PNG (Portable Network Graphics) Specification, Tenth Draft

Revision date: 5 May, 1995

Copyright 1995, Thomas Boutell. Permission is granted to reproduce this specification in complete and unaltered form. Excerpts may be printed with the following notice: "excerpted from the PNG (Portable Network Graphics) specification, tenth draft." No notice is required in software that follows this specification; notice is only required when reproducing or excerpting from the specification itself.

Contents

- 0. Status
- 1. Introduction
- 2. Data Representation
  - Integers and byte order
  - Color values
  - Image layout
  - Alpha channel
  - Filtering
  - Interlaced data order
  - Gamma correction
  - Text strings
- 3. File Structure
  - PNG file signature
  - Chunk layout
  - Chunk naming conventions
  - CRC algorithm
- 4. Chunk Specifications
  - Critical Chunks
  - Ancillary Chunks
  - Summary of Standard Chunks
  - Additional Chunk Types
- 5. Deflate/Inflate Compression
- 6. Filter Algorithms
  - Filter type 0: None
  - Filter type 1: Sub
  - Filter type 2: Up
  - Filter type 3: Average
  - Filter type 4: Paeth
- 7. Chunk Ordering Rules
- 8. Multi−Image Extension
- 9. Recommendations for Encoders
  - Bitdepth scaling
  - Encoder gamma handling
  - Alpha channel creation
  - Filter selection
  - Text chunk processing
  - Registering proprietary chunks
- 10. Recommendations for Decoders
  - Chunk error checking
  - Pixel dimensions
  - Truecolor image handling
  - Decoder gamma handling
  - Background color

    - Alpha channel processing
    - Progressive display
    - Palette histogram usage
    - Text chunk processing
- 11. Appendix: Rationale
    - Why a new file format?
    - Why these features?
    - Why not these features?
    - Why not use format XYZ?
    - Byte order
    - Interlacing
    - Why gamma encoding?
    - Non−premultiplied alpha
    - Filtering
    - Text strings
    - PNG file signature
    - Chunk layout
    - Chunk naming conventions
    - Palette histograms
- 12. Appendix: Sample CRC Code
- 13. Credits

0. Status

This is the tenth draft of the PNG specification. All future drafts will be backward−compatible with the graphics file format described by this document. Implementations are invited, and a reference implementation is under development; see below.

Archive sites

The latest versions of this document and related information can always be found at the PNG FTP archive site, `ftp.uu.net:/graphics/png/`. The maintainers of the PNG specification can be contacted by e−mail at `png−info@uunet.uu.net`.

At present, this document is available on the World Wide Web from `http://sunsite.unc.edu/boutell/png.html`, but this location may not be as permanent as the ones above.

Changes since draft 9

- Extensive editing
- Clarified interlaced representation of very small images: no filter bytes are present in an empty pass
- Chunk ordering rules clarified
- Limits set on `tEXt` chunk keyword length
- `cHRM` chunk description corrected: CIE x and y, not X and Y
- More extensive explanation of alpha and gamma processing
- Separate document created for extension chunk types
- Permanent archive site and e−mail contact point established

Reference implementation

It is anticipated that a reference implementation will be available before the end of May 1995. The reference implementation will be freely usable in all applications, including commercial applications.

When finished, the reference implementation will be available from the PNG FTP archive site,

`ftp.uu.net:/graphics/png/`.

A set of test images will also be prepared and will be available from the same site.

1. Introduction

The PNG format is intended to provide a portable, legally unencumbered, well−compressed, well−specified standard for lossless bitmapped image files.

Although the initial motivation for developing PNG was to replace GIF, the design provides some useful new features not available in GIF, with minimal cost to developers.

GIF features retained in PNG include:

- Palette−mapped images of up to 256 colors.
- Streamability: files can be read and written strictly serially, thus allowing the file format to be used as a communications protocol for on−the−fly generation and display of images.
- Progressive display: a suitably prepared image file can be displayed as it is received over a communications link, yielding a low−resolution image very quickly with gradual improvement of detail thereafter.
- Transparency: portions of the image can be marked as transparent, allowing the effect of a nonrectangular image area to be achieved.
- Ancillary information: textual comments and other data can be stored within the image file.
- Complete hardware and platform independence.
- Effective, 100% lossless compression.

Important new features of PNG, not available in GIF, include:

- Truecolor images of up to 48 bits per pixel.
- Grayscale images of up to 16 bits per pixel.
- Full alpha channel (general transparency masks).
- Image gamma indication, allowing automatic brightness/contrast adjustment.
- Reliable, straightforward detection of file corruption.
- Faster initial presentation in progressive display mode.

PNG is intended to be:

- Simple and portable: PNG should be widely implementable with reasonably small effort for developers.
- Legally unencumbered: to the best of the knowledge of the PNG authors, no algorithms under legal challenge are used.
- Well compressed: both palette−mapped and truecolor images are compressed as effectively as in any other widely used lossless format, and in most cases more effectively.
- Interchangeable: any standard−conforming PNG decoder will read all conforming PNG files.
- Flexible: the format allows for future extensions and private add−ons, without compromising interchangeability of basic PNG.
- Robust: the design supports full file integrity checking as well as simple, quick detection of common transmission errors.

The main part of this specification simply gives the definition of the file format. An appendix gives the rationale for many design decisions. Although the rationale is not part of the formal specification, reading it may help implementors to understand the design. Cross−references in the main text point to relevant parts of the rationale.

See Rationale: Why a new file format?, Why these features?, Why not these features?, Why not

use format XYZ?.

Pronunciation

PNG is pronounced "ping".

2. Data Representation

This chapter discusses basic data representations used in PNG files, as well as the expected representation of the image data.

Integers and byte order

All integers that require more than one byte will be in network byte order, which is to say the most significant byte comes first, then the less significant bytes in descending order of significance (MSB LSB for two−byte integers, B3 B2 B1 B0 for four−byte integers). The highest bit (value 128) of a byte is numbered bit 7; the lowest bit (value 1) is numbered bit 0. Values are unsigned unless otherwise noted. Values explicitly noted as signed are represented in two's complement notation.

See Rationale: Byte order.

Color values

All color values range from zero (representing black) to most intense at the maximum value for the bit depth. Color values may represent either grayscale or RGB color data. Note that the maximum value at a given bit depth is not 2^bitdepth, but rather (2^bitdepth)−1. Intensity is not necessarily linear; the `gAMA` chunk specifies the gamma characteristic of the source device, and viewers are strongly encouraged to properly compensate. See Gamma correction, below.

Source data with a precision not directly supported in PNG (for example, 5 bit/sample truecolor) must be scaled up to the next higher supported bit depth. Such scaling is reversible and hence incurs no loss of data, while it reduces the number of cases that decoders must cope with. See Recommendations for Encoders: Bitdepth scaling.

Image layout

PNG images are laid out as a rectangular pixel array, with pixels appearing left−to−right within each scanline, and scanlines appearing top−to−bottom. (For progressive display purposes, the data may not actually be transmitted in this order; see Interlaced data order.) The size of each pixel is determined by the *bit depth*, which is the number of bits per stored value in the image data.

Three types of pixels are supported:

- Palette−mapped pixels are represented by a single value that is an index into a supplied palette. The bit depth determines the maximum number of palette entries, not the color precision within the palette.
- Grayscale pixels are represented by a single value that is a grayscale level, where zero is black and the largest value for the bit depth is white.
- Truecolor pixels are represented by three−value sequences: red (zero = black, max = red) appears first, then green (zero = black, max = green), then blue (zero = black, max = blue). The bit depth specifies the size of each value, not the total pixel size.

Optionally, grayscale and truecolor pixels can also include an alpha value, as described in the next section.

In all cases, pixels are packed into scanlines consecutively, without wasted space between pixels. (The allowable bit depths are restricted so that the packing is simple and efficient.) When pixels

are less than 8 bits deep, they are packed into bytes with the leftmost pixel in the high−order bits of a byte, the rightmost in the low−order bits.

However, scanlines always begin on byte boundaries. When pixels are less than 8 bits deep, if the scanline width is not evenly divisible by the number of pixels per byte then the low−order bits in the last byte of each scanline are wasted. The contents of the padding bits added to fill out the last byte of a scanline are unspecified.

An additional "filter" byte is added to the beginning of every scanline, as described in detail below. The filter byte is not considered part of the image data, but it is included in the data stream sent to the compression step.

Alpha channel

An alpha channel, representing transparency levels on a per−pixel basis, may be included in grayscale and truecolor PNG images.

An alpha channel value of 0 represents full transparency, and a value of (2^bitdepth)−1 represents a fully opaque pixel. Intermediate values indicate partially transparent pixels that may be combined with a background image to yield a composite image.

Alpha channels may be included with images that have either 8 or 16 bits per sample, but not with images that have fewer than 8 bits per sample. Alpha values are represented with the same bit depth used for the image values. The alpha value is stored immediately following the grayscale or RGB values of the pixel.

The color stored for a pixel is not affected by the alpha value assigned to the pixel. This rule is sometimes called "unassociated" or "non premultiplied" alpha. (Another common technique is to store pixel values premultiplied by the alpha fraction; in effect, the image is already composited against a black background. PNG does *not* use premultiplied alpha.)

Transparency control is also possible without the storage cost of a full alpha channel. In a palette image, an alpha value may be defined for each palette entry. In grayscale and truecolor images, a single pixel value may be identified as being "transparent". These techniques are controlled by the `tRNS` ancillary chunk type.

If no alpha channel nor `tRNS` chunk is present, all pixels in the image are to be treated as fully opaque.

Viewers may support transparency control partially, or not at all.

See Rationale: Non−premultiplied alpha, Recommendations for Encoders: Alpha channel creation, and Recommendations for Decoders: Alpha channel processing.

Filtering

PNG allows the image data to be *filtered* before it is compressed. The purpose of filtering is to improve the compressibility of the data. The filter step itself does not reduce the size of the data. All PNG filters are strictly lossless.

PNG defines several different filter algorithms, including "none" which indicates no filtering. The filter algorithm is specified for each scanline by a filter type byte which precedes the filtered scanline in the precompression data stream. An intelligent encoder may switch filters from one scanline to the next. The method for choosing which filter to employ is up to the encoder.

See Filter Algorithms and Rationale: Filtering.

Interlaced data order

A PNG image can be stored in interlaced order to allow progressive display. The purpose of this feature is to allow images to "fade in" when they are being displayed on−the−fly. Interlacing slightly expands the file size on average, but it gives the user a meaningful display much more rapidly. Note that decoders are required to be able to read interlaced images, whether or not they actually perform progressive display.

With interlace type 0, pixels are stored sequentially from left to right, and scanlines sequentially from top to bottom (no interlacing).

Interlace type 1, known as Adam7 after its author, Adam M. Costello, consists of seven distinct passes over the image. Each pass transmits a subset of the pixels in the image. The pass in which each pixel is transmitted is defined by replicating the following 8−by−8 pattern over the entire image, starting at the upper left corner:

```
1 6 4 6 2 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
3 6 4 6 3 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
```

Within each pass, the selected pixels are transmitted left to right within a scanline, and selected scanlines sequentially from top to bottom. For example, pass 2 contains pixels 4, 12, 20, etc. of scanlines 0, 8, 16, etc. (numbering from 0,0 at the upper left corner). The last pass contains the entirety of scanlines 1, 3, 5, etc.

The data within each pass is laid out as though it were a complete image of the appropriate dimensions. For example, if the complete image is 8x8 pixels, then pass 3 will contain a single scanline containing two pixels. When pixels are less than 8 bits deep, each such scanline is padded to fill an integral number of bytes (see Image layout). Filtering is done on this reduced image in the usual way, and a filter type byte is transmitted before each of its scanlines (see Filter Algorithms). Notice that the transmission order is defined so that all the scanlines transmitted in a pass will have the same number of pixels; this is necessary for proper application of some of the filters.

Caution: If the image contains fewer than five columns or fewer than five rows, some passes will be entirely empty. Encoder and decoder authors must be careful to handle this case correctly. In particular, filter bytes are only associated with nonempty scanlines; no filter bytes are present in an empty pass.

See Rationale: Interlacing and Recommendations for Decoders: Progressive display.

