PNG-GROUP-DRAFT
MNG
Expires: 03 Mar 1999

PNG Development Group
mpng-list@dworkin.wustl.edu
03 Sep 1998

MNG (Multiple-image Network Graphics) Format Version 19980903

Draft 47, file: draft-mng-19980903.txt

Status of this Memo

This document is an informal draft of the PNG development group.

It is a **proposal**, and the format is subject to change.

Comments on this document can be sent to the PNG specification maintainers at one of the following addresses:

- mpng-list@dworkin.wustl.edu
- png-group@w3c.org
- png-info@uunet.uu.net

Distribution of this memo is unlimited.

At present, the latest version of this document is available on the World Wide Web from

`<URL:ftp://swrinde.nde.swri.edu/pub/mng/documents/>.`

Changes from forty-sixth MNG draft (draft-mng-19980816)

- Filled in section on minimum requirements for decoders.
- Removed brackets around “iCCP”.
- Fixed “tranparencies” typo.
- Added global PLTE specification.
- Clarified some behaviors of object 0.
- Relocated and revised SAVE and SEEK specifications.

- Added a requirement that viewers ignore the gIF* chunks. This is mainly to stifle any expectation that gIFt would result in text actually being drawn over an image.
- Revised Example 4 to demonstrate background image and global PLTE.

Abstract

This document presents the format of a MNG (Multiple-image Network Graphics) datastream. MNG is a multiple-image member of the PNG (Portable Network Graphics) format family, that can contain animations (slide shows) comprised of PNG single-image datastreams. It can also incorporate images in a highly compressible "Delta-PNG" format, defined herein, or a lossy JNG (JPEG Network Graphics) format, also defined herein.

The MNG format provides a mechanism for reusing image data without having to retransmit it. Multiple images can be composed into a "frame" and a group of images can be used as an animated "sprite" that moves from one location to another in subsequent frames. "Palette animations" are also possible.

A MNG frame normally contains a two-dimensional image or a two-dimensional layout of smaller images. It could also contain three-dimensional "voxel" data arranged as a series of two-dimensional planes (or tomographic slices), each plane being represented by a PNG or Delta-PNG datastream.

A Delta-PNG datastream defines an image in terms of a parent PNG or Delta-PNG image and the differences from that image. This has been demonstrated to provide a much more compact way of representing subsequent images than using a complete PNG datastream for each.

The MNG and JNG formats use the same chunk structure that is defined in the PNG specification, and they share other features of the PNG format. Any valid PNG or JNG datastream is also a valid MNG datastream.

This document includes examples that demonstrate various capabilities of MNG including simple movies, composite frames, loops, fades, tiling, scrolling, storage of voxel data, and converting GIF animations to MNG format.

Contents

1 Introduction 5
2 Terminology 7
3 Objects 10
3.1 Embedded objects 10
3.2 Object attributes 10
3.3 Object buffers 11
4 MNG chunks 13
4.1 Critical MNG control chunks 13
4.1.1 `MHDR` MNG datastream header 13
4.1.2 `MEND` End of MNG datastream 14

4.1.3 LOOP, ENDL Define a loop . . . 14
4.2 Critical MNG image defining chunks . . . 16
4.2.1 DEFI Define an object . . . 16
4.2.2 PLTE Global Palette . . . 18
4.2.3 IHDR, PNG chunks, IEND . . . 18
4.2.4 JHDR, JNG chunks, IEND . . . 19
4.2.5 BASI, PNG chunks, IEND . . . 19
4.2.6 CLON Clone an object . . . 21
4.2.7 DHDR, Delta-PNG chunks, IEND . . . 23
4.2.8 PAST Paste an image into another . . . 23
4.2.9 DISC Discard objects . . . 25
4.3 Critical MNG image displaying chunks . . . 26
4.3.1 BACK Background . . . 26
4.3.2 FRAM Delimit a frame . . . 28
4.3.3 MOVE New image location . . . 35
4.3.4 CLIP Object clipping boundaries . . . 35
4.3.5 SHOW Show images . . . 36
4.4 SAVE and SEEK chunks . . . 39
4.4.1 SAVE Save information . . . 39
4.4.2 SEEK Seek point . . . 41
4.5 Ancillary MNG chunks . . . 42
4.5.1 eXPI Export image . . . 42
4.5.2 fPRI Frame priority . . . 43
4.5.3 nEED Resources needed . . . 44
4.5.4 tERm Termination . . . 45
4.5.5 pHYs Physical pixel size . . . 45
4.5.6 PNG ancillary chunks . . . 46
5 The JPEG Network Graphics (JNG) format . . . 47
5.1 JNG critical chunks . . . 47
5.1.1 JHDR JNG header . . . 47
5.1.2 JDAT JNG image data . . . 48
5.1.3 IDAT JNG alpha data . . . 51
5.1.4 IEND End of JNG datastream . . . 51
5.2 JNG ancillary chunks . . . 51
6 The Delta-PNG format . . . 51
6.1 Delta-PNG critical chunks . . . 52
6.1.1 DHDR Delta-PNG datastream header . . . 52
6.1.2 IEND End of Delta-PNG datastream . . . 55
6.1.3 PROM Promotion of parent object . . . 55
6.1.4 IHDR . . . 57
6.1.5 IPNG Incomplete PNG . . . 57
6.1.6 PLTE and tRNS . . . 58
6.1.7 PPLT Incomplete palette . . . 58

