Extensions to the PNG Specification:
Special-Purpose Public Chunks
Revision: 27 August 1998

The latest versions of this document 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`.

1 Introduction

This document is an extension to the Portable Network Graphics (PNG) specification. It describes additional public chunk types and contains additional information for use in PNG images. The basic PNG specification is available from the PNG FTP archive mentioned above.

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. Additional chunk types may be proposed for inclusion in this list by contacting the PNG specification maintainers at `png-info@uunet.uu.net`.

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.

2 Data representation

2.1 Integer values

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

2.2 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.

3 Summary of Special-Purpose Chunks

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

Name	Multiple OK?	Ordering constraints
oFFs	No	Before IDAT
pCAL	No	Before IDAT
sCAL	No	Before IDAT
sPLT	Yes	Before IDAT
gIFg	Yes	None
gIFt	Yes	None
gIFx	Yes	None
fRAc	Yes	None

4 Chunk Descriptions

4.1 `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 chunk's contents are:

```
Image_position, X axis: 4 bytes (signed integer)
Image_position, Y axis: 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 25,400 micrometers. A micrometer (also called a micron) is 1/1,000,000th of a 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.

4.2 `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's contents are a text string (`Purpose`) that names the equation, original sample limits `X0` and `X1`, an `Equation_type`, a text string (`Unit_string`), and a set of parameters for the equation:

```
n bytes:  Purpose (Latin-1 text)

1 byte:   null separator

4 bytes:  X0 (signed integer) Lower limit of original
          sample range.  This original sample value is
          mapped to the stored sample value 0.

4 bytes:  X1 (signed integer) Upper limit of original
          sample range.  This original sample value is
          mapped to the stored sample value
          (2^Sample_depth - 1).  Must not equal X0.

1 byte:   Equation_type (unsigned integer).
     0:   Linear mapping
     1:   Base-e exponential mapping
     2:   Arbitrary-base exponential mapping
     3:   Hypberbolic mapping

1 byte:   N (unsigned integer), number of parameters.
```

```
u bytes:  Unit_string (Latin-1 string).  Symbol or description
          of the unit, eg. K, Population Density, MPa, etc).
          A zero-length string can be used for dimensionless
          data.

1 byte:   null separator

p0 bytes: P0 (ASCII text).  First parameter, a real number
          written as a text floating-point value, as
          described above.

1 byte:   null separator

p1 bytes: P1 (ASCII text).  Second parameter.

etc.
```

There is no null separator after the final parameter (or after the `Unit_string` field, if `N=0`). The number of parameters present must agree with the `N` field and must be correct for the specified `Equation_type`.

The `Purpose` identifies the equation, which can permit applications or people to choose the appropriate one when more than one `pCAL` chunk is present (this could occur in a multiple-image file, but not in a PNG file). The `Purpose` string must follow the format of a `tEXt` keyword, i.e., 1-79 printable Latin-1 characters, without leading, trailing, or consecutive blanks. One way the `Purpose` field could be used s to identify the system of units. A `Purpose` string such as "SI" or "English" could be used in the `pCAL` chunk as well as in other chunk types, to permit a decoder to select an appropriate set of chunks based on the contents of their `Purpose` fields.

The `Unit_string` can have any number of Latin-1 characters, or no characters at all, and there is no limitation on the number and position of blanks.

The `pCAL` chunk defines two mappings:

- A mapping from the stored samples, which are unsigned integers in the range [0..M], where `(M=2^Sample_depth-1)`, to the original samples, which are signed integers. `X0` and `X1`, together with the `Sample_depth` for the image, define this mapping.
- A mapping from the original samples to the physical values, which are usually real numbers. The remaining `pCAL` chunk parameters define this mapping.

Encoders will usually set `(X0 = 0)` and `(X1 = M)` 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.

The mapping algorithms are

```
Original_sample =
   (Stored_sample * (X1 - X0) + M/2) / M + X0
```

using integer arithmetic, where `(a/b)` means `(integer(floor(real(a)/real(b))))`. Note that this is the same as the "/" operator in the C programming language when "a" and "b" are nonnegative, but not necessarily when "a" or "b" is negative.