Gamma correction

Gamma is a way of defining the brightness reproduction curve of a camera or display device. When brightness levels are expressed as fractions in the range 0 to 1, such a device produces an output brightness level "`obright`" from an input brightness level "`ibright`" according to the equation

```
obright = ibright ^ gamma
```

PNG images may specify the gamma of the camera (or simulated camera) that produced the image, and thus the gamma of the image with respect to the original scene. To get accurate tone reproduction, the gamma of the display device and the gamma of the image file should be reciprocals of each other, since the overall gamma of the system is the product of the gammas of each component. So, for example, if an image with a gamma of 0.4 is displayed on a CRT with a

gamma of 2.5, the overall gamma of the system is 1.0. An overall gamma of 1.0 gives correct tone reproduction.

In practice, images of gamma around 1.0 and gamma around 0.45 are both widely found. PNG expects encoders to record the gamma if known, and it expects decoders to correct the image gamma if necessary for proper display on their display hardware. Failure to correct for image gamma leads to a too−dark or too−light display.

Gamma correction is not applied to the alpha channel, if any. Alpha values always represent a linear fraction of full opacity.

See Rationale: Why gamma encoding?, Recommendations for Encoders: Encoder gamma handling, and Recommendations for Decoders: Decoder gamma handling.

Text strings

A PNG file can store text associated with the image, such as an image description or copyright notice. Keywords are used to indicate what each text string represents.

ISO 8859−1 (Latin−1) is the character set recommended for use in text strings. This character set is a superset of 7−bit ASCII. Files defining the character set may be obtained from the PNG FTP archives, `ftp.uu.net:/graphics/png/`.

Character codes not defined in Latin−1 may be used, but are unlikely to port across platforms correctly. (For that matter, *any* characters beyond 7−bit ASCII will not display correctly on all platforms; but Latin−1 represents a set which is widely portable.)

Provision is also made for the storage of compressed text.

See Rationale: Text strings.

3. File Structure

A PNG file consists of a PNG *signature* followed by a series of *chunks*. This chapter defines the signature and the basic properties of chunks. Individual chunk types are discussed in the next chapter.

PNG file signature

The first eight bytes of a PNG file always contain the following (decimal) values:

```
137 80 78 71 13 10 26 10
```

This signature indicates that the remainder of the file contains a single PNG image, consisting of a series of chunks beginning with an `IHDR` chunk and ending with an `IEND` chunk.

See Rationale: PNG file signature.

Chunk layout

Each chunk consists of four parts:

Length
: A 4−byte unsigned integer giving the number of bytes in the chunk's data field. The length counts *only* the data field, *not* itself, the chunk type code, or the CRC. Zero is a valid length. Although encoders and decoders should treat the length as unsigned, its value may not exceed (2^31)−1 bytes.

Chunk Type

A 4−byte chunk type code. For convenience in description and in examining PNG files, type codes are restricted to consist of uppercase and lowercase ASCII letters (`A−Z`, `a−z`). However, encoders and decoders should treat the codes as fixed binary values, not character strings. For example, it would not be correct to represent the type code `IDAT` by the EBCDIC equivalents of those letters. Additional naming conventions for chunk types are discussed in the next section.

Chunk Data
: The data bytes appropriate to the chunk type, if any. This field may be of zero length.

CRC
: A 4−byte CRC (Cyclical Redundancy Check) calculated on the preceding bytes in that chunk, **including** the chunk type code and chunk data fields, but **not including** the length field. The CRC is **always** present, even for empty chunks such as `IEND`. The CRC algorithm is specified below.

The chunk data length may be any number of bytes up to the maximum; therefore, implementors may not assume that chunks are aligned on any boundaries larger than bytes.

Chunks may appear in any order, subject to the restrictions placed on each chunk type. (One notable restriction is that `IHDR` must appear first and `IEND` must appear last; thus the `IEND` chunk serves as an end−of−file marker.) Multiple chunks of the same type may appear, but only if specifically permitted for that type.

See Rationale: Chunk layout.

Chunk naming conventions

Chunk type codes are assigned in such a way that a decoder can determine some properties of a chunk even if it does not recognize the type code. These rules are intended to allow safe, flexible extension of the PNG format, by allowing a decoder to decide what to do when it encounters an unknown chunk. The naming rules are not normally of interest when a decoder does recognize the chunk's type.

Four bits of the type code, namely bit 5 (value 32) of each byte, are used to convey chunk properties. This choice means that a human can read off the assigned properties according to whether each letter of the type code is uppercase (bit 5 is 0) or lowercase (bit 5 is 1). However, decoders should test the properties of an unknown chunk by numerically testing the specified bits; testing whether a character is uppercase or lowercase is inefficient, and even incorrect if a locale−specific case definition is used.

It is also worth noting that the property bits are an inherent part of the chunk name, and hence are fixed for any chunk type. Thus, `TEXT` and `Text` are completely unrelated chunk type codes. Decoders should recognize codes by simple four−byte literal comparison; it is incorrect to perform case conversion on type codes.

The semantics of the property bits are:

First Byte: 0 (uppercase) = critical, 1 (lowercase) = ancillary
: Chunks which are not strictly necessary in order to meaningfully display the contents of the file are known as "ancillary" chunks. Decoders encountering an unknown chunk in which the ancillary bit is 1 may safely ignore the chunk and proceed to display the image. The time chunk (`tIME`) is an example of an ancillary chunk.

: Chunks which are critical to the successful display of the file's contents are called "critical" chunks. Decoders encountering an unknown chunk in which the ancillary bit is 0 must indicate to the user that the image contains information they cannot safely interpret. The image header chunk (`IHDR`) is an example of a critical chunk.

Second Byte: 0 (uppercase) = public, 1 (lowercase) = private

If the chunk is public (part of this specification or a later edition of this specification), its second letter is uppercase. If your application requires proprietary chunks, and you have *no* interest in seeing the software of other vendors recognize them, use a lowercase second letter in the chunk name. Such names will never be assigned in the official specification. Note that there is no need for software to test this property bit; it simply ensures that private and public chunk names will not conflict.

Third Byte: reserved, must be 0 (uppercase) always

The significance of the case of the third letter of the chunk name is reserved for possible future expansion. At the present time all chunk names must have uppercase third letters.

Fourth Byte: 0 (uppercase) = unsafe to copy, 1 (lowercase) = safe to copy

This property bit is not of interest to pure decoders, but it is needed by PNG editors (programs that modify a PNG file).

If a chunk's safe−to−copy bit is 1, the chunk may be copied to a modified PNG file whether or not the software recognizes the chunk type, and regardless of the extent of the file modifications.

If a chunk's safe−to−copy bit is 0, it indicates that the chunk depends on the image data. If the program has made *any* changes to *critical* chunks, including addition, modification, deletion, or reordering of critical chunks, then unrecognized unsafe chunks must **not** be copied to the output PNG file. (Of course, if the program **does** recognize the chunk, it may choose to output an appropriately modified version.)

A PNG editor is always allowed to copy all unrecognized chunks if it has only added, deleted, or modified ancillary chunks. This implies that it is not permissible to make ancillary chunks that depend on other ancillary chunks.

PNG editors that do not recognize a *critical* chunk must report an error and refuse to process that PNG file at all. The safe/unsafe mechanism is intended for use with ancillary chunks. The safe−to−copy bit will always be 0 for critical chunks.

Rules for PNG editors are discussed further under Chunk Ordering Rules.

For example, the hypothetical chunk type name "`bLOb`" has the property bits:

```
bLOb  <-- 32 bit Chunk Name represented in ASCII form
||||
||||- Safe to copy bit is 1 (lower case letter; bit 5 of byte is 1)
|||-- Reserved bit is 0     (upper case letter; bit 5 of byte is 0)
||--- Private bit is 0      (upper case letter; bit 5 of byte is 0)
|---- Ancillary bit is 1    (lower case letter; bit 5 of byte is 1)
```

Therefore, this name represents an ancillary, public, safe−to−copy chunk.

See Rationale: Chunk naming conventions.

CRC algorithm

Chunk CRCs are calculated using standard CRC methods with pre and post conditioning. The CRC polynomial employed is as follows:

```
x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+1
```

The 32−bit CRC register is initialized to all 1's, and then the data from each byte is processed from the least significant bit (1) to the most significant bit (128). After all the data bytes are processed, the CRC register is inverted (its ones complement is taken). This value is transmitted

(stored in the file) MSB first. For the purpose of separating into bytes and ordering, the least significant bit of the 32−bit CRC is defined to be the coefficient of the `x^31` term.

Practical calculation of the CRC always employs a precalculated table to greatly accelerate the computation. See Appendix: Sample CRC Code.

4. Chunk Specifications

This chapter defines the standard types of PNG chunks.

Critical Chunks

All implementations must understand and successfully render the standard critical chunks. A valid PNG image must contain an `IHDR` chunk, one or more `IDAT` chunks, and an `IEND` chunk.

`IHDR` Image Header

This chunk must appear FIRST. Its contents are:

```
Width:              4 bytes
Height:             4 bytes
Bit depth:          1 byte
Color type:         1 byte
Compression type:   1 byte
Filter type:        1 byte
Interlace type:     1 byte
```

Width and height give the image dimensions in pixels. They are 4−byte integers. Zero is an invalid value. The maximum for each is (2^31)−1 in order to accommodate languages which have difficulty with unsigned 4−byte values.

Bit depth is a single−byte integer giving the number of bits per pixel (for palette images) or per sample (for grayscale and truecolor images). Valid values are 1, 2, 4, 8, and 16, although not all values are allowed for all color types.

Color type is a single−byte integer that describes the interpretation of the image data. Color type values represent sums of the following values: 1 (palette used), 2 (color used), and 4 (full alpha used). Valid values are 0, 2, 3, 4, and 6.

Bit depth restrictions for each color type are imposed both to simplify implementations and to prohibit certain combinations that do not compress well in practice. Decoders must support all legal combinations of bit depth and color type. (Note that bit depths of 16 are easily supported on 8−bit display hardware by dropping the least significant byte.) The allowed combinations are:

Color Type	Allowed Bit Depths	Interpretation
0	1,2,4,8,16	Each pixel value is a grayscale level.
2	8,16	Each pixel value is an R,G,B series.
3	1,2,4,8	Each pixel value is a palette index; a PLTE chunk must appear.
4	8,16	Each pixel value is a grayscale level, followed by an alpha channel level.
6	8,16	Each pixel value is an R,G,B series, followed by an alpha channel level.

Compression type is a single−byte integer that indicates the method used to compress the image data. At present, only compression type 0 (deflate/inflate compression with a 32K

sliding window) is defined. All standard PNG images must be compressed with this scheme. The compression type code is provided for possible future expansion or proprietary variants. Decoders must check this byte and report an error if it holds an unrecognized code. See Deflate/Inflate Compression for details.

Filter type is a single−byte integer that indicates the preprocessing method applied to the image data before compression. At present, only filter type 0 (adaptive filtering with five basic filter types) is defined. As with the compression type code, decoders must check this byte and report an error if it holds an unrecognized code. See Filter Algorithms for details.

Interlace type is a single−byte integer that indicates the transmission order of the pixel data. Two values are currently defined: 0 (no interlace) or 1 (Adam7 interlace). See Interlaced data order for details.

`PLTE` Palette

This chunk's contents are from 1 to 256 palette entries, each a three−byte series of the form:

```
red:   1 byte (0 = black, 255 = red)
green: 1 byte (0 = black, 255 = green)
blue:  1 byte (0 = black, 255 = blue)
```

The number of entries is determined from the chunk length. A chunk length not divisible by 3 is an error.

This chunk must appear for color type 3, and may appear for color types 2 and 6. If this chunk does appear, it must precede the first `IDAT` chunk. There cannot be more than one `PLTE` chunk.

For color type 3 (palette data), the `PLTE` chunk is required. The first entry in `PLTE` is referenced by pixel value 0, the second by pixel value 1, etc. The number of palette entries must not exceed the range that can be represented by the bit depth (for example, 2^4 = 16 for a bit depth of 4). It is permissible to have fewer entries than the bit depth would allow. In that case, any out−of−range pixel value found in the image data is an error.

For color types 2 and 6 (truecolor), the `PLTE` chunk is optional. If present, it provides a recommended set of from 1 to 256 colors to which the truecolor image may be quantized if the viewer cannot display truecolor directly. If `PLTE` is not present, such a viewer must select colors on its own, but it is often preferable for this to be done once by the encoder.

