PNG (Portable Network Graphics) Specification
Version 1.0

Status of this Memo

   This memo provides information for the Internet community. This memo
   does not specify an Internet standard of any kind. Distribution of
   this memo is unlimited.

IESG Note:

   The IESG takes no position on the validity of any Intellectual
   Property Rights statements contained in this document.

Abstract

   This document describes PNG (Portable Network Graphics), an
   extensible file format for the lossless, portable, well-compressed
   storage of raster images. PNG provides a patent-free replacement for
   GIF and can also replace many common uses of TIFF. Indexed-color,
   grayscale, and truecolor images are supported, plus an optional alpha
   channel. Sample depths range from 1 to 16 bits.

   PNG is designed to work well in online viewing applications, such as
   the World Wide Web, so it is fully streamable with a progressive
   display option. PNG is robust, providing both full file integrity
   checking and simple detection of common transmission errors. Also,
   PNG can store gamma and chromaticity data for improved color matching
   on heterogeneous platforms.

   This specification defines the Internet Media Type image/png.

Table of Contents

1. Introduction

   The PNG format provides 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:

       * Indexed-color images of up to 256 colors.
       * Streamability: files can be read and written 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 followed by
         gradual improvement of detail.
       * Transparency: portions of the image can be marked as
         transparent, creating the effect of a non-rectangular image.
       * 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 information, which supports automatic display of
         images with correct brightness/contrast regardless of the
         machines used to originate and display the image.
       * Reliable, straightforward detection of file corruption.
       * Faster initial presentation in progressive display mode.

   PNG is designed to be:

       * Simple and portable: developers should be able to implement PNG
         easily.
       * Legally unencumbered: to the best knowledge of the PNG authors,
         no algorithms under legal challenge are used. (Some
         considerable effort has been spent to verify this.)
       * Well compressed: both indexed-color 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 must 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 gives the definition of the file
   format and recommendations for encoder and decoder behavior. An
   appendix gives the rationale for many design decisions. Although the
   rationale is not part of the formal specification, reading it can
   help implementors understand the design. Cross-references in the
   main text point to relevant parts of the rationale. Additional
   appendixes, also not part of the formal specification, provide
   tutorials on gamma and color theory as well as other supporting
   material.

   In this specification, the word "must" indicates a mandatory
   requirement, while "should" indicates recommended behavior.

   See Rationale: Why a new file format? (Section 12.1), Why these
   features? (Section 12.2), Why not these features? (Section 12.3), Why
   not use format X? (Section 12.4).

   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.

   2.1. Integers and byte order

      All integers that require more than one byte must be in network
      byte order: 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 (Section 12.5).

2.2. Color values

   Colors can be represented by either grayscale or RGB (red, green,
   blue) sample data. Grayscale data represents luminance; RGB data
   represents calibrated color information (if the cHRM chunk is
   present) or uncalibrated device-dependent color (if cHRM is
   absent). All color values range from zero (representing black) to
   most intense at the maximum value for the sample depth. Note that
   the maximum value at a given sample depth is (2^sampledepth)-1,
   not 2^sampledepth.

   Sample values are not necessarily linear; the gAMA chunk specifies
   the gamma characteristic of the source device, and viewers are
   strongly encouraged to compensate properly. See Gamma correction
   (Section 2.7).

   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. This scaling is reversible with no
   loss of data, and it reduces the number of cases that decoders
   have to cope with. See Recommendations for Encoders: Sample depth
   scaling (Section 9.1) and Recommendations for Decoders: Sample
   depth rescaling (Section 10.4).

2.3. Image layout

   Conceptually, a PNG image is 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 actually be transmitted in a different order; see
   Interlaced data order, Section 2.6.) The size of each pixel is
   determined by the bit depth, which is the number of bits per
   sample in the image data.

   Three types of pixel are supported:

       * An indexed-color pixel is represented by a single sample
         that is an index into a supplied palette. The image bit
         depth determines the maximum number of palette entries, but
         not the color precision within the palette.
       * A grayscale pixel is represented by a single sample that is
         a grayscale level, where zero is black and the largest value
         for the bit depth is white.
       * A truecolor pixel is represented by three samples: 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 sample, not the total pixel
         size.

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

Pixels are always packed into scanlines with no wasted bits
between pixels. Pixels smaller than a byte never cross byte
boundaries; they are packed into bytes with the leftmost pixel in
the high-order bits of a byte, the rightmost in the low-order
bits. Permitted bit depths and pixel types are restricted so that
in all cases the packing is simple and efficient.

PNG permits multi-sample pixels only with 8- and 16-bit samples,
so multiple samples of a single pixel are never packed into one
byte. 16-bit samples are stored in network byte order (MSB
first).

Scanlines always begin on byte boundaries. When pixels have fewer
than 8 bits and the scanline width is not evenly divisible by the
number of pixels per byte, the low-order bits in the last byte of
each scanline are wasted. The contents of these wasted bits are
unspecified.

An additional "filter type" byte is added to the beginning of
every scanline (see Filtering, Section 2.5). The filter type byte
is not considered part of the image data, but it is included in
the datastream sent to the compression step.

2.4. Alpha channel

An alpha channel, representing transparency information on a per-
pixel basis, can be included in grayscale and truecolor PNG
images.

An alpha value of zero represents full transparency, and a value
of (2^bitdepth)-1 represents a fully opaque pixel. Intermediate
values indicate partially transparent pixels that can be combined
with a background image to yield a composite image. (Thus, alpha
is really the degree of opacity of the pixel. But most people
refer to alpha as providing transparency information, not opacity
information, and we continue that custom here.)

Alpha channels can 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 samples are represented with the same bit
depth used for the image samples. The alpha sample for each pixel
is stored immediately following the grayscale or RGB samples of
the pixel.

The color values stored for a pixel are 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 sample values premultiplied by the alpha
fraction; in effect, such an 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 an indexed-color image, an alpha value
can be defined for each palette entry. In grayscale and truecolor
images, a single pixel value can 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 can support transparency control partially, or not at all.

See Rationale: Non-premultiplied alpha (Section 12.8),
Recommendations for Encoders: Alpha channel creation (Section
9.4), and Recommendations for Decoders: Alpha channel processing
(Section 10.8).

2.5. Filtering

PNG allows the image data to be filtered before it is compressed.
Filtering can 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 that precedes the filtered
scanline in the precompression datastream. An intelligent encoder
can switch filters from one scanline to the next. The method for
choosing which filter to employ is up to the encoder.

See Filter Algorithms (Chapter 6) and Rationale: Filtering
(Section 12.9).

2.6. 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 method 0, pixels are stored sequentially from left
to right, and scanlines sequentially from top to bottom (no
interlacing).

Interlace method 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 16 by 16 pixels, then pass 3 will contain two scanlines,
each containing four pixels. When pixels have fewer than 8 bits,
each such scanline is padded as needed to fill an integral number
of bytes (see Image layout, Section 2.3). 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,
Chapter 6). 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. Encoders and
decoders must handle this case correctly. In particular, filter
type bytes are only associated with nonempty scanlines; no filter
type bytes are present in an empty pass.

See Rationale: Interlacing (Section 12.6) and Recommendations for
Decoders: Progressive display (Section 10.9).

2.7. Gamma correction

PNG images can specify, via the gAMA chunk, the gamma
characteristic of the image with respect to the original scene.
Display programs are strongly encouraged to use this information,
plus information about the display device they are using and room
lighting, to present the image to the viewer in a way that
reproduces what the image's original author saw as closely as
possible. See Gamma Tutorial (Chapter 13) if you aren't already
familiar with gamma issues.

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

For high-precision applications, the exact chromaticity of the RGB
data in a PNG image can be specified via the cHRM chunk, allowing
more accurate color matching than gamma correction alone will
provide. See Color Tutorial (Chapter 14) if you aren't already
familiar with color representation issues.

See Rationale: Why gamma? (Section 12.7), Recommendations for
Encoders: Encoder gamma handling (Section 9.2), and
Recommendations for Decoders: Decoder gamma handling (Section
10.5).

2.8. 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 [ISO-8859]. This character set is a superset of 7-
bit ASCII.

Character codes not defined in Latin-1 should not be used, because
they have no platform-independent meaning. If a non-Latin-1 code
does appear in a PNG text string, its interpretation will vary
across platforms and decoders. Some systems might not even be
able to display all the characters in Latin-1, but most modern
systems can.

Provision is also made for the storage of compressed text.

See Rationale: Text strings (Section 12.10).

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.

   3.1. 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 (Section 12.11).

   3.2. 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 must 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 and a-z, or 65-90
         and 97-122 decimal). However, encoders and decoders must 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 can be of zero length.

CRC
     A 4-byte CRC (Cyclic Redundancy Check) calculated on the
     preceding bytes in the chunk, including the chunk type code and
     chunk data fields, but not including the length field. The CRC
     is always present, even for chunks containing no data. See CRC
     algorithm (Section 3.4).

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

Chunks can 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 can
appear, but only if specifically permitted for that type.

See Rationale: Chunk layout (Section 12.12).

3.3. Chunk naming conventions

Chunk type codes are assigned so that a decoder can determine some
properties of a chunk even when 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 the 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 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 would be unrelated chunk type codes, not the same
chunk with different properties. Decoders must recognize type
codes by a simple four-byte literal comparison; it is incorrect to
perform case conversion on type codes.

The semantics of the property bits are:

Ancillary bit: bit 5 of first byte
    0 (uppercase) = critical, 1 (lowercase) = ancillary.

    Chunks that are not strictly necessary in order to meaningfully
    display the contents of the file are known as "ancillary"
    chunks. A decoder encountering an unknown chunk in which the
    ancillary bit is 1 can safely ignore the chunk and proceed to
    display the image. The time chunk (tIME) is an example of an
    ancillary chunk.

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

Private bit: bit 5 of second byte
    0 (uppercase) = public, 1 (lowercase) = private.

    A public chunk is one that is part of the PNG specification or
    is registered in the list of PNG special-purpose public chunk
    types. Applications can also define private (unregistered)
    chunks for their own purposes. The names of private chunks
    must have a lowercase second letter, while public chunks will
    always be assigned names with uppercase second letters. Note
    that decoders do not need to test the private-chunk property
    bit, since it has no functional significance; it is simply an
    administrative convenience to ensure that public and private
    chunk names will not conflict. See Additional chunk types
    (Section 4.4) and Recommendations for Encoders: Use of private
    chunks (Section 9.8).

Reserved bit: bit 5 of third byte
    Must be 0 (uppercase) in files conforming to this version of
    PNG.

    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.
    (Decoders should not complain about a lowercase third letter,
    however, as some future version of the PNG specification could
    define a meaning for this bit. It is sufficient to treat a
    chunk with a lowercase third letter in the same way as any
    other unknown chunk type.)

     Safe-to-copy bit: bit 5 of 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 PNG files).
        This bit defines the proper handling of unrecognized chunks in
        a file that is being modified.

        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 can choose to
        output an appropriately modified version.)

        A PNG editor is always allowed to copy all unrecognized chunks
        if it has only added, deleted, modified, or reordered ancillary
        chunks. This implies that it is not permissible for ancillary
        chunks to 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 in Chunk Ordering
        Rules (Chapter 7).

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

        bLOb <-- 32 bit chunk type code represented in text form
        ||||
        |||+- Safe-to-copy bit is 1 (lower case letter; bit 5 is 1)
        ||+-- Reserved bit is 0   (upper case letter; bit 5 is 0)
        |+--- Private bit is 0   (upper case letter; bit 5 is 0)
        +---- Ancillary bit is 1   (lower case letter; bit 5 is 1)

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

See Rationale: Chunk naming conventions (Section 12.13).

3.4. CRC algorithm

Chunk CRCs are calculated using standard CRC methods with pre and post conditioning, as defined by ISO 3309 [ISO-3309] or ITU-T V.42 [ITU-V42]. The CRC polynomial employed is

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

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 Sample CRC Code (Chapter 15).

4. Chunk Specifications

This chapter defines the standard types of PNG chunks.

4.1. 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.