6.1.8 JHDR . . . 59
6.1.9 DROP Drop chunks . . . 59
6.1.10 DBYK Drop chunks by keyword . . . 60
6.1.11 ORDR Ordering restrictions . . . 60
6.1.12 Delta-PNG ancillary chunks . . . 61
6.1.13 gAMA, cHRM, iCCP, sRGB Colorspace chunks . . . 61
6.1.14 oFFs and pHYs . . . 61
6.2 Chunk ordering requirements . . . 61
7 Chunk copying rules . . . 62
8 Minimum requirements for MNG-compliant viewers . . . 63
8.1 Required PNG/JNG chunk support . . . 63
8.2 Required MNG chunk support . . . 63
8.3 Required Delta-PNG chunk support . . . 64
9 Recommendations for encoders . . . 65
9.1 Embedded images in LOOPs . . . 65
9.2 Including optional index in SAVE chunk . . . 65
10 Recommendations for decoders . . . 65
10.1 ENDL without matching LOOP . . . 65
10.2 Note on compositing . . . 65
10.3 Retaining object data . . . 66
10.4 Decoder handling of fatal errors . . . 67
10.5 Decoder handling of interlaced images . . . 67
10.6 Decoder handling of palettes . . . 67
10.7 Behavior of single-frame viewers . . . 68
10.8 Clipping . . . 68
11 Recommendations for editors . . . 69
12 Editing datastreams with optional index . . . 69
13 Miscellaneous topics . . . 69
13.1 File name extension . . . 69
13.2 Internet media type . . . 70
13.3 Uniform Resource Locator (URL) . . . 70
14 References . . . 70
15 Security considerations . . . 71
16 Appendix: Examples . . . 72
16.1 Example 1: A single image . . . 72
16.2 Example 2: Very simple movie . . . 72
16.3 Example 3: Simple slideshow . . . 73
16.4 Example 4: A more storage-efficient slideshow . . . 73
16.5 Example 5: Simple movie . . . 74
16.6 Example 6: Single composite frame . . . 75
16.7 Example 7: Movie with sprites . . . 76
16.8 Example 8: Movie with an animated sprite . . . 77
16.9 Example 9: “Fading in” a transparent image . . . 78

16.10 Example 10: Storing three-dimensional images 79
16.11 Example 11: Tiling 80
16.12 Example 12: Scrolling 81
16.13 Example 13: Converting a GIF animation to MNG 82
17 Credits 83

1 Introduction

This specification defines the format of the MNG (Multiple-image Network Graphics) and JNG (JPEG Network Graphics) datastreams.

Note: This specification depends on the PNG Portable Network Graphics specification [PNG]. The PNG specification is available at the PNG home page,

<URL:http://www.cdrom.com/pub/png/>

A MNG datastream describes a sequence of single frames, each of which can be composed of zero or more embedded images or directives to show previously defined images.

A typical MNG datastream consists of

- The 8-byte MNG signature.
- The MHDR chunk.
- Frame definitions. A frame can be
  - A single visible image.
  - A group of visible images delimited by FRAM chunks.
  - Nothing, terminated by a FRAM, SEEK, or MEND chunk, when Framing_mode is 2 or 3, to display a rectangle filled with the background color or background image (or filled with the underlying screen contents, if the BACK chunk has not appeared).
- LOOP–ENDL chunks that enclose frame definitions.
- SEEK chunks that mark points in the datastream where processing can be restarted.
- Various chunks for creating and manipulating images and other objects.
- The MEND chunk.

Images can be "concrete" or "abstract". The distinction allows decoders to use more efficient ways of manipulating images when it is not necessary to retain the image data in its original form or equivalent in order to show it properly on the target display system.

MNG is pronounced "Ming."

When a MNG datastream is stored in a file, it is recommended that ".mng" be used as the file suffix. In network applications, the Media Type "video/x-mng" can be used. Registration of the media type "video/mng" might be pursued at some future date.

The first eight bytes of a MNG datastream are

```
138 77 78 71 13 10 26 10
```

(decimal) which is similar to the PNG signature with "\212 M N G" instead of "\211 P N G" in bytes 0-3.