Note that the palette uses 8 bits (1 byte) per value regardless of the image bit depth specification. In particular, the palette is 8 bits deep even when it is a suggested quantization of a 16−bit truecolor image.

`IDAT` Image Data

This chunk contains the actual image data. To create this data, begin with image scanlines represented as described under Image layout; the layout and total size of this raw data are determinable from the `IHDR` fields. Then filter the image data according to the filtering method specified by the `IHDR` chunk. (Note that with filter method 0, the only one currently defined, this implies prepending a filter type byte to each scanline.) Finally, compress the filtered data using the compression method specified by the `IHDR` chunk. The `IDAT` chunk contains the output datastream of the compression algorithm. To read the image data, reverse this process.

There may be multiple `IDAT` chunks; if so, they must appear consecutively with no other intervening chunks. The compressed datastream is then the concatenation of the contents of all the `IDAT` chunks. The encoder may divide the compressed data stream into `IDAT` chunks as it wishes. (Multiple `IDAT` chunks are allowed so that encoders can work in a

fixed amount of memory; typically the chunk size will correspond to the encoder's buffer size.) It is important to emphasize that `IDAT` chunk boundaries have no semantic significance and can appear at any point in the compressed datastream. A PNG file in which each `IDAT` chunk contains only one data byte is legal, though remarkably wasteful of space. (For that matter, zero−length `IDAT` chunks are legal, though even more wasteful.)

See Filter Algorithms and Deflate/Inflate Compression for details.

`IEND` Image Trailer
This chunk must appear LAST. It marks the end of the PNG data stream. The chunk's data field is empty.

Ancillary Chunks

All ancillary chunks are optional, in the sense that encoders need not write them and decoders may ignore them. However, encoders are encouraged to write the standard ancillary chunks when the information is available, and decoders are encouraged to interpret these chunks when appropriate and feasible.

The standard ancillary chunks are listed in alphabetical order. This is not necessarily the order in which they would appear in a file.

`bKGD` Background Color
This chunk specifies a default background color against which the image may be presented. Note that viewers are not bound to honor this chunk; a viewer may choose to use a different background color.

For color type 3 (palette), the `bKGD` chunk contains:

```
palette index: 1 byte
```

The value is the palette index of the color to be used as background.

For color types 0 and 4 (grayscale, with or without alpha), `bKGD` contains:

```
gray:  2 bytes, range 0 .. (2^bitdepth) − 1
```

(For consistency, 2 bytes are used regardless of the image bit depth.) The value is the gray level to be used as background.

For color types 2 and 6 (RGB, with or without alpha), `bKGD` contains:

```
red:   2 bytes, range 0 .. (2^bitdepth) − 1
green: 2 bytes, range 0 .. (2^bitdepth) − 1
blue:  2 bytes, range 0 .. (2^bitdepth) − 1
```

(For consistency, 2 bytes per sample are used regardless of the image bit depth.) This is the RGB color to be used as background.

When present, the `bKGD` chunk must precede the first `IDAT` chunk, and must follow the `PLTE` chunk, if any.

See Recommendations for Decoders: Background color.

`cHRM` Primary Chromaticities and White Point
Applications that need precise specification of colors in a PNG file may use this chunk to specify the chromaticities of the red, green, and blue primaries used in the image, and the referenced white point. These values are based on the 1931 CIE (International Color

Committee) XYZ color space. Only the chromaticities (*x* and *y*) are specified. The chunk layout is:

```
White Point x: 4 bytes
White Point y: 4 bytes
Red x:         4 bytes
Red y:         4 bytes
Green x:       4 bytes
Green y:       4 bytes
Blue x:        4 bytes
Blue y:        4 bytes
```

Each value is encoded as a 4−byte unsigned integer, representing the *x* or *y* value times 100000.

If the `cHRM` chunk appears, it must precede the first `IDAT` chunk, and it must also precede the `PLTE` chunk if present.

`gAMA` Gamma Correction

The gamma correction chunk specifies the gamma of the camera (or simulated camera) that produced the image, and thus the gamma of the image with respect to the original scene. Note that this is *not* the same as the gamma of the display device that will reproduce the image correctly.

The chunk's contents are:

```
Image gamma value: 4 bytes
```

A value of 100000 represents a gamma of 1.0, a value of 45000 a gamma of 0.45, and so on (divide by 100000.0). Values around 1.0 and around 0.45 are common in practice.

If the encoder does not know the gamma value, it should not write a gamma chunk; the absence of a gamma chunk indicates the gamma is unknown.

If the `gAMA` chunk appears, it must precede the first `IDAT` chunk, and it must also precede the `PLTE` chunk if present.

See Gamma correction, Recommendations for Encoders: Encoder gamma handling, and Recommendations for Decoders: Decoder gamma handling.

`hIST` Image Histogram

The histogram chunk gives the approximate usage frequency of each color in the color palette. A histogram chunk may appear only when a palette chunk appears. If a viewer is unable to provide all the colors listed in the palette, the histogram may help it decide how to choose a subset of the colors for display.

This chunk's contents are a series of 2−byte (16 bit) unsigned integers. There must be exactly one entry for each entry in the `PLTE` chunk. Each entry is proportional to the fraction of pixels in the image that have that palette index; the exact scale factor is chosen by the encoder.

Histogram entries are approximate, with the exception that a zero entry specifies that the corresponding palette entry is not used at all in the image. It is required that a histogram entry be nonzero if there are any pixels of that color.

When the palette is a suggested quantization of a truecolor image, the histogram is necessarily approximate, since a decoder may map pixels to palette entries differently than the encoder did. In this situation, zero entries should not appear.

The `hIST` chunk, if it appears, must follow the `PLTE` chunk, and must precede the first `IDAT` chunk.

See Rationale: Palette histograms, and Recommendations for Decoders: Palette histogram usage.

`pHYs` Physical Pixel Dimensions

This chunk specifies the intended resolution for display of the image. The chunk's contents are:

```
4 bytes: pixels per unit, X axis (unsigned integer)
4 bytes: pixels per unit, Y axis (unsigned integer)
1 byte: unit specifier
```

The following values are legal for the unit specifier:

```
0: unit is unknown (pHYs defines pixel aspect ratio only)
1: unit is the meter
```

Conversion note: one inch is equal to exactly 0.0254 meters.

If this ancillary chunk is not present, pixels are assumed to be square, and the physical size of each pixel is unknown.

If present, this chunk must precede the first `IDAT` chunk.

See Recommendations for Decoders: Pixel dimensions.

`sBIT` Significant Bits

To simplify decoders, PNG specifies that only certain bit depth values be used, and further specifies that pixel values must be scaled to the full range of possible values at that bit depth. However, the `sBIT` chunk is provided in order to store the original number of significant bits, since this information may be of use to some decoders. We recommend that an encoder emit an `sBIT` chunk if it has converted the data from a different bit depth.

For color type 0 (grayscale), the `sBIT` chunk contains a single byte, indicating the number of bits which were significant in the source data.

For color type 2 (RGB truecolor), the `sBIT` chunk contains three bytes, indicating the number of bits which were significant in the source data for the red, green, and blue channels, respectively.

For color type 3 (palette color), the `sBIT` chunk contains three bytes, indicating the number of bits which were significant in the source data for the red, green, and blue components of the palette entries, respectively.

For color type 4 (grayscale with alpha channel), the `sBIT` chunk contains two bytes, indicating the number of bits which were significant in the source grayscale data and the source alpha channel data, respectively.

For color type 6 (RGB truecolor with alpha channel), the `sBIT` chunk contains four bytes, indicating the number of bits which were significant in the source data for the red, green, blue and alpha channels, respectively.

Note that `sBIT` does *not* have any implications for the interpretation of the stored image: the bit depth indicated by `IHDR` is the correct depth. `sBIT` is only an indication of the history of the image. However, an `sBIT` chunk showing a bit depth less than the `IHDR` bit depth does mean that not all possible color values occur in the image; this fact may be of

use to some decoders.

If the `sBIT` chunk appears, it must precede the first `IDAT` chunk, and it must also precede the `PLTE` chunk if present.

`tEXt` Textual Data

Any textual information that the encoder wishes to record with the image is stored in `tEXt` chunks. Each `tEXt` chunk contains a keyword and a text string, in the format:

```
Keyword:        n bytes (character string)
Null separator: 1 byte
Text:           n bytes (character string)
```

The keyword and text string are separated by a zero byte (null character). Neither the keyword nor the text string may contain a null character. Note that the text string is *not* null−terminated (the length of the chunk is sufficient information to locate the ending). The keyword must be at least one character and less than 80 characters long. The text string may be of any length from zero bytes up to the maximum permissible chunk size.

Any number of `tEXt` chunks may appear, and more than one with the same keyword is permissible.

The keyword indicates the type of information represented by the text string. The following keywords are predefined and should be used where appropriate:

```
Title            Short (one line) title or caption for image
Author           Name of image's creator
Copyright        Copyright notice
Description      Description of image (possibly long)
Software         Software used to create the image
Disclaimer       Legal disclaimer
Warning          Warning of nature of content
Source           Device used to create the image
Comment          Miscellaneous comment; conversion from GIF comment
```

Other keywords, containing any sequence of printable characters in the character set, may be invented for other purposes. Keywords of general interest may be registered with the maintainers of the PNG specification.

Keywords must be spelled exactly as registered, so that decoders may use simple literal comparisons when looking for particular keywords. In particular, keywords are considered case−sensitive.

Both keyword and text are interpreted according to the ISO 8859−1 (Latin−1) character set. Newlines in the text string should be represented by a single linefeed character (decimal 10); use of other ASCII control characters is discouraged.

See Recommendations for Encoders: Text chunk processing and Recommendations for Decoders: Text chunk processing.

`tIME` Image Last−Modification Time

This chunk gives the time of the last image modification (*not* the time of initial image creation). The chunk contents are:

```
2 bytes: Year (complete; for example, 1995, not 95)
1 byte: Month (1−12)
1 byte: Day (1−31)
1 byte: Hour (0−23)
1 byte: Minute (0−59)
1 byte: Second (0−60)    (yes, 60, for leap seconds; not 61, a common error)
```

Universal Time (UTC, also called GMT) should be specified rather than local time.

`tRNS` Transparency

Transparency is an alternative to the full alpha channel. Although transparency is not as elegant as the full alpha channel, it requires less storage space and is sufficient for many common cases.

For color type 3 (palette), this chunk's contents are a series of alpha channel bytes, corresponding to entries in the `PLTE` chunk:

```
Alpha for palette index 0:  1 byte
Alpha for palette index 1:  1 byte
etc.
```

Each entry indicates that pixels of that palette index should be treated as having the specified alpha value. Alpha values have the same interpretation as in an 8−bit full alpha channel: 0 is fully transparent, 255 is fully opaque, regardless of image bit depth. *The* `tRNS` *chunk may contain fewer alpha channel bytes than there are palette entries.* In this case, the alpha channel value for all remaining palette entries is assumed to be 255. In the common case where only palette index 0 need be made transparent, only a one−byte `tRNS` chunk is needed. The `tRNS` chunk may not contain more bytes than there are palette entries.

For color type 0 (grayscale), the `tRNS` chunk contains a single gray level value, stored in the format

```
gray:  2 bytes, range 0 .. (2^bitdepth) − 1
```

(For consistency, 2 bytes are used regardless of the image bit depth.) Pixels of the specified gray level are to be treated as transparent (equivalent to alpha value 0); all other pixels are to be treated as fully opaque (alpha value (2^bitdepth)−1).

For color type 2 (RGB), the `tRNS` chunk contains a single RGB color value, stored in the format

```
red:   2 bytes, range 0 .. (2^bitdepth) − 1
green: 2 bytes, range 0 .. (2^bitdepth) − 1
blue:  2 bytes, range 0 .. (2^bitdepth) − 1
```

(For consistency, 2 bytes per sample are used regardless of the image bit depth.) Pixels of the specified color value are to be treated as transparent (equivalent to alpha value 0); all other pixels are to be treated as fully opaque (alpha value (2^bitdepth)−1).

`tRNS` is prohibited for color types 4 and 6, since a full alpha channel is already present in those cases.

Note: when dealing with 16−bit grayscale or RGB data, it is important to compare both bytes of the sample values to determine whether a pixel is transparent. Although decoders may drop the low−order byte of the samples for display, this must not occur until after the data has been tested for transparency. For example, if the grayscale level 0x0001 is specified to be transparent, it would be incorrect to compare only the high−order byte and decide that 0x0002 is also transparent.