4.1.1. IHDR Image header

The IHDR chunk must appear FIRST. It contains:

```
Width:                4 bytes
Height:               4 bytes
Bit depth:            1 byte
Color type:           1 byte
Compression method: 1 byte
Filter method:        1 byte
Interlace method:     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 that have
difficulty with unsigned 4-byte values.

Bit depth is a single-byte integer giving the number of bits
per sample or per palette index (not per pixel). 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 codes represent
sums of the following values: 1 (palette used), 2 (color used),
and 4 (alpha channel used). Valid values are 0, 2, 3, 4, and 6.

Bit depth restrictions for each color type are imposed to
simplify implementations and to prohibit combinations that do
not compress well. Decoders must support all legal
combinations of bit depth and color type. The allowed
combinations are:

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

The sample depth is the same as the bit depth except in the
case of color type 3, in which the sample depth is always 8
bits.

Compression method is a single-byte integer that indicates the
method used to compress the image data. At present, only
compression method 0 (deflate/inflate compression with a 32K
sliding window) is defined. All standard PNG images must be
compressed with this scheme. The compression method field 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 (Chapter
5) for details.

Filter method is a single-byte integer that indicates the
preprocessing method applied to the image data before
compression. At present, only filter method 0 (adaptive
filtering with five basic filter types) is defined. As with
the compression method field, decoders must check this byte and
report an error if it holds an unrecognized code. See Filter
Algorithms (Chapter 6) for details.

Interlace method is a single-byte integer that indicates the
transmission order of the image data. Two values are currently
defined: 0 (no interlace) or 1 (Adam7 interlace). See
Interlaced data order (Section 2.6) for details.

4.1.2. PLTE Palette

The PLTE chunk contains 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 can appear for
color types 2 and 6; it must not appear for color types 0 and
4. If this chunk does appear, it must precede the first IDAT
chunk. There must not be more than one PLTE chunk.

For color type 3 (indexed color), 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 in the image
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 and truecolor with alpha),
the PLTE chunk is optional. If present, it provides a
suggested set of from 1 to 256 colors to which the truecolor
image can be quantized if the viewer cannot display truecolor
directly. If PLTE is not present, such a viewer will need to
select colors on its own, but it is often preferable for this

to be done once by the encoder. (See Recommendations for
Encoders: Suggested palettes, Section 9.5.)

Note that the palette uses 8 bits (1 byte) per sample
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.

There is no requirement that the palette entries all be used by
the image, nor that they all be different.

4.1.3. IDAT Image data

The IDAT chunk contains the actual image data. To create this
data:

* Begin with image scanlines represented as described in
  Image layout (Section 2.3); the layout and total size of
  this raw data are determined by the fields of IHDR.
* 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.)
* 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 can 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 can divide the compressed datastream
into IDAT chunks however 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 occur 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 (Chapter 6) and Deflate/Inflate
Compression (Chapter 5) for details.

4.1.4. IEND Image trailer

The IEND chunk must appear LAST. It marks the end of the PNG
datastream. The chunk's data field is empty.

4.2. Ancillary chunks

All ancillary chunks are optional, in the sense that encoders need
not write them and decoders can 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.

4.2.1. bKGD Background color

The bKGD chunk specifies a default background color to present
the image against. Note that viewers are not bound to honor
this chunk; a viewer can choose to use a different background.

For color type 3 (indexed color), 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 (truecolor, 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 (Section 10.7).

4.2.2. cHRM Primary chromaticities and white point

Applications that need device-independent specification of colors in a PNG file can use the cHRM chunk to specify the 1931 CIE x,y chromaticities of the red, green, and blue primaries used in the image, and the referenced white point. See Color Tutorial (Chapter 14) for more information.

The cHRM chunk contains:

```
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. For example, a value of 0.3127 would be stored as the integer 31270.

cHRM is allowed in all PNG files, although it is of little value for grayscale images.

If the encoder does not know the chromaticity values, it should not write a cHRM chunk; the absence of a cHRM chunk indicates that the image's primary colors are device-dependent.

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

See Recommendations for Encoders: Encoder color handling (Section 9.3), and Recommendations for Decoders: Decoder color handling (Section 10.6).

4.2.3. gAMA Image gamma

The gAMA 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. More precisely, the
gAMA chunk encodes the file_gamma value, as defined in Gamma
Tutorial (Chapter 13).

The gAMA chunk contains:

    Image gamma: 4 bytes

The value is encoded as a 4-byte unsigned integer, representing
gamma times 100000. For example, a gamma of 0.45 would be
stored as the integer 45000.

If the encoder does not know the image's gamma value, it should
not write a gAMA chunk; the absence of a gAMA chunk indicates
that 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 (Section 2.7), Recommendations for
Encoders: Encoder gamma handling (Section 9.2), and
Recommendations for Decoders: Decoder gamma handling (Section
10.5).

4.2.4. hIST Image histogram

The hIST chunk gives the approximate usage frequency of each
color in the color palette. A histogram chunk can 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.

The hIST chunk contains 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 (Section 12.14), and
Recommendations for Decoders: Suggested-palette and histogram
usage (Section 10.10).

4.2.5. pHYs Physical pixel dimensions

The pHYs chunk specifies the intended pixel size or aspect
ratio for display of the image. It contains:

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

The following values are legal for the unit specifier:

    0: unit is unknown
    1: unit is the meter

When the unit specifier is 0, the pHYs chunk defines pixel
aspect ratio only; the actual size of the pixels remains
unspecified.

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 (Section
10.2).

4.2.6. sBIT Significant bits

To simplify decoders, PNG specifies that only certain sample
depths can be used, and further specifies that sample values
should be scaled to the full range of possible values at the
sample depth. However, the sBIT chunk is provided in order to
store the original number of significant bits. This allows

decoders to recover the original data losslessly even if the
data had a sample depth not directly supported by PNG. We
recommend that an encoder emit an sBIT chunk if it has
converted the data from a lower sample depth.

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

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

For color type 3 (indexed color), the sBIT chunk contains three
bytes, indicating the number of bits that 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 that were
significant in the source grayscale data and the source alpha
data, respectively.

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

Each depth specified in sBIT must be greater than zero and less
than or equal to the sample depth (which is 8 for indexed-color
images, and the bit depth given in IHDR for other color types).

A decoder need not pay attention to sBIT: the stored image is a
valid PNG file of the sample depth indicated by IHDR. However,
if the decoder wishes to recover the original data at its
original precision, this can be done by right-shifting the
stored samples (the stored palette entries, for an indexed-
color image). The encoder must scale the data in such a way
that the high-order bits match the original data.

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

See Recommendations for Encoders: Sample depth scaling (Section
9.1) and Recommendations for Decoders: Sample depth rescaling
(Section 10.4).

4.2.7. tEXt Textual data

Textual information that the encoder wishes to record with the
image can be stored in tEXt chunks. Each tEXt chunk contains a
keyword and a text string, in the format:

    Keyword: 1-79 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 can
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 can be of any length from zero bytes up to the maximum
permissible chunk size less the length of the keyword and
separator.

Any number of tEXt chunks can 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
    Description   Description of image (possibly long)
    Copyright   Copyright notice
    Creation Time   Time of original image creation
    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

For the Creation Time keyword, the date format defined in
section 5.2.14 of RFC 1123 is suggested, but not required
[RFC-1123]. Decoders should allow for free-format text
associated with this or any other keyword.

Other keywords may be invented for other purposes. Keywords of
general interest can be registered with the maintainers of the
PNG specification. However, it is also permitted to use
private unregistered keywords. (Private keywords should be

reasonably self-explanatory, in order to minimize the chance
that the same keyword will be used for incompatible purposes by
different people.)

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

Keywords must contain only printable Latin-1 characters and
spaces; that is, only character codes 32-126 and 161-255
decimal are allowed. To reduce the chances for human
misreading of a keyword, leading and trailing spaces are
forbidden, as are consecutive spaces. Note also that the non-
breaking space (code 160) is not permitted in keywords, since
it is visually indistinguishable from an ordinary space.

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

See Recommendations for Encoders: Text chunk processing
(Section 9.7) and Recommendations for Decoders: Text chunk
processing (Section 10.11).

   4.2.8. tIME Image last-modification time

The tIME chunk gives the time of the last image modification
(not the time of initial image creation). It contains:

```
   Year:   2 bytes (complete; for example, 1995, not 95)
   Month:  1 byte (1-12)
   Day:    1 byte (1-31)
   Hour:   1 byte (0-23)
   Minute: 1 byte (0-59)
   Second: 1 byte (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.

The tIME chunk is intended for use as an automatically-applied
time stamp that is updated whenever the image data is changed.
It is recommended that tIME not be changed by PNG editors that
do not change the image data. See also the Creation Time tEXt
keyword, which can be used for a user-supplied time.

4.2.9. tRNS Transparency

The tRNS chunk specifies that the image uses simple
transparency: either alpha values associated with palette
entries (for indexed-color images) or a single transparent
color (for grayscale and truecolor images). Although simple
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 (indexed color), the tRNS chunk contains a
series of one-byte alpha values, 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 the corresponding palette
index must 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 must not contain
more alpha values than there are palette entries, but tRNS can
contain fewer values than there are palette entries. In this
case, the alpha value for all remaining palette entries is
assumed to be 255. In the common case in which only palette
index 0 need be made transparent, only a one-byte tRNS chunk is
needed.

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 (truecolor), 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 truecolor 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.

4.2.10. zTXt Compressed textual data

The zTXt chunk contains textual data, just as tEXt does;
however, zTXt takes advantage of compression. zTXt and tEXt
chunks are semantically equivalent, but zTXt is recommended for
storing large blocks of text.

A zTXt chunk contains:

```
Keyword:            1-79 bytes (character string)
Null separator:     1 byte
Compression method: 1 byte
Compressed text:    n bytes
```

The keyword and null separator are exactly the same as in the
tEXt chunk. Note that the keyword is not compressed. The
compression method byte identifies the compression method used
in this zTXt chunk. The only value presently defined for it is
0 (deflate/inflate compression). The compression method byte is

followed by a compressed datastream that makes up the remainder
of the chunk. For compression method 0, this datastream
adheres to the zlib datastream format (see Deflate/Inflate
Compression, Chapter 5). Decompression of this datastream
yields Latin-1 text that is identical to the text that would be
stored in an equivalent tEXt chunk.

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

See Recommendations for Encoders: Text chunk processing
(Section 9.7), and Recommendations for Decoders: Text chunk
processing (Section 10.11).

4.3. 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 OK?	Ordering constraints
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 OK?	Ordering constraints
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
Description       Description of image (possibly long)
Copyright         Copyright notice
Creation Time     Time of original image creation
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
```

4.4. Additional chunk types

Additional public PNG chunk types are defined in the document "PNG Special-Purpose Public Chunks" [PNG-EXTENSIONS]. Chunks described there are expected to be 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 can be proposed for inclusion in that list by contacting the PNG specification maintainers at png-info@uunet.uu.net or at png-group@w3.org.

New public chunks will 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.

Applications can also use private chunk types to carry data that is not of interest to other applications. See Recommendations for Encoders: Use of private chunks (Section 9.8).

Decoders must be prepared to encounter unrecognized public or private chunk type codes. Unrecognized chunk types must be handled as described in Chunk naming conventions (Section 3.3).

5. Deflate/Inflate Compression

PNG compression method 0 (the only compression method presently defined for PNG) specifies deflate/inflate compression with a 32K sliding 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.