MNG does not yet accommodate sound or complex sequencing information, but these capabilities might be added at a later date, in a backwards-compatible manner. These issues are being discussed in the `mpng-list@dworkin.wustl.edu` mailing list.

Chunk structure (length, name, data, CRC) and the chunk-naming system are identical to those defined in the PNG specification. As in PNG, all integers that require more than one byte must be in network byte order.

The chunk copying rules for MNG employ the same mechanism as PNG, but with rules that are explained more fully in Chapter 7, below. A MNG editor is not permitted to move unknown chunks across the `SAVE` and `SEEK` chunks, across any chunks that can cause images to be displayed, or into or out of a `IHDR-IEND` or similar sequence.

Note that decoders are not required to follow any decoding models described in this specification nor to follow the instructions in this specification, as long as they produce results identical to those that could be produced by a decoder that did use this model and did follow the instructions.

Each chunk of the MNG datastream or of any embedded object is an independent entity, i.e., no chunk is ever enclosed in the data segment of another chunk.

An independent PNG or JNG datastream, with a PNG or JNG signature, is also a valid MNG datastream that must be recognized and decoded by MNG-compliant decoders. This kind of MNG datastream will contain only a single embedded image.

Because the embedded objects making up a MNG are normally in PNG format, MNG shares the good features of PNG:

- It is unencumbered by patents.
- It is streamable.
- It has excellent, lossless compression.
- It stores up to four channels (red, green, blue, alpha), with up to 16 bits per channel.
- It provides transparency and an alpha channel.
- It provides platform-independent rendition of colors by inclusion of gamma and chromaticity information.
- It provides early detection of common file transmission errors and robust detection of file corruption.
- Single-image GIF files can be losslessly converted to PNG.

- It is complementary to JPEG and does not attempt to replace JPEG for lossy storage of images (however, MNG can accommodate JPEG-encoded images that are wrapped in the PNG-like JNG format that is defined herein).

In addition,

- It provides animation with variable Interframe delays.
- It allows composition of frames containing multiple images.
- It facilitates the use of images as "sprites" or groups of images as "animated sprites" that can be reused in subsequent frames.
- It capitalizes on frame-to-frame similarities to reduce the amount of data that must be included in a datastream.
- It provides "restart" points at which an animation can be safely resumed in case of data loss or corruption, or to which applications can jump if they have random access to the file.
- A "frame priority" chunk allows authors to indicate which frame should be displayed by single-image viewers, and a subset of the frames that should be displayed by slow viewers.
- Images and frames can be given names, allowing authors to mark them for export outside the scope of MNG, where they can be used for icons or similar purposes.
- A series of PNG and JNG images can be losslessly converted to MNG and back to a series of equivalent PNG or JNG images, even when the delta format is used to store them in the MNG.
- JNG provides JPEG with transparency, alpha, and colorspace information.
- Multiple-image GIF files can be losslessly converted to MNG.
- Most JPEG files can be losslessly converted to JNG or MNG, and all JNG datastreams can be losslessly converted to JPEG files.
- It is complementary to MPEG and does not attempt to replace MPEG for lossy storage of video (MNG does, however, provide the capability of including JPEG-encoded images that have been wrapped in the PNG format).

2 Terminology

See also the glossary in the PNG specification.

requirement levels

The words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119, and the word "CAN" is equivalent to the word "MAY" as described therein. "NOT ALLOWED" and "NOT PERMITTED" describe conditions that "MUST NOT" occur. "ALLOWED" and "PERMITTED" describe conditions that "CAN" occur.

abstract image or object

An image whose pixels have a hidden representation, and which does not necessarily carry PNG or JNG chunk data. An image delta cannot be applied to an abstract image. All abstract objects are viewable.

child, or child image

An image produced by applying an image delta to a parent object.

color encoding

File gamma and chromaticity values, or an sRGB rendering intent, iCCP profile, or whatever is involved in mapping between RGB values and colors.

concrete image or object

An image or object whose pixels have a publicly known representation, and which uses a publicly known color encoding. A concrete PNG or JNG image also carries data from other known PNG or JNG chunks that are present.

embedded object or image

A concrete object or image that appears in-line in a MNG datastream.

frame

A layout of a background and zero or more images that is to be displayed as a still frame or as part of an animation. An animation would ideally appear to a perfect observer (with an inhumanly fast visual system) as a sequence of still frames.

frame area

The portion of the display surface whose pixels are inside the frame clipping boundaries as defined by the `CLIP` chunk (see Paragraph 4.3.4, the CLIP chunk specification, below).

frame duration

The amount of time a frame should be visible when an animation is played. In reality, it takes a non-zero amount of time to display a frame. No matter which moment is picked as the "start" of the frame, the frame duration measures the time to the "start" of the next frame.