When present, the `tRNS` chunk must precede the first `IDAT` chunk, and must follow the `PLTE` chunk, if any.

`zTXt` Compressed Textual Data

A `zTXt` chunk contains textual data, just as `tEXt` does; however, `zTXt` takes advantage of compression.

A `zTXt` chunk begins with an uncompressed Latin−1 keyword followed by a null (0) character, just as in the `tEXt` chunk. The next byte after the null contains a compression type byte, for which the only presently legitimate value is zero (deflate/inflate compression). The compression−type byte is followed by a compressed data stream which makes up the remainder of the chunk. Decompression of this data stream yields Latin−1 text which is equivalent to the text stored in a `tEXt` chunk.

Any number of `zTXt` and `tEXt` chunks may appear in the same file. See the preceding definition of the `tEXt` chunk for the predefined keywords and the exact format of the text.

See Deflate/Inflate Compression, Recommendations for Encoders: Text chunk processing, and Recommendations for Decoders: Text chunk processing.

Summary of Standard Chunks

This table summarizes some properties of the standard chunk types.

```
Critical chunks (must appear in this order, except PLTE is optional):

        Name  Multiple  Ordering constraints
                OK?

        IHDR    No      Must be first
        PLTE    No      Before IDAT
        IDAT    Yes     Multiple IDATs must be consecutive
        IEND    No      Must be last

Ancillary chunks (need not appear in this order):

        Name  Multiple  Ordering constraints
                OK?

        cHRM    No      Before PLTE and IDAT
        gAMA    No      Before PLTE and IDAT
        sBIT    No      Before PLTE and IDAT
        bKGD    No      After PLTE; before IDAT
        hIST    No      After PLTE; before IDAT
        tRNS    No      After PLTE; before IDAT
        pHYs    No      Before IDAT
        tIME    No      None
        tEXt    Yes     None
        zTXt    Yes     None
```

Standard keywords for `tEXt` and `zTXt` chunks:

```
Title            Short (one line) title or caption for image
Author           Name of image's creator
Copyright        Copyright notice
Description      Description of image (possibly long)
Software         Software used to create the image
Disclaimer       Legal disclaimer
Warning          Warning of nature of content
Source           Device used to create the image
Comment          Miscellaneous comment; conversion from GIF comment
```

Additional Chunk Types

Additional public PNG chunk types are defined in the document "PNG Special−Purpose Public Chunks", available by FTP from `ftp.uu.net:/graphics/png/` or via WWW from `http://sunsite.unc.edu/boutell/pngextensions.html`.

Chunks described there are expected to be somewhat less widely supported than those defined in this specification. However, application authors are encouraged to use those chunk types whenever appropriate for their applications. Additional chunk types may be proposed for inclusion in that

list by contacting the PNG specification maintainers at png−info@uunet.uu.net.

Decoders must treat unrecognized chunk types as described under Chunk naming conventions.

5. Deflate/Inflate Compression

PNG compression type 0 (the only compression method presently defined for PNG) specifies deflate/inflate compression with a 32K window. Deflate compression is an LZ77 derivative used in zip, gzip, pkzip and related programs. Extensive research has been done supporting its patent−free status. Portable C implementations are freely available.

Documentation and C code for deflate are available from the Info−Zip archives at ftp.uu.net:/pub/archiving/zip/.

Deflate−compressed datastreams within PNG are stored in the "zlib" format, which has the structure:

```
Compression method/flags code: 1 byte
Additional flags/check bits:   1 byte
Compressed data blocks:        n bytes
Checksum:                      4 bytes
```

Further details on this format may be found in the zlib specification. At this writing, the zlib specification is at draft 3.1, and is available from ftp.uu.net:/pub/archiving/zip/doc/zlib−3.1.doc.

For PNG compression type 0, the zlib compression method/flags code must specify method code 8 ("deflate" compression) and an LZ77 window size of not more than 32K.

The checksum stored at the end of the zlib datastream is calculated on the uncompressed data represented by the datastream. Note that the algorithm used is not the same as the CRC calculation used for PNG chunk checksums. Verifying the chunk CRCs provides adequate confidence that the PNG file has been transmitted undamaged. The zlib checksum is useful mainly as a crosscheck that the deflate and inflate algorithms are implemented correctly.

The compressed data within the zlib datastream is stored as a series of blocks, each of which can represent raw (uncompressed) data, LZ77−compressed data encoded with fixed Huffman codes, or LZ77−compressed data encoded with custom Huffman codes. A marker bit in the final block identifies it as the last block, allowing the decoder to recognize the end of the compressed datastream. Further details on the compression algorithm and the encoding may be found in the deflate specification. At this writing, the deflate specification is at draft 1.1, and is available from ftp.uu.net:/pub/archiving/zip/doc/deflate−1.1.doc.

In a PNG file, the concatenation of the contents of all the `IDAT` chunks makes up a zlib datastream as specified above. This datastream decompresses to filtered image data as described elsewhere in this document.

It is important to emphasize that the boundaries between `IDAT` chunks are arbitrary and may fall anywhere in the zlib datastream. There is not necessarily any correlation between `IDAT` chunk boundaries and deflate block boundaries or any other feature of the zlib data. For example, it is entirely possible for the terminating zlib checksum to be split across `IDAT` chunks.

In the same vein, there is no required correlation between the structure of the image data (i.e., scanline boundaries) and deflate block boundaries or `IDAT` chunk boundaries. The complete image data is represented by a single zlib datastream that is stored in some number of `IDAT` chunks; a decoder that assumes any more than this is incorrect. (Of course, a particular encoder implementation may happen to emit files in which some of these structures are in fact related. But decoders may not rely on this.)

PNG also uses zlib datastreams in `zTXt` chunks. In a `zTXt` chunk, the remainder of the chunk following the compression type code byte is a zlib datastream as specified above. This datastream decompresses to the user−readable text described by the chunk's keyword. Unlike the image data, such datastreams are not split across chunks; each `zTXt` chunk contains an independent zlib datastream.

6. Filter Algorithms

This chapter describes the pixel filtering algorithms which may be applied in advance of compression. The purpose of these filters is to prepare the image data for optimum compression.

PNG defines five basic filtering algorithms, which are given numeric codes as follows:

Code	Name
0	None
1	Sub
2	Up
3	Average
4	Paeth

The encoder may choose which algorithm to apply on a scanline−by−scanline basis. In the image data sent to the compression step, each scanline is preceded by a filter type byte containing the numeric code of the filter algorithm used for that scanline.

Filtering algorithms are applied to *bytes*, not to pixels, regardless of the bit depth or color type of the image. The filtering algorithms work on the byte sequence formed by a scanline that has been represented as described under Image layout.

When the image is interlaced, each pass of the interlace pattern is treated as an independent image for filtering purposes. The filters work on the byte sequences formed by the pixels actually transmitted during a pass, and the "previous scanline" is the one previously transmitted in the same pass, not the one adjacent in the complete image. Note that the subimage transmitted in any one pass is always rectangular, but is of smaller width and/or height than the complete image. Filtering is not applied when this subimage is empty.

For all filters, the bytes "to the left of" the first pixel in a scanline must be treated as being zero. For filters that refer to the prior scanline, the entire prior scanline must be treated as being zeroes for the first scanline of an image (or of a pass of an interlaced image).

To reverse the effect of a filter, the decoder must use the decoded values of the prior pixel on the same line, the pixel immediately above the current pixel on the prior line, and the pixel just to the left of the pixel above. This implies that at least one scanline's worth of image data must be stored by the decoder at all times. Even though some filter types do not refer to the prior scanline, the decoder must always store each scanline as it is decoded, since the next scanline might use a filter that refers to it.

PNG imposes no restriction on which filter types may be applied to an image. However, the filters are not equally effective on all types of data. See Recommendations for Encoders: Filter selection.

See also Rationale: Filtering.

Filter type 0: None

With the None filter, the scanline is transmitted unmodified; it is only necessary to insert a filter type byte before the data.

Filter type 1: Sub

The Sub filter transmits the difference between each byte and the value of the corresponding byte of the prior pixel.

To compute the Sub filter, apply the following formula to each byte of each scanline:

```
Sub(x) = Raw(x) − Raw(x−bpp)
```

where `x` ranges from zero to the number of bytes representing that scanline minus one, `Raw(x)` refers to the raw data byte at that byte position in the scanline, and `bpp` is defined as the number of bytes per complete pixel, rounding up to one. For example, for color type 2 with a bit depth of 16, bpp is equal to 6 (three channels, two bytes per channel); for color type 0 with a bit depth of 2, bpp is equal to 1 (rounding up); for color type 4 with a bit depth of 16, bpp is equal to 4 (two−byte grayscale value, plus two−byte alpha channel).

Note this computation is done for each **byte**, regardless of bit depth. In a 16−bit image, MSBs are differenced from the preceding MSB and LSBs are differenced from the preceding LSB, because of the way that `bpp` is defined.

Unsigned arithmetic modulo 256 is used, so that both the inputs and outputs fit into bytes. The sequence of `Sub` values is transmitted as the filtered scanline.

For all $x < 0$, assume $\mathrm{Raw}(x) = 0$.

To reverse the effect of the Sub filter after decompression, output the following value:

```
Sub(x) + Raw(x−bpp)
```

(computed mod 256), where `Raw` refers to the bytes already decoded.

Filter type 2: Up

The Up filter is just like the Sub filter except that the pixel immediately above the current pixel, rather than just to its left, is used as the predictor.

To compute the Up filter, apply the following formula to each byte of each scanline:

```
Up(x) = Raw(x) − Prior(x)
```

where `x` ranges from zero to the number of bytes representing that scanline minus one, `Raw(x)` refers to the raw data byte at that byte position in the scanline, and `Prior(x)` refers to the unfiltered bytes of the prior scanline.

Note this is done for each **byte**, regardless of bit depth. Unsigned arithmetic modulo 256 is used, so that both the inputs and outputs fit into bytes. The sequence of `Up` values is transmitted as the filtered scanline.

On the first scanline of an image (or of a pass of an interlaced image), assume $\mathrm{Prior}(x) = 0$ for all x.

To reverse the effect of the Up filter after decompression, output the following value:

```
Up(x) + Prior(x)
```

(computed mod 256), where `Prior` refers to the decoded bytes of the prior scanline.

Filter type 3: Average

The Average filter uses the average of the two neighboring pixels (left and above) to predict the value of a pixel.

To compute the Average filter, apply the following formula to each byte of each scanline:

```
Average(x) = Raw(x) − floor((Raw(x−bpp)+Prior(x))/2)
```

where `x` ranges from zero to the number of bytes representing that scanline minus one, `Raw(x)` refers to the raw data byte at that byte position in the scanline, `Prior(x)` refers to the unfiltered bytes of the prior scanline, and `bpp` is defined as for the Sub filter.

Note this is done for each **byte**, regardless of bit depth. The sequence of `Average` values is transmitted as the filtered scanline.

The subtraction of the predicted value from the raw byte must be done modulo 256, so that both the inputs and outputs fit into bytes. However, the sum `Raw(x−bpp)+Prior(x)` must be formed without overflow (using at least nine−bit arithmetic). `floor()` indicates that the result of the division is rounded to the next lower integer if fractional; in other words, it is an integer division or right shift operation.

For all x < 0, assume Raw(x) = 0. On the first scanline of an image (or of a pass of an interlaced image), assume Prior(x) = 0 for all x.

To reverse the effect of the Average filter after decompression, output the following value:

```
Average(x) + floor((Raw(x−bpp)+Prior(x))/2)
```

where the result is computed mod 256, but the prediction is calculated in the same way as for encoding. `Raw` refers to the bytes already decoded, and `Prior` refers to the decoded bytes of the prior scanline.

Filter type 4: Paeth

The Paeth filter computes a simple linear function of the three neighboring pixels (left, above, upper left), then chooses as predictor the neighboring pixel closest to the computed value. This technique is taken from Alan W. Paeth's article "Image File Compression Made Easy" in Graphics Gems II, James Arvo, editor, Academic Press, 1991.

To compute the Paeth filter, apply the following formula to each byte of each scanline:

```
Paeth(x) = Raw(x) − PaethPredictor(Raw(x−bpp),Prior(x),Prior(x−bpp))
```

where `x` ranges from zero to the number of bytes representing that scanline minus one, `Raw(x)` refers to the raw data byte at that byte position in the scanline, `Prior(x)` refers to the unfiltered bytes of the prior scanline, and `bpp` is defined as for the Sub filter.

