail vertical pixel count
              (RGB)n     (3n bytes) Packed (24-bit) RGB values for the thumbnail
                                    pixels, n = Xthumbnail * Ythumbnail
         [ Optional JFIF extension APP0 marker segment(s) - see below ]
                o
                o
                o
         X'FF', SOFn, length, frame parameters
            Number of components Nf  = 1 or 3
            1st component    C1    = 1 = Y component
            2nd component    C2    = 2 = Cb component
            3rd component    C3    = 3 = Cr component
                 o
                 o
                 o
    X'FF', EOI
```

JFIF Extension APP0 Marker Segment

Immediately following the JFIF APP0 marker segment may be a JFIF extension APP0 marker. This JFIF extension APP0 marker segment may only be present for JFIF versions 1.02 and above. The syntax of the JFIF extension APP0 marker segment is:

```
          X'FF', APP0, length, identifier, extension_code, extension_data
             length   (2 bytes)    Total APP0 field byte count, including the byte
                                  count value (2 bytes), but excluding the APP0
```

```
                        marker itself
        identifier  (5 bytes)    = X'4A', X'46', X'58', X'58', X'00'
                        This zero terminated string ("JFXX") uniquely
                        identifies this APP0 marker.  This string shall
                        have zero parity (bit 7=0).
        extension_code (1 byte)    = Code which identifies the extension.  In this
                        version, the following extensions are defined:
                           = X'10'   Thumbnail coded using JPEG
                           = X'11'   Thumbnail stored using 1 byte/pixel
                           = X'13'   Thumbnail stored using 3 bytes/pixel
        extension_data (variable)    = The specification of the remainder of the JFIF
                        extension APP0 marker segment varies with the
                        extension. See below for a specification of
                        extension_data for each extension.
```

JFIF Extension: Thumbnail coded using JPEG

This extension supports thumbnails compressed using JPEG. The compressed thumbnail immediately follows the extension_code (X'10') in the extension_data field and the length of the compressed data must be included in the JFIF extension APP0 marker length field.

The syntax of the extension_data field conforms to the syntax for interchange format defined in Annex B of ISO DIS 10918-1. However, no "JFIF" or "JFXX" marker segments shall be present. As in the full resolution image of the JFIF file, the syntax of extension_data constrains parameters in the frame header as defined below:

```
    X'FF', SOI
                o
                o
                o
        X'FF', SOFn, length, frame parameters
             Number of components    Nf    = 1 or 3
             1st component    C1    = 1 = Y component
             2nd component    C2    = 2 = Cb component
             3rd component    C3    = 3 = Cr component
                o
                o
                o
    X'FF', EOI
```

JFIF Extension: Thumbnail stored using one byte per pixel

This extension supports thumbnails stored using one byte per pixel and a color palette in the extension_data field. The syntax of extension_data is:

```
        Xthumbnail       (1 byte)    Thumbnail horizontal pixel count
        Ythumbnail       (1 byte)    Thumbnail vertical pixel count
        palette          (768 bytes)  24-bit RGB pixel values for the color palette.
                                      The RGB values define the colors represented by
                                      each value of an 8-bit binary encoding (0 - 255).
        (pixel)n         (n bytes)    8-bit values for the thumbnail pixels
                                      n = Xthumbnail * Ythumbnail
```

JFIF Extension: Thumbnail stored using three bytes per pixel

This extension supports thumbnails stored using three bytes per pixel in the extension_data field. The syntax of extension_data is:

```
        Xthumbnail       (1 byte)    Thumbnail horizontal pixel count
        Ythumbnail       (1 byte)    Thumbnail vertical pixel count
        (RGB)n           (3n bytes)  Packed (24-bit) RGB values for the thumbnail
```

pixels, n = Xthumbnail * Ythumbnail

Useful tips

o you can identify a JFIF file by looking for the following sequence: X'FF', SOI, X'FF', APP0, <2 bytes to be skipped>, "JFIF", X'00'.

o if you use APP0 elsewhere, be sure not to have the strings "JFIF" or "JFXX" right after the APP0 marker.

o if you do not want to include a thumbnail, just program Xthumbnail = Ythumbnail = 0.

o be sure to check the version number in the special APP0 field. In general, if the major version number of the JFIF file matches that supported by the decoder, the file will be decodable.

o if you only want to specify a pixel aspect ratio, put 0 for the units field in the special APP0 field. Xdensity and Ydensity can then be programmed for the desired aspect ratio. Xdensity = 1, Ydensity = 1 will program a 1:1 aspect ratio. Xdensity and Ydensity should always be non-zero.

Disc Manufacturing, Inc.
A QUIXOTE COMPANY

Introduction to ISO 9660,

what it is, how it is implemented, and how it has been extended.

Clayton Summers

WHO IS DMI?

Disc Manufacturing, Inc. (DMI) manufactures all compact disc formats (i.e., CD-Audio, CD-ROM, CD-ROM XA, CDI, PHOTO CD, 3DO, KARAOKE, etc.) at two plant sites in the U.S.; Huntsville, AL, and Anaheim, CA. To help you, DMI has one of the largest Product Engineering/Technical Support staff and sales force dedicated solely to CD-ROM in the industry.

The company has had a long term commitment to optical disc technology and has performed developmental work and manufactured (laser) optical discs of various types since 1981. In 1983, DMI manufactured the first compact disc in the United States. DMI has developed extensive mastering expertise during this time and is frequently called upon by other companies to provide special mastering services for products in development.

In August 1991, DMI purchased the U.S. CD-ROM business from the Philips and Du Pont Optical Company (PDO). PDO employees in sales, marketing and technical services were retained.

DMI is a wholly-owned subsidiary of Quixote Corporation, a publicly owned corporation whose stock is traded on the NASDAQ exchange as QUIX. Quixote is a diversified technology company composed of Energy Absorption Systems, Inc. (manufactures highway crash cushions), Stenograph Corporation (manufactures shorthand machines and computer systems for court reporting) and Disc Manufacturing, Inc.

We would be pleased to help you with your CD project or answer any questions you may have. Please give us a call at 1-800-433-DISC for pricing or further information.

We have four additional technical papers available entitled

Integrating Mixed-Mode CD-ROM

An Overview to MultiMedia CD-ROM Production

Compact Disc Terminology - 2nd Edition

A Glossary of CD and CD-ROM Terms

These are available upon request
800-433-DISC
302-479-2500
Fax: 302-479-2527

This paper was written in response to the many questions we, as a CD-ROM manufacturer, have received concerning ISO 9660. Our intent was to provide clarity and simplification to a very technical subject. We hope you find it helpful.

For reprinting privileges, call or write to:
Nancy Klocko
Disc Manufacturing, Inc.
1409 Foulk Rd., Suite 102
Wilimington, DE 19803
800-433-DISC, 302-479-2527

Acknowledgments

Without the help of many people, this paper would not have been possible.

Special Thanks go to:

Wendy Upham and Breck Rowell for working on how MS-DOS and the Macintosh implement ISO 9660 and what some of the quirks are

John Sands at Young Minds

Apple Computer

Doug Carson and Associates

and Nancy Klocko and Pam Sansbury for invaluable help editing and revising and Lori Magno for formatting.

Table of Contents

Tables

Figures

Introduction to ISO 9660

The Digital Audio Compact Disc has been called the most successful consumer product ever launched. Since it's introduction in June of 1980, the CD has come to dominate the music industry and become the format of choice for millions of music listeners due to the ultra high fidelity afforded by the digital recording technique and the near indestructibility afforded by the optical design. These same features make the CD very attractive as a carrier of other types of digital information. Another feature that makes the Compact Disc attractive as a medium for digital information is that CDs can be manufactured in large quantities quickly and inexpensively. Also, due to the industry standards defined by the Red Book, Yellow Book, and ISO 9660, any CD can be used on almost any hardware/software platform. It, therefore, comes as no surprise that this 15 grams of poly-carbonate and aluminum which contains billions of bits of data would be embraced by the computer industry to store and distribute huge amounts of data. However, creating a disc that works on multiple platforms is not as simple as copying files to a floppy.

Background

Before ISO 9660, all CD-ROM discs could be read by all CD-ROM drives; however, CD-ROM drives were not supported by any readily available computer operating system. Application developers were required to have software device drivers for each computer and CD-ROM drive combination that they wanted to support. In addition, most applications require a file structure and this had to be provided by each developer as well.

Typically, application developers used a systems house to provide device drivers and file system software as well as build and retrieval engines. The result was that application developers requested both enhancements to the build and retrieval engines and support for additional drive types. The systems houses had to spend critical resources to develop these drivers when they would have preferred to spend those resources in other areas. A committee called High Sierra was formed to develop an industry standard to address the file system software.

The High Sierra proposal was designed to enable data interchange between computers using standardized software. When a computer is equipped for either High Sierra or ISO 9660, data on any properly encoded CD-ROM may be read using standard operating system instructions such as list directory, open, read and close. This reduces the amount of effort required to bring an application to market. In addition, discs may be read by any drive that has standard driver software.

High Sierra was defined and submitted to the **I**nternational **S**tandards **O**rganization in May of 1986. During the time it was being debated and approved, the companies involved in creating the High Sierra proposal went ahead and implemented High Sierra. ISO 9660 was published in April of 1988. ISO made several minor changes during the process that made the ISO 9660 standard incompatible with the High Sierra proposal. The changes involved rearranging the order of the information in the directory record and changing the code that identifies the disc as a High Sierra or ISO 9660 disc, among other, more esoteric, items.

