Extensions to the PNG 1.1 Specification, Version 1.1.0

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

Abstract

This document is an extension to the Portable Network Graphics (PNG) specification, version 1.1. It describes additional public chunk types and contains additional information for use in PNG images.

This document, together with the PNG specification, contains the entire list of registered "public" PNG chunks. The additional registered chunks appearing in this document are the oFFs, pCAL, sCAL, gIFg, gIFs, and fRAc chunks, plus the deprecated gIFt chunk. Additional chunk types may be proposed for inclusion in this list by contacting the PNG specification maintainers at png-info@uunet.uu.net. Chunks described here are expected to be less widely supported than those defined in the basic specification. However, application authors are encouraged to use these chunk types whenever appropriate for their applications.

This document also describes data representations that do not occur in the core PNG format, but are used in one or more special-purpose chunks. New chunks should use these representations whenever applicable, in order to maximize portability and simplify decoders.

Table of Contents

1 Data Representation

1.1 Integer values

Refer to Section 2.1 of the PNG specification for the format and range of integer values.

1.2 Floating-point values

The core of PNG does not use floating-point numbers anywhere; it uses integers or, where applicable, fixed-point fractional values. However, special-purpose chunks may need to represent values that do not fit comfortably in fixed-point notation. The textual floating-point notation defined here is recommended for use in all such cases. This representation is simple, has no a priori limits on range or precision, and is portable across all machines.

A floating-point value in this notation is represented by an ASCII text string in a standardized decimal floating-point format. The string is variable-length and must be terminated by a null (zero) character unless it is the last item in its chunk. The string consists of an optional sign ("+" or "-"), an integer part, a fraction part beginning with a decimal point ("."), and an exponent part beginning with an "E" or "e" and optional sign. The integer, fraction, and exponent parts each contain one or more digits (ASCII "0" to "9"). Either the integer part or the fraction part, but not both, may be omitted. A decimal point is allowed, but not required, if there is no fraction part. The exponent part may be omitted. No spaces or any other character besides those specified may appear.

Note in particular that C-language "F" and "L" suffixes are not allowed, the string "." is not allowed as a shorthand for 0 as in some other programming languages, and no commas or underscores are allowed. This format ought to be easily readable in all programming environments.

2 Summary of Special-Purpose Chunks

This table summarizes some properties of the chunks described in this document.

```
Name  Multiple  Ordering constraints
        OK?

oFFs    No      Before IDAT
pCAL    No      Before IDAT
sCAL    No      Before IDAT
gIFg    Yes     None
gIFt    Yes     None (this chunk is deprecated)
gIFx    Yes     None
fRAc    Yes     None
```

3 Chunk Descriptions

3.1 oFFs Image offset

The oFFs chunk gives the position on a printed page at which the image should be output when printed alone. It can also be used to define the image's location with respect to a larger screen or other application-specific coordinate system.

The oFFs chunk contains:

```
X position:     4 bytes (signed integer)
Y position:     4 bytes (signed integer)
Unit specifier: 1 byte
```

Both position values are signed. The following values are legal for the unit specifier:

```
0: unit is the pixel (true dimensions unspecified)
1: unit is the micrometer
```

Conversion note: one inch is equal to exactly 25400 micrometers. A micrometer (also called a micron) is `10^-6` meter.

The X position is measured rightwards from the left edge of the page to the left edge of the image; the Y position is measured downwards from the top edge of the page to the top edge of the image. Note that negative values are permitted, and denote displacement in the opposite directions. Although oFFs can specify an image placement that is partially or wholly outside the page boundaries, the result of such placement is application-dependent.

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

3.2 pCAL Calibration of pixel values

When a PNG file is being used to store physical data other than color values, such as a two-dimensional temperature field, the pCAL chunk can be used to record the relationship (mapping) between stored pixel samples, original samples, and actual physical values. The pCAL data might be used to construct a reference color bar beside the image, or to extract the original physical data values from the file. It is not expected to affect the way the pixels are displayed. Another method should be used if the encoder wants the decoder to modify the sample values for display purposes.