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
Check value:                   4 bytes
```

Further details on this format are given in the zlib specification
[RFC-1950].

For PNG compression method 0, the zlib compression method/flags code
must specify method code 8 ("deflate" compression) and an LZ77 window
size of not more than 32K. Note that the zlib compression method
number is not the same as the PNG compression method number. The
additional flags must not specify a preset dictionary.

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 are given in the
deflate specification [RFC-1951].

The check value 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 check values. The zlib check value is useful
mainly as a cross-check that the deflate and inflate algorithms are
implemented correctly. Verifying the chunk CRCs provides adequate
confidence that the PNG file has been transmitted undamaged.

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 can 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 check value
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, some encoder implementations may emit files
in which some of these structures are indeed related. But decoders
cannot rely on this.)

PNG also uses zlib datastreams in zTXt chunks. In a zTXt chunk, the
remainder of the chunk following the compression method 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.

Additional documentation and portable C code for deflate and inflate
are available from the Info-ZIP archives at
<URL:ftp://ftp.uu.net/pub/archiving/zip/>.

6. Filter Algorithms

This chapter describes the filter algorithms that can be applied
before compression. The purpose of these filters is to prepare the
image data for optimum compression.

6.1. Filter types

   PNG filter method 0 defines five basic filter types:

        Type   Name

        0   None
        1   Sub
        2   Up
        3   Average
        4   Paeth

   (Note that filter method 0 in IHDR specifies exactly this set of
   five filter types. If the set of filter types is ever extended, a
   different filter method number will be assigned to the extended
   set, so that decoders need not decompress the data to discover
   that it contains unsupported filter types.)

   The encoder can choose which of these filter algorithms 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
   that specifies 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 in Image layout
(Section 2.3). If the image includes an alpha channel, the alpha
data is filtered in the same way as the image data.

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 will have to be stored by
the decoder at all times. Even though some filter types do not
refer to the prior scanline, the decoder will always need to 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 can be applied to
an image. However, the filters are not equally effective on all
types of data. See Recommendations for Encoders: Filter selection
(Section 9.6).

See also Rationale: Filtering (Section 12.9).

6.2. 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.

   6.3. 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 the scanline:

         Sub(x) = Raw(x) - Raw(x-bpp)

      where x ranges from zero to the number of bytes representing the
      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
      samples, two bytes per sample); 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 sample, plus
      two-byte alpha sample).

      Note this computation is done for each byte, regardless of bit
      depth. In a 16-bit image, each MSB is predicted from the
      preceding MSB and each LSB 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 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.

   6.4. 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 the scanline:

         Up(x) = Raw(x) - Prior(x)

where x ranges from zero to the number of bytes representing the
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 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.

6.5. 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 the scanline:

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

where x ranges from zero to the number of bytes representing the
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.

6.6. 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 due to Alan W. Paeth [PAETH].

To compute the Paeth filter, apply the following formula to each
byte of the 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 the
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 then return a
     else if pb <= pc then return b
     else 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 byte value.

Note that the order in which ties are broken is critical 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 and
wishes to preserve as much as possible of the ancillary information
in the file. Two examples of PNG editors are a program that adds or
modifies text chunks, and a program that adds a suggested palette to
a truecolor PNG file. Ordinary image editors are not PNG editors in
this sense, because they usually discard all 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. If our program to add a suggested
PLTE 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 all unknown chunks, but
that is a very unattractive solution. Instead, PNG requires
ancillary chunks not to have ordering restrictions like this.

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.

7.1. Behavior of PNG editors

   The rules for PNG editors are:

         * When copying an unknown unsafe-to-copy ancillary chunk, a
           PNG editor must not move the chunk relative to any critical
           chunk. It can 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
           must not add, delete, modify, or reorder critical chunks if
           it is preserving unknown unsafe-to-copy chunks.)
         * When copying an unknown safe-to-copy ancillary chunk, a PNG
           editor must 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.
         * When copying a known ancillary chunk type, an editor need
           only honor the specific chunk ordering rules that exist for
           that chunk type. However, it can always choose to apply the
           above general rules instead.
         * PNG editors must give up on encountering an unknown critical
           chunk type, because there is no way to be certain that a
           valid file will result from modifying a file containing such
           a chunk. (Note that simply discarding the chunk is not good
           enough, because it might have unknown implications for the
           interpretation of other chunks.)

These rules are expressed in terms of copying chunks from an input
file to an output file, but they apply in the obvious way if a PNG
file is modified in place.

See also Chunk naming conventions (Section 3.3).

7.2. Ordering of ancillary chunks

The ordering rules for an ancillary chunk type cannot be any
stricter than this:

* Unsafe-to-copy chunks can have ordering requirements
  relative to critical chunks.
* Safe-to-copy chunks can have ordering requirements relative
  to IDAT.

The actual ordering rules for any particular ancillary chunk type
may be weaker. See for example the ordering rules for the
standard ancillary chunk types (Summary of standard chunks,
Section 4.3).

Decoders must 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 after it. But you can
safely assume that your chunk will remain somewhere between IDAT
and IEND.)

7.3. Ordering of critical chunks

Critical chunks can have arbitrary ordering requirements, because
PNG editors are required to give up if they encounter unknown
critical chunks. For example, IHDR has the special ordering rule
that it must always appear first. A PNG editor, or indeed any
PNG-writing program, must know and follow the ordering rules for
any critical chunk type that it can emit.

8. Miscellaneous Topics

   8.1. File name extension

      On systems where file names customarily include an extension
      signifying file type, the extension ".png" is recommended for PNG
      files. Lower case ".png" is preferred if file names are case-
      sensitive.

   8.2. Internet media type

      The Internet Assigned Numbers Authority (IANA) has registered
      "image/png" as the Internet Media Type for PNG [RFC-2045, RFC-
      2048]. For robustness, decoders may choose to also support the
      interim media type "image/x-png" which was in use before
      registration was complete.

   8.3. Macintosh file layout

      In the Apple Macintosh system, the following conventions are
      recommended:

         * The four-byte file type code for PNG files is "PNGf". (This
           code has been registered with Apple for PNG files.) The
           creator code will vary depending on the creating
           application.
         * The contents of the data fork must be a PNG file exactly as
           described in the rest of this specification.
         * The contents of the resource fork are unspecified. It may
           be empty or may contain application-dependent resources.
         * When transferring a Macintosh PNG file to a non-Macintosh
           system, only the data fork should be transferred.

   8.4. Multiple-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. In the future, a
      multiple-image format based on PNG may be defined. Such a format
      will be considered a separate file format and will have a
      different signature. PNG-supporting applications may or may not
      choose to support the multiple-image format.

      See Rationale: Why not these features? (Section 12.3).

8.5. Security considerations

A PNG file or datastream is composed of a collection of explicitly
typed "chunks". Chunks whose contents are defined by the
specification could actually contain anything, including malicious
code. But there is no known risk that such malicious code could
be executed on the recipient's computer as a result of decoding
the PNG image.

The possible security risks associated with future chunk types
cannot be specified at this time. Security issues will be
considered when evaluating chunks proposed for registration as
public chunks. There is no additional security risk associated
with unknown or unimplemented chunk types, because such chunks
will be ignored, or at most be copied into another PNG file.

The tEXt and zTXt chunks contain data that is meant to be
displayed as plain text. It is possible that if the decoder
displays such text without filtering out control characters,
especially the ESC (escape) character, certain systems or
terminals could behave in undesirable and insecure ways. We
recommend that decoders filter out control characters to avoid
this risk; see Recommendations for Decoders: Text chunk processing
(Section 10.11).

Because every chunk's length is available at its beginning, and
because every chunk has a CRC trailer, there is a very robust
defense against corrupted data and against fraudulent chunks that
attempt to overflow the decoder's buffers. Also, the PNG
signature bytes provide early detection of common file
transmission errors.

A decoder that fails to check CRCs could be subject to data
corruption. The only likely consequence of such corruption is
incorrectly displayed pixels within the image. Worse things might
happen if the CRC of the IHDR chunk is not checked and the width
or height fields are corrupted. See Recommendations for Decoders:
Error checking (Section 10.1).

A poorly written decoder might be subject to buffer overflow,
because chunks can be extremely large, up to (2^31)-1 bytes long.
But properly written decoders will handle large chunks without
difficulty.

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
   that conform to the format specified in the preceding chapters.
   However, best results will usually be achieved by following these
   recommendations.

   9.1. Sample depth scaling

      When encoding input samples that have a sample depth that cannot
      be directly represented in PNG, the encoder must scale the samples
      up to a sample depth that is allowed by PNG. The most accurate
      scaling method is the linear equation

         output = ROUND(input * MAXOUTSAMPLE / MAXINSAMPLE)

      where the input samples range from 0 to MAXINSAMPLE and the
      outputs range from 0 to MAXOUTSAMPLE (which is (2^sampledepth)-1).

      A close approximation to the linear scaling method can be achieved
      by "left bit replication", which is shifting the valid bits to
      begin in the most significant bit and repeating the most
      significant bits into the open bits. This method is often faster
      to compute than linear scaling. As an example, assume that 5-bit
      samples are being scaled up to 8 bits. If the source sample value
      is 27 (in the range from 0-31), then the original bits are:

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

      Left bit replication gives a value of 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

      which matches the value computed by the linear equation. Left bit
      replication usually gives the same value as linear scaling, and is
      never off by more than one.

A distinctly less accurate approximation is obtained by simply
left-shifting the input value and filling the low order bits with
zeroes. This scheme cannot reproduce white exactly, since it does
not generate an all-ones maximum value; the net effect is to
darken the image slightly. This method is not recommended in
general, but it does have the effect of improving compression,
particularly when dealing with greater-than-eight-bit sample
depths. Since the relative error introduced by zero-fill scaling
is small at high sample depths, some encoders may choose to use
it. Zero-fill must not be used for alpha channel data, however,
since many decoders will special-case alpha values of all zeroes
and all ones. It is important to represent both those values
exactly in the scaled data.

When the encoder writes an sBIT chunk, it is required to do the
scaling in such a way that the high-order bits of the stored
samples match the original data. That is, if the sBIT chunk
specifies a sample depth of S, the high-order S bits of the stored
data must agree with the original S-bit data values. This allows
decoders to recover the original data by shifting right. The
added low-order bits are not constrained. Note that all the above
scaling methods meet this restriction.

When scaling up source data, it is recommended that the low-order
bits be filled consistently for all samples; that is, the same
source value should generate the same sample value at any pixel
position. This improves compression by reducing the number of
distinct sample values. However, this is not a requirement, and
some encoders may choose not to follow it. For example, an
encoder might instead dither the low-order bits, improving
displayed image quality at the price of increasing file size.

In some applications the original source data may have a range
that is not a power of 2. The linear scaling equation still works
for this case, although the shifting methods do not. It is
recommended that an sBIT chunk not be written for such images,
since sBIT suggests that the original data range was exactly
$0..2^S-1$.

9.2. Encoder gamma handling

See Gamma Tutorial (Chapter 13) if you aren't already familiar
with gamma issues.

Proper handling of gamma encoding and the gAMA chunk in an encoder
depends on the prior history of the sample values and on whether
these values have already been quantized to integers.

If the encoder has access to sample intensity values in floating-
point or high-precision integer form (perhaps from a computer
image renderer), then it is recommended that the encoder perform
its own gamma encoding before quantizing the data to integer
values for storage in the file. Applying gamma encoding at this
stage results in images with fewer banding artifacts at a given
sample depth, or allows smaller samples while retaining the same
visual quality.

A linear intensity level, expressed as a floating-point value in
the range 0 to 1, can be converted to a gamma-encoded sample value
by

    sample = ROUND((intensity ^ encoder_gamma) * MAXSAMPLE)

The file_gamma value to be written in the PNG gAMA chunk is the
same as encoder_gamma in this equation, since we are assuming the
initial intensity value is linear (in effect, camera_gamma is
1.0).

If the image is being written to a file only, the encoder_gamma
value can be selected somewhat arbitrarily. Values of 0.45 or 0.5
are generally good choices because they are common in video
systems, and so most PNG decoders should do a good job displaying
such images.

Some image renderers may simultaneously write the image to a PNG
file and display it on-screen. The displayed pixels should be
gamma corrected for the display system and viewing conditions in
use, so that the user sees a proper representation of the intended
scene. An appropriate gamma correction value is

    screen_gc = viewing_gamma / display_gamma

If the renderer wants to write the same gamma-corrected sample
values to the PNG file, avoiding a separate gamma-encoding step
for file output, then this screen_gc value should be written in
the gAMA chunk. This will allow a PNG decoder to reproduce what
the file's originator saw on screen during rendering (provided the
decoder properly supports arbitrary values in a gAMA chunk).

However, it is equally reasonable for a renderer to apply gamma
correction for screen display using a gamma appropriate to the
viewing conditions, and to separately gamma-encode the sample
values for file storage using a standard value of gamma such as
0.5. In fact, this is preferable, since some PNG decoders may not
accurately display images with unusual gAMA values.

Computer graphics renderers often do not perform gamma encoding,
instead making sample values directly proportional to scene light
intensity. If the PNG encoder receives sample values that have
already been quantized into linear-light integer values, there is
no point in doing gamma encoding on them; that would just result
in further loss of information. The encoder should just write the
sample values to the PNG file. This "linear" sample encoding is
equivalent to gamma encoding with a gamma of 1.0, so graphics
programs that produce linear samples should always emit a gAMA
chunk specifying a gamma of 1.0.

When the sample values come directly from a piece of hardware, the
correct gAMA value is determined by the gamma characteristic of
the hardware. In the case of video digitizers ("frame grabbers"),
gAMA should be 0.45 or 0.5 for NTSC (possibly less for PAL or
SECAM) since video camera transfer functions are standardized.
Image scanners are less predictable. Their output samples may be
linear (gamma 1.0) since CCD sensors themselves are linear, or the
scanner hardware may have already applied gamma correction
designed to compensate for dot gain in subsequent printing (gamma
of about 0.57), or the scanner may have corrected the samples for
display on a CRT (gamma of 0.4-0.5). You will need to refer to
the scanner's manual, or even scan a calibrated gray wedge, to
determine what a particular scanner does.

File format converters generally should not attempt to convert
supplied images to a different gamma. Store the data in the PNG
file without conversion, and record the source gamma if it is
known. Gamma alteration at file conversion time causes re-
quantization of the set of intensity levels that are represented,
introducing further roundoff error with little benefit. It's
almost always better to just copy the sample values intact from
the input to the output file.

In some cases, the supplied image may be in an image format (e.g.,
TIFF) that can describe the gamma characteristic of the image. In
such cases, a file format converter is strongly encouraged to
write a PNG gAMA chunk that corresponds to the known gamma of the
source image. Note that some file formats specify the gamma of
the display system, not the camera. If the input file's gamma
value is greater than 1.0, it is almost certainly a display system
gamma, and you should use its reciprocal for the PNG gAMA.

If the encoder or file format converter does not know how an image
was originally created, but does know that the image has been
displayed satisfactorily on a display with gamma display_gamma
under lighting conditions where a particular viewing_gamma is
appropriate, then the image can be marked as having the
file_gamma:

    file_gamma = viewing_gamma / display_gamma

This will allow viewers of the PNG file to see the same image that
the person running the file format converter saw. Although this
may not be precisely the correct value of the image gamma, it's
better to write a gAMA chunk with an approximately right value
than to omit the chunk and force PNG decoders to guess at an
appropriate gamma.

On the other hand, if the image file is being converted as part of
a "bulk" conversion, with no one looking at each image, then it is
better to omit the gAMA chunk entirely. If the image gamma has to
be guessed at, leave it to the decoder to do the guessing.

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

See also Recommendations for Decoders: Decoder gamma handling
(Section 10.5).

9.3. Encoder color handling

See Color Tutorial (Chapter 14) if you aren't already familiar
with color issues.

If it is possible for the encoder to determine the chromaticities
of the source display primaries, or to make a strong guess based
on the origin of the image or the hardware running it, then the
encoder is strongly encouraged to output the cHRM chunk. If it
does so, the gAMA chunk should also be written; decoders can do
little with cHRM if gAMA is missing.

Video created with recent video equipment probably uses the CCIR
709 primaries and D65 white point [ITU-BT709], which are:

```
              R           G           B          White
    x       0.640       0.300       0.150       0.3127
    y       0.330       0.600       0.060       0.3290
```

An older but still very popular video standard is SMPTE-C [SMPTE-
170M]:

```
              R           G           B          White
    x       0.630       0.310       0.155       0.3127
    y       0.340       0.595       0.070       0.3290
```

The original NTSC color primaries have not been used in decades.
Although you may still find the NTSC numbers listed in standards
documents, you won't find any images that actually use them.

Scanners that produce PNG files as output should insert the filter
chromaticities into a cHRM chunk and the camera_gamma into a gAMA
chunk.

In the case of hand-drawn or digitally edited images, you have to
determine what monitor they were viewed on when being produced.
Many image editing programs allow you to specify what type of
monitor you are using. This is often because they are working in
some device-independent space internally. Such programs have
enough information to write valid cHRM and gAMA chunks, and should
do so automatically.

If the encoder is compiled as a portion of a computer image
renderer that performs full-spectral rendering, the monitor values
that were used to convert from the internal device-independent
color space to RGB should be written into the cHRM chunk. Any
colors that are outside the gamut of the chosen RGB device should
be clipped or otherwise constrained to be within the gamut; PNG
does not store out of gamut colors.

If the computer image renderer performs calculations directly in
device-dependent RGB space, a cHRM chunk should not be written
unless the scene description and rendering parameters have been
adjusted to look good on a particular monitor. In that case, the
data for that monitor (if known) should be used to construct a
cHRM chunk.

There are often cases where an image's exact origins are unknown,
particularly if it began life in some other format. A few image
formats store calibration information, which can be used to fill
in the cHRM chunk. For example, all PhotoCD images use the CCIR
709 primaries and D65 whitepoint, so these values can be written
into the cHRM chunk when converting a PhotoCD file. PhotoCD also
uses the SMPTE-170M transfer function, which is closely
approximated by a gAMA of 0.5. (PhotoCD can store colors outside
the RGB gamut, so the image data will require gamut mapping before
writing to PNG format.) TIFF 6.0 files can optionally store
calibration information, which if present should be used to
construct the cHRM chunk. GIF and most other formats do not store
any calibration information.

It is not recommended that file format converters attempt to
convert supplied images to a different RGB color space. Store the
data in the PNG file without conversion, and record the source
primary chromaticities if they are known. Color space
transformation at file conversion time is a bad idea because of
gamut mismatches and rounding errors. As with gamma conversions,
it's better to store the data losslessly and incur at most one
conversion when the image is finally displayed.

See also Recommendations for Decoders: Decoder color handling
(Section 10.6).

9.4. Alpha channel creation

The alpha channel can 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.

Image authors should keep in mind the possibility that a decoder
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 decoders 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 sample value by the corresponding alpha value, then
multiplying by the maximum value for the image bit depth, and
rounding to the nearest integer. In valid premultiplied data, the
sample 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).

9.5. Suggested palettes

A PLTE chunk can appear in truecolor PNG files. In such files,
the chunk is not an essential part of the image data, but simply
represents a suggested palette that viewers may use to present the
image on indexed-color display hardware. A suggested palette is
of no interest to viewers running on truecolor hardware.

If an encoder chooses to provide a suggested palette, it is
recommended that a hIST chunk also be written to indicate the
relative importance of the palette entries. The histogram values
are most easily computed as "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.)

For images of color type 2 (truecolor without alpha channel), it
is recommended that the palette and histogram be computed with
reference to the RGB data only, ignoring any transparent-color
specification. If the file uses transparency (has a tRNS chunk),
viewers can easily adapt the resulting palette for use with their
intended background color. They need only replace the palette
entry closest to the tRNS color with their background color (which
may or may not match the file's bKGD color, if any).

For images of color type 6 (truecolor with alpha channel), it is
recommended that a bKGD chunk appear and that the palette and
histogram be computed with reference to the image as it would
appear after compositing against the specified background color.
This definition is necessary to ensure that useful palette entries
are generated for pixels having fractional alpha values. The
resulting palette will probably only be useful to viewers that
present the image against the same background color. It is
recommended that PNG editors delete or recompute the palette if
they alter or remove the bKGD chunk in an image of color type 6.

If PLTE appears without bKGD in an image of color type 6, the
circumstances under which the palette was computed are
unspecified.

9.6. Filter selection

For images of color type 3 (indexed color), filter type 0 (None)
is usually the most effective. Note that color images with 256 or
fewer colors should almost always be stored in indexed color
format; truecolor format is likely to be much larger.

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 uses a fixed filter, 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 that 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.

9.7. Text chunk processing

A nonempty keyword must be provided for each text chunk. The
generic keyword "Comment" can be used if no better description of
the text is available. If a user-supplied keyword is used, be
sure to check that it meets the restrictions on keywords.

PNG text strings are expected to use the Latin-1 character set.
Encoders should avoid storing characters that are not defined in
Latin-1, and should provide character code remapping if the local
system's character set is not Latin-1.

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

It is recommended that text items less than 1K (1024 bytes) in
size should be output using uncompressed tEXt chunks. In
particular, it is recommended that the basic title and author
keywords should always be output using uncompressed tEXt chunks.
Lengthy disclaimers, on the other hand, are ideal candidates for
zTXt.

Placing large tEXt and zTXt chunks after the image data (after
IDAT) can 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.

9.8. Use of private chunks

Applications can use PNG private chunks to carry information that
need not be understood by other applications. Such chunks must be
given names with lowercase second letters, to ensure that they can
never conflict with any future public chunk definition. Note,
however, that there is no guarantee that some other application
will not use the same private chunk name. If you use a private
chunk type, it is prudent to store additional identifying
information at the beginning of the chunk data.

Use an ancillary chunk type (lowercase first letter), not a
critical chunk type, for all private chunks that store information
that is not absolutely essential to view the image. Creation of
private critical chunks is discouraged because they render PNG
files unportable. Such chunks should not be used in publicly
available software or files. If private critical chunks are
essential for your application, it is recommended that one appear
near the start of the file, so that a standard decoder need not
read very far before discovering that it cannot handle the file.

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, Section 4.4). Note that a proposed public
chunk name (with uppercase second letter) must not be used in
publicly available software or files until registration has been
approved.

If an ancillary chunk contains textual information that might be
of interest to a human user, you should not create a special chunk
type for it. Instead use a tEXt chunk and define a suitable
keyword. That way, the information will be available to users not
using your software.

Keywords in tEXt chunks should be reasonably self-explanatory,
since the idea is to let other users figure out what the chunk
contains. If of general usefulness, new keywords can be
registered with the maintainers of the PNG specification. But it
is permissible to use keywords without registering them first.

9.9. Private type and method codes

This specification defines the meaning of only some of the
possible values of some fields. For example, only compression
method 0 and filter types 0 through 4 are defined. Numbers
greater than 127 must be used when inventing experimental or
private definitions of values for any of these fields. Numbers
below 128 are reserved for possible future public extensions of
this specification. Note that use of private type codes may
render a file unreadable by standard decoders. Such codes are
strongly discouraged except for experimental purposes, and should
not appear in publicly available software or files.

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.

10.1. Error checking

To ensure early detection of common file-transfer problems,
decoders should verify that all eight bytes of the PNG file
signature are correct. (See Rationale: PNG file signature,
Section 12.11.) A decoder can have additional confidence in the
file's integrity if the next eight bytes are an IHDR chunk header
with the correct chunk length.

Unknown chunk types must be handled as described in Chunk naming
conventions (Section 3.3). An unknown chunk type is not to be
treated as an error unless it is a critical chunk.

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

In some situations it is desirable to check chunk headers (length
and type code) before reading the chunk data and CRC. The chunk
type can be checked for plausibility by seeing whether all four
bytes are ASCII letters (codes 65-90 and 97-122); note that this
need only be done for unrecognized type codes. If the total file

size is known (from file system information, HTTP protocol, etc),
the chunk length can be checked for plausibility as well.

If CRCs are not checked, dropped/added data bytes or an erroneous
chunk length can cause the decoder to get out of step and
misinterpret subsequent data as a chunk header. Verifying that
the chunk type contains letters is an inexpensive way of providing
early error detection in this situation.

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 new information.

Unexpected values in fields of known chunks (for example, an
unexpected compression method in the IHDR chunk) must be checked
for and treated as errors. However, it is recommended that
unexpected field values be treated as fatal errors only in
critical chunks. An unexpected value in an ancillary chunk can be
handled by ignoring the whole chunk as though it were an unknown
chunk type. (This recommendation assumes that the chunk's CRC has
been verified. In decoders that do not check CRCs, it is safer to
treat any unexpected value as indicating a corrupted file.)

10.2. Pixel dimensions

Non-square pixels can be represented (see the pHYs chunk), but
viewers are not required to account for them; a viewer can 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.

10.3. Truecolor image handling

To achieve PNG's goal of universal interchangeability, decoders
are required to accept all types of PNG image: indexed-color,
truecolor, and grayscale. Viewers running on indexed-color
display hardware need to be able to reduce truecolor images to
indexed format 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 that can 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, libpng, includes code for the
purpose.

10.4. Sample depth rescaling

Decoders may wish to scale PNG data to a lesser sample depth (data
precision) for display. For example, 16-bit data will need to be
reduced to 8-bit depth for use on most present-day display
hardware. Reduction of 8-bit data to 5-bit depth is also common.

The most accurate scaling is achieved by the linear equation

    output = ROUND(input * MAXOUTSAMPLE / MAXINSAMPLE)

where

    MAXINSAMPLE = (2^sampledepth)-1
    MAXOUTSAMPLE = (2^desired_sampledepth)-1

A slightly less accurate conversion is achieved by simply shifting
right by sampledepth-desired_sampledepth places. For example, to
reduce 16-bit samples to 8-bit, one need only discard the low-
order byte. In many situations the shift method is sufficiently
accurate for display purposes, and it is certainly much faster.
(But if gamma correction is being done, sample rescaling can be
merged into the gamma correction lookup table, as is illustrated
in Decoder gamma handling, Section 10.5.)

When an sBIT chunk is present, the original pre-PNG data can be
recovered by shifting right to the sample depth specified by sBIT.
Note that linear scaling will not necessarily reproduce the
original data, because the encoder is not required to have used
linear scaling to scale the data up. However, the encoder is
required to have used a method that preserves the high-order bits,
so shifting always works. This is the only case in which shifting
might be said to be more accurate than linear scaling.

When comparing pixel values to tRNS chunk values to detect
transparent pixels, it is necessary to do the comparison exactly.
Therefore, transparent pixel detection must be done before
reducing sample precision.

10.5. Decoder gamma handling

See Gamma Tutorial (Chapter 13) if you aren't already familiar
with gamma issues.

To produce correct tone reproduction, a good image display program
should take into account the gammas of the image file and the
display device, as well as the viewing_gamma appropriate to the
lighting conditions near the display. This can be done by
calculating

```
gbright = insample / MAXINSAMPLE
bright = gbright ^ (1.0 / file_gamma)
vbright = bright ^ viewing_gamma
gcvideo = vbright ^ (1.0 / display_gamma)
fbval = ROUND(gcvideo * MAXFBVAL)
```

where MAXINSAMPLE is the maximum sample value in the file (255 for
8-bit, 65535 for 16-bit, etc), MAXFBVAL is the maximum value of a
frame buffer sample (255 for 8-bit, 31 for 5-bit, etc), insample
is the value of the sample in the PNG file, and fbval is the value
to write into the frame buffer. The first line converts from
integer samples into a normalized 0 to 1 floating point value, the
second undoes the gamma encoding of the image file to produce a
linear intensity value, the third adjusts for the viewing
conditions, the fourth corrects for the display system's gamma
value, and the fifth converts to an integer frame buffer sample.
In practice, the second through fourth lines can be merged into

```
gcvideo = gbright^(viewing_gamma / (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 possible sample value. This requires only
256 calculations per image (for 8-bit accuracy), not one or three
calculations per pixel. For an indexed-color image, a one-time
correction of the palette is sufficient, unless the image uses
transparency and is being displayed against a nonuniform
background.

In some cases even the cost of 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 0.5 with some reasonable choice of viewing_gamma and
display_gamma, 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 CRT_gamma of 2.5 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. Gamma Tutorial
(Chapter 13) gives some examples.

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 fourth and fifth lines of the
computation above can be replaced by a lookup in these
measurements, to find the actual frame buffer value that most
nearly gives the desired brightness.

The value of viewing_gamma depends on lighting conditions; see
Gamma Tutorial (Chapter 13) for more detail. Ideally, a viewer
would allow the user to specify viewing_gamma, either directly
numerically, or via selecting from "bright surround", "dim
surround", and "dark surround" conditions. Viewers that don't
want to do this should just assume a value for viewing_gamma of
1.0, since most computer displays live in brightly-lit rooms.