To accurately and repeatedly create ISO-9660 discs requires an understanding of what ISO-9660 is and how it is implemented by different platforms. First a little background information will be presented that helps put the concept of a common format for data interchange into perspective. This will be accomplished by discussing file systems and how they relate to the computer's operating system. Then ISO-9660 will be covered in general terms and some of the more commonly used features and data structures will be covered in some detail. A description, at a conceptual level, of how some operating systems implement support for ISO-9660 and notes regarding some of the peculiarities this causes will then be presented. Then some of the ways ISO-9660 has been extended to provide better support for two particular operating systems, the Macintosh and UNIX environments, will be introduced. Finally, the Frankfurt proposal, an extension to ISO 9660 that allows updating information on a recordable CD, will be discussed.

File Systems

Most operating systems store information in both fast, short term memory usually referred to as Random Access Memory or RAM, as well as in relatively slow, long term memory. Typically, the slow, long term memory takes the form of a floppy disk, or hard disk and is stored as files. If we compare this to someone's office, the fast, short term memory can be compared to the desktop, where things are actually being worked on. The slow, long term disk can be compared to the file cabinet, where unused items are put until needed. The way the operating system keeps track of where files are located is called the file system. Examples of file systems are MS-DOS's FAT (File Allocation Table), the Macintosh HFS (Hierarchical File System), OS/2's HPFS (High Performance File System), and the UNIX File System. All of these file systems are specific to, and optimized for, a particular operating system. ISO-9660 is also a file system. However, it was designed to be independent of any operating system, and because it was designed for CD-ROM, is also "read only". This means that unlike the other file systems mentioned, it does not provide any way to add to or change the information in it. Since ISO 9660 was intended to be used on a diverse group of operating systems, it includes only the minimum information that can be utilized by the widest variety of systems.

Overview of ISO-9660 structure.

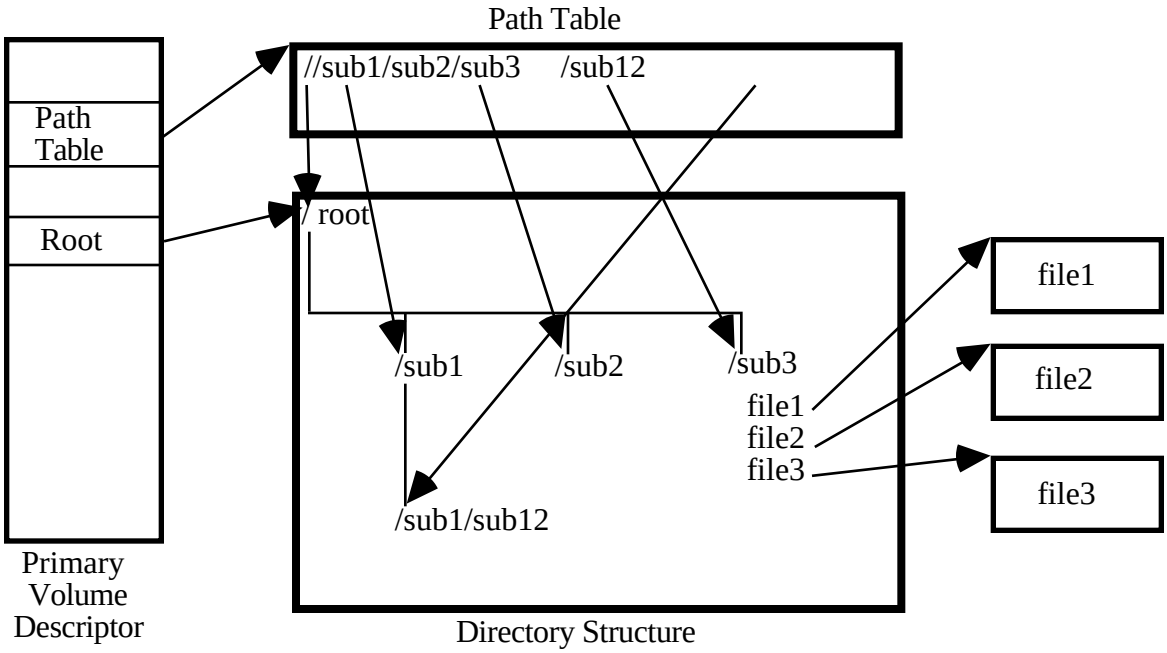


Figure 1. ISO 9660 structures

ISO 9660 data structures fall into three main categories: the Volume Descriptors, the Directory Structures, and the Path Tables. These structures are interrelated as shown in figure 1. The Volume Descriptor tells where the directory structure and the Path Table are located, the directories tell us where the actual files are located, and the Path table gives us short cuts to each directory .

The Volume Descriptors

There are currently four types of Volume Descriptors defined in ISO 9660. Only one of these, the Primary Volume Descriptor, is commonly used. The other types are the Boot Record, the Supplementary Volume Descriptor, and the Volume Partition Descriptor. The Boot Record can be used for systems that must perform some type of initialization before the user can access the volume, although ISO 9660 does not specify what information must be in the Boot Record or how it is to be used. The Supplementary Volume Descriptor can be used to identify an alternate character set for use by systems that do not support the ISO 646 character set. The Volume Partition Descriptor can be used to logically divide the volume into smaller volume partitions, although ISO 9660 does not

specify how to do this, only that it can be done.[1] The Volume Descriptors are recorded starting at Logical Sector 16 (which corresponds to two seconds and sixteen sectors into the CD, or in CD "Atime", 00:02:16).

The Primary Volume Descriptor

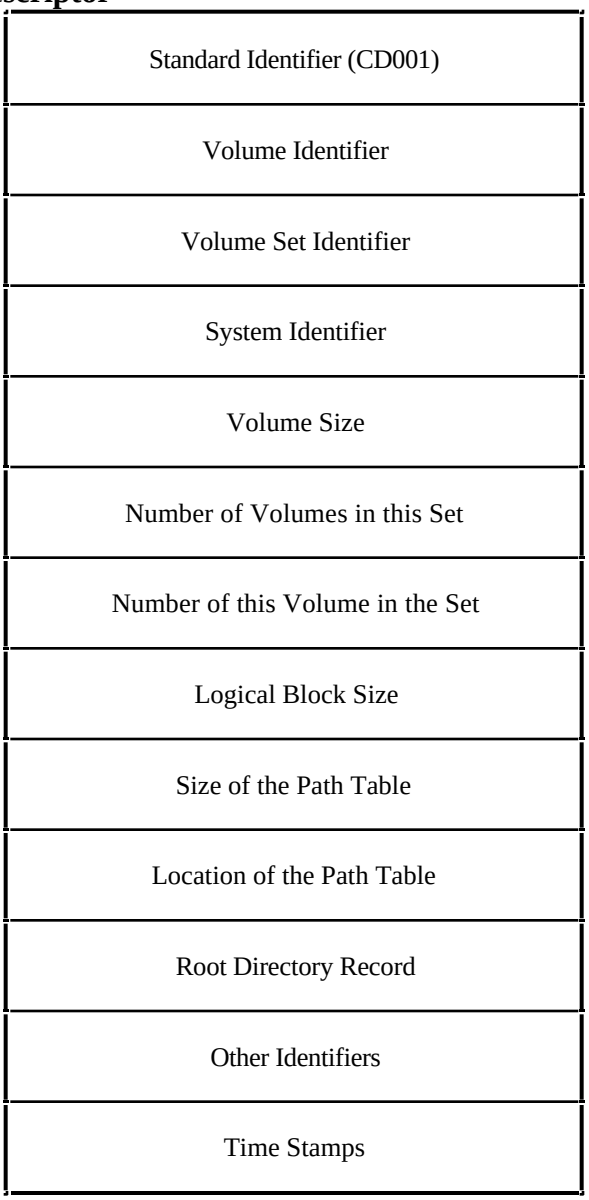


Figure 2. The Primary Volume Descriptor

The Primary Volume Descriptor as seen in figure 2 is the starting point in identifying a CD-ROM. It contains the Standard Identifier, the Volume Identifier, the Volume Set Identifier, the System

[1] ISO 9660-1988, pp. 11, section 8.1.1

Identifier ,the size of the Volume, the number of Volumes in the Volume Set it belongs to, the sequence within the Volume Set that this Volume belongs, the Logical Block size of the blocks in this volume, the size of the Path Table, the location of the Path Table, the Directory record for the Root Directory, other identifiers and important times relating to the Volume.[2]

The Standard Identifier is a set of characters, defined by ISO 9660 to be `CD001`, that tells the Operating System that this is an ISO 9660 disc. This is to distinguish the volume from other file systems that use a similar layout, such as High Sierra, whose Standard Identifier is CDROM, and Compact Disc Interactive, whose Standard Identifier is CD-I.

The Volume Identifier is simply the name that is given to the ISO 9660 volume.
The characters that can be used in the Volume Identifier are restricted to what ISO 9660 calls d-characters and the length is restricted to 31 characters. The d-characters are shown in figure 3.

```
A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _
```

Figure 3. The d-characters

Some systems, such as the Macintosh, use the Volume Identifier extensively. Others, such as MS-DOS, use it somewhat, and some, such as UNIX, barely use it at all.

The Volume Set Identifier is the name given to the Multiple Volume Set that this Volume belongs to. Like the Volume Identifier, it is restricted to the d-characters and cannot be more than 31 characters long. For example, if this Volume where named DICTIONARY_E_H, it might have a Volume Set Identifier of DICTIONARY, meaning that this Volume contains the words starting with the letter E through the letter H, and the Volume Set is the set of discs for the entire alphabet.

---

[2] see Appendix A: Table 14 and ISO 9660-1988, pp. 12-16, section 8.4

The System Identifier identifies a system that can recognize and act on logical sectors 0 through 15.[3] While ISO 9660 specifies that this is what the System Identifier is used for, it does not specify what is in sectors 0 through 15, nor does it specify how the data is used. The characters that can be used in the System ID are what ISO 9660 calls a-characters and the length is restricted to 31 characters. The a-characters are shown in figure 4.