The pCAL chunk contains:

```
Calibration name:      1-79 bytes (character string)
Null separator:        1 byte
Original zero (x0):    4 bytes (signed integer)
Original max  (x1):    4 bytes (signed integer)
Equation type:         1 byte
Number of parameters: 1 byte
Unit name:             0 or more bytes (character string)
Null separator:        1 byte
Parameter 0 (p0):      1 or more bytes (ASCII floating-point)
Null separator:        1 byte
Parameter 1 (p1):      1 or more bytes (ASCII floating-point)
...etc...
```

There is no null separator after the final parameter (or after the unit name, if there are zero parameters). The number of parameters field must agree with the actual number of parameters present in the chunk, and must be correct for the specified equation type (see below).

The calibration name can be any convenient name for referring to the mapping, and is subject to the same restrictions as a tEXt keyword: it must contain only printable Latin-1 [ISO/IEC-8859-1] characters (33–126 and 161–255) and spaces (32), but no leading, trailing, or consecutive spaces. The calibration name can permit applications or people to choose the appropriate pCAL chunk when more than one is present (this could occur in a multiple-image file, but not in a PNG file). For example, a calibration name of "SI" or "English" could be used to identify the system of units in the pCAL chunk as well as in other chunk types, to permit a decoder to select an appropriate set of chunks based on their names.

The pCAL chunk defines two mappings:

1. A mapping from the stored samples, which are unsigned integers in the range `0..max`, where `max=(2^bitdepth)-1`, to the original samples, which are signed integers. The `x0` and `x1` fields, together with the bit depth for the image, define this mapping.
2. A mapping from the original samples to the physical values, which are usually real numbers with units. This mapping is defined by `x0`, `x1`, the equation type, parameters, and unit name.

The mapping between the stored samples and the original samples is given by the following equations:

```
original_sample =
   (stored_sample * (x1-x0) + max/2) / max + x0

stored_sample =
    ((original_sample - x0) * max + (x1-x0)/2) / (x1-x0)
    clipped to the range 0..max
```

In these equations, "/" means integer division that rounds toward negative infinity, so `n/d = integer(floor(real(a)/real(b)))`. Note that this is the same as the "/" operator in the C programming language when `n` and `d` are nonnegative, but not necessarily when `n` or `d` is negative.

Notice that `x0` and `x1` are the original samples that correspond to the stored samples 0 and `max`, respectively. Encoders will usually set `x0=0` and `x1=max` to indicate that the stored samples are equal to the original samples. Note that `x0` is not constrained to be less than `x1`, and neither is constrained to be positive, but they must be different from each other.

This mapping is lossless and reversible when `abs(x1-x0) <= max` and the original sample is in the range `x0..x1`. If `abs(x1-x0) > max` then there can be no lossless reversible mapping, but the functions provide the best integer approximations to floating-point affine transformations.

The mapping between the original samples and the physical values is given by one of several equations, depending on the equation type, which may have the following values:

```
0: Linear mapping
1: Base-e exponential mapping
2: Arbitrary-base exponential mapping
3: Hyperbolic mapping
```

For equation type 0:

```
physical_value = p0 + p1 * original_sample / (x1-x0)
```

For equation type 1:

```
physical_value =
   p0 + p1 * exp(p2 * original_sample / (x1-x0))
```

For equation type 2:

```
physical_value =
   p0 + p1 * pow(p2, (original_sample / (x1-x0)))
```

For equation type 3:

```
physical_value =
   p0 + p1 * sinh(p2 * (original_sample - p3) / (x1-x0))
```

For these physical value equations, "/" means floating-point division.

The function `exp(x)` is `e` raised to the power of `x`, where `e` is the base of the natural logarithms, approximately 2.71828182846. The exponential function `exp()` is the inverse the natural logarithm function `ln()`.

The function `pow(x,y)` is `x` raised to the power of `y`.

```
pow(x,y) = exp(y * ln(x))
```

The function `sinh(x)` is the hyperbolic sine of `x`.

```
sinh(x) = 0.5 * (exp(x) - exp(-x))
```

The units for the physical values are given by the unit name, which may contain any number of printable Latin-1 characters, with no limitation on the number and position of blanks. For example, “K”, “population density”, “MPa”. A zero-length string can be used for dimensionless data.

For color types 0 (gray) and 4 (gray-alpha), the mappings apply to the gray sample values (but not to the alpha sample). For color types 2 (RGB), 3 (indexed RGB), and 6 (RGBA), the mappings apply independently to each of the red, green, and blue sample values (but not the alpha sample). In the case of color type 3 (indexed RGB), the mapping refers to the RGB samples and not to the index values.