```
if Equation_type = 0 then
Physical_value = P0 + P1 * Original_sample/(X1-X0)

else if Equation_type = 1 then
Physical_value =
    P0 + P1 * EXP(P2 * Original_sample/(X1-X0))

else if Equation_type = 2 then
Physical_value = P0 + P1 * P2^(Original_sample/(X1-X0))

else if Equation_type = 3 then
Physical_value =
    P0 + P1*SINH(P2*(Original_sample - P3)/(X1-X0))
```

using floating-point arithmetic, in which

- EXP(x) is the exponential function, e^x, in which "e" is the base of natural logarithms (approximately 2.718282)
- SINH(x) is the hyperbolic sine of x,

```
SINH(x) = 0.5 * ( EXP(x) - EXP(-x) )
```

To map an original sample to a stored sample, the function is:

```
Stored_sample =
    ((Original_sample - X0) * M + (X1 - X0) / 2) / (X1 - X0)

    (limited to the range [0..M])
```

This mapping is lossless and reversible when `(ABS(X1-X0) <= M)` and the original sample is in the range [X0..X1]. If `(ABS(X1-X0) > M)` then there can be no lossless reversible mapping, but the functions provide the best integer approximations to floating-point affine transformations.

For color_types 2, 3, and 6, the mapping algorithms are applied independently to each of the color sample values. In the case of color type 3 (indexed color), 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 either with either equation_type 1 or `Equation_type 2`:

```
X0 = 0                              X0 = 0
X1 = M                              X1 = M
Equation_type = 1                   Equation_type = 2
N  = 3                 or with      N  = 3
P0 = 0                              P0 = 0
P1 = min                            P1 = min
P2 = LOGe(max/min)                  P2 = max/min
```

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 in the range [-v0..v1] 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 floating-point numbers on 32-bit machines) can be represented with

```
X0 = 0
X1 = 65535
Equation_type = 3
N  = 4
P0 = 0.0
P1 = 1.0e-30
P2 = 280.0
P3 = 32767.0
```

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

Applications should use double precision arithmetic (or take other precautions) while performing the mappings for `Equation_type 1, 2, and 3`, to prevent overflow of intermediate results when the parameter `P1` is small and the exponential or power functions are 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 stream.

4.3 `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. The `sCAL` chunk's contents are

```
Unit_specifier: 1 byte
Pixel_width:    m bytes (floating value as ASCII string)
Null separator: 1 byte
Pixel_height:   n bytes (floating value as ASCII string)
```

Valid values for the Unit_specifier field are:

```
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 for the final string. Each of these strings contains a floating-point constant in the format specified above (Floating-Point Values). Both values are required to be greater than zero.

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

4.4 sPLT Suggested Palette

This chunk can be used to suggest a reduced palette to be used when the display device is not capable of displaying the full range of colors present in the image. If present, it provides a recommended set of colors, with alpha and frequency information, that can be used to construct a reduced palette to which the PNG image can be quantized.

This chunk's contents are a zero-byte-terminated text string that names the palette and a 1-byte sPLT_sample_depth integer, followed by a series of palette entries, each a six-byte or ten-byte series containing five unsigned integers:

```
n bytes:     (Latin-1 text) Palette_name
1 byte:      (null) terminator

1 byte:      (unsigned integer)  sPLT_sample_depth
             must be 8 or 16

1 or 2 bytes:  (unsigned integer)  Red_level
           0:   black
           etc.
           255 or 65535: full red intensity

1 or 2 bytes:  (unsigned integer)  Green_level

1 or 2 bytes:  (unsigned integer)  Blue_level

1 or 2 bytes:  (unsigned integer)  Alpha
           0: fully transparent
           etc.
           255 or 65535: fully opaque

2 bytes:   (unsigned integer)  Frequency
           (relative frequency of occurrence)
etc.
```

There can be any number of entries; a decoder determines the number of entries from the remaining chunk length after the Palette_name field and its zero-byte terminator, and the sPLT_sample_depth byte. This length not divisible by six (if sPLT_sample_depth=8) or by ten (if sPLT_sample_depth=16) is an error. Entries must appear in decreasing order of "frequency". There is no requirement that the entries all be used by the image, nor that they all be different.