Note this is done for each **byte**, regardless of bit depth. Unsigned arithmetic modulo 256 is used, so that both the inputs and outputs fit into bytes. The sequence of `Paeth` values is transmitted as the filtered scanline.

The PaethPredictor function is defined by the following pseudocode:

```
function PaethPredictor (a, b, c)
begin
     ; a = left, b = above, c = upper left
     p := a + b − c        ; initial estimate
     pa := abs(p − a)      ; distances to a, b, c
     pb := abs(p − b)
     pc := abs(p − c)
     ; return nearest of a,b,c,
     ; breaking ties in order a,b,c.
     if pa <= pb AND pa <= pc
     begin
```

```
            return a
        end
        if pb <= pc
        begin
            return b
        end
        return c
    end
```

The calculations within the PaethPredictor function must be performed exactly, without overflow. Arithmetic modulo 256 is to be used only for the final step of subtracting the function result from the target pixel value.

Note that the order in which ties are broken is fixed and must not be altered. The tie break order is: pixel to the left, pixel above, pixel to the upper left. (This order differs from that given in Paeth's article.)

For all x < 0, assume Raw(x) = 0 and Prior(x) = 0. On the first scanline of an image (or of a pass of an interlaced image), assume Prior(x) = 0 for all x.

To reverse the effect of the Paeth filter after decompression, output the following value:

```
Paeth(x) + PaethPredictor(Raw(x-bpp),Prior(x),Prior(x-bpp))
```

(computed mod 256), where `Raw` and `Prior` refer to bytes already decoded. Exactly the same PaethPredictor function is used by both encoder and decoder.

7. Chunk Ordering Rules

To allow new chunk types to be added to PNG, it is necessary to establish rules about the ordering requirements for all chunk types. Otherwise a PNG editing program cannot know what to do when it encounters an unknown chunk.

We define a "PNG editor" as a program that modifies a PNG file, but wishes to preserve as much as possible of the ancillary information in the file. Examples of PNG editors are a program that adds or modifies text chunks, and a program that adds a suggested palette to a 24-bit RGB PNG file. Ordinary image editors are not PNG editors in this sense, because they usually discard any unrecognized information while reading in an image. (Note: we strongly encourage programs handling PNG files to preserve ancillary information whenever possible.)

As an example of possible problems, consider a hypothetical new ancillary chunk type that is safe-to-copy and is required to appear after `PLTE` if `PLTE` is present (presumably `PLTE` affects the meaning of the chunk in some way). If our program to add a suggested `PLTE` to a file does not recognize this new chunk, it may insert `PLTE` in the wrong place, namely after the new chunk. We could prevent such problems by requiring PNG editors to discard any unknown chunks, but that is a very unattractive solution.

To prevent this type of problem while allowing for future extension, we put some constraints on both the behavior of PNG editors and the allowed ordering requirements for chunks. The rules for PNG editors are:

1. When copying an unknown unsafe-to-copy ancillary chunk, a PNG editor may not move the chunk relative to any critical chunk. It may relocate the chunk freely relative to other ancillary chunks that occur between the same pair of critical chunks. (This is well defined since the editor may not add, delete, or reorder critical chunks if it is preserving unsafe-to-copy chunks.)
2. When copying an unknown safe-to-copy ancillary chunk, a PNG editor may not move the chunk from before `IDAT` to after `IDAT` or vice versa. (This is well defined because `IDAT` is always present.) Any other reordering is permitted.

3. When copying a *known* chunk type, an editor need only honor the specific chunk ordering rules that exist for that chunk type. However, it may always choose to apply the above general rules instead.

Therefore, the actual ordering rules for any ancillary chunk type cannot be any stricter than this:

- Unsafe−to−copy chunks may have ordering requirements relative to critical chunks.
- Safe−to−copy chunks may have ordering requirements relative to `IDAT`.

Note that critical chunks can have arbitrary ordering requirements, because PNG editors are required to give up if they encounter unknown critical chunks. A PNG editor must always know the ordering rules for any critical chunk type that it deals with. For example, `IHDR` has the special ordering rule that it must always appear first.

Decoders may not assume more about the positioning of any ancillary chunk than is specified by the chunk ordering rules. In particular, it is never valid to assume that a specific ancillary chunk type occurs with any particular positioning relative to other ancillary chunks. (For example, it is unsafe to assume that your private ancillary chunk occurs immediately before `IEND`. Even if your application always writes it there, a PNG editor might have inserted some other ancillary chunk in between. But you can safely assume that your chunk will remain somewhere between `IDAT` and `IEND`.)

See also Chunk naming conventions.

8. Multi−Image Extension

PNG itself is strictly a single−image format. However, it may be necessary to store multiple images within one file; for example, this is needed to convert some GIF files. For this purpose, a multi−image format will be defined in the near future. PNG decoders will not be required to support the multi−image extension.

9. Recommendations for Encoders

This chapter gives some recommendations for encoder behavior. The only absolute requirement on a PNG encoder is that it produce files which conform to the format specified in the preceding chapters. However, best results will usually be achieved by following these recommendations.

Bitdepth scaling

When scaling input values that have a bit depth that cannot be directly represented in PNG, an excellent approximation to the correct value can be achieved by shifting the valid bits to begin in the most significant bit and repeating the most significant bits into the open bits.

For example, if 5 bits per channel are available in the source data, conversion to a bitdepth of 8 can be achieved as follows:

If the value for a sample in the source data is 27 (in a range from 0−31), then the original bits are:

```
4 3 2 1 0
---------
1 1 0 1 1
```

Converted to a bitdepth of 8, the best value is 222:

```
7 6 5 4 3  2 1 0
----------------
1 1 0 1 1  1 1 0
|=======|  |===|
    |      Leftmost Bits Repeated to Fill Open Bits
    |
```

```
Original Bits
```

Note that this scaling can be reversed simply by shifting right.

Scaling by simply shifting left by three bits is incorrect, since the resulting data would have a range less than the desired full range. (For 5−bit input data, the maximum output would be 248 = 11111000, which is not full brightness.)

It is recommended that the `sBIT` chunk be included when bitdepth scaling has been performed, to record the original data depth.

Encoder gamma handling

If it is possible for the encoder to determine the image gamma, or to make a strong guess based on the hardware on which it runs, then the encoder is strongly encouraged to output the `gAMA` chunk.

A linear brightness level, expressed as a floating−point value in the range 0 to 1, may be converted to a gamma−corrected pixel value by

```
gbright := bright ^ gamma
pixelval := ROUND(gbright * MAXPIXVAL)
```

Computer graphics renderers often do not perform gamma encoding, instead making pixel values directly proportional to scene brightness. This "linear" pixel encoding is equivalent to gamma encoding with a gamma of 1.0, so graphics programs that produce linear pixels should always put out a `gAMA` chunk specifying a gamma of 1.0.

If the encoder knows that the image has been displayed satisfactorily on a display of gamma display_gamma, then the image can be marked as having gamma 1.0/display_gamma.

It is **not** recommended that encoders attempt to convert supplied images to a different gamma. Store the data in the file without conversion, and record the source gamma. Gamma conversion at encode time is a bad idea because gamma adjustment of digital pixel data is inherently lossy, due to roundoff error (8 or so bits is not really enough accuracy). Thus encode−time conversion permanently degrades the image. Worse, if the eventual decoder wants the data with some other gamma, then two conversions occur, each introducing roundoff error. Better to store the data losslessly and incur at most one conversion when the image is finally displayed.

Gamma does not apply to alpha channel values; alpha is always represented linearly.

See Recommendations for Decoders: Decoder gamma handling for more details.

Alpha channel creation

The alpha channel may be regarded either as a mask that temporarily hides transparent parts of the image, or as a means for constructing a non−rectangular image. In the first case, the color values of fully transparent pixels should be preserved for future use. In the second case, the transparent pixels carry no useful data and are simply there to fill out the rectangular image area required by PNG. In this case, fully transparent pixels should all be assigned the same color value for best compression.

Encoders should keep in mind the possibility that a viewer will ignore transparency control. Hence, the colors assigned to transparent pixels should be reasonable background colors whenever feasible.

For applications that do not require a full alpha channel, or cannot afford the price in compression efficiency, the `tRNS` transparency chunk is also available.

If the image has a known background color, this color should be written in the `bKGD` chunk. Even viewers that ignore transparency may use the `bKGD` color to fill unused screen area.

If the original image has premultiplied (also called "associated") alpha data, convert it to PNG's non−premultiplied format by dividing each RGB value by the corresponding alpha value, then multiplying by the maximum value for the image bit depth. In valid premultiplied data, the RGB values never exceed their corresponding alpha values, so the result of the division should always be in the range 0 to 1. If the alpha value is zero, output black (zeroes).

Filter selection

For images of color type 3 (palette−based color), filter type 0 (none) is usually the most effective.

Filter type 0 is also recommended for images of bit depths less than 8. For low−bit−depth grayscale images, it may be a net win to expand the image to 8−bit representation and apply filtering, but this is rare.

For truecolor and grayscale images, any of the five filters may prove the most effective. If an encoder wishes to use a fixed filter choice, the Paeth filter is most likely to be the best.

For best compression of truecolor and grayscale images, we recommend an adaptive filtering approach in which a filter is chosen for each scanline. The following simple heuristic has performed well in early tests: compute the output scanline using all five filters, and select the filter which gives the smallest sum of absolute values of outputs. (Consider the output bytes as signed differences for this test.) This method usually outperforms any single fixed filter choice. However, it is likely that much better heuristics will be found as more experience is gained with PNG.

Filtering according to these recommendations is effective on interlaced as well as noninterlaced images.

Text chunk processing

Note that a nonempty keyword *must* be provided for each text chunk. The generic keyword "Comment" may be used if no better description of the text is available.

Encoders should discourage the creation of single lines of text longer than 79 characters, in order to facilitate easy reading.

If an encoder chooses to support output of `zTXt` compressed text chunks, it is recommended that text less than 1K (1024 bytes) in size be output using uncompressed `tEXt` chunks. In particular, it is recommended that the basic title and author keywords always be output using uncompressed `tEXt` chunks. Lengthy disclaimers, on the other hand, are an ideal candidate for `zTXt`.

Placing large `tEXt` and `zTXt` chunks after the image data (after `IDAT`) may speed up image display in some situations, since the decoder won't have to read over the text to get to the image data. But it is recommended that small text chunks, such as the image title, appear before `IDAT`.

Registering proprietary chunks

If you want others outside your organization to understand a chunk type that you invent, contact the maintainers of the PNG specification to submit a proposed chunk name and definition for addition to the list of special−purpose public chunks (see Additional Chunk Types).

New public chunks will be only be registered if they are of use to others and do not violate the design philosophy of PNG. Chunk registration is not automatic, although it is the intent of the authors that it be straightforward when a new chunk of potentially wide application is needed. Note that the creation of new critical chunk types is discouraged unless absolutely necessary.

If you do not desire that others outside your organization understand the chunk type, you may use a private chunk name by specifying a lowercase letter for the second character. Such chunk types need not be registered. But note that others may use the same private chunk name, so it is prudent to store additional identifying information at the beginning of the chunk data.

Please note that if you want to use a private chunk for information that is not essential to view the image, and have any desire whatsoever that others not using your own viewer software be able to view the image, you should use an ancillary chunk type (first character is lowercase) rather than a critical chunk type (first character uppercase).

If an ancillary chunk is to contain textual information that might be of interest to a human user, it is recommended that a special chunk type **not** be used. Instead use a `tEXt` chunk and define a suitable keyword. In this way, the information will be available to users not using your software.

If of general usefulness, new keywords for `tEXt` chunks may be registered with the maintainers of the PNG specification. Keywords should be chosen to be reasonably self−explanatory, since the idea is to let other users figure out what the chunk contains.

10. Recommendations for Decoders

This chapter gives some recommendations for decoder behavior. The only absolute requirement on a PNG decoder is that it successfully read any file conforming to the format specified in the preceding chapters. However, best results will usually be achieved by following these recommendations.

Chunk error checking

Unknown chunk types must be handled as described under Chunk naming conventions.

It is strongly recommended that decoders verify the CRC on each chunk.