When viewing images that are digitized from video, or that are
destined to become video frames, the user might want to set the
viewing_gamma to about 1.25 regardless of the actual level of room
lighting. This value of viewing_gamma is "built into" NTSC video
practice, and displaying an image with that viewing_gamma allows
the user to see what a TV set would show under the current room
lighting conditions. (This is not the same thing as trying to
obtain the most accurate rendition of the content of the scene,
which would require adjusting viewing_gamma to correspond to the
room lighting level.) This is another reason viewers might want
to allow users to adjust viewing_gamma directly.

10.6. Decoder color handling

See Color Tutorial (Chapter 14) if you aren't already familiar
with color issues.

In many cases, decoders will treat image data in PNG files as
device-dependent RGB data and display it without modification
(except for appropriate gamma correction). This provides the
fastest display of PNG images. But unless the viewer uses exactly
the same display hardware as the original image author used, the
colors will not be exactly the same as the original author saw,
particularly for darker or near-neutral colors. The cHRM chunk
provides information that allows closer color matching than that
provided by gamma correction alone.

Decoders can use the cHRM data to transform the image data from
RGB to XYZ and thence into a perceptually linear color space such
as CIE LAB. They can then partition the colors to generate an
optimal palette, because the geometric distance between two colors
in CIE LAB is strongly related to how different those colors
appear (unlike, for example, RGB or XYZ spaces). The resulting
palette of colors, once transformed back into RGB color space,
could be used for display or written into a PLTE chunk.

Decoders that are part of image processing applications might also
transform image data into CIE LAB space for analysis.

In applications where color fidelity is critical, such as product
design, scientific visualization, medicine, architecture, or
advertising, decoders can transform the image data from source_RGB
to the display_RGB space of the monitor used to view the image.
This involves calculating the matrix to go from source_RGB to XYZ
and the matrix to go from XYZ to display_RGB, then combining them
to produce the overall transformation. The decoder is responsible
for implementing gamut mapping.

Decoders running on platforms that have a Color Management System
(CMS) can pass the image data, gAMA and cHRM values to the CMS for
display or further processing.

Decoders that provide color printing facilities can use the
facilities in Level 2 PostScript to specify image data in
calibrated RGB space or in a device-independent color space such
as XYZ. This will provide better color fidelity than a simple RGB
to CMYK conversion. The PostScript Language Reference manual
gives examples of this process [POSTSCRIPT]. Such decoders are
responsible for implementing gamut mapping between source_RGB
(specified in the cHRM chunk) and the target printer. The
PostScript interpreter is then responsible for producing the
required colors.

Decoders can use the cHRM data to calculate an accurate grayscale
representation of a color image. Conversion from RGB to gray is
simply a case of calculating the Y (luminance) component of XYZ,
which is a weighted sum of the R G and B values. The weights
depend on the monitor type, i.e., the values in the cHRM chunk.
Decoders may wish to do this for PNG files with no cHRM chunk. In
that case, a reasonable default would be the CCIR 709 primaries
[ITU-BT709]. Do not use the original NTSC primaries, unless you
really do have an image color-balanced for such a monitor. Few
monitors ever used the NTSC primaries, so such images are probably
nonexistent these days.

10.7. Background color

The background color given by bKGD 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 will need to make its own decision about a
suitable background color.

Viewers that have a specific background against which to present
the image (such as Web browsers) should ignore the bKGD chunk, in
effect overriding bKGD with their preferred background color or
background image.

The background color given by bKGD is not to be considered
transparent, even if it happens to match the color given by tRNS
(or, in the case of an indexed-color image, refers to a palette
index that is marked as transparent by tRNS). Otherwise one would
have to imagine something "behind the background" to composite
against. The background color is either used as background or
ignored; it is not an intermediate layer between the PNG image and
some other background.

Indeed, it will be common that bKGD and tRNS specify the same
color, since then a decoder that does not implement transparency
processing will give the intended display, at least when no
partially-transparent pixels are present.

10.8. 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 sample 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-encoded) 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 sample depths and gamma values of foreground image, background
image, and frame buffer/CRT all to be different. Don't assume
they are the same without checking.

This code is standard C, with line numbers added for reference in the comments below.

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

     /* Get max sample values in data and frame buffer */
10   fg_maxsample = (1 << fg_sample_depth) - 1;
11   bg_maxsample = (1 << bg_sample_depth) - 1;
12   fb_maxsample = (1 << frame_buffer_sample_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_maxsample) {
         /*
          * Copy foreground pixel to frame buffer.
          */
17       for (i = 0; i < 3; i++) {
18           gamfg = (float) foreground[i] / fg_maxsample;
19           linfg = pow(gamfg, 1.0/fg_gamma);
20           comppix = linfg;
21           gcvideo = pow(comppix,viewing_gamma/display_gamma);
22           fbpix[i] = (int) (gcvideo * fb_maxsample + 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_maxsample;
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_maxsample;
29          linfg = pow(gamfg, 1.0/fg_gamma);
30          gambg = (float) background[i] / bg_maxsample;
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,viewing_gamma/display_gamma);
34          fbpix[i] = (int) (gcvideo * fb_maxsample + 0.5);
35      }
36  }
```

Variations:

* 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 sample value.
     */
    gamout = pow(comppix, outfile_gamma);
    outpix[i] = (int) (gamout * out_maxsample + 0.5);
```

  Also, it becomes necessary to process background pixels when
  alpha is zero, rather than just skipping pixels. Thus, line
  15 will need to be replaced by copies of lines 17-23, but
  processing background instead of foreground pixel values.

       * If the sample depths 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.
       * When the sample depths and gamma values all match, it may
         appear attractive to skip the gamma decoding and encoding
         (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.
       * 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 need
         to extract intensity from the frame buffer pixel values
         using code like

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

         However, some roundoff error can result, so it is better to
         have the original background pixels available if at all
         possible.
       * 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 (different) lookup tables.
       * 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 sample 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.)

Even if the decoder author does not wish to implement true
compositing logic, it is simple to deal with images that contain
only zero and one alpha values. (This is implicitly true for
grayscale and truecolor PNG files that use a tRNS chunk; for
indexed-color PNG files, it is easy to check whether tRNS contains
any values other than 0 and 255.) In this simple case,
transparent pixels are replaced by the background color, while
others are unchanged. If a decoder contains only this much
transparency capability, it should deal with a full alpha channel
by treating all nonzero alpha values as fully opaque; that is, do
not replace partially transparent pixels by the background. This
approach will not yield very good results for images converted
from associated-alpha formats, but it's better than doing nothing.

10.9. 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.

   10.10. Suggested-palette and histogram usage

   In truecolor PNG files, the encoder may have provided a suggested
   PLTE chunk for use by viewers running on indexed-color hardware.

   If the image has a tRNS chunk, the viewer will need to adapt the
   suggested palette for use with its desired background color. To
   do this, replace the palette entry closest to the tRNS color with
   the desired background color; or just add a palette entry for the
   background color, if the viewer can handle more colors than there
   are PLTE entries.

For images of color type 6 (truecolor with alpha channel), any
suggested palette should have been designed for display of the
image against a uniform background of the color specified by bKGD.
Viewers should probably ignore the palette if they intend to use a
different background, or if the bKGD chunk is missing. Viewers
can use a suggested palette for display against a different
background than it was intended for, but the results may not be
very good.

If the viewer presents a transparent truecolor image against a
background that is more complex than a single color, it is
unlikely that the suggested palette will be optimal for the
composite image. In this case it is best to perform a truecolor
compositing step on the truecolor PNG image and background image,
then color-quantize the resulting image.

The histogram chunk is useful when the viewer cannot provide as
many colors as are used in the image's palette. 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 palette or histogram chunk is provided, a decoder can
develop its own, at the cost of an extra pass over the image data.
Alternatively, a default palette (probably a color cube) can be
used.

See also Recommendations for Encoders: Suggested palettes (Section
9.5).

10.11. 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 might be
able to understand it.

PNG text is not supposed to contain any characters outside the ISO
8859-1 "Latin-1" character set (that is, no codes 0-31 or 127-
159), except for the newline character (decimal 10). But decoders
might encounter such characters anyway. Some of these characters
can be safely displayed (e.g., TAB, FF, and CR, decimal 9, 12, and
13, respectively), but others, especially the ESC character
(decimal 27), could pose a security hazard because unexpected

actions may be taken by display hardware or software. To prevent
such hazards, decoders should not attempt to directly display any
non-Latin-1 characters (except for newline and perhaps TAB, FF,
CR) encountered in a tEXt or zTXt chunk. Instead, ignore them or
display them in a visible notation such as "\nnn". See Security
considerations (Section 8.5).

Even though encoders are supposed to represent newlines as LF, it
is recommended that decoders not rely on this; it's best to
recognize all the common newline combinations (CR, LF, and CR-LF)
and display each as a single newline. TAB can be expanded to the
proper number of spaces needed to arrive at a column multiple of
8.

Decoders running on systems with non-Latin-1 character set
encoding should provide character code remapping so that Latin-1
characters are displayed correctly. Some systems may not provide
all the characters defined in Latin-1. Mapping unavailable
characters to a visible notation such as "\nnn" is a good
fallback. In particular, character codes 127-255 should be
displayed only if they are printable characters on the decoding
system. Some systems may interpret such codes as control
characters; for security, decoders running on such systems should
not display such characters literally.

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

11. Glossary

   a^b
      Exponentiation; a raised to the power b. C programmers should be
      careful not to misread this notation as exclusive-or. Note that
      in gamma-related calculations, zero raised to any power is valid
      and must give a zero result.

   Alpha
      A value representing a pixel's degree of transparency. The more
      transparent a pixel, the less it hides the background against
      which the image is presented. In PNG, alpha is really the degree
      of opacity: zero alpha represents a completely transparent pixel,
      maximum alpha represents a completely opaque pixel. But most
      people refer to alpha as providing transparency information, not
      opacity information, and we continue that custom here.

Ancillary chunk
    A chunk that provides additional information. A decoder can still
    produce a meaningful image, though not necessarily the best
    possible image, without processing the chunk.

Bit depth
    The number of bits per palette index (in indexed-color PNGs) or
    per sample (in other color types). This is the same value that
    appears in IHDR.

Byte
    Eight bits; also called an octet.

Channel
    The set of all samples of the same kind within an image; for
    example, all the blue samples in a truecolor image. (The term
    "component" is also used, but not in this specification.) A
    sample is the intersection of a channel and a pixel.

Chromaticity
    A pair of values x,y that precisely specify the hue, though not
    the absolute brightness, of a perceived color.

Chunk
    A section of a PNG file. Each chunk has a type indicated by its
    chunk type name. Most types of chunks also include some data.
    The format and meaning of the data within the chunk are determined
    by the type name.

Composite
    As a verb, to form an image by merging a foreground image and a
    background image, using transparency information to determine
    where the background should be visible. The foreground image is
    said to be "composited against" the background.

CRC
    Cyclic Redundancy Check. A CRC is a type of check value designed
    to catch most transmission errors. A decoder calculates the CRC
    for the received data and compares it to the CRC that the encoder
    calculated, which is appended to the data. A mismatch indicates
    that the data was corrupted in transit.

Critical chunk
    A chunk that must be understood and processed by the decoder in
    order to produce a meaningful image from a PNG file.

CRT
    Cathode Ray Tube: a common type of computer display hardware.

Datastream
    A sequence of bytes. This term is used rather than "file" to
    describe a byte sequence that is only a portion of a file. We
    also use it to emphasize that a PNG image might be generated and
    consumed "on the fly", never appearing in a stored file at all.

Deflate
    The name of the compression algorithm used in standard PNG files,
    as well as in zip, gzip, pkzip, and other compression programs.
    Deflate is a member of the LZ77 family of compression methods.

Filter
    A transformation applied to image data in hopes of improving its
    compressibility. PNG uses only lossless (reversible) filter
    algorithms.

Frame buffer
    The final digital storage area for the image shown by a computer
    display. Software causes an image to appear onscreen by loading
    it into the frame buffer.

Gamma
    The brightness of mid-level tones in an image. More precisely, a
    parameter that describes the shape of the transfer function for
    one or more stages in an imaging pipeline. The transfer function
    is given by the expression

        output = input ^ gamma

    where both input and output are scaled to the range 0 to 1.

Grayscale
    An image representation in which each pixel is represented by a
    single sample value representing overall luminance (on a scale
    from black to white). PNG also permits an alpha sample to be
    stored for each pixel of a grayscale image.

Indexed color
    An image representation in which each pixel is represented by a
    single sample that is an index into a palette or lookup table.
    The selected palette entry defines the actual color of the pixel.

Lossless compression
    Any method of data compression that guarantees the original data
    can be reconstructed exactly, bit-for-bit.

Lossy compression
    Any method of data compression that reconstructs the original data
    approximately, rather than exactly.

LSB
    Least Significant Byte of a multi-byte value.

Luminance
    Perceived brightness, or grayscale level, of a color. Luminance
    and chromaticity together fully define a perceived color.