```
A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ sp
! " % & ' ( ) * + , - . / : ; < = > ?
```

Figure 4. The a-characters

The Volume Size is a number that tells the operating system how many Logical Blocks are in this Volume. A Logical Block is the basic way of locating things in the Volume. All locations are given as Logical Block Numbers. If the Volume is pictured as an Interstate highway, then the Logical Block Numbers are the mile markers.

Volume Set Size is a number that tells the operating system how many volumes are in the Volume Set to which this Volume belongs. The Volume Sequence Number is the place within a multiple volume set that this volume belongs. For example on a disc with Volume Set Size of 5 and Volume Sequence Number of 3, this disc is the third disc of a five disc set.

The Logical Block Size is the number of bytes that make up the smallest amount of space that is allocated in this volume. This number can be 512, 1024, or 2048 bytes. Most ISO 9660 discs use a Logical Block Size of 2048, the same as the Sector Size.

The Path Table Size tells the operating system how many bytes are in the Path Table. Most operating systems that use the Path Table keep it in fast, local memory (RAM), and this number is a quick way

[3]ISO 9660-1988, pp.13, section 8.4.5

for the operating system to know how much memory it needs to allocate before it reads the Path Table. This way the Operating system only reads the Path Table once, saving time. The location of the Path Table must be in the Primary Volume Descriptor since the Path Table itself may be anywhere in the Volume.

The Root Directory record contains the information the operating system needs to locate and read the top level directory. It is formatted exactly the same as any other directory record.[4]

Other identifiers in the Primary Volume Descriptor contain information about who published this Volume, who prepared the data, what the application is, and what the names of the files are that contain the copyright notice, the abstract, and the bibliography.

The time stamps are fields in the Primary Volume Descriptor that contain information about when the Volume was created, when it may have been modified, when the data becomes effective, and when the data becomes obsolete.

[4]See Appendix A, Table 15, page IV

The Directory Structure

The ISO 9660 directory structure is organized in a hierarchical manner similar to most modern file systems.[5] At the top of the hierarchy is the Root Directory, the location of which is identified in the Primary Volume Descriptor. When drawn hierarchically, the directory structure resemble the roots of a tree, with the Root directory at the top of the structure, as shown in figure 5.

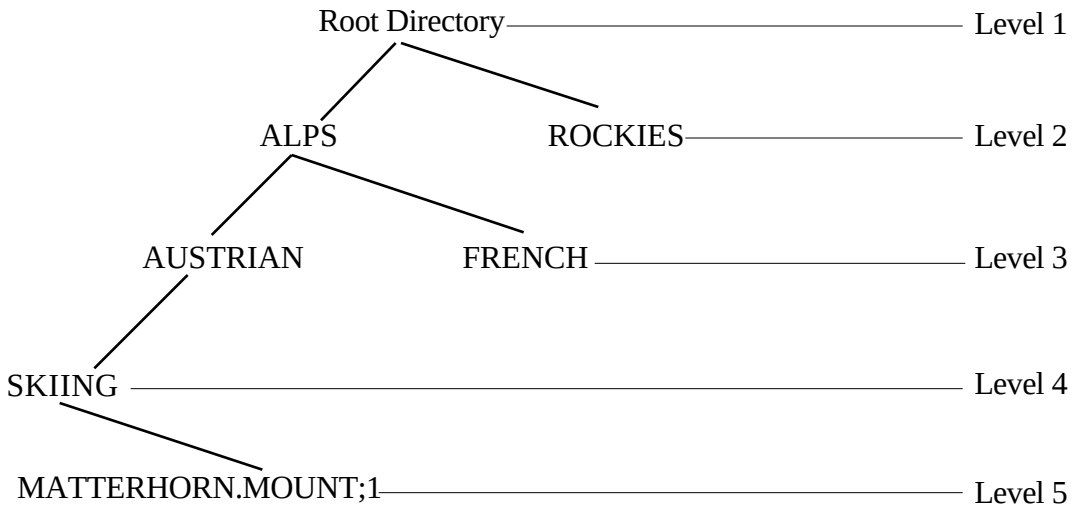


Figure 5. The Directory Hierarchy

[5]ISO 9660:1988, pp.7, section 6.8.2

As shown in figure 5, there are distinct levels in this hierarchy. The Root Directory is the only directory at level 1. In the example illustrated by figure 5, Subdirectories ALPS and ROCKIES are at level 2, Subdirectories AUSTRIAN and FRENCH are at level 3, Subdirectory SKIING is at level 4, and the file MATTERHORN.MOUNT;1 is at level 5. To insure compatibility, ISO 9660 imposes a limit of eight levels to the depth of the directory structure.[6] It also imposes a limit on the length of the path to each file. The length of the path is the sum of the lengths of all relevant directories ,the length of the File Identifier, and the number of relevant directories. The length of the path cannot exceed 255. Again, in figure 5, the sum of the length of the File Identifier ,the lengths of the relevant directories , and the number of relevant directories is 39 as shown in table 1.

Table 1. Length of the Path

Identifier	Length
ALPS	4
AUSTRIAN	8
SKIING	6
MATTERHORN.MOUNT;1	18
number of directories	3
sum of lengths and number of directories	39

A directory in an ISO-9660 volume is recorded as a file containing a set of directory records. Each directory record describes a file or another directory. Every directory has a parent directory. The parent directory contains the directory record that identifies that directory, as shown in figure 6.

[6]ISO 9660:1988, pp.7, section 6.8.2.1

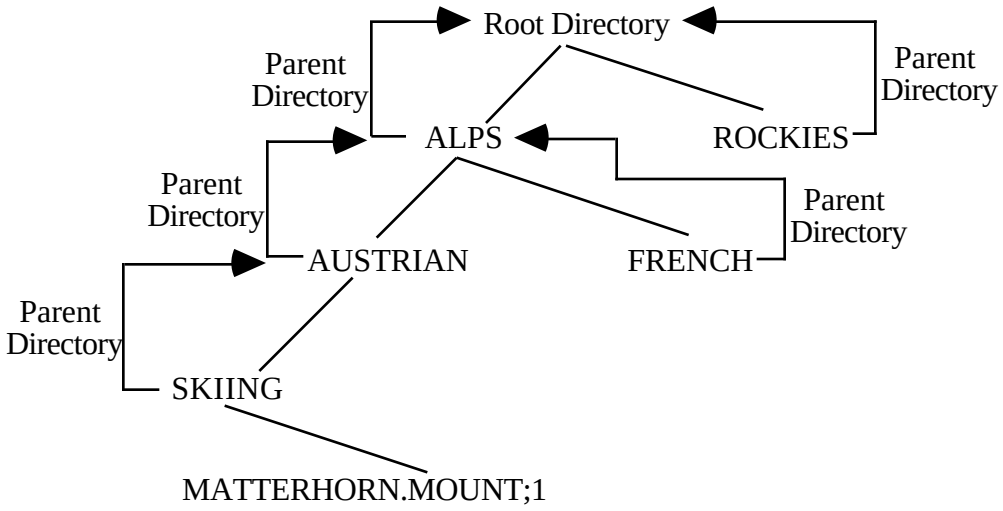


Figure 6. Parent Directories

The Root directory's parent is the Root directory itself.

Each directory also contains a record for its parent directory. A given directory may contain entries for several files as well as for several directories, all of which have the same parent.

File Names

Every file and directory in an ISO 9660 Volume has a name an identifying name associated with it. This name is called the File Identifier. An ISO 9660 File Identifier actually consists of five parts as shown in table 2.[7]

Table 2. The File Identifier

	1) File Name	2) SEPARATOR 1	3) File Name Extension	4) SEPARATOR 2	5) File Version Number
contents	d-characters (see figure 3)	.	d-characters (see figure 3)	;	a number from 1 to 32767
file 1	MATTERHORN	.	MOUNT	;	1
file 2	PIKES_PEAK	.		;	1
file 3		.	HILLS	;	1
directory	SKIING				

The File Identifier must also meet the following conditions:[8]

- If the File Name has no characters, then the File Name Extension must have at least one character, as shown in table 2, file 3.

- If the File Name Extension has no characters, then the File Name must have at least one character, as shown in table 2, file 2.

- The sum of the lengths of the File Name and the File Name Extension cannot exceed 30.

[7]ISO 9660:1988, pp. 10, section 7.5.1
[8]ISO 9660:1988, pp. 10, section 7.5.1

An ISO 9660 directory name is limited to only a Name; it cannot have a SEPARATOR 1 (.), an Extension, a SEPARATOR 2 (;) or a version number, as shown in table 2, directory.[9]

Order of Directory Records

ISO 9660 also specifies the order of the records in a directory.[10] They must be sorted by the relative value of the File Identifier field. Table 3 shows a set of unsorted File Identifiers.

Table 3. File Identifiers

1) File Name	2) SEPARATOR 1	3) File Name Extension	4) SEPARATOR 2	5) File Version Number
MATTERHORN	.	MOUNT	;	1
PIKES_PEAK	.		;	1
	.	HILLS	;	1
SKIING				

The relative value of two File Identifiers is determined in the following way:

- If two File Names have the same content in all byte positions, then these two file names are said to be equal in value.

- If two File Names do not contain the same number of byte positions, the shorter File Name shall be treated as if it were padded on the right with all padding bytes set to (20) (the SPACE character) and as if both File Names contained the same number of byte positions.

[9] ISO 9660:1988, pp. 11, section 7.6.1
[10] ISO 9660:1988, pp.21-22, section 9.3

- After any necessary padding to make the two File Names the same length, the corresponding characters are compared, starting with the first byte position, until a byte position is found that does not have the same character in both File Names. The greater File Name is the one that contains the character whose ASCII value is greater. Table 4 shows the File Names from Table 3 and their relative values.