For known−length chunks such as `IHDR`, decoders should treat an unexpected chunk length as an error. Future extensions to this specification will not add new fields to existing chunks; instead, new chunk types will be added to carry any new information.

Unexpected values in fields of known chunks (for example, an unexpected compression type in the `IHDR` chunk) should be checked for and treated as errors.

Pixel dimensions

Non−square pixels can be represented (see the `pHYs` chunk), but viewers are not required to account for them; a viewer may present any PNG file as though its pixels are square.

Conversely, viewers running on display hardware with non−square pixels are strongly encouraged to rescale images for proper display.

Truecolor image handling

To achieve PNG’s goal of universal interchangeability, decoders are required to accept all types of PNG image: palette, truecolor, and grayscale. Viewers running on palette−mapped display hardware need to be able to reduce truecolor images to palette form for viewing. This process is usually called "color quantization".

A simple, fast way of doing this is to reduce the image to a fixed palette. Palettes with uniform color spacing ("color cubes") are usually used to minimize the per−pixel computation. For photograph−like images, dithering is recommended to avoid ugly contours in what should be smooth gradients; however, dithering introduces graininess which may be objectionable.

The quality of rendering can be improved substantially by using a palette chosen specifically for the image, since a color cube usually has numerous entries that are unused in any particular image. This approach requires more work, first in choosing the palette, and second in mapping individual pixels to the closest available color. PNG allows the encoder to supply a suggested palette in a `PLTE` chunk, but not all encoders will do so, and the suggested palette may be unsuitable in any case (it may have too many or too few colors). High−quality viewers will therefore need to have a palette selection routine at hand. A large lookup table is usually the most feasible way of mapping individual pixels to palette entries with adequate speed.

Numerous implementations of color quantization are available. The PNG reference implementation will include code for the purpose.

Decoder gamma handling

To produce correct tone reproduction, a good image display program must take into account the gammas of both the image file and the display device. This can be done by calculating

```
gbright := pixelval / MAXPIXVAL
bright := gbright ^ (1.0 / file_gamma)
gcvideo := bright ^ (1.0 / display_gamma)
fbval := ROUND(gcvideo * MAXFBVAL)
```

where `MAXPIXVAL` is the maximum pixel value in the file (255 for 8−bit, 65535 for 16−bit, etc), `MAXFBVAL` is the maximum value of a frame buffer pixel (255 for 8−bit, 31 for 5−bit, etc), `pixelval` is the value of the pixel in the PNG file, and `fbval` is the value to write into the frame buffer. The first line converts from pixel code into a normalized 0 to 1 floating point value, the second undoes the encoding of the image file to produce a linear brightness value, the third line pre−corrects for the monitor's gamma response, and the fourth converts to an integer frame buffer pixel. In practice the second and third lines can be merged into

```
gcvideo := gbright ^ (1.0 / (file_gamma * display_gamma))
```

so as to perform only one power calculation. For color images, the entire calculation is performed separately for R, G, and B values.

It is *not* necessary to perform transcendental math for every pixel! Instead, compute a lookup table that gives the correct output value for every pixel value. This requires only 256 calculations per image (for 8−bit accuracy), not one calculation per pixel. For palette−based images, a one−time correction of the palette is sufficient.

In some cases even computing a gamma lookup table may be a concern. In these cases, viewers are encouraged to have precomputed gamma correction tables for file_gamma values of 1.0 and 0.45 and some reasonable single display_gamma value, and to use the table closest to the gamma indicated in the file. This will produce acceptable results for the majority of real files.

When the incoming image has unknown gamma (no `gAMA` chunk), choose a likely default file_gamma value, but allow the user to select a new one if the result proves too dark or too light.

In practice, it is often difficult to determine what value of display_gamma should be used. In systems with no built−in gamma correction, the display_gamma is determined entirely by the CRT. Assuming a value of 2.2 is recommended, unless you have detailed calibration measurements of this particular CRT available.

However, many modern frame buffers have lookup tables that are used to perform gamma correction, and on these systems the display_gamma value should be the gamma of the lookup table and CRT combined. You may not be able to find out what the lookup table contains from within an image viewer application, so you may have to ask the user what the system's gamma value is. Unfortunately, different manufacturers use different ways of specifying what should go

into the lookup table, so interpretation of the system gamma value is system−dependent.

Here are examples of how to deal with some known systems:

- On many Macintosh systems, there is a "gamma" control panel that lets you select one of a small set of gamma values (1.0, 1.4, 1.8, 2.2). These numbers are the combined gamma of the hardware table and the CRT, and so they are exactly the value that the decoder needs to use as display_gamma. With the "gamma" control panel turned off, or not present at all, the default Macintosh system gamma is 1.8.
- On recent SGI systems, there is a hardware gamma−correction table whose contents are controlled by the (privileged) "gamma" program. The gamma of the table is actually the reciprocal of the number that "gamma" prints, and does not include the CRT gamma. To obtain the display_gamma, you need to find the SGI system gamma (either by looking in a file, or asking the user) and then calculating

  ```
  display_gamma = 2.2 / SGI_system_gamma
  ```

  You will find SGI systems with the system gamma set to 1.0 and 2.2 (or higher), but the default when machines are shipped is 1.7.
- On frame buffers that have hardware gamma correction tables, and which are calibrated to display linear pixels correctly, display_gamma is 1.0.
- Many workstations and Xterms and PC displays lack gamma correction hardware. Here, assume that display_gamma is 2.2.

We should point out that there is a fudge factor built into the use of the magic value "2.2" as the assumed CRT gamma in the calculations above. Real CRTs usually have a higher gamma than this, around 2.8 in fact. By doing the display gamma correction for a CRT gamma of only 2.2, we get an image on screen that is slightly higher in contrast than the original scene. This is normal TV and film practice, and we are continuing it here. Generally, writers of display programs are best to assume that CRT gamma is 2.2 rather than using actual measurements.

If you have carefully measured the gamma of your CRT, you might want to set display_gamma to your_CRT_gamma/1.25, in order to preserve this intentional contrast boost.

Finally, note that the response of real displays is actually more complex than can be described by a single number (display_gamma). If actual measurements of the monitor’s light output as a function of voltage input are available, the third and fourth lines of the computation above may be replaced by a lookup in these measurements, to find the actual frame buffer value that most nearly gives the desired brightness.

Background color

Viewers which have a specific background against which to present the image will ignore the `bKGD` chunk, but viewers with no preset background color may choose to honor it. The background color will typically be used to fill unused screen space around the image, as well as any transparent pixels within the image. (Thus, `bKGD` is valid and useful even when the image does not use transparency.) If no `bKGD` chunk is present, the viewer must make its own decision about a suitable background color.

Alpha channel processing

In the most general case, the alpha channel can be used to composite a foreground image against a background image; the PNG file defines the foreground image and the transparency mask, but not the background image. Decoders are *not* required to support this most general case. It is expected that most will be able to support compositing against a single background color, however.

The equation for computing a composited pixel value is

```
output := alpha * foreground + (1−alpha) * background
```

where alpha and the input and output sample values are expressed as fractions in the range 0 to 1. This computation should be performed with linear (non−gamma−corrected) sample values. For color images, the computation is done separately for R, G, and B samples.

The following code illustrates the general case of compositing a foreground image over a background image. It assumes that you have the original pixel data available for the background image, and that output is to a frame buffer for display. Other variants are possible; see the comments below the code. The code allows the bit depths and gamma values of foreground image, background image, and frame buffer/CRT to all be different. Don't assume they are the same without checking!

There are line numbers for referencing code in the comments below. Other than that, this is standard C.

```
01  int foreground[4];      /* file pixel: R, G, B, A */
02  int background[3];      /* file background color: R, G, B */
03  int fbpix[3];           /* frame buffer pixel */
04  int fg_maxpixval;       /* foreground max pixel */
05  int bg_maxpixval;       /* background max pixel */
06  int fb_maxpixval;       /* frame buffer max pixel */
07  int ialpha;
08  float alpha, compalpha;
09  float gamfg, linfg, gambg, linbg, comppix, gcvideo;

    /* Get max pixel value in files and frame buffer */
10  fg_maxpixval = (1 << fg_bit_depth) − 1;
11  bg_maxpixval = (1 << bg_bit_depth) − 1;
12  fb_maxpixval = (1 << frame_buffer_bit_depth) − 1;
    /*
     * Get integer version of alpha.
     * Check for opaque and transparent special cases;
     * no compositing needed if so.
     *
     * We show the whole gamma decode/correct process in
     * floating point, but it would more likely be done
     * with lookup tables.
     */
13  ialpha = foreground[3];
14  if (ialpha == 0) {
        /*
         * Foreground image is transparent here.
         * If the background image is already in the frame
         * buffer, there is nothing to do.
         */
15      ;
16  } else if (ialpha == fg_maxpixval) {
17      for (i = 0; i < 3; i++ {
18          gamfg = (float) foreground[i] / fg_maxpixval;
19          linfg = pow(gamfg, 1.0/fg_gamma);
20          comppix = linfg;
21          gcvideo = pow(comppix, 1.0/display_gamma);
22          fbpix[i] = (int) (gcvideo * fb_maxpixval + 0.5);
23      }
24  } else {
        /*
         * Compositing is necessary.
         * Get floating−point alpha and its complement.
         * Note: alpha is always linear; gamma does not
         * affect it.
         */
25      alpha = (float) ialpha / fg_maxpixval;
26      compalpha = 1.0 − alpha;

27      for (i = 0; i < 3; i++ {
            /*
             * Convert foreground and background to floating point,
```

```
             * then linearize (undo gamma encoding).
             */
28          gamfg = (float) foreground[i] / fg_maxpixval;
29          linfg = pow(gamfg, 1.0/fg_gamma);
30          gambg = (float) background[i] / bg_maxpixval;
31          linbg = pow(gambg, 1.0/bg_gamma);
            /*
             * Composite.
             */
32          comppix = linfg * alpha + linbg * compalpha;
            /*
             * Gamma correct for display.
             * Convert to integer frame buffer pixel.
             */
33          gcvideo = pow(comppix, 1.0/display_gamma);
34          fbpix[i] = (int) (gcvideo * fb_maxpixval + 0.5);
35      }
36  }
```

Variations:

1. If output is to another PNG image file instead of a frame buffer, lines 21, 22, 33, and 34 should be changed to be something like:

```
            /*
             * Gamma encode for storage in output file.
             * Convert to integer pixel value.
             */
            gamout = pow(comppix, outfile_gamma);
            outpix[i] = (int) (gamout * out_maxpixval + 0.5);
```

   Also, it becomes necessary to process background pixels when alpha is zero, rather than just skipping pixels. Thus, line 15 must be replaced by copies of lines 18−22, but processing background instead of foreground pixel values.
2. If the bit depth of the output file, foreground file, and background file are all the same, and the three gamma values also match, then the no−compositing code in lines 14−23 reduces to nothing more than copying pixel values from the input file to the output file if alpha is one, or copying pixel values from background to output file if alpha is zero. Since alpha is typically either zero or one for the vast majority of pixels in an image, this is a great savings. No gamma computations are needed for most pixels.
3. When the bit depths and gamma values all match, it may appear attractive to skip the gamma decorrection and correction (lines 28−31, 33−34) and just perform line 32 using gamma−encoded sample values. Although this doesn't hurt image quality too badly, the time savings are small if alpha values of zero and one are special−cased as recommended here.
4. If the original pixel values of the background image are no longer available, only processed frame buffer pixels left by display of the background image, then lines 30 and 31 must extract intensity from the frame buffer pixel values using code like:

```
            /*
             * Decode frame buffer value back into linear space.
             */
            gcvideo = (float) (fbpix[i] / fb_maxpixval);
            linbg = pow(gcvideo, display_gamma);
```

   However, some roundoff error can result, so it is better to have the original background pixels available if at all possible.
5. Note that lines 18−22 are performing exactly the same gamma computation that is done when no alpha channel is present. So, if you handle the no−alpha case with a lookup table, you can use the same lookup table here. Lines 28−31 and 33−34 can also be done with lookup tables.
6. Of course, everything here can be done in integer arithmetic. Just be careful to maintain sufficient precision all the way through.

Note: in floating point, no overflow or underflow checks are needed, because the input pixel values are guaranteed to be between 0 and 1, and compositing always yields a result that is in between the input values (inclusive). With integer arithmetic, some roundoff−error analysis might be needed to guarantee no overflow or underflow.