Linear data can be expressed with equation type 0.

Pure logarithmic data can be expressed with either equation type 1 or 2:

```
Equation type 1        Equation type 2
x0 = 0                 x0 = 0
x1 = max               x1 = max
p0 = 0                 p0 = 0
p1 = bottom            p1 = bottom
p2 = ln(top/bottom)    p2 = top/bottom
```

Equation types 1 and 2 are functionally equivalent; both are defined because authors may find one or the other more convenient.

Using equation type 3, floating-point data can be reduced (with loss) to a set of integer samples such that the resolution of the stored data is roughly proportional to its magnitude. For example, floating-point data ranging from `-10^31` to `10^31` (the usual range of 32-bit floating-point numbers) can be represented with:

```
Equation type 3
x0 = 0
x1 = 65535
p0 = 0.0
p1 = 1.0e-30
p2 = 280.0
p3 = 32767.0
```

The resolution near zero is about `10^-33`, while the resolution near `10^31` or `-10^31` is about `10^28`. Everywhere the resolution is about 0.4 percent of the magnitude.

Note that those floating-point paramaters could be stored in the chunk more compactly as follows:

```
p0 = 0
p1 = 1e-30
p2 = 280
p3 = 32767
```

Applications should use double precision arithmetic (or take other precautions) while performing the mappings for equation types 1, 2, and 3, to prevent overflow of intermediate results when p1 is small and the `exp()`, `pow()`, or `sinh()` function is large.

If present, the pCAL chunk must appear before the first IDAT chunk. Only one instance of the pCAL chunk is permitted in a PNG datastream.

3.3 sCAL Physical scale of image subject

While the pHYs chunk is used to record the physical size of the image itself as it was scanned or as it should be printed, certain images (such as maps, photomicrographs, astronomical surveys, floor plans, and others) may benefit from knowing the actual physical dimensions of the image's subject for remote measurement and other purposes. The sCAL chunk serves this need. It contains:

```
Unit specifier: 1 byte
Pixel width:    1 or more bytes (ASCII floating-point)
Null separator: 1 byte
Pixel height:   1 or more bytes (ASCII floating-point)
```

The following values are legal for the unit specifier:

```
1: unit is the meter
2: unit is the radian
```

Following the unit specifier are two ASCII strings. The first string defines the physical width represented by one image pixel; the second string defines the physical height represented by one pixel. The two strings are separated by a zero byte (null character). As in the tEXt chunk, there is no trailing zero byte for the final string. Each of these strings contains a floating-point constant in the format specified above (Floating-point values, Section 1.2). Both values are required to be greater than zero.

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

3.4 gIFg GIF Graphic Control Extension

The gIFg chunk is provided for backward compatibility with the GIF89a Graphic Control Extension. It contains:

```
Disposal Method: 1 byte
User Input Flag: 1 byte
Delay Time:      2 bytes (byte order converted from GIF)
```

The Disposal Method indicates the way in which the graphic is to be treated after being displayed. The User Input Flag indicates whether user input is required before continuing. The Delay Time specifies the number of hundredths (1/100) of a second to delay before continuing with the processing of the datastream. Note that this field is to be byte-order-converted.

The "Transparent Color Flag" and "Transparent Color Index" fields found in the GIF89a Graphic Control Extension are omitted from gIFg. These fields should be converted using the transparency features of basic PNG.

The GIF specification allows at most one Graphic Control Extension to preceed each graphic rendering block. Because each PNG file holds only one image, it is expected that gIFg will appear at most once, before IDAT, but there is no strict requirement.

3.5 gIFx GIF Application Extension

The gIFx chunk is provided for backward compatibility with the GIF89a Application Extension. The Application Extension contains application-specific information. This chunk contains:

```
Application Identifier: 8 bytes
Authentication Code:    3 bytes
Application Data:       n bytes
```

The Application Identifier is a sequence of eight printable ASCII characters used to identify the application creating the Application Extension. The Authentication Code is three additional bytes that the application may use to further validate the Application Extension. The remainder of the chunk is application-specific data whose content is not defined by the GIF specification.

Note that GIF-to-PNG converters should not attempt to perform byte reordering on the contents of the Application Extension. The data is simply transcribed without any processing except for de-blocking GIF sub-blocks.