Table 4. Relative Value of File Names

1) File Name	ASCII Code of first character	Relative value
PIKES_PEAK	80	3
MATTERHORN	77	2
	32	1
SKIING	83	4

The File Name Extensions and Directory Names are also sorted in this manner. The File Version Number is sorted by padding the values with (30) (ASCII "0") on the left to make the lengths equal, then comparing them as above. Table 5 shows the Extensions and Version Numbers from table 3 and their relative values.

Table 5. Relative Value of Extensions and Version Numbers

3) File Name Extension	ASCII Code of first character	Relative Value	5) File Version Number	ASCII Code of first character	Relative Value
MOUNT	77	3	1	31	2
	32	1	1	31	2
HILLS	72	2	1	31	2
	32	1		30	1

The Directory records are then sorted as follows:

- First in ascending order relative to the File Name (or Directory Name)
- Second in ascending order relative to the File Name Extension.
- Third, in descending order relative to the File Version Number.
- Forth, in descending order relative to the value of the Associated File Flag in the File Flags Field. (The associated file comes first, then the file it is associated with).
- Last, in the order of the File Sections of the file. (Only valid if the file is recorded in interleaved mode).

following the above rules, the example started in table 3 is sorted as shown in table 6.

Table 6. Sorted File Identifiers

1) File Name	2) SEPERATOR 1	3) File Name Extension	4) SEPERATOR 2	5) File Version Number
	.	HILLS	;	1
MATTERHORN	.	MOUNT	;	1
PIKES_PEAK	.		;	1
SKIING				

The Path Table

The Path Table indicates to the operating system a short cut to each directory on the disc rather than making the operating system read through each directory to get to the file it needs. This is done primarily to enhance performance. For each directory other than the Root directory, the path table contains a record that identifies the directory, its parent directory, and its location[11].

[11] See Appendix A, Table 16, page VII

Most operating systems read the Path Table once and keep it in memory, rather than reading it over and over again. In the example shown in figure 5, page 9 (the Directory Hierarchy), a system that does not make use of the path table would have to read the root directory to find the location of the ALPS directory, then read the ALPS directory to find the location of the AUSTRIAN directory, then read the SKIING directory to find the location of the file MATTERHORN.MOUNT;1. By making use of the Path Table, the operating system can look up the location of the SKIING directory in the Path Table, read the SKIING directory and find the location of the file. This requires only one seek on the CD-ROM, rather than four. The time difference, for a typical drive with a seek time of 250 msec, is 3/4 of a second. When accessing many files, this difference can significantly affect performance.

Levels of Interchange

ISO 9660 defines three nested levels of interchange which affect the length of the File Identifiers and whether the files must be contiguous or not. Level 1 imposes the most restrictions above and beyond what is specified in ISO 9660. Level 2 impose fewer restrictions, and Level 3 imposes none beyond what is specified in ISO 9660.

Level 1:

- each file shall consist of only one File Section. This means that the files must be contiguous.
- a File Name cannot contain more than eight d-characters or d1-characters.
- a File Name Extension cannot contain more than three d-characters or d1-characters.
- a Directory Identifier cannot contain more than eight d-characters or d1-characters.

An example of Level 1 would be an ISO 9660 disc for the MS-DOS environment, restricted to Level 1 interchange to accommodate MS-DOS's file naming limitations. An important point to note here is that the Level 1 restrictions are more restrictive than the MS-DOS naming conventions. File

Identifiers are still limited to the d-characters as shown in table 1, and Directory Identifiers are limited to d-characters and cannot have an extension.

Level 2:

- each file shall consist of only one File Section. This means that the files must be contiguous.

An example of Level 2 would be an ISO 9660 disc for the Macintosh and UNIX environments, restricted to Level 2 interchange, to allow longer file names.

Level 3:

no restrictions apply beyond what is specified by ISO 9660.

An example of Level 3 would be a CD-ROM-XA disc, which has interleaved data and audio files.

ISO 9660 Implementation Requirements

ISO 9660 also defines requirements for systems that originate or create ISO 9660 volumes and for systems that receive or read ISO 9660 volumes. The requirements for the originating system are primarily of interest to people writing pre-mastering software and will be only briefly mentioned. Pre-mastering is the actual process of creating an ISO 9660 volume. In general, an originating, or pre-mastering system must be able to record a set of files and all of the descriptors described by ISO 9660. The originating system may, however, be restricted to one of the Levels of Interchange. Anyone interested in this level of understanding needs to read the actual ISO 9660:1988 specification.

The receiving system is the system that reads an ISO 9660 disc and makes the data accessible to the user. The receiving system is expected to be able to read the files and descriptors on a volume that

meets at least one of the above interchange levels. It is also expected to make the information in these files available to the user. All of them except for the Associated Files, which we will discuss in more detail later. The receiving system is also expected to make available to the user much of the information in the Volume Descriptors and Path Table. All of the following information should be visible to the user:

From the Primary Volume Descriptor:

- Volume Identifier
- Volume Set Identifier
- Copyright File Identifier
- Abstract File Identifier
- Bibliographic File Identifier

From each Supplementary Volume Descriptor:

- Volume Identifier
- Bit 0 of the Volume Flags field, which indicates if the escape sequences used are registered
- Escape Sequences, which define the d1-character set
- Volume Set Identifier
- Copyright File Identifier
- Abstract File Identifier
- Bibliographic File Identifier

From each Path Table Record:

- Parent Directory Number
- Directory Identifier

From each Directory Record:

- File Name of a File Identifier
- File Name Extension of a File Identifier
- Directory Bit of the File Flags field

Similar to the Levels of Interchange, which apply to the originating system, ISO 9660 also defines Levels of Implementation, which apply to the receiving system:

Level 1 implementation:

- At level 1, the receiving system does not have to make any of the data contained in and pointed to by the Supplementary Volume Descriptors available to the user. Most implementations are Level 1.

Level 2 implementation has no such restrictions.

Implementations of ISO 9660

As figure 7. shows, each operating system has its own specific file system. Independent of the operating systems is ISO 9660.

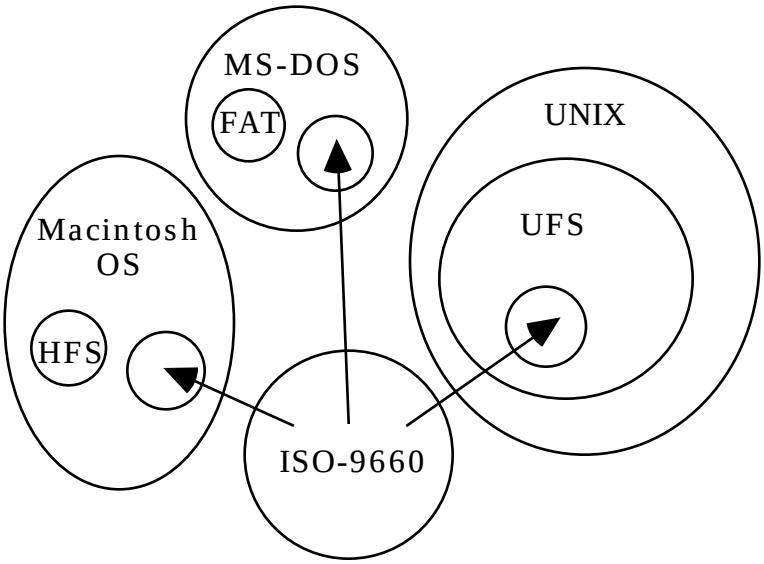


Figure 7. ISO 9660 World View

In order for a given platform to transparently implement ISO-9660, the operating system must convert the ISO-9660 file system into something that looks like its normal file system. This way, the application programs can use the ISO-9660 disc as if were a native read only file system.

DOS

Under MS-DOS, a program called Microsoft CD-ROM Extensions (MSCDEX.EXE) intercepts operating system calls to the CD-ROM and makes the ISO-9660 volume look like a normal, read only, hard disk. However, MS-DOS does not support File Version Numbers, which are part of an ISO 9660 File Identifier. To properly handle the File Version number, when MS-DOS requests a File Identifier from the ISO-9660 volume, Extensions modifies the File Identifier it returns to the operating system by stripping off the File Version Number and only returning the file with the highest version number. To the operating system and the user, the ISO-9660 disc is accessed using a drive letter, just like a write protected hard disk or floppy disk, or network drive.

When creating discs for the DOS environment, the biggest pitfall that developers encounter is that the ISO 9660 file naming conventions. ISO-9660 is much more restrictive than DOS as to the characters that may be used, but much less so as to the length of the names.

MS-DOS file names are limited to eight characters in the file name, and 3 characters in the file name extension. ISO-9660 File Identifiers can be up to 30 characters, and does not specifically limit the length of the Name or the Extension. If the pre-mastering system is DOS based, the length is not a problem, but if the disc is being pre-mastered on a Macintosh or UNIX machine, for example, the file names may be longer than DOS allows. When this disc is read on a DOS machine, the names appear to be truncated, but DOS cannot access the files, as shown in table 7.

Table 7. Long ISO File Identifiers under MS-DOS

Original ISO-9660 File Identifier	File Name as seen by MS-DOS	Response when accessed
MATTERHORN.MOUNT;1	MATTERHO.RNM	File Not Found
EVEREST.MNT;1	EVEREST.MNT	OK

MS-DOS allows a much larger group of characters to be included in file names than does ISO -9660. If an ISO-9660 Volume is made using characters that are allowed under MS-DOS, but do not conform to the d-character set, as shown in figure 2, there is a good chance that some files will not be readable. The files appear normally in the directory, but DOS may or may not be able to open all of the files. This problem is particularly perplexing since the file with the illegal character can be read, but the next file after it may not be. This problem occurs if you have a set of files, which have the same names up to some point, after which one file has an illegal character, and one file a SEPARATOR 1 (period, "."), as demonstrated in table 8.