frame origin

The upper left corner of the output device (frame buffer, screen, window, page, etc.) where the pixels are to be displayed. This is the $\{0,0\}$ position for the purpose of defining frame clipping boundaries, image locations, and image clipping boundaries. Note that in a windowing system, the frame origin might be moved offscreen, but MOVE and CLIP values would still be measured from this offscreen origin.

frozen object

An object whose object attributes set and object buffer are not allowed to be discarded, replaced, or modified.

image delta

An object that can be applied to a concrete image or object to produce another concrete image. For any two concrete images, there exists an image delta that will produce one from the other.

image N or object N

Shorthand for "the object with the object attribute set pointed to by `'Object_id=N'`".

object, Object_id

An image or a nonviewable basis object. The Object_id is an unsigned sixteen-bit number that serves as the identifier of a set of object attributes.

object attributes

Properties of an object such as its existence, potential visibility, location, clipping boundaries, and a pointer to an object buffer. See Object attributes below.

object buffer

A 2D array of pixels or pixel deltas, each of which has color and transparency information. More than one object can point to a given object buffer. See Object buffers below.

parent, parent object, or parent image

An object to which a delta is applied.

potentially visible image

- An not-yet-defined viewable object that is "marked" for on-the-fly display while the embedded image that defines it is being decoded, by setting its `Do_not_show` flag to zero.
- An existing viewable object that is potentially visible (i.e., it is "marked" for being made visible by subsequent `SHOW` chunks), by setting its `Do_not_show` flag to zero.

prologue segment

The first segment, when there is more than one segment.

regular segment

Any segment other than the first (also the first segment, when there is only one).

segment

A part of a MNG datastream starting with the `MHDR` chunk or with a `SEEK` chunk and extending to just before the next `SEEK` chunk (or the `MEND` chunk if there is no next `SEEK` chunk). The `MHDR`, `MEND`, `SAVE`, and `SEEK` chunks are not considered to be a part of any segment.

signal

An entity with a number that can arrive asynchronously at the decoder. More detailed semantics, like whether multiple signals of the same number (or even different numbers) can be queued, are beyond the scope of this specification.

viewable image

A stored object that is capable of being made visible. An image is viewable, while some objects resulting from decoding a `BASI` datastream are not viewable.

visible image

Actually drawn on a display. If an object is visible, a person looking at the display can see it.

3 Objects

An "object", which is identified by an `Object_id`, is an image or it is a nonviewable entity that is created by the `BASI` chunk. The `Object_id` is an unsigned sixteen-bit number that serves as the identifier of a set of object attributes.

An "image" is a viewable object.

3.1 Embedded objects

An embedded object is

- A PNG datastream (`IHDR`, PNG chunks, `IEND`).
- A JNG datastream (`JHDR`, JNG chunks, `IEND`).
- A BASI datastream (`BASI`, PNG chunks, `IEND`).

3.2 Object attributes

Objects have `object attributes` that can be defined and modified by the contents of various MNG chunks. Decoders are responsible for keeping track of them. The simplest decoder might simply establish a 65,536-element array for each attribute, but real applications will undoubtedly use a more memory-efficient method. Object attributes include:

Object exists or does not exist.

- An object comes into existence when
  - an embedded object appears in the MNG datastream, and the most recent `DEFI` chunk specified a nonzero `Object_id`..
  - a `CLON` chunk creates it.
- An object ceases to exist when it does not have the "frozen" attribute and
  - It is the subject of a `DISC` chunk.
  - An empty `DISC` chunk appears.

- A SEEK chunk appears.
- The MEND chunk appears (or the IEND chunk appears in a simple PNG or JNG file).
- A DEFI chunk with the same Object_id appears.
- A new embedded object with the same Object_id replaces it without an intervening DEFI chunk. (in this case, the new object inherits the object attribute set from the previous object with the same nonzero Object_id; when Object_id = 0, it receives the default DEFI attribute set).
- It has Object_id = 0 and its IEND chunk appears (in this case, the object attribute set can be immediately discarded).

Pointer to an object buffer.

Every object (except for object 0) has an object buffer. Multiple objects can point to the same object buffer. The representation of a pointer is decided by the application; pointers never appear explicitly in a MNG datastream. Decoders can also create an object buffer for object 0, if that is more convenient, but the information in that buffer cannot be depended upon to exist after the image has been displayed, nor can that buffer become "frozen".

Frozen or not frozen.

All objects are initially "not frozen". Any objects in existence (except for object 0) when the SAVE chunk is encountered become "frozen", along with the object buffers that they point to.

Potential visibility.

The "potential visibility" of an object is determined by the Do_not_show byte of the DEFI or CLON chunk that introduced it. The "potential visibility" of viewable objects can be changed by the SHOW chunk. When an embedded object is "potentially visible" it can be displayed "on-the-fly" as it is being decoded. Later, the SHOW chunk can direct that a "potentially visible" object be displayed.

Location.

The X and Y location of an object is determined by the DEFI chunk that introduced it, and can be changed by the MOVE chunk.