Applications that formerly used GIF Application Extensions may define special-purpose PNG chunks to replace their application extensions. If a GIF-to-PNG converter recognizes the Application Identifier and is aware of a corresponding PNG chunk, it may choose to convert the Application Extension into that PNG chunk type rather than using gIFx.

4 Chunks Not Described Here

The definitions of some public chunks are being maintained by groups other than the core PNG group. In general, these are chunks that are useful to more than one application (and thus are not private chunks), but are considered too specialized to list in the core PNG documentation.

4.1 fRAc Fractal image parameters

The fRAc chunk will describe the parameters used to generate a fractal image. The specification for the contents of the fRAc chunk is being developed by Tim Wegner, twegner@phoenix.net.

In the future, chunks will be fully specified before they are registered.

5 Text Chunk Keywords

It is expected that special-purpose keywords for tEXt and zTXt chunks will be registered and will appear in this document. However, no such keywords have yet been assigned.

6 Deprecated Chunks

The chunks listed in this section are registered, but deprecated. Encoders are discouraged from using them, and decoders are not encouraged to support them.

6.1 gIFt GIF Plain Text Extension

The gIFt chunk was originally provided for backward compatibility with the GIF89a Plain Text Extension, but gIFt is now deprecated because it suffers from some fundamental design flaws.

- GIF considers a Plain Text Extension to be a Graphic Rendering Block, just like an image, so a GIF datastream containing an image and a Plain Text Extension is really a multi-image datastream with ordering issues (like associating each Graphic Control Extension with the proper Graphic Rendering Block). PNG, being a single-image format with no provisions for handling these ordering issues, is not equipped to contain both IDAT and gIFt simultaneously. Since IDAT is required, gIFt must be discouraged.
- The Text Foreground Color and Text Background Color fields of the Plain Text Extension are converted to RGB, rather than being converted to RGBA or left as palette indexes. Therefore, transparency information can be lost.

The gIFt chunk contains:

```
Text Grid Left Position: 4 bytes (signed integer,
                            byte order and size converted)
Text Grid Top Position:  4 bytes (signed integer,
                            byte order and size converted)
Text Grid Width:         4 bytes (unsigned integer,
                            byte order and size converted)
Text Grid Height:        4 bytes (unsigned integer,
                            byte order and size converted)
Character Cell Width:    1 byte
Character Cell Height:   1 byte
Text Foreground Color:   3 bytes (R,G,B samples)
Text Background Color:   3 bytes (R,G,B samples)
Plain Text Data:         n bytes
```

Text Grid Left Position, Top Position, Width, and Height specify the text area position and size in pixels. The converter must reformat these fields from 2-byte LSB-first unsigned integers to 4-byte MSB-first signed or unsigned integers. Note that GIF defines the position to be relative to the upper left corner of the logical screen. If an oFFs chunk is also present, a decoder should assume that the oFFs chunk defines the offset of the image relative to the GIF logical screen; hence subtracting the oFFs values (converted from micrometers to pixels if necessary) from the Text Grid Left and Top Positions gives the text area position relative to the main PNG image.

Character Cell Width and Height give the dimensions of each character in pixels.

Text Foreground and Background Color give the colors to be used to render text foreground and background. Note that the GIF-to-PNG converter must replace the palette index values found in the GIF Plain Text Extension block with the corresponding palette entry.

The remainder of the chunk is the text to be displayed. Note that this data is not in GIF sub-block format, but is a continuous datastream.

7 Security Considerations

The normal precautions (see the Security considerations section of the PNG specification) should be taken when displaying text contained in the sCAL calibration name, pCAL unit name, or any ASCII floating-point fields.

Applications must take care to avoid underflow and overflow of intermediate results when converting data from one form to another according to the pCAL mappings.

8 Appendix: Sample code

This appendix provides some sample code that can be used in encoding and decoding PNG chunks. It does not form a part of the specification. In the event of a discrepancy between the sample code in this appendix and the chunk definition, the chunk definition prevails.