When displaying a PNG image with full alpha channel, it is important to be able to composite the image against some background, even if it's only black. Ignoring the alpha channel will cause PNG images that have been converted from an associated−alpha representation to look wrong. (Of course, if the alpha channel is a separate transparency mask, then ignoring alpha is a useful option: it allows the hidden parts of the image to be recovered.)

When dealing with PNG images that have a `tRNS` chunk, it is reasonable to assume that the transparency information is a mask rather than associated−alpha coverage data. In this case, it is an acceptable shortcut to interpret all nonzero alpha values as fully opaque (no background). This approach is simple to implement: transparent pixels are replaced by the background color, others are unchanged. A viewer with no particular background color preference may even choose to ignore the `tRNS` chunk; but if a `bKGD` chunk is provided, it is better to use the specified background color.

Progressive display

When receiving images over slow transmission links, decoders can improve perceived performance by displaying interlaced images progressively. This means that as each pass is received, an approximation to the complete image is displayed based on the data received so far. One simple yet pleasing effect can be obtained by expanding each received pixel to fill a rectangle covering the yet−to−be−transmitted pixel positions below and to the right of the received pixel. This process can be described by the following pseudocode:

```
Starting_Row [1..7] =  { 0, 0, 4, 0, 2, 0, 1 }
Starting_Col [1..7] =  { 0, 4, 0, 2, 0, 1, 0 }
Row_Increment [1..7] = { 8, 8, 8, 4, 4, 2, 2 }
Col_Increment [1..7] = { 8, 8, 4, 4, 2, 2, 1 }
Block_Height [1..7] =  { 8, 8, 4, 4, 2, 2, 1 }
Block_Width [1..7] =   { 8, 4, 4, 2, 2, 1, 1 }

pass := 1
while pass <= 7
begin
    row := Starting_Row[pass]

    while row < height
    begin
        col := Starting_Col[pass]

        while col < width
        begin
            visit (row, col,
                   min (Block_Height[pass], height − row),
                   min (Block_Width[pass], width − col))
            col := col + Col_Increment[pass]
        end
        row := row + Row_Increment[pass]
    end

    pass := pass + 1
end
```

Here, the function "visit(row,column,height,width)" obtains the next transmitted pixel and paints a rectangle of the specified height and width, whose upper−left corner is at the specified row and column, using the color indicated by the pixel. Note that row and column are measured from 0,0 at the upper left corner.

If the decoder is merging the received image with a background image, it may be more convenient just to paint the received pixel positions; that is, the "visit()" function sets only the pixel at the specified row and column, not the whole rectangle. This produces a "fade−in" effect as the new image gradually replaces the old. An advantage of this approach is that proper alpha or transparency processing can be done as each pixel is replaced. Painting a rectangle as described above will overwrite background−image pixels that may be needed later, if the pixels eventually received for those positions turn out to be wholly or partially transparent. Of course, this is only a problem if the background image is not stored anywhere offscreen.

Palette histogram usage

If the viewer is only short a few colors, it is usually adequate to drop the least−used colors from the palette. To reduce the number of colors substantially, it's best to choose entirely new representative colors, rather than trying to use a subset of the existing palette. This amounts to performing a new color quantization step; however, the existing palette and histogram can be used as the input data, thus avoiding a scan of the image data.

If no histogram chunk is provided, a decoder can of course develop its own, at the cost of an extra pass over the image data.

Text chunk processing

If practical, decoders should have a way to display to the user all `tEXt` and `zTXt` chunks found in the file. Even if the decoder does not recognize a particular text keyword, the user may well be able to understand it.

Decoders should be prepared to display text chunks which contain any number of printing characters between newline characters, even though encoders are encouraged to avoid creating lines in excess of 79 characters.

11. Appendix: Rationale

(This appendix is not part of the formal PNG specification.)

This appendix gives the reasoning behind some of the design decisions in PNG. Many of these decisions were the subject of considerable debate. The authors freely admit that another group might have made different decisions; however, we believe that our choices are defensible and consistent.

Why a new file format?

Does the world really need yet another graphics format? We believe so. GIF is no longer freely usable, but no other commonly used format can directly replace it, as is discussed in more detail below. We might have used an adaptation of an existing format, for example GIF with an unpatented compression scheme. But this would require new code anyway; it would not be all that much easier to implement than a whole new file format. (PNG is designed to be simple to implement, with the exception of the compression engine, which would be needed in any case.) We feel that this is an excellent opportunity to design a new format that fixes some of the known limitations of GIF.

Why these features?

The features chosen for PNG are intended to address the needs of applications that previously used the special strengths of GIF. In particular, GIF is well adapted for on−line communications because of its streamability and progressive display capability. PNG shares those attributes.

We have also addressed some of the widely known shortcomings of GIF. In particular, PNG

supports truecolor images. We know of no widely used image format that losslessly compresses truecolor images as effectively as PNG does. We hope that PNG will make use of truecolor images more practical and widespread.

Some form of transparency control is desirable for applications in which images are displayed against a background or together with other images. GIF provided a simple transparent−color specification for this purpose. PNG supports a full alpha channel as well as transparent−color specifications. This allows both highly flexible transparency and compression efficiency.

Robustness against transmission errors has been an important consideration. For example, images transferred across Internet are often mistakenly processed as text, leading to file corruption. PNG is designed so that such errors can be detected quickly and reliably.

PNG has been expressly designed not to be completely dependent on a single compression technique. Although inflate/deflate compression is mentioned in this document, PNG would still exist without it.

Why not these features?

Some features have been deliberately omitted from PNG. These choices were made to simplify implementation of PNG, promote portability and interchangeability, and make the format as simple and foolproof as possible for users. In particular:

- There is no uncompressed variant of PNG. It is possible to store uncompressed data by using only uncompressed deflate blocks (a feature normally used to guarantee that deflate does not make incompressible data much larger). However, any software that does not support full deflate/inflate will not be considered compliant with the PNG standard. The two most important features of PNG−−−portability and compression−−−are absolute requirements for online applications, and users demand them. Failure to support full deflate/inflate compromises both of these objectives.
- There is no lossy compression in PNG. Existing formats such as JFIF already handle lossy compression well. Furthermore, available lossy compression methods (e.g., JPEG) are far from foolproof to use −−− a poor choice of quality level can ruin an image. To avoid user confusion and unintentional loss of information, we feel it is best to keep lossy and lossless formats strictly separate. Also, lossy compression is complex to implement. Adding JPEG support to a PNG decoder might increase its size by an order of magnitude. This would certainly cause some decoders to omit support for the feature, which would destroy our goal of interchangeability.
- There is no support for CMYK or other unusual color spaces. Again, this is in the name of promoting portability. CMYK, in particular, is far too device−dependent to be useful as a portable image representation.
- There is no standard chunk for thumbnail views of images. In discussions with software vendors who use thumbnails in their products, it has become clear that most would not use a "standard" thumbnail chunk. This is partly because every vendor has a distinct idea of what the dimensions and characteristics of a thumbnail should be, and partly because vendors who keep thumbnails in separate files to accommodate varied image formats are not going to stop doing that simply because of a thumbnail chunk in one new format. Proprietary chunks containing vendor−specific thumbnails appear to be more practical than a common thumbnail format.

It is worth noting that private extensions to PNG could easily add these features. We will not, however, include them as part of the basic PNG standard.

Basic PNG also does not support multiple images in one file. This restriction is a reflection of the reality that many applications do not need and will not support multiple images per file. (While the GIF standard nominally allows multiple images per file, few applications actually support it.)

In any case, single images are a fundamentally different sort of object from sequences of images. Rather than make false promises of interchangeability, we have drawn a clear distinction between single−image and multi−image formats. PNG is a single−image format.

Why not use format XYZ?

Numerous existing formats were considered before deciding to develop PNG. None could meet the requirements we felt were important for PNG.

GIF is no longer suitable as a universal standard because of legal entanglements. Although just replacing GIF's compression method would avoid that problem, GIF does not support truecolor images, alpha channels, or gamma correction. The spec has more subtle problems too. Only a small subset of the GIF89 spec is actually portable across a variety of implementations, but there is no codification of the most portable part of the spec.

TIFF is far too complex to meet our goals of simplicity and interchangeability. Defining a TIFF subset would meet that objection, but would frustrate users making the reasonable assumption that a file saved as TIFF from Software XYZ would load into a program supporting our flavor of TIFF. Furthermore, TIFF is not designed for stream processing, has no provision for progressive display, and does not currently provide any good, legally unencumbered, lossless compression method.

IFF has also been suggested, but is not suitable in detail: available image representations are too machine−specific or not adequately compressed. The overall chunk structure of IFF is a useful concept which PNG has liberally borrowed from, but we did not attempt to be bit−for−bit compatible with IFF chunk structure. Again this is due to detailed issues, notably the fact that IFF FORMs are not designed to be serially writable.

Lossless JPEG is not suitable because it does not provide for the storage of palette−color images. Furthermore, its lossless truecolor compression is often inferior to that of PNG.

Byte order

It has been asked why PNG uses network byte order. We have selected one byte ordering and used it consistently. Which order in particular is of little relevance, but network byte order has the advantage that routines to convert to and from it are already available on any platform that supports TCP/IP networking, **including** all PC platforms. The functions are trivial and will be included in the reference implementation.

Interlacing

PNG's two−dimensional interlacing scheme is more complex to implement than GIF's line−wise interlacing. It also costs a little more in file size. However, it yields an initial image *eight times* faster than GIF (the first pass transmits only 1/64th of the pixels, compared to 1/8th for GIF). Although this initial image is coarse, it is useful in many situations. For example, if the image is a World Wide Web imagemap that the user has seen before, PNG's first pass is enough to determine where to click. The PNG scheme also looks better than GIF's, because horizontal and vertical resolution never differ by more than a factor of two; this avoids the odd "stretched" look seen when interlaced GIFs are filled in by replicating scanlines.

Why gamma encoding?

Although gamma 1.0 (linear brightness response) might seem a natural standard, it is common for images to have a gamma of less than 1. There are two good reasons for this:

- CRT hardware typically has a gamma between 2 and 3. Hence, "gamma correction" is a standard part of all video signals. The transmitted image usually has a gamma of 0.45 (NTSC) or 0.36 (PAL/SECAM), so images obtained by frame−grabbing video already

have this value of gamma.
- An image gamma less than 1 allocates more of the available pixel codes or voltage range to darker areas of the image. This allows photographic−quality images to be stored in only 24 bits/pixel without banding artifacts in the darker areas (in most cases). This makes "gamma encoding" a much better way of storing digital images than the simpler linear encoding.

In practice, image gamma values around 1.0 and around 0.45 are both widely found. Older image standards such as GIF often do not account for this fact, leading to widespread problems with images coming out "too dark" or "too light".

PNG expects viewers to compensate for image gamma at the time that the image is displayed. Another possible approach is to expect encoders to convert all images to a uniform gamma at encoding time. While that method would speed viewers slightly, it has fundamental flaws:

- Gamma correction is inherently lossy due to roundoff error. Requiring conversion at encoding time thus causes irreversible loss. Since PNG is intended to be a lossless storage format, this is undesirable; we should store unmodified source data.
- The encoder might not know the image gamma. If the decoder does gamma correction at viewing time, it can adjust the gamma (correct the displayed brightness) in response to feedback from a human user. The encoder has no such option.
- Whatever "standard" gamma we settled on would be wrong for some displays. Hence viewers would still need gamma correction capability.

Since there will always be images with no gamma or an incorrect recorded gamma, good viewers will need to incorporate gamma correction logic anyway. Gamma correction at viewing time is thus the right way to go.

Non−premultiplied alpha

PNG uses "unassociated" or "non−premultiplied" alpha so that images with separate transparency masks can be stored losslessly. Another common technique, "premultiplied alpha", stores pixel values premultiplied by the alpha fraction; in effect, the image is already composited against a black background. Any image data hidden by the transparency mask is irretrievably lost by that method, since multiplying by a zero alpha value always produces zero.

Some image rendering techniques generate images with premultiplied alpha (the alpha value actually represents how much of the pixel is covered by the image). This representation can be converted to PNG by dividing the RGB values by alpha, except where alpha is zero. The result will look good if displayed by a viewer that handles alpha properly, but will not look very good if the viewer ignores the alpha channel.