LUT
    Look Up Table. In general, a table used to transform data. In
    frame buffer hardware, a LUT can be used to map indexed-color
    pixels into a selected set of truecolor values, or to perform
    gamma correction. In software, a LUT can be used as a fast way of
    implementing any one-variable mathematical function.

MSB
    Most Significant Byte of a multi-byte value.

Palette
    The set of colors available in an indexed-color image. In PNG, a
    palette is an array of colors defined by red, green, and blue
    samples. (Alpha values can also be defined for palette entries,
    via the tRNS chunk.)

Pixel
    The information stored for a single grid point in the image. The
    complete image is a rectangular array of pixels.

PNG editor
    A program that modifies a PNG file and preserves ancillary
    information, including chunks that it does not recognize. Such a
    program must obey the rules given in Chunk Ordering Rules (Chapter
    7).

Sample
    A single number in the image data; for example, the red value of a
    pixel. A pixel is composed of one or more samples. When
    discussing physical data layout (in particular, in Image layout,
    Section 2.3), we use "sample" to mean a number stored in the image
    array. It would be more precise but much less readable to say
    "sample or palette index" in that context. Elsewhere in the
    specification, "sample" means a color value or alpha value. In
    the indexed-color case, these are palette entries not palette
    indexes.

Sample depth
   The precision, in bits, of color values and alpha values. In
   indexed-color PNGs the sample depth is always 8 by definition of
   the PLTE chunk. In other color types it is the same as the bit
   depth.

Scanline
   One horizontal row of pixels within an image.

Truecolor
   An image representation in which pixel colors are defined by
   storing three samples for each pixel, representing red, green, and
   blue intensities respectively. PNG also permits an alpha sample
   to be stored for each pixel of a truecolor image.

White point
   The chromaticity of a computer display's nominal white value.

zlib
   A particular format for data that has been compressed using
   deflate-style compression. Also the name of a library
   implementing this method. PNG implementations need not use the
   zlib library, but they must conform to its format for compressed
   data.

12. 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.

12.1. 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.

12.2. 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 online 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 deflate/inflate
   compression is mentioned in this document, PNG would still exist
   without it.

12.3. 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,
           PNG software must support full deflate/inflate; any software
           that does not is not 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 --- 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. For one thing, every vendor
           has a different idea of what the dimensions and
           characteristics of a thumbnail ought to be. Also, some
           vendors keep thumbnails in separate files to accommodate
           varied image formats; they 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.

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. 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. (But see Multiple-image extension, Section 8.4.)

12.4. Why not use format X?

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 their existing software
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 that 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 indexed-color images. Furthermore, its lossless
truecolor compression is often inferior to that of PNG.

   12.5. 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.

   12.6. 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 often 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.
      Preliminary results show that small text in an interlaced PNG
      image is typically readable about twice as fast as in an
      equivalent GIF, i.e., after PNG's fifth pass or 25% of the image
      data, instead of after GIF's third pass or 50%. This is again due
      to PNG's more balanced increase in resolution.

   12.7. Why gamma?

      It might seem natural to standardize on storing sample values that
      are linearly proportional to light intensity (that is, have gamma
      of 1.0). But in fact, it is common for images to have a gamma of
      less than 1. There are three good reasons for this:

         * For reasons detailed in Gamma Tutorial (Chapter 13), all
           video cameras apply a "gamma correction" function to the
           intensity information. This causes the video signal to have
           a gamma of about 0.5 relative to the light intensity in the
           original scene. Thus, images obtained by frame-grabbing
           video already have a gamma of about 0.5.
         * The human eye has a nonlinear response to intensity, so
           linear encoding of samples either wastes sample codes in
           bright areas of the image, or provides too few sample codes
           to avoid banding artifacts in dark areas of the image, or
           both. At least 12 bits per sample are needed to avoid

        visible artifacts in linear encoding with a 100:1 image
        intensity range. An image gamma in the range 0.3 to 0.5
        allocates sample values in a way that roughly corresponds to
        the eye's response, so that 8 bits/sample are enough to
        avoid artifacts caused by insufficient sample precision in
        almost all images. This makes "gamma encoding" a much
        better way of storing digital images than the simpler linear
        encoding.
   *   Many images are created on PCs or workstations with no gamma
        correction hardware and no software willing to provide gamma
        correction either. In these cases, the images have had
        their lighting and color chosen to look best on this
        platform --- they can be thought of as having "manual" gamma
        correction built in. To see what the image author intended,
        it is necessary to treat such images as having a file_gamma
        value in the range 0.4-0.6, depending on the room lighting
        level that the author was working in.

   In practice, image gamma values around 1.0 and around 0.5 are both
   widely found. Older image standards such as GIF often do not
   account for this fact. The JFIF standard specifies that images in
   that format should use linear samples, but many JFIF images found
   on the Internet actually have a gamma somewhere near 0.4 or 0.5.
   The variety of images found and the variety of systems that people
   display them on have led to widespread problems with images
   appearing "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 quantization and
        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 source gamma value. If the
        decoder does gamma correction at viewing time, it can adjust
        the gamma (change the displayed brightness) in response to
        feedback from a human user. The encoder has no such
        recourse.
   *   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
adjustment code anyway. Gamma correction at viewing time is thus
the right way to go.

See Gamma Tutorial (Chapter 13) for more information.

12.8. 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 sample 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 lossless and
more general case.

12.9. 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
indexed-color 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 sample,
so providing pixelwise filtering for such images would be
pointless. For 16 bit/sample 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 defiltering logic for every filter
type anyway. The only cost is an extra byte per scanline in the
pre-compression datastream. 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.

12.10. 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 private 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 textual information be stored in
standard tEXt chunks with suitable keywords. Use of tEXt tells
any PNG viewer that the chunk contains text that might 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 presently no provision for text employing character sets
other than Latin-1. We recognize 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, we expect that text chunks for other
character sets, such as Unicode, eventually will be registered and
increase gradually in popularity.

12.11. 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
    (hexadecimal)            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.

A decoder may further verify that the next eight bytes contain an
IHDR chunk header with the correct chunk length; this will catch
bad transfers that drop or alter null (zero) bytes.

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 explained in Chunk naming conventions (Section 12.13).

12.12. 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 from the CRC so that the CRC can be
   calculated as the data is generated; this avoids a second pass
   over the data in cases where the chunk length is not known in
   advance. 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 set of bytes and then be tested against the wrong
   value from the file.

12.13. 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. We decided against having an
   overall format version number because 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 should
   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 the image
   represents. 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 could be in use somewhere. However, this does not protect
users of private chunk names from the possibility that someone
else may 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 file editor 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 can be made critical
chunks, with consequent loss of portability to other applications.
In general, ancillary chunks can 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 are
insufficient to represent this case, 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 can be useful for other purposes. For example,
if a program relies on the palette being in a particular order, it
can 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.

12.14. 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 could be
   reserved by a window system.) To produce the best results in this
   situation, it is helpful to have information about the frequency
   with which each palette index actually appears, in order to choose
   the best palette for dithering or to 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.

   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
   needs to 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.

13. Appendix: Gamma Tutorial

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

   It would be convenient for graphics programmers if all of the
   components of an imaging system were linear. The voltage coming from
   an electronic camera would be directly proportional to the intensity
   (power) of light in the scene, the light emitted by a CRT would be
   directly proportional to its input voltage, and so on. However,
   real-world devices do not behave in this way. All CRT displays,
   almost all photographic film, and many electronic cameras have
   nonlinear signal-to-light-intensity or intensity-to-signal
   characteristics.

   Fortunately, all of these nonlinear devices have a transfer function
   that is approximated fairly well by a single type of mathematical
   function: a power function. This power function has the general
   equation

       output = input ^ gamma

   where ^ denotes exponentiation, and "gamma" (often printed using the
   Greek letter gamma, thus the name) is simply the exponent of the
   power function.

   By convention, "input" and "output" are both scaled to the range
   0..1, with 0 representing black and 1 representing maximum white (or
   red, etc). Normalized in this way, the power function is completely
   described by a single number, the exponent "gamma".

   So, given a particular device, we can measure its output as a
   function of its input, fit a power function to this measured transfer
   function, extract the exponent, and call it gamma. We often say
   "this device has a gamma of 2.5" as a shorthand for "this device has
   a power-law response with an exponent of 2.5". We can also talk
   about the gamma of a mathematical transform, or of a lookup table in
   a frame buffer, so long as the input and output of the thing are
   related by the power-law expression above.

   How do gammas combine?

      Real imaging systems will have several components, and more than
      one of these can be nonlinear. If all of the components have
      transfer characteristics that are power functions, then the
      transfer function of the entire system is also a power function.
      The exponent (gamma) of the whole system's transfer function is
      just the product of all of the individual exponents (gammas) of
      the separate stages in the system.

Also, stages that are linear pose no problem, since a power
function with an exponent of 1.0 is really a linear function. So
a linear transfer function is just a special case of a power
function, with a gamma of 1.0.

Thus, as long as our imaging system contains only stages with
linear and power-law transfer functions, we can meaningfully talk
about the gamma of the entire system. This is indeed the case
with most real imaging systems.

What should overall gamma be?

If the overall gamma of an imaging system is 1.0, its output is
linearly proportional to its input. This means that the ratio
between the intensities of any two areas in the reproduced image
will be the same as it was in the original scene. It might seem
that this should always be the goal of an imaging system: to
accurately reproduce the tones of the original scene. Alas, that
is not the case.

When the reproduced image is to be viewed in "bright surround"
conditions, where other white objects nearby in the room have
about the same brightness as white in the image, then an overall
gamma of 1.0 does indeed give real-looking reproduction of a
natural scene. Photographic prints viewed under room light and
computer displays in bright room light are typical "bright
surround" viewing conditions.

However, sometimes images are intended to be viewed in "dark
surround" conditions, where the room is substantially black except
for the image. This is typical of the way movies and slides
(transparencies) are viewed by projection. Under these
circumstances, an accurate reproduction of the original scene
results in an image that human viewers judge as "flat" and lacking
in contrast. It turns out that the projected image needs to have
a gamma of about 1.5 relative to the original scene for viewers to
judge it "natural". Thus, slide film is designed to have a gamma
of about 1.5, not 1.0.

There is also an intermediate condition called "dim surround",
where the rest of the room is still visible to the viewer, but is
noticeably darker than the reproduced image itself. This is
typical of television viewing, at least in the evening, as well as
subdued-light computer work areas. In dim surround conditions,
the reproduced image needs to have a gamma of about 1.25 relative
to the original scene in order to look natural.

The requirement for boosted contrast (gamma) in dark surround
conditions is due to the way the human visual system works, and
applies equally well to computer monitors. Thus, a PNG viewer
trying to achieve the maximum realism for the images it displays
really needs to know what the room lighting conditions are, and
adjust the gamma of the displayed image accordingly.

If asking the user about room lighting conditions is inappropriate
or too difficult, just assume that the overall gamma
(viewing_gamma as defined below) should be 1.0 or 1.25. That's
all that most systems that implement gamma correction do.

What is a CRT's gamma?

All CRT displays have a power-law transfer characteristic with a
gamma of about 2.5. This is due to the physical processes
involved in controlling the electron beam in the electron gun, and
has nothing to do with the phosphor.

An exception to this rule is fancy "calibrated" CRTs that have
internal electronics to alter their transfer function. If you
have one of these, you probably should believe what the
manufacturer tells you its gamma is. But in all other cases,
assuming 2.5 is likely to be pretty accurate.

There are various images around that purport to measure gamma,
usually by comparing the intensity of an area containing
alternating white and black with a series of areas of continuous
gray of different intensity. These are usually not reliable.
Test images that use a "checkerboard" pattern of black and white
are the worst, because a single white pixel will be reproduced
considerably darker than a large area of white. An image that
uses alternating black and white horizontal lines (such as the
"gamma.png" test image at
ftp://ftp.uu.net/graphics/png/images/suite/gamma.png) is much
better, but even it may be inaccurate at high "picture" settings
on some CRTs.

If you have a good photometer, you can measure the actual light
output of a CRT as a function of input voltage and fit a power
function to the measurements. However, note that this procedure
is very sensitive to the CRT's black level adjustment, somewhat
sensitive to its picture adjustment, and also affected by ambient
light. Furthermore, CRTs spread some light from bright areas of
an image into nearby darker areas; a single bright spot against a
black background may be seen to have a "halo". Your measuring
technique will need to minimize the effects of this.

Because of the difficulty of measuring gamma, using either test
images or measuring equipment, you're usually better off just
assuming gamma is 2.5 rather than trying to measure it.

What is gamma correction?

A CRT has a gamma of 2.5, and we can't change that. To get an
overall gamma of 1.0 (or somewhere near that) for an imaging
system, we need to have at least one other component of the "image
pipeline" that is nonlinear. If, in fact, there is only one
nonlinear stage in addition to the CRT, then it's traditional to
say that the CRT has a certain gamma, and that the other nonlinear
stage provides "gamma correction" to compensate for the CRT.
However, exactly where the "correction" is done depends on
circumstance.