8.1 pCAL

The latest version of this code, including test routines not shown here, is available at ftp://ftp.uu.net/graphics/png/src/pcal.c.

```
#if 0
pcal.c 0.2.2 (Sat 19 Dec 1998)
Adam M. Costello <amc @ cs.berkeley.edu>

This is public domain example code for computing
```

```
the mappings defined for the PNG pCAL chunk.

#endif
#if __STDC__ != 1
#error This code relies on ANSI C conformance.
#endif


#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>


/* In this program a type named uintN denotes an unsigned    */
/* type that handles at least all values 0 through (2^N)-1.  */
/* A type named intN denotes a signed type that handles at   */
/* least all values 1-2^(N-1) through 2^(N-1)-1.  It is not  */
/* necessarily the smallest such type; we are more concerned */
/* with speed.                                               */

typedef unsigned int uint16;

#if UINT_MAX >= 0xffffffff
  typedef unsigned int uint32;
#else
  typedef unsigned long uint32;
#endif

#if INT_MAX >= 0x7fffffff && INT_MIN + 0x7fffffff <= 0
  typedef int int32;
#else
  typedef long int32;
#endif

/* Testing for 48-bit integers is tricky because we cannot */
/* safely use constants greater than 0xffffffff.  Also,    */
/* shifting by the entire width of a type is undefined, so */
/* for unsigned int, which might be only 16 bits wide, we  */
/* must shift in two steps.                                */

#if (UINT_MAX - 0xffff) >> 8 >> 8 >= 0xffffffff
  typedef unsigned int uint48;
  #define HAVE_UINT48 1
#elif (ULONG_MAX - 0xffff) >> 16 >= 0xffffffff
  typedef unsigned long uint48;
  #define HAVE_UINT48 1
#elif defined(ULLONG_MAX)
  #if (ULLONG_MAX - 0xffff) >> 16 >= 0xffffffff
```

```
    typedef unsigned long long uint48;
    #define HAVE_UINT48 1
  #endif
#else
  #define HAVE_UINT48 0
#endif


/********************/
/* Program failure */

void
fail(const char *msg)
{
  fputs(msg,stderr);
  fputc('\n', stderr);
  exit(EXIT_FAILURE);
}


/***************************/
/* Check max, x0, and x1 */

int
samp_params_ok(uint16 max, int32 x0, int32 x1)

/* Returns 1 if max, x0, and x1 have */
/* allowed values, 0 otherwise.      */

{
  const int32 xlimit = 0x7fffffff;

  return    max >  0      && max <=  0xffff
         &&  x0 <= xlimit &&  x0 >= -xlimit
         &&  x1 <= xlimit &&  x1 >= -xlimit
         &&  x0 != x1;
}


/************************************************/
/* Map from stored samples to original samples */

int32
stored_to_orig(uint16 stored, uint16 max, int32 x0, int32 x1)

#if 0

  Returns the original sample corresponding to the given stored
  sample, which must be <= max.  The parameters max, x0, and x1
```

```
must have been approved by samp_params_ok().

The pCAL spec says:

    orig = (stored * (x1-x0) + max/2) / max + x0           [1]

Equivalently:

    orig = (stored * (x1-x0) + max/2) / max
           + (x0-x1) - (x0-x1) + x0
    orig = (stored * (x1-x0) + max * (x0-x1) + max/2) / max
           - (x0-x1) + x0
    orig = ((max - stored) * (x0-x1) + max/2) / max + x1

So we can check whether x0 < x1 and coerce the formula so that
the numerators and denominators are always nonnegative:

    orig = (offset * xspan + max/2) / max + xbottom        [2]

This will come in handy later.

But the multiplication and the subtraction can overflow, so we
have to be trickier.  For the subtraction, we can convert to
unsigned integers.  For the multiplication, we can use 48-bit
integers if we have them, otherwise observe that:

       b    =   (b/c)*c +     b%c
     a*b    = a*(b/c)*c +  a*(b%c)  ; let d = a*(b%c)
    (a*b)/c = a*(b/c)   +     d/c   remainder d%c          [3]

These are true no matter which way the division rounds.  If
(a*b)/c is in-range, a*(b/c) is guaranteed to be in-range if
b/c rounds toward zero.  Here is another observation:

    sum{x_i} / c = sum{x_i / c} + sum{x_i % c} / c       [4]

This one also avoids overflow if the division rounds toward
zero.  The pCAL spec requires rounding toward -infinity.  ANSI
C leaves the rounding direction implementation-defined except
when both the numerator and denominator are nonnegative, in
which case it rounds downward.  So if we arrange for all
numerators and denominators to be nonnegative, everything
works.  Starting with equation 2 and applying identity 4, then
3, we obtain the final formula:

          d = offset * (xspan % max)
    xoffset = offset * (xspan / max) + d/max
              + (d%max + max/2) / max
       orig = xoffset + xbottom
```

```
#endif

{
  uint16 offset;
  uint32 xspan, q, r, d, xoffset;
  int32 xbottom;

  if (stored > max) fail("stored_to_orig: stored > max");

  if (x1 >= x0) {
    xbottom = x0;
    xspan = (uint32)x1 - (uint32)x0;
    offset = stored;
  }
  else {
    xbottom = x1;
    xspan = (uint32)x0 - (uint32)x1;
    offset = max - stored;
  }