Table 8. Illegal d-characters and Microsoft extensions

INSTALL-1.TXT;1	Reads OK
INSTALL.BAT;1	Reports "File not found"
INSTALL.EXE;1	Reports "File not found"
INSTALL.TXT;1	Reports "File not found"

In the example shown in table 10, the first name has an illegal ISO-9660 character, the dash (-). This file is readable. However, when attempting to read any of the other three files, MS-DOS reports "File not found". This appears to be occurring because Microsoft Extensions assumes that the directory records are sorted properly and when it sees a name that, if sorted properly, would come after the name it is searching for, it stops searching and proclaims that it cannot find the file. In the example shown in table 9, if MS-DOS is searching for INSTALL.BAT, it reads the first File Name and compares it to INSTALL.BAT. If properly sorted according to the rules specified in ISO 9660, INSTALL.BAT would appear before INSTALL-1.TXT, as shown in table 11. However, the actual order of the records has INSTALL-1.TXT first, before any of other INSTALL files. Therefore, when MS-DOS searches for INSTALL.BAT, it sees INSTALL-1.TXT, which should come after INSTALL.BAT, and comes to the conclusion that INSTALL.BAT is not there.

Table 9. Sorting illegal ISO-9660 File Identifiers

Order of directory records in ISO-9660 Volume	Correctly sorted directory records
INSTALL-1.TXT;1	INSTALL.BAT;1
INSTALL.BAT;1	INSTALL.EXE;1
INSTALL.EXE;1	INSTALL.TXT;1
INSTALL.TXT;1	INSTALL-1.TXT;1

What appears to be causing this is that most pre-mastering packages do not sort the names properly **IF** they are told to put illegal d-characters in the ISO 9660 volume. In particular, they appear to no longer sort the File Name and the File Name Extension separately, but treat the entire File Identifier as the File Name. The best way to avoid this is to not have any illegal d-characters in the File Names, Directory Names, or File Name Extensions.

Macintosh

The Macintosh supports ISO 9660 by adding an extension onto the operating system, called the foreign file access extension. This operating system extension, along with code that tells it how to convert ISO-9660, makes an ISO-9660 disc appear on the desktop just like any other write-protected HFS Volume, with its own icon on the Macintosh desktop.

Even though the ISO 9660 disc looks like a normal volume, not all of the data that the Macintosh needs to display the volume on the desktop is available in the ISO 9660 data. The additional information gets created by the ISO-9660 access software when an ISO directory is opened. The result of this is that the user has no control over the placement of folders and files, and an ISO 9660 disc loses some of the look and feel so important to the Macintosh.

Another problem that occurs with ISO 9660 on the Macintosh has to do with how the Macintosh file system implements executable files, or applications as they are known to Mac users. All of the files on a standard ISO 9660 disc will appear on the desktop as generic documents. To correct this deficiency, Apple uses the reserved system use field in the directory record and associated files in ISO 9660 to add extra information that allows applications to run from an ISO 9660 disc. See the discussion on extensions to ISO 9660 for more details, page 29.

For some developers, another source of difficulties is the File Version Number. The File Version Number is added to each file when it is pre-mastered to make the File Identifiers comply with the ISO specification. Applications that worked from the hard disk may not work from the CD-ROM for the simple reason that they are trying to find a file whose name now has a ';1' on the end of it.

UNIX

UNIX systems typically support ISO 9660 by including the programs into the operating system, rather than using external conversion programs or extension drivers. This is normally done by recompiling the operating system along with the new code to support CD-ROM and ISO 9660. The CD-ROM can then be "mounted" on an existing directory in the UNIX directory structure. On UNIX style systems, instead of having different drive letters or volumes, different drives, or devices, are "mounted" on a directory. This means that each UNIX system only has one directory structure, and all devices are part of that structure as shown in figure 8.

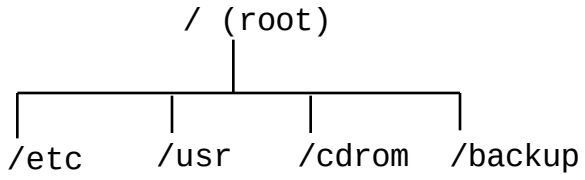


Figure 8. UNIX directory

In this example, the `etc` directory may be on the hard disk with the operating system, the `usr` directory may be a second hard disk used to store user's files, the `cdrom` directory may be an ISO-9660 disc, and the `backup` directory may be a network drive on an entirely different system.

Unfortunately, because of the way UNIX has historically been implemented, there is quite a bit of variation among UNIX systems as to how they support CD-ROM and how the files on the CD-ROM

will appear to the user. As there are well over 100 UNIX compatible Operating Systems currently available, no attempt will be made to address each one separately. The best way to find out how a system will react is to read the manual pages for the mount command and for cdrom and the CD-ROM file system. Some of the most common idiosyncrasies will be mentioned, however.

Some UNIX systems do not modify the file names at all and the user will see upper case files names, with the version number appended. This seems reasonable, since it shows the complete File Identifier, but on many systems the semi-colon character has special meaning and causes access difficulties. Other systems will strip off the version number in much the same way that MSDOS does, and have the option to either convert the characters to lower case or to leave them uppercase. Of the systems that have this option, some default to converting to lower case, while others default to leaving everything upper case. See table 10 for examples of what these different options look like.

Table 10. UNIX File name conversions

ISO 9660 File Identifier	Conversion being applied	File Name as it appears to user
MANPAGES.1;1	none	MANPAGES.1;1
MANPAGES.1;1	no version number	MANPAGES.1
MANPAGES.1;1	lower case, no version number	manpages.1

This provides the system administrator with a lot of flexibility, but makes it difficult for applications that have file names "hard coded" into the programs. These discs have to include clear instructions to the user to have the system mount the disc with filenames in the proper case. Anyone who has been involved in porting applications to different UNIX systems will find this situation not at all unusual.

Extensions to ISO 9660

ISO 9660 works well for a majority of operating systems. In some cases, however, it has proven to be difficult or impossible to use. To make it more usable in their respective environments, Apple and the UNIX community established extensions to ISO 9660. The Apple Extensions were created by Apple and are generally known as Apple ISO 9660. The UNIX community assembled a group of people created what is known as the Rock Ridge Proposal.

Recordable CD, or CD-R, opens up the possibility of updating information already on a CD-ROM. ISO-9660 was never intended to support this function. To meet this new requirement, a group met at Frankfurt, Germany and developed an update to ISO 9660. This update is called the Frankfurt Group Proposal - Volume and File Structure for Read-Only and Write-Once Compact Disc. At this point in time, the Frankfurt Proposal has been approved by the European Computer Manufacturers Association as standard ECMA 168. The Frankfurt Proposal is very complex and difficult to implement. To simply provide a way to update information on an ISO 9660 volume, a much simpler method has been implemented by several companies. This method is commonly known as Updatable ISO 9660 or Multi-Session ISO 9660.

Apple ISO 9660[12]

The Macintosh operating system requires a lot of specialized data to support its graphical user interface. The information needed to implement these features is stored in two places by the HFS file system. Some information is stored in special fields in the HFS directory record. Most of the data is stored in the file itself, in a specially formatted area called the resource fork. The data in the resource

[12] For further information, see "CD-ROM & the Macintosh Computer, A Gentle, Technical Introduction to Creating CD-ROMS for the Apple Macintosh computer family" and "The Apple CD-ROM Handbook"

fork is normally only accessible through system calls to a part of the Macintosh Operating system known as the Resource Manager. The resource fork contains things like the Menu layout, the Window definitions, user preferences, and text messages that the application displays, and the actual executable code in the case of an application. The remaining data in a file is called the data fork. The data fork is the same as a DOS or UNIX file.

Most Macintosh files have both forks, a data fork and a resource fork. ISO 9660 accommodates the resource fork very well through the use of the associated file. When a Macintosh file is recorded in an ISO volume, the resource fork is recorded as an associated file, and the data fork is recorded as a normal file.

Unlike the resource fork, there is some information vital to the Macintosh environment that can not be stored on a standard ISO 9660 volume. The native Macintosh file system, HFS, stores information regarding the file icon's position on the screen, what the icon looks like, what type of file it is, what application created it, and file attributes. The file attributes contain information such as if the file is visible, if the file is locked, and if it is an alias. To capture this information on an ISO 9660 volume, Apple created an extension to ISO 9660, making use of the System Use Field in the Directory Record. This field is allocated in ISO 9660, but how it is used is left open.

Apple was not, however, able to make an ISO disc look exactly like an HFS disk. Currently, there is no way to record the positions of icons on the desktop in an ISO 9660 volume, so these are created when a directory is opened, with no way to control where they are located. Also, because of the way the Finder works, files and folders on an ISO disc only appear with generic icons, as seen in figure 9.

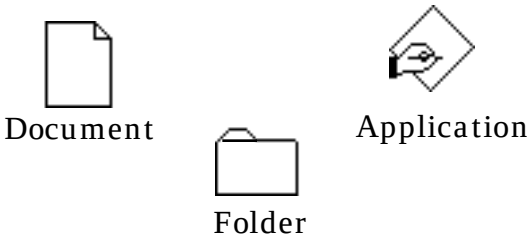


Figure 9. Apple Macintosh generic Icons

These generic icons are displayed even if the correct icons are in the Apple extensions area because the Finder assumes there is a desktop database from which to retrieve these icons and uses a special call to retrieve them. This part of the Finder was written to be very HFS specific and does not work with ISO 9660 volumes even if there is a file called desktop in the volume. If these files are copied to an HFS volume, however, the correct icon will be shown. If the Apple extensions are not present, all files will show as generic documents.