In all broadcast video systems, gamma correction is done in the
camera. This choice was made in the days when television
electronics were all analog, and a good gamma-correction circuit
was expensive to build. The original NTSC video standard required
cameras to have a transfer function with a gamma of 1/2.2, or
about 0.45. Recently, a more complex two-part transfer function
has been adopted [SMPTE-170M], but its behavior can be well
approximated by a power function with a gamma of 0.5. When the
resulting image is displayed on a CRT with a gamma of 2.5, the
image on screen ends up with a gamma of about 1.25 relative to the
original scene, which is appropriate for "dim surround" viewing.

These days, video signals are often digitized and stored in
computer frame buffers. This works fine, but remember that gamma
correction is "built into" the video signal, and so the digitized
video has a gamma of about 0.5 relative to the original scene.

Computer rendering programs often produce linear samples. To
display these correctly, intensity on the CRT needs to be directly
proportional to the sample values in the frame buffer. This can
be done with a special hardware lookup table between the frame
buffer and the CRT hardware. The lookup table (often called LUT)
is loaded with a mapping that implements a power function with a
gamma of 0.4, thus providing "gamma correction" for the CRT gamma.

Thus, gamma correction sometimes happens before the frame buffer,
sometimes after. As long as images created in a particular
environment are always displayed in that environment, everything
is fine. But when people try to exchange images, differences in
gamma correction conventions often result in images that seem far
too bright and washed out, or far too dark and contrasty.

Gamma-encoded samples are good

   So, is it better to do gamma correction before or after the frame
   buffer?

   In an ideal world, sample values would be stored in floating
   point, there would be lots of precision, and it wouldn't really
   matter much. But in reality, we're always trying to store images
   in as few bits as we can.

   If we decide to use samples that are linearly proportional to
   intensity, and do the gamma correction in the frame buffer LUT, it
   turns out that we need to use at least 12 bits for each of red,
   green, and blue to have enough precision in intensity. With any
   less than that, we will sometimes see "contour bands" or "Mach
   bands" in the darker areas of the image, where two adjacent sample
   values are still far enough apart in intensity for the difference
   to be visible.

   However, through an interesting coincidence, the human eye's
   subjective perception of brightness is related to the physical
   stimulation of light intensity in a manner that is very much like
   the power function used for gamma correction. If we apply gamma
   correction to measured (or calculated) light intensity before
   quantizing to an integer for storage in a frame buffer, we can get
   away with using many fewer bits to store the image. In fact, 8
   bits per color is almost always sufficient to avoid contouring
   artifacts. This is because, since gamma correction is so closely
   related to human perception, we are assigning our 256 available
   sample codes to intensity values in a manner that approximates how
   visible those intensity changes are to the eye. Compared to a
   linear-sample image, we allocate fewer sample values to brighter
   parts of the tonal range and more sample values to the darker
   portions of the tonal range.

   Thus, for the same apparent image quality, images using gamma-
   encoded sample values need only about two-thirds as many bits of
   storage as images using linear samples.

General gamma handling

When more than two nonlinear transfer functions are involved in
the image pipeline, the term "gamma correction" becomes too vague.
If we consider a pipeline that involves capturing (or calculating)
an image, storing it in an image file, reading the file, and
displaying the image on some sort of display screen, there are at
least 5 places in the pipeline that could have nonlinear transfer
functions. Let's give each a specific name for their
characteristic gamma:

camera_gamma
    the characteristic of the image sensor

encoding_gamma
    the gamma of any transformation performed by the software
    writing the image file

decoding_gamma
    the gamma of any transformation performed by the software
    reading the image file

LUT_gamma
    the gamma of the frame buffer LUT, if present

CRT_gamma
    the gamma of the CRT, generally 2.5

In addition, let's add a few other names:

file_gamma
    the gamma of the image in the file, relative to the original
    scene. This is

        file_gamma = camera_gamma * encoding_gamma

display_gamma
    the gamma of the "display system" downstream of the frame
    buffer. This is

        display_gamma = LUT_gamma * CRT_gamma

viewing_gamma
    the overall gamma that we want to obtain to produce pleasing
    images --- generally 1.0 to 1.5.

The file_gamma value, as defined above, is what goes in the gAMA
chunk in a PNG file. If file_gamma is not 1.0, we know that gamma
correction has been done on the sample values in the file, and we
could call them "gamma corrected" samples. However, since there
can be so many different values of gamma in the image display
chain, and some of them are not known at the time the image is
written, the samples are not really being "corrected" for a
specific display condition. We are really using a power function
in the process of encoding an intensity range into a small integer
field, and so it is more correct to say "gamma encoded" samples
instead of "gamma corrected" samples.

When displaying an image file, the image decoding program is
responsible for making the overall gamma of the system equal to
the desired viewing_gamma, by selecting the decoding_gamma
appropriately. When displaying a PNG file, the gAMA chunk
provides the file_gamma value. The display_gamma may be known for
this machine, or it might be obtained from the system software, or
the user might have to be asked what it is. The correct
viewing_gamma depends on lighting conditions, and that will
generally have to come from the user.

Ultimately, you should have

   file_gamma * decoding_gamma * display_gamma = viewing_gamma

Some specific examples

In digital video systems, camera_gamma is about 0.5 by declaration
of the various video standards documents. CRT_gamma is 2.5 as
usual, while encoding_gamma, decoding_gamma, and LUT_gamma are all
1.0. As a result, viewing_gamma ends up being about 1.25.

On frame buffers that have hardware gamma correction tables, and
that are calibrated to display linear samples correctly,
display_gamma is 1.0.

Many workstations and X terminals and PC displays lack gamma
correction lookup tables. Here, LUT_gamma is always 1.0, so
display_gamma is 2.5.

On the Macintosh, there is a LUT. By default, it is loaded with a
table whose gamma is about 0.72, giving a display_gamma (LUT and
CRT combined) of about 1.8. Some Macs have a "Gamma" control
panel that allows gamma to be changed to 1.0, 1.2, 1.4, 1.8, or
2.2. These settings load alternate LUTs that are designed to give
a display_gamma that is equal to the label on the selected button.
Thus, the "Gamma" control panel setting can be used directly as
display_gamma in decoder calculations.

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 it 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 = 2.5 / 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.

A note about video gamma

The original NTSC video standards specified a simple power-law
camera transfer function with a gamma of 1/2.2 or 0.45. This is
not possible to implement exactly in analog hardware because the
function has infinite slope at x=0, so all cameras deviated to
some degree from this ideal. More recently, a new camera transfer
function that is physically realizable has been accepted as a
standard [SMPTE-170M]. It is

    Vout = 4.5 * Vin   if Vin < 0.018
    Vout = 1.099 * (Vin^0.45) - 0.099   if Vin >= 0.018

where Vin and Vout are measured on a scale of 0 to 1. Although
the exponent remains 0.45, the multiplication and subtraction
change the shape of the transfer function, so it is no longer a
pure power function. If you want to perform extremely precise
calculations on video signals, you should use the expression above
(or its inverse, as required).

However, PNG does not provide a way to specify that an image uses
this exact transfer function; the gAMA chunk always assumes a pure
power-law function. If we plot the two-part transfer function
above along with the family of pure power functions, we find that
a power function with a gamma of about 0.5 to 0.52 (not 0.45) most
closely approximates the transfer function. Thus, when writing a

PNG file with data obtained from digitizing the output of a modern
video camera, the gAMA chunk should contain 0.5 or 0.52, not 0.45.
The remaining difference between the true transfer function and
the power function is insignificant for almost all purposes. (In
fact, the alignment errors in most cameras are likely to be larger
than the difference between these functions.) The designers of
PNG deemed the simplicity and flexibility of a power-law
definition of gAMA to be more important than being able to
describe the SMPTE-170M transfer curve exactly.

The PAL and SECAM video standards specify a power-law camera
transfer function with a gamma of 1/2.8 or 0.36 --- not the 1/2.2
of NTSC. However, this is too low in practice, so real cameras
are likely to have their gamma set close to NTSC practice. Just
guessing 0.45 or 0.5 is likely to give you viewable results, but
if you want precise values you'll probably have to measure the
particular camera.

   Further reading

If you have access to the World Wide Web, read Charles Poynton's
excellent "Gamma FAQ" [GAMMA-FAQ] for more information about
gamma.

14. Appendix: Color Tutorial

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

   About chromaticity

The cHRM chunk is used, together with the gAMA chunk, to convey
precise color information so that a PNG image can be displayed or
printed with better color fidelity than is possible without this
information. The preceding chapters state how this information is
encoded in a PNG image. This tutorial briefly outlines the
underlying color theory for those who might not be familiar with
it.

Note that displaying an image with incorrect gamma will produce
much larger color errors than failing to use the chromaticity
data. First be sure the monitor set-up and gamma correction are
right, then worry about chromaticity.

   The problem

The color of an object depends not only on the precise spectrum of
light emitted or reflected from it, but also on the observer ---
their species, what else they can see at the same time, even what

they have recently looked at! Furthermore, two very different
spectra can produce exactly the same color sensation. Color is
not an objective property of real-world objects; it is a
subjective, biological sensation. However, by making some
simplifying assumptions (such as: we are talking about human
vision) it is possible to produce a mathematical model of color
and thereby obtain good color accuracy.

   Device-dependent color

      Display the same RGB data on three different monitors, side by
      side, and you will get a noticeably different color balance on
      each display. This is because each monitor emits a slightly
      different shade and intensity of red, green, and blue light. RGB
      is an example of a device-dependent color model --- the color you
      get depends on the device. This also means that a particular
      color --- represented as say RGB 87, 146, 116 on one monitor ---
      might have to be specified as RGB 98, 123, 104 on another to
      produce the same color.

   Device-independent color

      A full physical description of a color would require specifying
      the exact spectral power distribution of the light source.
      Fortunately, the human eye and brain are not so sensitive as to
      require exact reproduction of a spectrum. Mathematical, device-
      independent color models exist that describe fairly well how a
      particular color will be seen by humans. The most important
      device-independent color model, to which all others can be
      related, was developed by the International Lighting Committee
      (CIE, in French) and is called XYZ.

      In XYZ, X is the sum of a weighted power distribution over the
      whole visible spectrum. So are Y and Z, each with different
      weights. Thus any arbitrary spectral power distribution is
      condensed down to just three floating point numbers. The weights
      were derived from color matching experiments done on human
      subjects in the 1920s. CIE XYZ has been an International Standard
      since 1931, and it has a number of useful properties:

         * two colors with the same XYZ values will look the same to
           humans
         * two colors with different XYZ values will not look the same
         * the Y value represents all the brightness information
           (luminance)
         * the XYZ color of any object can be objectively measured

Color models based on XYZ have been used for many years by people
who need accurate control of color --- lighting engineers for film
and TV, paint and dyestuffs manufacturers, and so on. They are
thus proven in industrial use. Accurate, device-independent color
started to spread from high-end, specialized areas into the
mainstream during the late 1980s and early 1990s, and PNG takes
notice of that trend.

Calibrated, device-dependent color

Traditionally, image file formats have used uncalibrated, device-
dependent color. If the precise details of the original display
device are known, it becomes possible to convert the device-
dependent colors of a particular image to device-independent ones.
Making simplifying assumptions, such as working with CRTs (which
are much easier than printers), all we need to know are the XYZ
values of each primary color and the CRT_gamma.

So why does PNG not store images in XYZ instead of RGB? Well, two
reasons. First, storing images in XYZ would require more bits of
precision, which would make the files bigger. Second, all
programs would have to convert the image data before viewing it.
Whether calibrated or not, all variants of RGB are close enough
that undemanding viewers can get by with simply displaying the
data without color correction. By storing calibrated RGB, PNG
retains compatibility with existing programs that expect RGB data,
yet provides enough information for conversion to XYZ in
applications that need precise colors. Thus, we get the best of
both worlds.

What are chromaticity and luminance?

Chromaticity is an objective measurement of the color of an
object, leaving aside the brightness information. Chromaticity
uses two parameters x and y, which are readily calculated from
XYZ:

    x = X / (X + Y + Z)
    y = Y / (X + Y + Z)

XYZ colors having the same chromaticity values will appear to have
the same hue but can vary in absolute brightness. Notice that x,y
are dimensionless ratios, so they have the same values no matter
what units we've used for X,Y,Z.

The Y value of an XYZ color is directly proportional to its
absolute brightness and is called the luminance of the color. We
can describe a color either by XYZ coordinates or by chromaticity
x,y plus luminance Y. The XYZ form has the advantage that it is
linearly related to (linear, gamma=1.0) RGB color spaces.

How are computer monitor colors described?

The "white point" of a monitor is the chromaticity x,y of the
monitor's nominal white, that is, the color produced when
R=G=B=maximum.

It's customary to specify monitor colors by giving the
chromaticities of the individual phosphors R, G, and B, plus the
white point. The white point allows one to infer the relative
brightnesses of the three phosphors, which isn't determined by
their chromaticities alone.

Note that the absolute brightness of the monitor is not specified.
For computer graphics work, we generally don't care very much
about absolute brightness levels. Instead of dealing with
absolute XYZ values (in which X,Y,Z are expressed in physical
units of radiated power, such as candelas per square meter), it is
convenient to work in "relative XYZ" units, where the monitor's
nominal white is taken to have a luminance (Y) of 1.0. Given this
assumption, it's simple to compute XYZ coordinates for the
monitor's white, red, green, and blue from their chromaticity
values.