The Palette_name (e.g., "256 color including Macintosh default", "256 color including Windows-3.1 default", "Optimal 512") identifies the palette. It may help applications or people to choose the appropriate suggested palette when more than one appears in a PNG file. The Palette_name string must follow the same restrictions imposed on tEXt keywords: it must consist only of printable Latin-1 characters and must not have leading or trailing blanks, but can have single embedded blanks. There must be at least one and no more than 79 characters in the name. Names are case-sensitive.

The Red_level, Green_level, and Blue_level values are not premultiplied by alpha, nor are they precomposited against any background. This differs from the treatment of suggested palettes represented by PLTE: in a truecolor image that uses transparency, sPLT provides suggested uncomposited color values, while a suggested-palette PLTE provides suggested colors after compositing against the bKGD color. A decoder can build a target palette by compositing sPLT palette entries against any background color or set of background colors that it chooses. It can then composite the truecolor PNG image against the desired background, and finally quantize the composited image into the target palette.

Each Frequency entry is proportional to the fraction of pixels in the image that are closest to that palette entry, without regard to any compositing against a background palette. The exact scale factor is chosen by the encoder, but should be chosen so that the range of individual values reasonably fills the range 0 to 65535. It is acceptable to artificially inflate the Frequency values for "important" colors such as those in a company logo or in the facial features of a portrait. Zero is a valid value for Frequency, meaning the color is "least important" or that it is rarely if ever used. But when all of the Frequency values are zero, the "frequency" is undefined.

The palette uses 8 bits or 16 bits (1 or 2 bytes) per value according to the number given in the sPLT_sample_depth field, regardless of the image bit depth specification. Decoders wishing to construct a palette with a different bit depth can accomplish this by scaling the RGBA entries, as described under "Sample depth rescaling" in the PNG specification. The palette samples have the same gamma and chromaticity values as those of the PNG image.

The sPLT chunk can appear for any color type. Note that entries in sPLT can fall outside the colorspace of the PNG image; for example, in a grayscale PNG, sPLT entries would typically have R=G=B, but this is not required and decoders must not fail if it is not so. (But a decoder can simply ignore a sPLT that is not to its liking.) Similarly, sPLT entries can have nonopaque alpha values even when the PNG image does not use transparency.

In an indexed-color (color type 3) PNG, PLTE defines the full color range of the PNG image, but sPLT can be used to define alternative reduced palettes for viewers that are unable to display all the colors present in the PLTE chunk.

If sPLT appears, it must precede the first IDAT chunk. There can be multiple sPLT chunks, but if so they must have different palette_names.

Note: The core PNG specification recommends the use of PLTE and hIST chunks to define suggested palettes for truecolor images. This is still allowed, but sPLT provides a more flexible solution:

- With sPLT, there can be multiple suggested palettes. A decoder may choose an appropriate palette based on name or number of entries.
- In an RGBA (color type 6) PNG, PLTE represents a palette already composited against the bKGD color, so it is only useful for display against that background color. The sPLT chunk provides an uncomposited palette, which is useful for display against backgrounds of the decoder's choice.
- Since sPLT is a noncritical chunk, a PNG editor can add or modify suggested palettes without being forced to discard unknown unsafe-to-copy chunks.
- sPLT is allowed in PNG files of color types 0, 3, and 4, whereas PLTE cannot be used to provide reduced palettes in these cases.
- More than 256 entries can appear in sPLT.