  /* We knew xspan would fit in a uint32, but we needed to   */
  /* cast x0 and x1 before subtracting because otherwise the */
  /* subtraction could overflow, and ANSI doesn't say what   */
  /* the result will be in that case.                        */

  /* Let's optimize two common simple cases */
  /* before handling the general case:      */

  if (xspan == max) {
    xoffset = offset;
  }
  else if (xspan <= 0xffff) {
    /* Equation 2 won't overflow and does only one division. */
    xoffset = (offset * xspan + (max>>1)) / max;
  }
  else {
    #if HAVE_UINT48
      /* We can use equation 2 and do one uint48     */
      /* division instead of three uint32 divisions. */
      xoffset = (offset * (uint48)xspan + (max>>1)) / max;
    #else
      q = xspan / max;
      r = xspan % max;
      /* Hopefully those were compiled into one instruction. */
      d = offset * r;
      xoffset = offset * q + d/max + (d%max + (max>>1)) / max;
    #endif
  }
```

```
  /* xoffset might not fit in an int32, but we know the sum */
  /* xbottom + xoffset will, so we can do the addition on   */
  /* unsigned integers and then cast.                        */

  return (int32)((uint32)xbottom + xoffset);
}


/*********************************************/
/* Map from original samples to stored samples */

uint16
orig_to_stored(int32 orig, uint16 max, int32 x0, int32 x1)

#if 0

  Returns the stored sample corresponding to the given original
  sample.  The parameters max, x0, and x1 must have been
  approved by samp_params_ok().

  The pCAL spec says:

  stored = ((orig - x0) * max + (x1-x0)/2) / (x1-x0)
           clipped to the range 0..max

  Notice that all three terms are nonnegative, or else all
  are nonpositive.  Just as in stored_to_orig(), we can avoid
  overflow and rounding problems by transforming the equation to
  use unsigned quantities:

  stored = (xoffset * max + xspan/2) / xspan

#endif

{
  uint32 xoffset, xspan;

  if (x0 < x1) {
    if (orig < x0) return 0;
    if (orig > x1) return max;
    xspan = (uint32)x1 - (uint32)x0;
    xoffset = (uint32)orig - (uint32)x0;
  }
  else {
    if (orig < x1) return 0;
    if (orig > x0) return max;
    xspan = (uint32)x0 - (uint32)x1;
    xoffset = (uint32)x0 - (uint32)orig;
  }
```

```
  /* For 16-bit xspan the calculation is straightforward: */

  if (xspan <= 0xffff)
    return (xoffset * max + (xspan>>1)) / xspan;

  /* Otherwise, the numerator is more than 32 bits and the   */
  /* denominator is more than 16 bits.  The tricks we played */
  /* in stored_to_orig() depended on the denominator being   */
  /* 16-bit, so they won't help us here.                     */

  #if HAVE_UINT48
    return ((uint48)xoffset * max + (xspan>>1)) / xspan;
  #else

    /* Doing the exact integer calculation with 32-bit         */
    /* arithmetic would be very difficult.  But xspan > 0xffff */
    /* implies xspan > max, in which case the pCAL spec says   */
    /* "there can be no lossless reversible mapping, but the   */
    /* functions provide the best integer approximations to    */
    /* floating-point affine transformations."  So why insist  */
    /* on using the integer calculation?  Let's just use       */
    /* floating-point.                                         */

    return ((double)xoffset * max + (xspan>>1)) / xspan;

  #endif
}


/********************************************/
/* Check x0, x1, eqtype, n, and p[0]..p[n-1] */

int
phys_params_ok(int32 x0, int32 x1, int eqtype, int n, double *p)

/* Returns 1 if x0, x1, eqtype, n, and p[0]..p[n-1] */
/* have allowed values, 0 otherwise.                */

{
  if (!samp_params_ok(1,x0,x1)) return 0;

  switch (eqtype) {
    case 0: return n == 2;
    case 1: return n == 3;
    case 2: break;
    case 3: return n == 4;
  }

  /* eqtype is 2, check for pow() domain error: */
```

```
  if (p[2] > 0) return 1;
  if (p[2] < 0) return 0;
  return (x0 <= x1) ? (x0 > 0 && x1 > 0) : (x0 < 0 && x1 < 0);
}


/************************************************/
/* Map from original samples to physical values */

double
orig_to_phys(int32 orig, int32 x0, int32 x1,
             int eqtype, double *p)

/* Returns the physical value corresponding to the given    */
/* original sample.  The parameters x0, x1, eqtype, and p[] */
/* must have been approved by phys_params_ok().  The array  */
/* p[] must hold enough parameters for the equation type.   */

{
  double xdiff, f;

  xdiff = (double)x1 - x0;

  switch (eqtype) {
    case 0:  f = orig / xdiff;
             break;
    case 1:  f = exp(p[2] * orig / xdiff);
             break;
    case 2:  f = pow(p[2], orig / xdiff);
             break;
    case 3:  f = sinh(p[2] * (orig - p[3]) / xdiff);
             break;
    default: fail("orig_to_phys: unknown equation type");
  }

  return p[0] + p[1] * f;
}
```