Why does cHRM use x,y rather than XYZ? Simply because that is how
manufacturers print the information in their spec sheets!
Usually, the first thing a program will do is convert the cHRM
chromaticities into relative XYZ space.

What can I do with it?

If a PNG file has the gAMA and cHRM chunks, the source_RGB values
can be converted to XYZ. This lets you:

   * do accurate grayscale conversion (just use the Y component)
   * convert to RGB for your own monitor (to see the original
     colors)
   * print the image in Level 2 PostScript with better color
     fidelity than a simple RGB to CMYK conversion could provide
   * calculate an optimal color palette
   * pass the image data to a color management system
   * etc.

How do I convert from source_RGB to XYZ?

   Make a few simplifying assumptions first, like the monitor really
   is jet black with no input and the guns don't interfere with one
   another. Then, given that you know the CIE XYZ values for each of
   red, green, and blue for a particular monitor, you put them into a
   matrix m:

```
              Xr Xg Xb
        m =   Yr Yg Yb
              Zr Zg Zb
```

   Here we assume we are working with linear RGB floating point data
   in the range 0..1. If the gamma is not 1.0, make it so on the
   floating point data. Then convert source_RGB to XYZ by matrix
   multiplication:

```
        X       R
        Y = m   G
        Z       B
```

   In other words, $X = Xr*R + Xg*G + Xb*B$, and similarly for Y and Z.
   You can go the other way too:

```
        R        X
        G = im   Y
        B        Z
```

   where im is the inverse of the matrix m.

What is a gamut?

   The gamut of a device is the subset of visible colors which that
   device can display. (It has nothing to do with gamma.) The gamut
   of an RGB device can be visualized as a polyhedron in XYZ space;
   the vertices correspond to the device's black, blue, red, green,
   magenta, cyan, yellow and white.

   Different devices have different gamuts, in other words one device
   will be able to display certain colors (usually highly saturated
   ones) that another device cannot. The gamut of a particular RGB
   device can be determined from its R, G, and B chromaticities and
   white point (the same values given in the cHRM chunk). The gamut
   of a color printer is more complex and can only be determined by
   measurement. However, printer gamuts are typically smaller than
   monitor gamuts, meaning that there can be many colors in a
   displayable image that cannot physically be printed.

Converting image data from one device to another generally results
in gamut mismatches --- colors that cannot be represented exactly
on the destination device. The process of making the colors fit,
which can range from a simple clip to elaborate nonlinear scaling
transformations, is termed gamut mapping. The aim is to produce a
reasonable visual representation of the original image.

Further reading

References [COLOR-1] through [COLOR-5] provide more detail about
color theory.

15. Appendix: Sample CRC Code

The following sample code represents a practical implementation of
the CRC (Cyclic Redundancy Check) employed in PNG chunks. (See also
ISO 3309 [ISO-3309] or ITU-T V.42 [ITU-V42] for a formal
specification.)

The sample code is in the ANSI C programming language. Non C users
may find it easier to read with these hints:

&
    Bitwise AND operator.

^

    Bitwise exclusive-OR operator. (Caution: elsewhere in this
    document, ^ represents exponentiation.)

>>
    Bitwise right shift operator. When applied to an unsigned
    quantity, as here, right shift inserts zeroes at the left.

!
    Logical NOT operator.

++
    "n++" increments the variable n.

0xNNN
    0x introduces a hexadecimal (base 16) constant. Suffix L
    indicates a long value (at least 32 bits).

    /* Table of CRCs 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++) {
            if (c & 1)
              c = 0xedb88320L ^ (c >> 1);
            else
              c = 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;
        int n;

        if (!crc_table_computed)
          make_crc_table();
        for (n = 0; n < len; n++) {
          c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
        }
        return c;
      }

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

16. Appendix: Online Resources

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

   This appendix gives the locations of some Internet resources for PNG
   software developers. By the nature of the Internet, the list is
   incomplete and subject to change.

   Archive sites

      The latest released versions of this document and related
      information can always be found at the PNG FTP archive site,
      ftp://ftp.uu.net/graphics/png/. The PNG specification is
      available in several formats, including HTML, plain text, and
      PostScript.

   Reference implementation and test images

      A reference implementation in portable C is available from the PNG
      FTP archive site, ftp://ftp.uu.net/graphics/png/src/. The
      reference implementation is freely usable in all applications,
      including commercial applications.

      Test images are available from
      ftp://ftp.uu.net/graphics/png/images/.

   Electronic mail

      The maintainers of the PNG specification can be contacted by e-
      mail at png-info@uunet.uu.net or at png-group@w3.org.

   PNG home page

      There is a World Wide Web home page for PNG at
      http://quest.jpl.nasa.gov/PNG/. This page is a central location
      for current information about PNG and PNG-related tools.

17. Appendix: Revision History

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

   The PNG format has been frozen since the Ninth Draft of March 7,
   1995, and all future changes are intended to be backwards compatible.
   The revisions since the Ninth Draft are simply clarifications,
   improvements in presentation, and additions of supporting material.

   On 1 October 1996, the PNG specification was approved as a W3C (World
   Wide Web Consortium) Recommendation.

   Changes since the Tenth Draft of 5 May, 1995

            * Clarified meaning of a suggested-palette PLTE chunk in a
              truecolor image that uses transparency
            * Clarified exact semantics of sBIT and allowed sample depth
              scaling procedures
            * Clarified status of spaces in tEXt chunk keywords
            * Distinguished private and public extension values in type
              and method fields
            * Added a "Creation Time" tEXt keyword
            * Macintosh representation of PNG specified
            * Added discussion of security issues
            * Added more extensive discussion of gamma and chromaticity
              handling, including tutorial appendixes
            * Clarified terminology, notably sample depth vs. bit depth
            * Added a glossary
            * Editing and reformatting

18. References

   [COLOR-1]
      Hall, Roy, Illumination and Color in Computer Generated Imagery.
      Springer-Verlag, New York, 1989. ISBN 0-387-96774-5.

   [COLOR-2]
      Kasson, J., and W. Plouffe, "An Analysis of Selected Computer
      Interchange Color Spaces", ACM Transactions on Graphics, vol 11 no
      4 (1992), pp 373-405.

   [COLOR-3]
      Lilley, C., F. Lin, W.T. Hewitt, and T.L.J. Howard, Colour in
      Computer Graphics. CVCP, Sheffield, 1993. ISBN 1-85889-022-5.
      Also available from
      <URL:http://info.mcc.ac.uk/CGU/ITTI/Col/colour_announce.html>

   [COLOR-4]
      Stone, M.C., W.B. Cowan, and J.C. Beatty, "Color gamut mapping and
      the printing of digital images", ACM Transactions on Graphics, vol
      7 no 3 (1988), pp 249-292.

   [COLOR-5]
      Travis, David, Effective Color Displays --- Theory and Practice.
      Academic Press, London, 1991. ISBN 0-12-697690-2.

   [GAMMA-FAQ]
      Poynton, C., "Gamma FAQ".
      <URL:http://www.inforamp.net/%7Epoynton/Poynton-colour.html>

   [ISO-3309]
      International Organization for Standardization, "Information
      Processing Systems --- Data Communication High-Level Data Link
      Control Procedure --- Frame Structure", IS 3309, October 1984, 3rd
      Edition.

   [ISO-8859]
      International Organization for Standardization, "Information
      Processing --- 8-bit Single-Byte Coded Graphic Character Sets ---
      Part 1: Latin Alphabet No. 1", IS 8859-1, 1987.
      Also see sample files at
      ftp://ftp.uu.net/graphics/png/documents/iso_8859-1.*

   [ITU-BT709]
      International Telecommunications Union, "Basic Parameter Values
      for the HDTV Standard for the Studio and for International
      Programme Exchange", ITU-R Recommendation BT.709 (formerly CCIR
      Rec. 709), 1990.

   [ITU-V42]
      International Telecommunications Union, "Error-correcting
      Procedures for DCEs Using Asynchronous-to-Synchronous Conversion",
      ITU-T Recommendation V.42, 1994, Rev. 1.

   [PAETH]
      Paeth, A.W., "Image File Compression Made Easy", in Graphics Gems
      II, James Arvo, editor. Academic Press, San Diego, 1991. ISBN
      0-12-064480-0.

   [POSTSCRIPT]
      Adobe Systems Incorporated, PostScript Language Reference Manual,
      2nd edition. Addison-Wesley, Reading, 1990. ISBN 0-201-18127-4.

   [PNG-EXTENSIONS]
      PNG Group, "PNG Special-Purpose Public Chunks". Available in
      several formats from
      ftp://ftp.uu.net/graphics/png/documents/pngextensions.*

   [RFC-1123]
      Braden, R., Editor, "Requirements for Internet Hosts ---
      Application and Support", STD 3, RFC 1123, USC/Information
      Sciences Institute, October 1989.
      <URL:ftp://ds.internic.net/rfc/rfc1123.txt>

   [RFC-2045]
      Freed, N., and N. Borenstein, "Multipurpose Internet Mail
      Extensions (MIME) Part One: Format of Internet Message Bodies",
      RFC 2045, Innosoft, First Virtual, November 1996.
      <URL:ftp://ds.internic.net/rfc/rfc2045.txt>

   [RFC-2048]
      Freed, N., Klensin, J., and J. Postel, "Multipurpose Internet Mail
      Extensions (MIME) Part Four: Registration Procedures", RFC 2048,
      Innosoft, MCI, USC/Information Sciences Institute, November 1996.
      <URL:ftp://ds.internic.net/rfc/rfc2048.txt>

   [RFC-1950]
      Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format
      Specification version 3.3", RFC 1950, Aladdin Enterprises, May
      1996.
      <URL:ftp://ds.internic.net/rfc/rfc1950.txt>

   [RFC-1951]
      Deutsch, P., "DEFLATE Compressed Data Format Specification version
      1.3", RFC 1951, Aladdin Enterprises, May 1996.
      <URL:ftp://ds.internic.net/rfc/rfc1951.txt>

   [SMPTE-170M]
      Society of Motion Picture and Television Engineers, "Television
      --- Composite Analog Video Signal --- NTSC for Studio
      Applications", SMPTE-170M, 1994.

19. Credits

   Editor

      Thomas Boutell, boutell@boutell.com

   Contributing Editor

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

   Authors

      Authors' names are presented in alphabetical order.

            * Mark Adler, madler@alumni.caltech.edu
            * Thomas Boutell, boutell@boutell.com
            * Christian Brunschen, cb@df.lth.se
            * Adam M. Costello, amc@cs.berkeley.edu
            * Lee Daniel Crocker, lee@piclab.com
            * Andreas Dilger, adilger@enel.ucalgary.ca
            * Oliver Fromme, fromme@rz.tu-clausthal.de
            * Jean-loup Gailly, gzip@prep.ai.mit.edu
            * Chris Herborth, chrish@qnx.com
            * Alex Jakulin, Aleks.Jakulin@snet.fri.uni-lj.si
            * Neal Kettler, kettler@cs.colostate.edu
            * Tom Lane, tgl@sss.pgh.pa.us
            * Alexander Lehmann, alex@hal.rhein-main.de
            * Chris Lilley, chris@w3.org
            * Dave Martindale, davem@cs.ubc.ca
            * Owen Mortensen, 104707.650@compuserve.com
            * Keith S. Pickens, ksp@swri.edu
            * Robert P. Poole, lionboy@primenet.com
            * Glenn Randers-Pehrson, glennrp@arl.mil or
              randeg@alumni.rpi.edu
            * Greg Roelofs, newt@pobox.com
            * Willem van Schaik, willem@gintic.gov.sg
            * Guy Schalnat
            * Paul Schmidt, pschmidt@photodex.com
            * Tim Wegner, 71320.675@compuserve.com
            * Jeremy Wohl, jeremyw@anders.com

      The authors wish to acknowledge the contributions of the Portable
      Network Graphics mailing list, the readers of comp.graphics, and
      the members of the World Wide Web Consortium (W3C).

The Adam7 interlacing scheme is not patented and it is not the
intention of the originator of that scheme to patent it. The
scheme may be freely used by all PNG implementations. The name
"Adam7" may be freely used to describe interlace method 1 of the
PNG specification.

Trademarks

GIF is a service mark of CompuServe Incorporated. IBM PC is a
trademark of International Business Machines Corporation.
Macintosh is a trademark of Apple Computer, Inc. Microsoft and
MS-DOS are trademarks of Microsoft Corporation. PhotoCD is a
trademark of Eastman Kodak Company. PostScript and TIFF are
trademarks of Adobe Systems Incorporated. SGI is a trademark of
Silicon Graphics, Inc. X Window System is a trademark of the
Massachusetts Institute of Technology.

The name and trademarks of copyright holders may NOT be used in
advertising or publicity pertaining to the specification without
specific, written prior permission. Title to copyright in this
specification and any associated documentation will at all times
remain with copyright holders.

Security Considerations

Security issues are discussed in Security considerations (Section
8.5).

Author's Address

Thomas Boutell
PO Box 20837
Seattle, WA 98102

Phone: (206) 329-4969
EMail: boutell@boutell.com