Clipping boundaries.

The clipping boundaries of an object are determined by the DEFI chunk that introduced it, and can be changed by means of the CLIP chunk.

3.3 Object buffers

An object buffer is created by the appearance of an embedded object in the datastream, with a nonzero Object_id, or by the appearance of a CLON chunk that specifies a "full clone". The contents of an object buffer can be modified by processing an image delta or a PAST chunk.

Object buffers contain a 2D array of pixel data and can contain additional information. In addition, decoders are responsible for keeping track of some properties of the data in the object buffer:

Object is viewable.

Any object that points to a viewable object buffer can be made potentially visible, but one that points to a nonviewable one cannot. Any attempt to do so must be ignored.

- A PNG or JNG datastream always has the "viewable" attribute.
- The "viewable" attribute of a `BASI` datastream is defined in the `BASI` chunk. Only `BASI` datastreams that produce legal PNG files can be declared "viewable".
- When a Delta-PNG is applied to a parent object, the resulting object buffer always has the "viewable" attribute.

Format of data in the object buffer.

The data format can be

- A concrete PNG or JNG object.

  A concrete object must be stored by the decoder in a form that retains the complete object description, sufficient to regenerate the original object description or its equivalent without loss. Its pixels have a publicly known representation, and it uses a publicly known color encoding.

  – In the case of a PNG image, the object also carries data from other known PNG chunks that are present. This means that the decoder must store sufficient information to make it possible to restore exactly the original decoded and unfiltered pixel samples as they existed prior to any gamma correction (but not the original compressed datastream or line-by-line filter selections and "zlib" compression flags), and data from the `IHDR` and `PLTE` chunks and any additional recognized PNG chunks such as `gAMA`, `cHRM`, and `tRNS` that the application plans to use.

  – In the case of a JNG image, the object also carries data from other known JNG chunks that are present. This means that the decoder must store sufficient information to make it possible to restore exactly the original JPEG datastream [or set of DCTs?] and decoded alpha channel as they existed in the original JNG file, and data from the `JHDR` chunk and any additional recognized JNG chunks such as `gAMA` and `cHRM` that the application plans to use.

  – A decoder that recreates PNG or JNG files from a series of PNG, JNG, and Delta-PNG datastreams will also have to store the contents of any unknown chunks that it finds, in case they turn out to be safe to copy (see DROP, Paragraph 6.1.10, DBYK, and Paragraph 6.1.11, ORDR, below).

- An abstract image.

  An abstract image can be stored by the decoder in any form that is convenient, such as an X Window System "pixmap", even though that form might not have sufficient resolution for exact, lossless conversion. In the case of a PNG image, the pixels could be stored after the gamma and chromaticity corrections have been made, and the sample depth could be the same as the display hardware, even though it is smaller than the original sample depth. Similarly, a JNG image could be stored in the same form, after the pixels have been decoded, converted to RGB form, and

gamma-corrected. It is always safe, however, to store an abstract image as though it were concrete, if decoders do not wish to take advantage of the distinction between abstract and concrete objects.

Frozen or not frozen.

All object buffers are initially "not frozen". Any object buffers in existence when the SAVE chunk is encountered become "frozen". Decoders do not actually have to store this flag except as a sanity check, because they can depend on the fact that a "frozen" object buffer will always have at least one "frozen" object whose "buffer pointer" points to it.

A reference count.

When an object buffer is first created, its reference count is set to 1.

When a partial clone is made of an object via the CLON chunk, the reference count for the object buffer is incremented, and no new object buffer is created.

When an object is discarded, the reference count of the object buffer that it points to is decremented, and the object buffer is also discarded if the resulting reference count is zero.

4 MNG chunks

This chapter describes chunks that can appear at the top level of a MNG datastream. Unless otherwise specified in the Delta-PNG chapter of this specification, they need not be recognized there.

4.1 Critical MNG control chunks

This section describes critical MNG control chunks. MNG-compliant decoders must recognize and process them.

4.1.1 MHDR MNG datastream header

The MHDR chunk is always first in all MNG datastreams except for those that consist of a PNG datastream with a PNG signature. The MHDR chunk contains exactly 12 bytes:

```
4 bytes: Frame_width  (unsigned integer).

4 bytes: Frame_height (unsigned integer).

4 bytes: Ticks_per_second  (unsigned integer).
         This is the unit used by the FRAM chunk to
         specify Frame_duration and Sync_timeout.
```

The Frame_width and Frame_height fields give the intended display size (measured in pixels) and provide default clipping boundaries (see Paragraph 4.3.4, below). It is strongly recommended that these be set to zero if the MNG datastream contains no visible images.

The Ticks_per_second field must be nonzero if the datastream contains more than one frame. It is strongly recommended that this be set to zero if the datastream contains exactly one frame, so simple applications that only examine file headers can use this field to identify single-frame MNGs.