8.2 Fixed-point gamma correction

The latest version of this code, including test routines not shown here, is available at ftp://ftp.uu.net/graphics/png/src/gamma-lookup.c.

```
#if 0
gamma-lookup.c 0.1.4 (Sat 19 Dec 1998)
by Adam M. Costello <amc @ cs.berkeley.edu>
```

```
This is public domain example code for computing gamma
correction lookup tables using integer arithmetic.

#endif
#if __STDC__ != 1
#error This code relies on ANSI C conformance.
#endif


#include <limits.h>
#include <math.h>


/* In this program a type named uintN denotes the  */
/* smallest unsigned type we can find that handles */
/* at least all values 0 through (2^N)-1.          */

typedef unsigned char uint8;

#if UCHAR_MAX >= 0xffff
  typedef unsigned char uint16;
#else
  typedef unsigned short uint16;
#endif

#if UCHAR_MAX >= 0xffffffff
  typedef unsigned char uint32;
#elif USHRT_MAX >= 0xffffffff
  typedef unsigned short uint32;
#elif UINT_MAX >= 0xffffffff
  typedef unsigned int uint32;
#else
  typedef unsigned long uint32;
#endif


/*********************/
/* 16-bit arithmetic */

void
precompute16(uint16 L[511])

/* Precomputes the log table (this requires floating point). */

{
  int j;
  double f;

  /* L[j] will hold an integer representation of -log(j / */
```

```
  /* 510.0).  Knowing that L[1] (the largest) is 0xfe00   */
  /* will help avoid overflow later, so we set the scale  */
  /* factor accordingly.                                  */

  f = 0xfe00 / log(1 / 510.0);

  for (j = 1;  j <= 510;  ++j)
    L[j] = log(j / 510.0) * f + 0.5;
}


void
gamma16(uint16 L[511], uint8 G[256], uint16 g)

/* Makes a 256-entry gamma correction lookup table G[] with */
/* exponent g/pow(2,14), where g must not exceed 0xffff.    */

{
  int i, j;
  uint16 x, y, xhi, ghi, xlo, glo;

  j = 1;
  G[0] = 0;

  for (i = 1;  i <= 255;  ++i) {
    x = L[i << 1];
    xhi = x >> 8;
    ghi = g >> 8;
    y = xhi * ghi;

    if (y > 0x3f80) {
      /* We could have overflowed later. */
      /* But now we know y << 2 > L[1].  */
      G[i] = 0;
      continue;
    }

    xlo = x & 0xff;
    glo = g & 0xff;
    y = (y << 2) + ((xhi * glo) >> 6) + ((xlo * ghi) >> 6);
    while (L[j] > y) ++j;
    G[i] = j >> 1;
  }
}


/********************/
/* 32-bit arithmetic */
```

```
void
precompute32(uint32 L[511])

/* Precomputes the log table (this requires floating point). */

{
  int j;
  double f;

  /* L[j] will hold an integer representation of -log(j /   */
  /* 510.0).  Knowing that L[1] (the largest) is 0x3fffffff */
  /* will help avoid overflow later, so we set the scale    */
  /* factor accordingly.                                    */

  f = 0x3fffffff / log(1 / 510.0);

  for (j = 1;  j <= 510;  ++j)
    L[j] = log(j / 510.0) * f + 0.5;
}


void
gamma32(uint32 L[511], uint8 G[256], uint16 g)

/* Makes a 256-entry gamma correction lookup table G[] with */
/* exponent g/pow(2,14), where g must not exceed 0xffff.    */

{
  int i, j;
  uint32 x, y;

  j = 1;
  G[0] = 0;

  for (i = 1;  i <= 255;  ++i) {
    x = L[i << 1];
    y = (x >> 14) * g;
    while (L[j] > y) ++j;
    G[i] = j >> 1;
  }
}


/***********************************************/
/* Floating-point arithmetic (for comparison) */

void
gamma_fp(uint8 G[256], double g)
```

```
/* Makes a 256-entry gamma correction */
/* lookup table G[i] with exponent g. */

{
  int i;

  G[0] = 0;

  for (i = 1;  i <= 255;  ++i)
    G[i] = pow(i/255.0, g) * 255 + 0.5;
}
```