In PNG files of color types 2 and 6, if both PLTE and sPLT appear, the decoder can choose from among the palettes suggested by both, bearing in mind the different transparency semantics mentioned above.

An encoder that uses sPLT may choose to write a PLTE/hIST suggested palette as well, for backwards compatibility with decoders that do not recognize sPLT. Note that hIST defines the frequency of PLTE entries, not the entries of sPLT.

5 GIF Conversion Chunk Descriptions

The chunks described in this section are intended specifically for preserving information found in GIF extension blocks when converting a GIF file to a PNG file. GIF-to-PNG converters should simply duplicate these blocks with the reformatting specified below. Applications that formerly made use of this information when reading a GIF file may continue to use it when reading PNG files. Since many applications ignored GIF extension blocks, it is expected that many applications will ignore these PNG chunks.

When converting single-image GIFs to PNG, these chunks should be placed in the same relative position within the PNG file as the extension block has within the GIF file. Thus converters can translate GIF extensions as they are encountered. Note, however, that these chunks are ancillary, safe-to-copy; therefore it is possible that a PNG editor will rearrange their order. Conversion of multiple-image GIFs is a more complex problem that will be addressed in a future document. For now, such files can be converted to a sequence of single-image PNG files.

In a GIF file, an extension block is organized as a series of sub-blocks, where each sub-block is preceded by a length byte. PNG does not use sub-blocks, so the GIF data must be de-blocked by removal of the sub-block length bytes. In addition, the converter is expected to convert certain fields from GIF byte order (LSB first) to network byte order (MSB first), and to expand image-size-related fields from two bytes to four. Fields requiring format changes are marked in the descriptions below.

For brevity, the following chunk descriptions simply define the layout of the chunk fields and indicate the reformatting required. The semantics of the field contents remain the same as in GIF. Readers are referred to the GIF89a specification for details. It is available on CompuServe from GRAPHSUPPORT library 16, file GIF89A.DOC, and is also widely available on Internet.

5.1 `gIFg` GIF Graphic Control Extension

The `gIFg` chunk is provided for backwards compatibility with the GIF89a Graphic Control Extension. The contents of this chunk are:

```
Disposal_method:      1 byte
User_input:           1 byte
Delay_time:           2 bytes (byte order converted from GIF)
```

`Disposal_method` indicates the way in which the graphic is to be treated after being displayed. `User_input` indicates whether user input is required before continuing. `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 Transparency 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.

5.2 `gIFt` GIF Plain Text Extension

The `gIFt` chunk is provided for backwards compatibility with the GIF89a Plain Text Extension. The contents of this chunk are:

```
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`, `Text_grid_top_position`, `Text_grid_width`, and `Text_grid_height` specify the text area position and size in pixels. The converter must reformat these fields from 2 bytes LSB-first unsigned integers to 4 bytes 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 with Unit_specifier = 0 (pixels), 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 from the Text_grid_left_position and Text_grid_top_position gives the text area position relative to the main PNG image (when converting a GIF that has a text rendering block to PNG, it is best to convert the local image Left and Top values to oFFs values in pixels, rather than in micrometers.)

Character_cell_width and Character_cell_height give the dimensions of each character in pixels.

Text_foreground_color and Text_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.

5.3 gIFx GIF Application Extension

The gIFx chunk is provided for backwards compatibility with the GIF89a Application Extension. The Application Extension contains application-specific information. The contents of this chunk are:

```
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.

6 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.

6.1 fRAc Fractal Image Parameters

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

This chunk was registered with good intentions, on the understanding that a full specification would be forthcoming in a timely manner. Implementation and testing of the fRAc chunk have been delayed by unforeseen complications. The PNG maintainers do not intend to register any other future chunks until they have been fully specified.

7 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.