4.1.2 MEND End of MNG datastream

The MEND chunk's data length is zero. It signifies the end of a MNG datastream.

4.1.3 LOOP, ENDL Define a loop

The LOOP chunk provides a "shorthand" notation that can be used to avoid having to repeat identical chunks in a MNG datastream. Its contents are the first three, five, six, or more of the following fields:

```
1  byte: Nest_level (unsigned integer).
1  byte: Termination_condition (unsigned integer).
      0: deterministic.
      1: decoder discretion.
      2: user discretion.
      3: external signal
4 bytes: Repeat_count (unsigned integer), range
         [0..2^31-1].
4 bytes: Iteration_min (unsigned integer).  Omit
    if Termination_condition=0.
4 bytes: Iteration_max (unsigned integer).  Omit
    if Termination_condition=0.
4 bytes: Signal_number (unsigned integer).  Omit
    if Termination_condition != 3.
4 bytes: Additional Signal_number.  Omit if
    Termination_condition != 3.
etc.
```

Decoders must treat the chunks enclosed in a loop exactly as if they had been repeatedly spelled out. Therefore, during the first iteration of the loop, the parent objects for any Delta-PNG datastreams in the loop are the images in existence prior to entering the LOOP chunk, but in subsequent iterations these parent objects might have been modified. The Termination_condition field can be used to inform decoders that it is safe to change the number of loop iterations.

Simple decoders can ignore all fields except for the Repeat_count.

When the LOOP chunk is present, an ENDL chunk with the same Nest_level must be present later in the MNG datastream. Loops can be nested. Each inner loop must have a higher value of Nest_level than the loop that encloses it, though not necessarily exactly one greater.

The termination condition specifies how the actual number of iterations is determined. It is very similar to the termination condition field of FRAM, and can take the same values:

deterministic

The loop terminates after exactly the number of iterations specified by the iteration count. This value must be used if altering the number of repetitions would mess up the MNG stream, but can be used merely to preserve the author's intent.

decoder-discretion

The number of iterations can be chosen by the decoder, and must fall between the Iteration_min and Iteration_max. If the decoder has no reason to choose its own value, it should use the iteration count. One example of a decoder wishing to choose its own value is a real-time streaming decoder hovering at a loop while waiting for its input buffer to fill to a comfortable level.

user-discretion

The number of iterations should be chosen by the user (e.g., by pressing the <escape> key), but the decoder must enforce the Iteration_min and Iteration_max limits. Some decoders might not be able to interact with the user, and many decoders will find that nested user-discretion loops present too great of a user-interface challenge, so the <user-discretion> condition will probably usually degenerate into the <decoder-discretion> condition.

external-signal

The number of iterations must lie between the Iteration_min and Iteration_max limits, but the exact number can be determined by the arrival of a signal whose number matches one of the Signal_number fields. There can be any number of Signal_number fields. Signal_number = 0 is reserved to represent any input from a keyboard or pointing device, and 1-255 are reserved to represent the corresponding ASCII character, received from a keyboard or simulated keyboard, and values 256-1023 are reserved for future definition by this specification.

The Iteration_min and Iteration_max can be omitted only if the condition is <deterministic>. The Repeat_count, Iteration_min, and Iteration_max can be any non-negative integers or <infinity>, but they must satisfy Iteration_min <= Repeat_count <= Iteration_max.

If Repeat_count is zero, the loop is done zero times. Upon encountering a LOOP chunk with Repeat_count=0, decoders simply skip chunks until the matching ENDL chunk is found, and resume processing with the chunk immediately following it.

The Signal_number can be omitted only if the termination condition is not <external-signal>. Signal_number=0 is reserved to represent a user input from a keyboard or pointing device.

The ENDL chunk ends a loop that begins with the LOOP chunk. It contains a single one-byte field:

```
1 byte: nest_level (unsigned integer),
        range [0..255].
```

When the ENDL chunk is encountered, the loop Repeat_count is decremented, if it is not already zero. If the result is nonzero, processing resumes at the beginning of the loop. Otherwise processing resumes with the chunk immediately following the ENDL chunk.

When the ENDL chunk is present, a LOOP chunk with the same nest_level must be present earlier in the MNG datastream. See below.

The SEEK chunk is permitted inside a LOOP–ENDL pair, but not when Repeat_count=0. The SAVE chunk is not permitted inside a LOOP–ENDL pair. To rerun an entire datastream that includes a SAVE chunk, use the tERm chunk instead, with termination action 4. See Paragraph 4.5.4, below.