9 Appendix: Rationale

This appendix gives the reasoning behind some of the design decisions in the PNG extension chunks. It does not form a part of the specification.

9.1 pCAL

This section gives the reasoning behind some of the design decisions in the pCAL chunk. It does not form a part of the specification.

Redundant equation types

Equation types 1 and 2 seem to be equivalent. Why have both?

- We don't want to force people to do the exponentiation using `ln()` and `exp()`, since `pow()` may provide better accuracy in some floating-point math libraries. We also don't want to force people using base-10 logs to store a sufficiently accurate value of `ln(10)` in the pCAL chunk.
- When the base is `e`, we don't want to force people to encode a sufficiently accurate value of `e` in the pCAL chunk, or to use `pow()` when `exp()` is sufficient.

What are x0 and x1 for?

- First, `x0` and `x1` provide a way to recover the original data, losslessly, when the original range is not a power of two. Sometimes the digitized values do not have a range that fills the full depth of a PNG. For example, if the original samples range from 0 (corresponding to black) to 800 (corresponding to white), PNG requires that these samples be scaled to the range 0 to 65535. By recording `x0=0` and `x1=800` we can recover the original samples, and we indicate the precision of the data.

- Even if the original data had a range identical to a valid PNG image sample, like 0 (black) to 65535 (white), one might want to create a derived image by stretching the contrast in a limited intensity range containing the important details. For example, we might want to scale the samples so that 46000 becomes 0 (black) and 47000 becomes 65535 (white). As in the previous case, by recording `x0=46000` and `x1=47000`, we can recover the original data samples that fell between 46000 and 47000.

Integer division

Why define integer divison to round toward negative infinity? This is different from many C implementations and from all Fortran implementations, which round toward zero.

We cannot leave the choice unspecified. If we were to specify rounding toward zero, we'd have to account for a discontinuity at zero. A division by positive `d` would map the `2d-1` values from `-(d-1)` through `d-1` to zero, but would map only d values to any other value; for example, `3d` through `4d-1` would be mapped to `3`. Achieving lossless mappings in spite of this anomaly would be difficult.

10 Appendix: Revision History

- March 11, 1997: First public release
- December 22, 1998:
  - Added pCAL chunk and related sample code
  - Deprecated the gIFT chunk
  - Added sample gamma-correction code that uses integer arithmetic

11 References

[ISO/IEC-8859]
International Organization for Standardization and International Electrotechnical Commission, "Information Technology—8-bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1", IS 8859-1, 1998.
Also see sample files at
ftp://ftp.uu.net/graphics/png/documents/iso_8859-1.*

12 Credits

Editors

- Glenn Randers-Pehrson, randeg @ alumni.rpi.edu

- Tom Lane, tgl @ sss.pgh.pa.us (edited the first release of this document)

Contributors

Names of contributors not already listed in the PNG specification are presented in alphabetical order:

- Todd French, tfrench @ sandia.gov
- Alaric B. Williams, png @ abwillms.demon.co.uk

Trademarks

GIF is a service mark of CompuServe Incorporated.

12.1 Copyright Notice

End of Extensions to the PNG Specification