8 Security considerations

The normal precautions (see the Security considerations section of the PNG specification) should be taken when displaying text contained in the Purpose, Unit_string, and parameter fields of the pCAL chunk and the Palette_name field of the sPLT chunk.

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.

9 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.

9.1 pCAL

This section provides some sample code for the PNG pCAL chunk, written in the "C" Programming Language. The pCAL_encode subroutine takes an array of floating-point numbers and produces an array of 16-bit samples that can be stored as grayscale PNG pixels. The pCAL_make_lut procedure produces a lookup table that can be used to extract the original physical values from PNG pixels.

```
/* Sample code for the PNG pCAL chunk */
#include <math.h>
/* Math.h supplies double precision exp(), pow(), and
 * sinh() functions.  If your math.h doesn't supply sinh(),
 * use sinh(x)=(exp(x)+exp(-x))/2.
 */
unsigned short limit(long low, double x, long high)
{
    if (low < high){
       if(x < low)return low;
       else if (x > high) return high;
       else return x;
    }
    else {
       if(x < high)return high;
       else if( x > low) return low;
       else return x;
    }
}
int pCAL_encode (unsigned short *stored_sample, long n,
 float *physical_value, unsigned int m, int equation_type,
 long x0, long x1, float *p)

/* returns 0 (success)
 *        -1 (error, x0==x1)
 *        -2 (unknown equation type)
 * input:
 *         n: number of samples
 *         physical_value[0..n-1]
 *         m: PNG sample depth
 *         equation_type: from pCAL chunk
 *         x0, x1: stored sample to original sample mapping
 *         p[]: equation parameters, from pCAL chunk
 * output:
 *         stored_samples[0..n-1] (caller must allocate space)
 */

{
    double d,dm;  /* force double precision arithmetic */
    long isample, osample, k;
    d=x1-x0;
    dm=m;
```

```
    if (x1 != x0) {
        if(equation_type == 0){
            for (k=0; k<n; k++) {
               isample=.5+d*(physical_value[k] - p[0])/p[1];
               osample=limit(x0, isample, x1);
               stored_sample[k]= floor(((osample-x0)*dm
                                +floor(d/2))/d);
            }
        }

        else if(equation_type == 1){
            for (k=0; k<n; k++) {
               isample= .5+d*(log(physical_value[k]
                         - p[0])/p[1])/p[2];
               osample=limit(x0, isample, x1);
               stored_sample[k]= floor(((osample-x0)*dm
                                +floor(d/2))/d);
            }
        }

        else if(equation_type == 2){
            double factor;
            factor=d/log(p[2]);
            for (k=0; k<n; k++) {
               isample=.5+log((physical_value[k]
                        -p[0])/p[1])*factor;
               osample=limit(x0, isample, x1);
               stored_sample[k]= floor(((osample-x0)*dm
                                +floor(d/2))/d);
            }
        }

        else if(equation_type == 3){
            for (k=0; k<n; k++) {
               isample= .5+p[3]+d*asinh((physical_value[k]
                        -p[0])/p[1])/p[2];
               osample=limit(x0, isample, x1);
               stored_sample[k]= floor(((osample-x0)*dm
                                +floor(d/2))/d);
            }
        }
        else return (-2); /* ERROR, unknown equation type */
    }
    else return (-1); /* ERROR, x0 == x1 */
    return (0);
}

int pCAL_make_lut (float *physical_value, unsigned int m,
             int equation_type, long x0, long x1, float *p)
```

```
/* returns 0 (success)
 *        -1 (error, x0==x1)
 *        -2 (unknown equation type)
 * input:
 *         m: PNG sample depth
 *         equation_type: from pCAL chunk
 *         x0, x1: stored sample to original sample mapping
 *         p[]: equation parameters, from pCAL chunk
 * output:
 *         physical_value[0..m] (caller must allocate space)
 */

{
    double d, dm; /* force double precision arithmetic */
    long sample, osample;
    d=x1-x0;
    dm=m;

    if (x1 != x0) {
        if(equation_type == 0){
            for (sample=0; sample<=m; sample++){
               osample=floor((sample*d+floor(dm/2))/dm) + x0;
               physical_value[sample] = p[0] + p[1]*osample/d;
            }
        }

        else if(equation_type == 1){
            for (sample=0; sample<=m; sample++){
               osample=floor((sample*d+floor(dm/2))/dm) + x0;
               physical_value[sample] = p[0]
                        + p[1]*exp(p[2]*osample/d);
            }
        }
        else if(equation_type == 2){
            for (sample=0; sample<=m; sample++){
               osample=floor((sample*d+floor(dm/2))/dm) + x0;
               physical_value[sample] = p[0]
                        + p[1]*pow(p[2],osample/d);
            }
        }
```

```
            else if(equation_type == 3){
                for (sample=0; sample<=m; sample++){
                   osample=floor((sample*d+floor(dm/2))/dm) + x0;
                   physical_value[sample] =
                      p[0] + p[1]*sinh(p[2]*(osample-p[3])/d);
                }
            }
            else return (-2); /* ERROR, unknown equation type */
        }
        else return (-1); /* ERROR, X0 == X1 */
        return (0);
    }
```