The Protocol Identifier

To identify a volume as having the Apple extensions and inform the operating system what type of file name translation to perform, the System Identifier field in the Primary Volume Descriptor is defined to be the following[13]:

"APPLE COMPUTER, INC., TYPE: " followed by a four byte identifier.

The four byte identifier tells the system what version of Apple Extensions this is and whether or not to perform automatic file name translation for ProDOS (used on the Apple //GS). A typical identifier, that tells the system not to perform ProDOS translation and that the version number is 2 is "0002".

[13] "CD-ROM & the Macintosh Computer...", pp. 23

The Directory Record System Use Field

The System Use Field[14] of an ISO 9660 Directory record is used to store the additional information needed by the Macintosh Operating system. The additional fields are defined as shown in table 11[15].

Table 11. Apple ISO 9660 Directory Record System Use Field

BP	Field Name	Content
1 to 2	Signature ID	(41)(41) "AA" Apple signature
3	SystenUse Extension Length	(0E) bytes
4	System Use ID	(02) for HFS
5 to 8	HFS fileType	(MSB-LSB)
9 to 12	HFS fileCreator	(MSB-LSB)
13 to 16	HFS finder flags	(MSB-LSB)

Apple also allows more than one System Use Extension field in a single directory entry, limited only by the size of the directory entry (it cannot be larger than a logical block). Apple intended for the Signature ID to be used to identify extensions for different systems. Each system could then ignore the fields whose Signature ID it did not recognize. This method of sharing the System Use field was later adopted by both the Rock Ridge Group and the Frankfurt Group for extending the ISO 9660 Directory record.

[14] See ISO 9660:1988, pp. 21, section 9.1.13
[15] "CD-ROM & the Macintosh Computer...", pp. 25

The Rock Ridge Proposals

Rock Ridge is a group of companies that began meeting in July 1990 to resolve issues with ISO 9660 that make it difficult to use as a distribution medium for some operating systems (UNIX based systems being their primary concern). Some of the issues addressed include long filenames with lower case characters, directory structures much deeper than the eight levels allowed by ISO 9660, different file types and access privileges. The Rock Ridge proposals offer industry standard solutions for the distribution of data and software on CD-ROM media by extending the ISO 9660:1988 specification while remaining completely compliant with it.

These proposals deal with two main areas. First, it establishes a standardized way for multiple file system extensions to coexist in one ISO 9660 Directory record. I then defines a way to record POSIX[16] files and directories in an ISO volume without modifying their original directory information. POSIX is a standard for a Portable Operating System Interface much of which is based on how the UNIX operating system works. This allows standard UNIX style file names and directories to be recorded in an ISO 9660 volume without any modification.

Rock Ridge System Use Sharing Protocol (SUSP)

The System Use Sharing Protocol provides a standard way for multiple systems to record system specific extensions in the System Use field by defining a generic field format for System Use Fields, and a set of generic System Use Fields for that can be used to:

- continue the System Use Fields in an area outside of the directory record
- do additional padding
- identify that SUSP is being used

[16] Institute of Electrical and Electronic Engineers Portable Operating System Interface IEEE Std. 1003.1-1990

- terminate the SUSP area
- identify which system specific extension is being used.

The System Use Field Format is as shown in table 12.

Table 12. SUSP System Use Field

BP	Field Name	Content
1 to 2	Signature Word	Identifies what type of System Use Field this is
3	Length of System Use Field (LEN_SUF)	bytes
4	System Use Field Version	Version Number of System Use Field
5 to LEN_SUF	Data	Content of System Use Field

The Signature Word identifies what the following system use field will be used for. The Signature Words that are defined by SUSP are:

"CE" Continuation Area. This field points to the Logical Block Number where the System Use Area is continued. This allows the System Use data to extend beyond a single Logical Block.

"PD" Padding Field. More than one of this field may appear in any given System Use Area. This is used to fill up empty areas such as might occur at the end of a Logical Block before going on to the Continuation Area.

"SP" System Use Sharing Protocol Indicator. Only one of this field can be in a Volume. It must be in the System Use Area of the first directory record of Root directory. This identifies the Volume as adhering to the System Use Sharing Protocol.

"ST" System Use Sharing Protocol Terminator. This field is an optional field to mark the end of SUSP's use of the System Use area.

"ER" Extensions reference. This field may be mandatory or not, depending on the specification which it describes. This is determined by specifications that make use of the System Use Sharing Protocol.

For more information regarding these System Use Fields, see the System Use Sharing Protocol proposed specification, available on the DMI "demo" disc. The System Use Sharing Protocol does not provide new features by itself, but provides a common base on which to establish new ISO 9660 extensions, such as the Rock Ridge Interchange Protocol.

Rock Ridge Interchange Protocol (RRIP)

Rock Ridge Interchange Protocol was designed to allow users of POSIX and other UNIX like systems to retain much of the directory information that is in the native file system. These systems use directory entries for much more than just pointing to files. Directory entries can point to other entries (symbolic links or aliases) or to device drivers that are linked to peripheral devices such as hard disks, tape drives and CD-ROM drives (device files). The directory entry includes information that lets the system know the file type. Is it a regular file, a directory, a symbolic link, or a device file? The Directory entry also has information regarding who has permission to read, write and execute each file. Most of these systems are multi-user systems, and must be able to limit who can

write to the device file that contains the operating system, or it could accidentally (or purposely?) be erased.

File Names

The Rock Ridge Interchange Protocol is intended to be portable across a variety of POSIX compliant systems, so it makes suggestions that increase portability. However, unlike ISO 9660, it does not set hard and fast limits on the character set that can be used in file names and directory names. Systems that support RRIP must, however, treat file names that are the same except for the case of the letters as being different files (a file named `alpine` is not the same as a file named `Alpine`). RRIP suggests that the only the characters in table 13 be used for maximum portability.

Table 13. Suggested Characters for RRIP File Identifiers

Suggested Characters	ASCII Hexadecimal Code
'A' through 'Z"	(41) - (5A)
'a' through 'z'	(61) - (7A)
period .	(2E)
underscore _	(5F)
hyphen -	(2D)

Deep directories

On most POSIX type systems, not only do the file names tend to be long, but the directories tend to get much deeper than the 8 levels allowed by ISO 9660. For this reason, RRIP defines a way to remap deep directories that allows it to be ISO compliant, while at the same time retaining the deep directory structure on systems supporting RRIP.

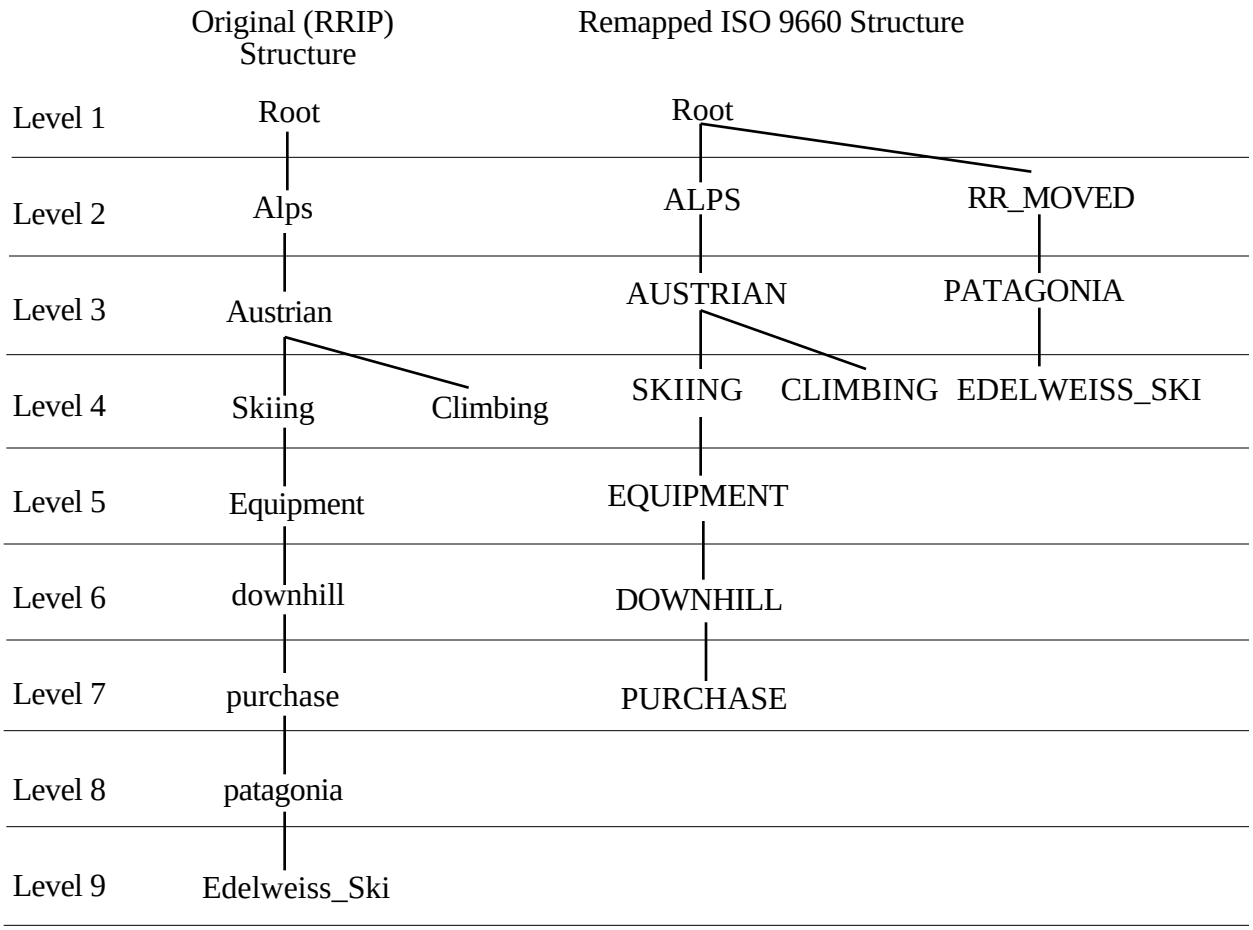


Figure 10. Remapped Directory structure

A directory that was originally at the eighth level is relocated higher up in the directory structure as shown in figure 10. Generally, the pre-mastering software will handle relocating directories automatically. Probably the most common pre-mastering package that supports RRIP, Makedisc from Young Minds, Inc., creates a new directory at the root level, called RR_MOVED, and places all relocated directories here. This directory is only visible on systems that do not support RRIP. On RRIP systems, you see the original, deep, directory structure.