4.2 Critical MNG image defining chunks

4.2.1 DEFI Define an object

The DEFI chunk sets the default object attribute set (Object_id, Potential_visibility, Concrete_flag, location, and clipping boundaries) for any subsequent images that are defined with IHDR–IEND, JHDR–IEND, or BASI–IEND datastreams. The DEFI chunk contains 2, 3, 4, 12, or 28 bytes:

```
2 bytes: Object_id (unsigned integer) identifier
         to be given to the objects that follow the
         DEFI chunk.

1 byte:  Do_not_show (unsigned integer)
     0:  Make the objects potentially visible.
     1:  Do not make the objects potentially visible.
     This field can be omitted if the Concrete_flag,
     location, and clipping boundary fields
     are also omitted.  When it is omitted, the image
     is potentially visible (Do_not_show=0).

1 byte:  Concrete_flag (unsigned integer)
     0:  Make the objects "abstract" (image can not
         be the source for a Delta-PNG)
     1:  Make the objects "concrete" (object can be
         modified by a Delta-PNG).
     This field can be omitted if the location and
     clipping boundary fields are also omitted.
     When it is omitted, the object is made "abstract"
     (Concrete_flag=0).

4 bytes: X_location (signed integer).
     The X_location and Y_location fields can be
     omitted if the clipping boundaries are also
     omitted.  If so, decoders must assume default
     values {X_location,Y_location}={0,0}.
```

```
4 bytes: Y_location (signed integer).

4 bytes: Left_cb (signed integer). Left clipping
     boundary.  The Left_cb, Right_cb, Top_cb, and
     Bottom_cb fields can be omitted as a group.
     If so, decoders must assume default values
     {0, Frame_width, 0, Frame_height}.

4 bytes: Right_cb (signed integer).

4 bytes: Top_cb (signed integer).

4 bytes: Bottom_cb (signed integer).
```

If the object number for an object is nonzero, subsequent chunks can use this number to identify it.

When the object number for an object is zero, its object buffer can be discarded immediately after it has been processed, and it can be treated as an "abstract" image, regardless of the contents of the "Concrete_flag" field.

Negative values are permitted for the X and Y location and clipping boundaries. The positive directions are downward and rightward from the frame origin.

Multiple IHDR-IEND, JHDR-IEND, and BASI-IEND objects can follow a single DEFI chunk. When Object_id is nonzero, the DEFI chunk values remain in effect until another DEFI chunk or a SEEK chunk appears, unless they are modified by SHOW, MOVE, or CLIP chunks. When Object_id=0, the DEFI chunk values are discarded after the object's IEND chunk is processed. The Object_id and .Concrete_flag can only be changed by using another DEFI chunk. If no DEFI chunk is in effect (either because there is none in the datastream, or because a SEEK chunk has caused it to be discarded), the decoder must use the following default values:

```
      Object_id = 0
    Do_not_show = 0
Concrete_flag = 0
   X_location = 0
   Y_location = 0
      Left_cb = 0
     Right_cb = Frame_width
       Top_cb = 0
    Bottom_cb = Frame_height
```

If Object_id is an identifier that already exists when a DEFI, the object attribute set (except for the pointer to the object buffer) is immediately replaced, but the contents of the object buffer do not change until an IHDR, JHDR, or BASI chunk is encountered; then the contents of the object buffer previously associated with the identifier are replaced with the new embedded object data. Note that if the object has partial clones, the clones will also be affected.

4.2.2 PLTE Global Palette

The PLTE chunk has the same format as a PNG PLTE chunk. It provides a global palette that is inherited by PNG datastreams that contain an empty PLTE chunk.

4.2.3 IHDR, PNG chunks, IEND

A PNG (Portable Network Graphics) datastream.

See the PNG specification [PNG] and the PNG Special Purpose Chunks document [PNG-EXT] for the format of the PNG chunks.

Any chunks between IHDR and IEND are written and decoded according to the PNG specification.

If a global PLTE chunk appears in the top-level MNG datastream, the PNG datastream can have an empty PLTE chunk, which directs that the global PLTE data be used. If an empty PLTE chunk is not present, the data is not inherited. MNG applications that recreate PNG files must write the global PLTE chunk rather than the empty one in the output PNG file. It is an error for the PNG datastream to contain an empty PLTE chunk when the global PLTE chunk is not present or has been nullified.

If the PNG sRGB, gAMA, iCCP, or cHRM chunks appear in the top-level MNG datastream (and have not been nullified), but none of them appear in the PNG datastream, then the values are inherited from the top level as though the chunks had actually appeared in the PNG datastream. Data from such chunks appearing in the PNG datastream take preference over the inherited values. If any one of these chunks, or any future chunk that defines the color space, appears in the PNG datastream, none of them is inherited. MNG applications that recreate PNG files must write these chunks, if they are inherited, in the output PNG files.

If the PNG sPLT chunk appears in the top-level MNG datastream, it takes preference over any sPLT chunk appearing in the PNG datastream. MNG applications that recreate PNG files should not copy top-level sPLT chunks to the output PNG files, because a suggested palette for rendering a group of images is not necessarily the best palette for rendering a single image.