10 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.

10.1 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 `log()` and `exp()`, since `pow()` may provide better accuracy in some floating-point math libraries. Thus we have `Equation_type 2`, which uses `pow(p2, value/(X1-X0))`. We also don't want to force people using base-10 logs to store a sufficiently accurate value of log(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. Thus we have `Equation_type 1`.

What are `X0` and `X1` for?

The {`X0,X1`} mechanism is useful for three things.

- First, the {`X0,X1`} mechanism provides a way to map stored values back to digitizer output and second, as a way to label a limited range image, and to help 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 full range of an image is 0-800 and one wants to stretch the contrast and increase the bit depth to 16 bits, one can do it and leave a record that this has been done, by setting `X0=0` and `X1=800`. The reason to do this is to know the original granularity in the data, which is useful at least as a lower limit on the error in the data.

- A second use is that the {`X0,X1`} parameters can be used to mark the limited range of an original image that is contained in the current image. In this scenario there is an original image that has important detail in a limited intensity range. The limited range is extracted and contrast stretched to fill the PNG bit depth. There is no way in `pCAL` to distinguish these cases but at least `pCAL` can store the relevant information. People can distinguish these cases by including a text chunk labelling the image as original or derived.
- Finally, `X0` is necessary in several of the `Equation_types` to allow a zero offset. Without it, we'd need an extra parameter in `Equation_types` 1 and 2. So eliminating the notion of an "original sample" wouldn't really simplify implementations—and having it allows lossless recovery in the cases where there *is* an intermediate exact integer representation.

Integer division

Why define integer division (`(a / b)` means `(integer(floor(real(a) / real(b))))`?. This is different from many "C" implementations and from all Fortran implementations, which truncate toward zero.

We want to avoid any surprises due to differences in "C" compiler implementations. Also, if we were to depend upon division truncating toward zero, we'd have to account for a discontinuity at `Original_sample` value zero. Zero would represent twice as large a range of physical values:

- zero would represent (-1+epsilon to 1-epsilon)
- positive n would represent (n to n+1-epsilon)
- negative n would represent (n-1+epsilon to n)

where "epsilon" is the difference between "n" and the closest representable real number that is different from "n". But, as we have defined integer division, all samples represent the same range:

- any n represents (n to n+1-epsilon)

11 Appendix: Revision history

- Version 19980827 is the second release of this document. It introduces the `pCAL` and `sPLT` chunks.

12 Contributors

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

- John Bowler, jbowler@acm.org
- Todd French, tfrench@sandia.gov
- Alaric B. Williams, png@abwillms.demon.co.uk

13 Editor

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

End of PNG Special-Purpose Chunk Listing