For more information on the exact layout of the RRIP System Use Fields, see the Rock Ridge Interchange Protocol proposed specification, available on the DMI "demo" disc.

Updatable ISO 9660

A simple method for allowing CD-WO discs to be updated has been implemented by several companies. This technique involves writing a new, complete directory structure each time the disc is updated. The new structure is recorded as an ISO 9660 structure, starting at 00:02:16 into the latest recording session, another session being recorded each time the disc is updated. For an explanation of recording sessions and CD-WO in general, see the Compact Disc Terminology paper, available from DMI, and the Philips and Sony Orange book, *Recordable Compact Disc Systems.* A simple volume that has not been updated, and then the same disc after updating is shown in figure 11. The ISO 9660 directory structure in Session 1 only reflects files 1 and 2. The directory structure recorded in session 2, however, reflects not only files 3 and 4, recorded during session 2, but also files 1 and 2, recorded in the session 1.

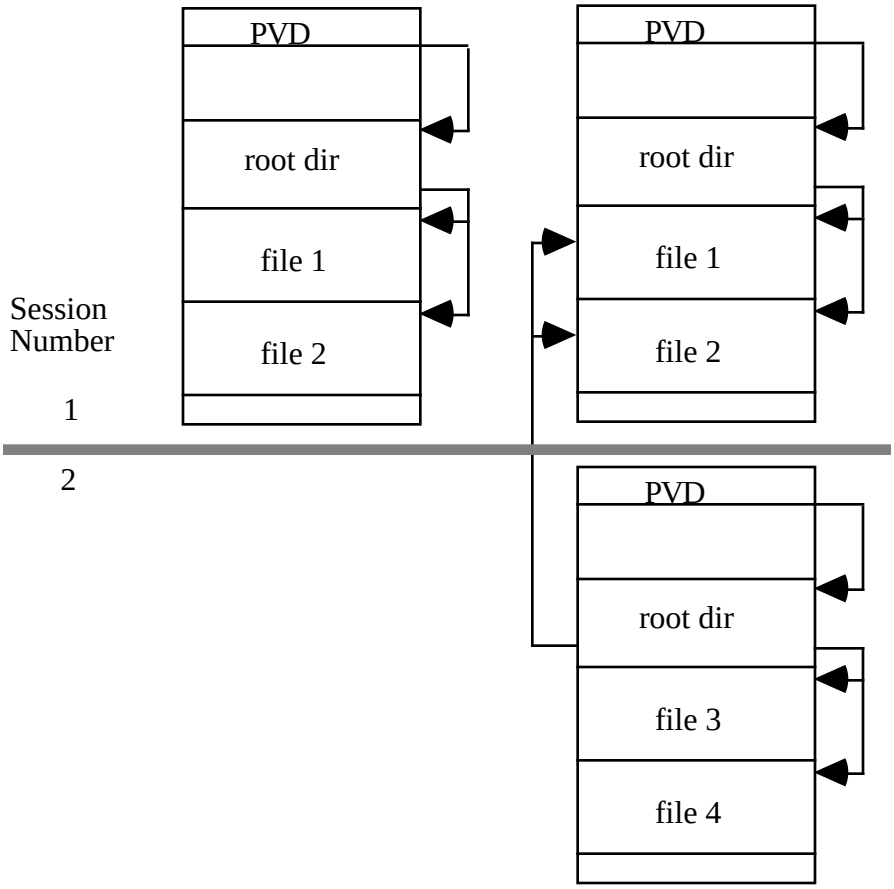


Figure 11. Updatable ISO 9660

In order for a CD-ROM drive to recognize the updated areas, the drive must be able to recognize that there are multiple sessions on the disc, and the ISO 9660 implementation must be able to use the Primary Volume Descriptor from the last session on the disc. This is commonly known as "Multi-Session" compatibility.

The Frankfurt Group Proposal, ECMA 168

The Frankfurt Group is an ad hoc group of companies which share common interests and goals concerning Read Only and Write Once Compact Disc Technology. The Write Once Technology is governed by the Philips and Sony Orange Book, *Recordable Compact Disc Systems*. The Frankfurt Group Proposal covers two types of Volume and File Structures. Type 1 is the original ISO 9660 specification, and is used only for Read-Only discs. Type 2, the Frankfurt Proposal, is an extension to Type 1 that allows incremental recording of a Compact Disc Write Once (CD-WO or CD-R) and updates to be recorded to an existing CD-WO. This proposal is much more complex and flexible than the Updatable ISO 9660. In addition to providing a way to incrementally record and update discs, Type 2 also defines a framework that furnishes the same type of functionality that the Rock Ridge Proposals provide. The specification, however, is not the same as the Rock Ridge Proposals. For more information regarding the Frankfurt Proposal, see ECMA 168 - Volume and File Structure for Read-Only and Write-Once Compact Disc.

Summary of ISO 9660

- An ISO 9660 volume consists of the following data structures:
  - Volume descriptors (what is this volume, and where are the other data structures)
  - The Directory Structure (where are and what are the names of the files and directories)
  - The Path Table (what are the locations and parent directories of each directory)
- ISO 9660 defines three levels of interchange for the Originating System:
  - Level 1 (File names limited to 8.3, directory names limited to 8, and all files must be contiguous) To insure compatibility across the most platforms, restrict the ISO volume to Level 1 Interchange.
  - Level 2 (All files must be contiguous)
  - Level 3 (no limitations beyond the ISO specification itself)
- ISO 9660 defines two levels of Implementation for the Receiving System:
  - Level 1 (Receiving System can ignore any Supplementary Volume Descriptors)
  - Level 2 (Receiving System must utilize all data available)
- Example of ISO 9660 Implementations:
  - MS-DOS:
    - MS-DOS supports ONLY Level 1 Interchange.
    - Access to ISO 9660 volumes is performed by Microsoft Extensions, MSCDEX.EXE.
    - Extensions strips off the version number and only recognizes the highest version.
    - Extensions can not find files and directories with File Identifiers longer than 8.3.
    - Extensions does not handle illegal ISO characters (non d-characters) very well.
  - Apple Macintosh:
    - Apple Macintosh supports Level 2 Interchange. Volumes that must be usable under MS-DOS, as well as Macintosh, must be restricted to Level 1 Interchange.

- Access to ISO 9660 volumes performed by system extensions.
- Layout of folders and files on desktop is created when an ISO 9660 directory is opened.
- ISO 9660 disc must include Apple extensions to run Macintosh applications from the ISO volume.
- ISO 9660 file identifiers include the version number.

- UNIX
  - Most UNIX type systems support Level 2 Interchange. Volumes that must be usable under MS-DOS, as well as UNIX, must be restricted to Level 1 Interchange.
  - Access to ISO 9660 volumes is usually incorporated into the operating system.
  - There is considerable variation between UNIX type implementations as to how the File Identifiers appear to the user.
  - Some systems have options to convert File Identifiers to lower case, and remove the File Version Number, so the same volume can appear different, even on the same machine.

• Extensions to ISO 9660
  - Apple ISO 9660 provides the Macintosh system with additional data needed to launch applications from an ISO 9660 volume.
  - The Rock Ridge Proposals provide a more UNIX like environment for distributing data to a variety of UNIX like platforms.
  - Updatable ISO 9660 provides a simple way to add more data to a previously recorded CD-WO.
  - The Frankfurt Group Proposal, ECMA 168, provides a way to append data to a CD-WO, and provide a more UNIX like environment for distributing data to a variety of UNIX like platforms.

Appendix A: ISO 9660 Structures

Table 14. Primary Volume Descriptor

BytePosition	Field Name	Content
1	Volume Descriptor Type	1
2 to 6	Standard Identifier	CD001
7	Volume Descriptor Version	1
8	Unused Field	(00)[17] byte
9 to 40	System Identifier	a-characters allowed[18]
41 to 72	Volume Identifier	d-characters allowed[19]
73 to 80	Unused Field	(00) bytes
81 to 88	Volume Space Size	Number of logical blocks in the Volume
89 to 120	Unused Field	(00) bytes
121 to 124	Volume Set Size	The assigned Volume Set size of the Volume
125 to 128	Volume Sequence Number	The ordinal number of the volume in the Volume Set
129 to 132	Logical Block Size	The size in bytes of a Logical Block
133 to 140	Path Table Size	Length in bytes of the path table
141 to 144	Location of Type L Path Table	Logical Block Number of first Block allocated to the Type L Path Table, Type L meaning multiple byte numerical values are recorded with least significant byte first. This value is also recorded with least significant byte first.
145 to 148	Location of Optional Type L Path Table	0 if Optional Path Table was not recorded, otherwise, Logical Block Number of first Block allocated to the Optional Type L Path Table.
149 to 152	Location of Type M Path Table	Logical Block Number of first Block allocated to the Type M Path Table, Type M meaning multiple byte numerical values are recorded with most significant byte first. This value is also recorded with most significant byte first.
153 to 156	Location of Optional Type M Path Table	0 if Optional Path Table was not recorded, otherwise, Logical Block Number of first Block allocated to the Type M Path Table.

[17] Numbers surrounded by parentheses () are hexadecimal numbers.