Although each form of alpha storage has its advantages, we did not want to require all PNG viewers to handle both forms. We standardized on non−premultiplied alpha as being the more general case.

Filtering

PNG includes filtering capability because filtering can significantly reduce the compressed size of truecolor and grayscale images. Filtering is also sometimes of value on palette images, although this is less common.

The filter algorithms are defined to operate on bytes, rather than pixels; this gains simplicity and speed with very little cost in compression performance. Tests have shown that filtering is usually ineffective for images with fewer than 8 bits per pixel, so providing pixelwise filtering for such images would be pointless. For 16 bit/pixel data, bytewise filtering is nearly as effective as pixelwise filtering, because MSBs are predicted from adjacent MSBs, and LSBs are predicted from adjacent

LSBs.

The encoder is allowed to change filters for each new scanline. This creates no additional complexity for decoders, since a decoder is required to contain unfiltering logic for every filter type anyway. The only cost is an extra byte per scanline in the pre−compression data stream. Our tests showed that when the same filter is selected for all scanlines, this extra byte compresses away to almost nothing, so there is little storage cost compared to a fixed filter specified for the whole image. And the potential benefits of adaptive filtering are too great to ignore. Even with the simplistic filter−choice heuristics so far discovered, adaptive filtering usually outperforms fixed filters. In particular, an adaptive filter can change behavior for successive passes of an interlaced image; a fixed filter cannot.

The basic filters offered by PNG have been chosen on both theoretical and experimental grounds. In particular, it is worth noting that all the filters (except "none" and "average") operate by encoding the difference between a pixel and one of its neighboring pixels. This is usually superior to conventional linear prediction equations because the prediction is certain to be one of the possible pixel values. When the source data is not full depth (such as 5−bit data scaled up to 8−bit depth), this restriction ensures that the number of prediction delta values is no more than the number of distinct pixel values present in the source data. A linear equation can produce intermediate values not actually present in the source data, and thus reduce compression efficiency.

Text strings

Most graphics file formats include the ability to store some textual information along with the image. But many applications need more than that: they want to be able to store several identifiable pieces of text. For example, a database using PNG files to store medical X−rays would likely want to include patient’s name, doctor’s name, etc. A simple way to do this in PNG would be to invent new proprietary chunks holding text. The disadvantage of such an approach is that other applications would have no idea what was in those chunks, and would simply ignore them. Instead, we recommend that text information be stored in standard `tEXt` chunks with suitable keywords. Use of `tEXt` tells any PNG viewer that the chunk contains text that may be of interest to a human user. Thus, a person looking at the file with another viewer will still be able to see the text, and even understand what it is if the keywords are reasonably self−explanatory. (To this end, we recommend spelled−out keywords, not abbreviations that will be hard for a person to understand. Saving a few bytes on a keyword is false economy.)

The ISO 8859−1 (Latin−1) character set was chosen as a compromise between functionality and portability. Some platforms cannot display anything more than 7−bit ASCII characters, while others can handle characters beyond the Latin−1 set. We felt that Latin−1 represents a widely useful and reasonably portable character set. Latin−1 is a direct subset of character sets commonly used on popular platforms such as Microsoft Windows and X Windows. It can also be handled on Macintosh systems with a simple remapping of characters.

There is at present no provision for text employing character sets other than the Latin−1 character set. It is recognized that the need for other character sets will increase. However, PNG already requires that programmers implement a number of new and unfamiliar features, and text representation is not PNG’s primary purpose. Since PNG provides for the creation and public registration of new ancillary chunks of general interest, it is expected that text chunks for other character sets, such as Unicode, eventually will be registered and increase gradually in popularity.

PNG file signature

The first eight bytes of a PNG file always contain the following values:

```
(decimal)              137  80  78  71  13  10  26  10
(hex)                   89  50  4e  47  0d  0a  1a  0a
```

```
(ASCII C notation)    \211   P   N   G  \r  \n \032 \n
```

This signature both identifies the file as a PNG file and provides for immediate detection of common file−transfer problems. The first two bytes distinguish PNG files on systems that expect the first two bytes to identify the file type uniquely. The first byte is chosen as a non−ASCII value to reduce the probability that a text file may be misrecognized as a PNG file; also, it catches bad file transfers that clear bit 7. Bytes two through four name the format. The CR−LF sequence catches bad file transfers that alter newline sequences. The control−Z character stops file display under MS−DOS. The final line feed checks for the inverse of the CR−LF translation problem.

Additional confidence in correct file transfer can be had by checking that the next eight bytes are an `IHDR` chunk header with the correct chunk length.

Note that there is no version number in the signature, nor indeed anywhere in the file. This is intentional: the chunk mechanism provides a better, more flexible way to handle format extensions, as is described below.

Chunk layout

The chunk design allows decoders to skip unrecognized or uninteresting chunks: it is simply necessary to skip the appropriate number of bytes, as determined from the length field.

Limiting chunk length to (2^31)−1 bytes avoids possible problems for implementations that cannot conveniently handle 4−byte unsigned values. In practice, chunks will usually be much shorter than that anyway.

A separate CRC is provided for each chunk in order to detect badly−transferred images as quickly as possible. In particular, critical data such as the image dimensions can be validated before being used. The chunk length is excluded in order to permit CRC calculation while data is generated (possibly before the length is known, in the case of variable−length chunks); this may avoid an extra pass over the data. Excluding the length from the CRC does not create any extra risk of failing to discover file corruption, since if the length is wrong, the CRC check will fail (the CRC will be computed on the wrong bytes and then tested against the wrong value from the file).

Chunk naming conventions

The chunk naming conventions allow safe, flexible extension of the PNG format. This mechanism is much better than a format version number, because it works on a feature−by−feature basis rather than being an overall indicator. Decoders can process newer files if and only if the files use no unknown critical features (as indicated by finding unknown critical chunks). Unknown ancillary chunks can be safely ignored. Experience has shown that format version numbers hurt portability as much as they help. Version numbers tend to be set unnecessarily high, leading to older decoders rejecting files that they could have processed (this was a serious problem for several years after the GIF89 spec came out, for example). Furthermore, private extensions can be made either critical or ancillary, and standard decoders will react appropriately; overall version numbers are no help for private extensions.

A hypothetical chunk for vector graphics would be a critical chunk, since if ignored, important parts of the intended image would be missing. A chunk carrying the Mandelbrot set coordinates for a fractal image would be ancillary, since other applications could display the image without understanding what it was. In general, a chunk type should be made critical only if it is impossible to display a reasonable representation of the intended image without interpreting that chunk.

The public/private property bit ensures that any newly defined public chunk type name cannot conflict with proprietary chunks that may be in use somewhere. However, this does not protect users of private chunk names from the possibility that someone else may re−use the same chunk name for a different purpose. It is a good idea to put additional identifying information at the start

of the data for any private chunk type.

When a PNG file is modified, certain ancillary chunks may need to be changed to reflect changes in other chunks. For example, a histogram chunk needs to be changed if the image data changes. If the encoder does not recognize histogram chunks, copying them blindly to a new output file is incorrect; such chunks should be dropped. The safe/unsafe property bit allows ancillary chunks to be marked appropriately.

Not all possible modification scenarios are covered by the safe/unsafe semantics. In particular, chunks that are dependent on the total file contents are not supported. (An example of such a chunk is an index of `IDAT` chunk locations within the file: adding a comment chunk would inadvertently break the index.) Definition of such chunks is discouraged. If absolutely necessary for a particular application, such chunks may be made critical chunks, with consequent loss of portability to other applications. In general, ancillary chunks may depend on critical chunks but not on other ancillary chunks. It is expected that mutually dependent information should be put into a single chunk.

In some situations it may be unavoidable to make one ancillary chunk dependent on another. Although the chunk property bits do not allow this case to be represented, a simple solution is available: in the dependent chunk, record the CRC of the chunk depended on. It can then be determined whether that chunk has been changed by some other program.

The same technique may be useful for other purposes. For example, if a program relies on the palette being in a particular order, it may store a private chunk containing the CRC of the `PLTE` chunk. If this value matches when the file is again read in, then it provides high confidence that the palette has not been tampered with. Note that it is not necessary to mark the private chunk unsafe−to−copy when this technique is used; thus, such a private chunk can survive other editing of the file.

Palette histograms

A viewer may not be able to provide as many colors as are listed in the image’s palette. (For example, some colors may be reserved by a window system.) To produce the best results in this situation, it is helpful to have information on the frequency with which each palette index actually appears, in order to choose the best palette for dithering or drop the least−used colors. Since images are often created once and viewed many times, it makes sense to calculate this information in the encoder, although it is not mandatory for the encoder to provide it.

The same rationale holds good for palettes which are suggested quantizations of truecolor images. In this situation, it is recommended that the histogram values represent "nearest neighbor" counts, that is, the approximate usage of each palette entry if no dithering is applied. (These counts will often be available for free as a consequence of developing the suggested palette.)

Other image formats have usually addressed this problem by specifying that the palette entries should appear in order of frequency of use. That is an inferior solution, because it doesn’t give the viewer nearly as much information: the viewer can’t determine how much damage will be done by dropping the last few colors. Nor does a sorted palette give enough information to choose a target palette for dithering, in the case that the viewer must reduce the number of colors substantially. A palette histogram provides the information needed to choose such a target palette without making a pass over the image data.

12. Appendix: Sample CRC Code

The following sample code represents a practical implementation of the CRC (Cyclic Redundancy Check) employed in PNG chunks.

The sample code provided is in the C programming language. (See also ISO 3309 and ITU−T V.42

for a formal specification.)

```
/* table of crc's of all 8-bit messages */
unsigned long crc_table[256];

/* Flag: has the table been computed? Initially false. */
int crc_table_computed = 0;

/* make the table for a fast crc */
void make_crc_table(void)
{
  unsigned long c;
  int n, k;

  for (n = 0; n < 256; n++) {
    c = (unsigned long) n;
    for (k = 0; k < 8; k++)
      c = (c & 1) ? (0xedb88320L ^ (c >> 1)) : (c >> 1);
    crc_table[n] = c;
  }
  crc_table_computed = 1;
}

/* update a running crc with the bytes buf[0..len-1]--the crc should be
   initialized to all 1's, and the transmitted value is the 1's complement
   of the final running crc (see the crc() routine below)). */

unsigned long update_crc(unsigned long crc, unsigned char *buf, int len)
{
  unsigned long c = crc;
  unsigned char *p = buf;
  int n = len;

  if (!crc_table_computed)
    make_crc_table();
  if (n > 0) do {
    c = crc_table[(c ^ (*p++)) & 0xff] ^ (c >> 8);
  } while (--n);
  return c;
}

/* return the crc of the bytes buf[0..len-1] */
unsigned long crc(unsigned char *buf, int len)
{
  if (!crc_table_computed)
    make_crc_table();
  return update_crc(0xffffffffL, buf, len) ^ 0xffffffffL;
}
```

13. Credits

Editor:

Thomas Boutell, boutell@netcom.com

Contributing Editor:

Tom Lane, tgl@sss.pgh.pa.us

Authors:

Authors' names are presented in alphabetical order.

- Mark Adler, madler@cco.caltech.edu
- Thomas Boutell, boutell@netcom.com
- Adam M. Costello, amc@cs.wustl.edu
- Lee Daniel Crocker, lee@piclab.com
- Oliver Fromme, fromme@rz.tu-clausthal.de

- Jean−Loup Gailly, jloup@chorus.fr
- Alex Jakulin, alex@hermes.si
- Neal Kettler, kettler@cs.colostate.edu
- Tom Lane, tgl@sss.pgh.pa.us
- Dave Martindale, davem@cs.ubc.ca
- Owen Mortensen, ojm@csi.compuserve.com
- Glenn Randers−Pehrson, glennrp@arl.mil
- Greg Roelofs, newt@uchicago.edu
- Paul Schmidt, photodex@netcom.com
- Tim Wegner, 71320.675@compuserve.com
- Jeremy Wohl, jeremy@cs.sunysb.edu

The authors wish to acknowledge the contributions of the Portable Network Graphics mailing list and the readers of comp.graphics.

Trademarks

GIF is a service mark of CompuServe Incorporated. Macintosh is a trademark of Apple Computer, Inc. Microsoft and MS−DOS are trademarks of Microsoft Corporation. SGI is a trademark of Silicon Graphics, Inc. X Window System is a trademark of the Massachusetts Institute of Technology.

End of PNG Specification