[18] a-characters are A-Z, 0-9, _, space, !, ", %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?
see ISO-9660:1988, Annex A, Table 15

[19] d-characters are A-Z, 0-9, _
see ISO-9660:1988, Annex A, Table 14

157 to 190	Directory record for Root Directory	This is the actual directory record for the top of the directory structure. See the section on directory records for the format of this data.
191 to 318	Volume Set Identifier	Name of the multiple volume set of which this volume is a member. d-characters allowed.
319 to 446	Publisher Identifier	Identifies who provided the actual data contained in the files. a-characters allowed.
447 to 574	Data Preparer Identifier	Identifies who performed the actual creation of the current volume. a-characters allowed.
575 to 702	Application Identifier	Identifies the specification of how the data in the files are recorded. For example, this field might contain SGML if the files were recorded according to the Standard Generalized Markup Language
703 to 739	Copyright File Identifier	Identifies the file in the root directory that contains the copyright notice for this volume. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.[20]
740 to 776	Abstract File Identifier	Identifies the file in the root directory that contains the abstract statement for this volume. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.
777 to 813	Bibliographic File Identifier	Identifies the file in the root directory that contains bibliographic records. ISO-9660 does not specify the format of these records. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.
814 to 830	Volume Creation Date and Time	Date and time at which the volume was created. Represented by seven bytes: 1: Number of years since 1900 2: Month of the year from 1 to 12 3: Day of the Month from 1 to 31 4: Hour of the day from 0 to 23 5: Minute of the hour from 0 to 59 6: second of the minute from 0 to 59 7: Offset from Greenwich Mean Time in number of 15 minute intervals from -48(West) to +52(East)
831 to 847	Volume Modification Date and Time	Date and time at which the volume was last modified. Represented the same as the Volume Creation Date and Time
848 to 864	Volume Expiration Date and Time	Date and Time at which the information in the volume may be considered obsolete. Represented the same as the Volume Creation Date and Time
865 to 881	Volume Effective Date and Time	Date and Time at which the information in the volume may be used. Represented the same as the Volume Creation Date and Time

[20] For a description of the level 1 interchange restrictions, see page <?>

882	File Structure Version	1
883	Reserved for future standardization	(00)
884 to 1395	Application Use	This field is reserved for application use. Its content is not specified by ISO-9660.
1396 to 2048	Reserved for future standardization	All bytes must be set to (00).

Table 15. Directory Record

BP	Field Name	Content
1	Length of directory Record (LEN_DR)	Bytes
2	Extended Attribute Record Length	Bytes - this field refers to the Extended Attribute Record, which provides additional information about a file to systems that know how to use it. Since few systems use it, we will not discuss it here. Refer to ISO 9660:1988 for more information.
3 to 10	Location of Extent	This is the Logical Block Number of the first Logical Block allocated to the file.
11 to 18	Data Length	Length of the file section in bytes
19 to 25	Recording Date and Time	This is recorded in the same format as the Volume Creation Date and Time
26	File Flags	One Byte, each bit of which is a Flag: Bit 0 File is Hidden if this bit is 1 1 Entry is a Directory if this bit is 1 2 Entry is an Associated file is this bit is 1 3 Information is structured according to the extended attribute record if this bit is 1 4 Owner, group and permissions are specified in the extended attribute record if this bit is 1 5 Reserved (0) 6 Reserved (0) 7 File has more than one directory record if this bit is 1
27	File Unit Size	This field is only valid if the file is recorded in interleave mode. Otherwise this field is (00)
28	Interleave Gap Size	This field is only valid if the file is recorded in interleave mode. Otherwise this field is (00)
29 to 32	Volume Sequence Number	The ordinal number of the volume in the Volume Set on which the file described by the directory record is recorded.
33	Length of File Identifier (LEN_FI)	Byte

34 to (33 + LEN_FI)	File Identifier	Interpretation depends on the setting of the directory bit in the File Flags
		
		If set to ZERO, then
		
		The field refers to a File Identifier, as described below
		
		If set to ONE, then
		
		The field refers to a Directory Identifier, as described below.

34 + LEN_FI	Padding Field	Present only if the length of the File Identifier is an even number. If present, value is (00)
LEN_DR - LEN_SU + 1	System Use (LEN_SU)	Reserved for system use. If necessary, so that the length of the directory record is an even number of bytes, a (00) byte may be added to terminate this field.

The Path Table Record contains the following fields:[21]

Table 16. Path Table Record

BP	Field Name	Content
1	Length of Directory Identifier (LEN_DI)	Length in Bytes
2	Extended Attribute Record Length	If an Extended Attribute Record is recorded, this is the length in Bytes. Otherwise, this is (00)
3 to 6	Location of Extent	Logical Block Number of the first Logical Block allocated to the Directory
7 to 8	Parent Directory Number	The record number in the Path Table for the parent directory of this directory
9 to (8 + LEN_DI)	Directory Identifier	This field is the same as in the Directory Record
(9 + LEN_DI)	Padding Field	Present only if LEN_DI is an odd number. (00)

[21] ISO 9660:1988, pp. 22, section 9.4

Appendix B: Common Q&A

1. What do I need to do to make my MS-DOS data ready for ISO-9660?

There are three things that need to be checked to insure that your disc will painlessly translate to ISO-9660:

1- The characters used in the File Names must only be A-Z, 0-9, and _. See figure 3, on page 6.

2- The depth of the directory structure can not exceed 8 levels. See figure 5, on page 9.

3- The length of the path to any file can not exceed 255 characters. See Table 1, on page 10.

To improve performance, you may also want to minimize the number of files in each directory. If there are over 50 files in a directory, you may notice some slowing while reading files in this directory, If there are over 250 files, you are very likely to notice that it takes much longer to read some files in this directory.

2. What do I need to do to make my Macintosh data ready for ISO-9660?

See question 1. In order for a disc to have Macintosh applications on it, the ISO-9660 data must include the Apple extensions, otherwise, every file on the disc will appear to be a text file to the Macintosh. Also, ISO-9660 discs will have the version number appended to each file name. See Macintosh implementation of ISO-9660, page 23, and Apple Extensions to ISO-9660, page 26.

3. Why use ISO 9660 on the Mac and what are the affects?

The primary reason for using ISO-9660 on a disc that will be running on the Macintosh is that the same disc will need to run on other platforms, such as MS-DOS. The primary disadvantage to using ISO-9660 on the Macintosh is the loss of some of the "look and feel" of the Macintosh user interface. In particular, you must live with the restrictions on the file names (see File Names, page 12), and you can not control where the folders open up to

and how they are viewed (see Macintosh implementation of ISO-9660, page 23, and Apple Extensions to ISO-9660, page 26).

4. What is a hybrid disc; what are the issues to consider?

In an effort to alleviate the disadvantages to using ISO-9660 on the Macintosh (see question 3), a scheme called the hybrid disc has been developed. A hybrid disc is basically a disc with two partitions on it. It has both an ISO-9660 partition and an HFS partition. When this disc is mounted on the Macintosh, the Macintosh only sees the HFS partition. On other platforms, only the ISO-9660 partition is visible.

Until recently, the primary disadvantage to a hybrid disc was that any data that was common to the different platforms had to be duplicated in both partitions. This could significantly limit the amount of data you could place on a hybrid disc. There is now becoming available a type of hybrid disc that blends the two partitions. This allows us to only put data on the disc once, and have both ISO-9660 and HFS directory structures point to the same data.

5. How do I make one disc that works on both my PC and Macintosh?

see questions 3 and 4. Often, a disc such as this is created entirely on the Macintosh. An important point to note here is that if the ISO-9660 partition is created on the Macintosh, any files that must be readable by MS-DOS must meet ISO-9660 level 1 Interchange. That is, the file names can not be longer than 8.3 (see Levels of Interchange, page 16 and Implementations of ISO-9660, DOS, page 20).

6 .What do I need to do to make my UNIX data ready for ISO-9660?

See question 1. The primary differences between ISO-9660 for UNIX and MS-DOS is that UNIX generally can support Interchange level 2 (file names longer than 8.3), and UNIX typically can handle larger subdirectories with less performance degradation. An important point to remember when dealing with ISO-9660 and UNIX is that the file names can appear

different on different systems. Even on the same system, different mount options can affect how, or if, the system translates the ISO-9660 file names. (See Implementations of ISO-9660, UNIX, page 24).

7. Will ISO discs work on UNIX, MAC and DOS?

Yes, but... Only files that meet the proper level of Interchange will be readable on any given platform, i.e., UNIX and Mac files that are recorded longer than 8.3 will not be readable by DOS (see Implementations of ISO-9660, DOS, page 19).

8. What is an Image file?

In the jargon of the industry, an Image file is a single data file that contains all of the data that will be recorded to a CD. Typically, this file will contain a complete ISO-9660 volume, or an HFS volume, or a hybrid image with both ISO and HFS volumes. If you can generate an image file, your data is *image ready.*

9. How do I create an Image file?

There are software packages available that will create an ISO-9660 volume and record it to an Image file. These packages are normally call pre-mastering packages and are available from companies such as Dataware Technologies, Meridian Data, Optical Media International, and Trace. There are too many packages to try and list them all here. The best way to choose one is to read reviews and talk to people who are using the package you are considering purchasing.

10. How do I send the Image file to the Mastering Facility?

Most pre-mastering packages contain options to write an ISO-9660 volume to a SCSI tape drive as well as to an Image file. Some packages also support writing to a CD Recordable device. Most mastering houses will accept 8mm Exabyte tapes, 4mm DDS tapes, and CD-R discs as standard input. If none of these options are available, but you do have an Image file, the Image file may be backed up with a standard backup program such as tar on UNIX systems, Retrospect on the Macintosh, and Sytos+ or Novaback on DOS. If you will be sending your Image file this way, be sure and call the mastering house to make sure they can restore the particular backup format you are using.