When Framing_mode != 1 and Framing_mode != 4 from the MNG FRAM chunk, the PNG oFFs and pHYs chunks and any future chunks that attempt to set the pixel dimensions or the drawing location must be ignored by MNG viewers and simply copied (according to the copying rules) by MNG editors. When Framing_mode=1 or Framing_mode=4 (i.e., when each PNG image is an individual frame), these chunks must be treated as described in the PNG specification.

The PNG gIFg, gIFt, and gIFx chunks must be ignored by viewers and must be copied according to the copying rules by MNG editors.

If Do_not_show=0 for the image when the IHDR chunk is encountered, a viewer can choose to display the image while it is being decoded, perhaps taking advantage of the PNG interlacing method, or to display it after decoding is complete.

If Object_id=0, there is no need to store the pixel data after displaying it.

If Concrete_flag=1 and Object_id != 0, the decoder must store the original pixel data losslessly, along with data from other recognized PNG chunks, because it is possible that a subsequent Delta-PNG stream might want to modify it. If Concrete_flag=0, the decoder can store the pixel data in any form that it chooses.

If an object already exists with the same Object_id, the contents of its object buffer are replaced with the new data.

4.2.4 JHDR, JNG chunks, IEND

A JNG (JPEG Network Graphics) datastream.

See the JNG specification below for the format of the JNG datastream.

Any chunks between JHDR and IEND are written and decoded according to the JNG specification.

The remaining discussion in the previous paragraph about PNG datastreams also applies to JNG datastreams.

4.2.5 BASI, PNG chunks, IEND

The BASI chunk introduces a basis object, which, while it might be incomplete, can serve as a parent object to which a delta image can be applied.

The first 13 bytes of the BASI chunk are identical to those of the IHDR chunk. An optional 8 additional bytes provide 16-bit {red, green, blue, alpha} values, that are used to fill the entire basis object when the IDAT chunk is not present, and a 1-byte "viewable" flag can be present.

```
4 bytes: Width (unsigned integer)
4 bytes: Height (unsigned integer)
1 byte:  Sample_depth (unsigned integer)
1 byte:  Color_type (unsigned integer)
1 byte:  Compression_method (unsigned integer)
1 byte:  Filter_type (unsigned integer)
1 byte:  Interlace_type (unsigned integer)
2 bytes: Red_sample (unsigned integer)
2 bytes: Green_sample (unsigned integer)
2 bytes: Blue_sample (unsigned integer)
2 bytes: Alpha_sample (unsigned integer)
1 byte:  Viewable (unsigned integer)
      0: basis object is not viewable
      1: basis object is viewable
```

The Alpha_sample can be omitted if the Viewable field is also omitted. If so, and the Color_type is one that requires alpha, the alpha value corresponding to an opaque pixel will be used. If the color samples are omitted, zeroes will be used. The decoder is responsible for converting the color and alpha samples to the appropriate format and sample depth for the specified Color_type. When Color_type=3,

the decoder must generate a palette of length 2^Sample_depth, whose first entry contains the given {Red_sample, Green_sample, Blue_sample} triple, and whose remaining entries are filled with zeroes. If the Viewable field is omitted, the object is not viewable.

The color and alpha samples are written as four 16-bit samples regardless of the Color_type and Sample_depth. When the Sample_depth is less than sixteen, the least significant bits are used and the remaining bits must be zero filled. When Color_type=3, the least significant byte of each color sample is used and the upper byte must be zero.

The BASI datastream contains PNG chunks, but is not necessarily a PNG datastream. It can be incomplete or empty and it can deviate in certain ways from the PNG specification. It can serve as a parent object for a Delta-PNG datastream, which must supply the missing data or correct the other deviations before the image is displayed. The end of the datastream is denoted by an IEND chunk.

The permitted deviations from the PNG format are:

- Multiple instances of some chunks can be present even though the PNG specification allows only one. The subsequent Delta-PNG that uses this as the parent object must select only one, through the DBYK or similar mechanism.
- The IDAT chunk can be omitted or there can be a single empty IDAT chunk. If so, all of the the pixels are filled with the given color and alpha samples. A subsequent Delta-PNG that uses this as the parent object can supply the IDAT chunk or chunks.
- The PLTE chunk can be omitted or incomplete even when the Color_type=3. If so, the subsequent Delta-PNG that uses this as the parent object can supply a complete PLTE chunk, if the single-entry palette that is generated is not desired.

The BASI chunk can be used to introduce such things as a library of faLT chunks from which one or another can be selected for use with any single image, or it can be used to introduce a simple blank or colored rectangle into which other images will be pasted by means of the PAST chunk.

A BASI chunk appearing in a MNG datastream must be preceded by a DEFI chunk that gives the Object_id, location, and potential visibility for the basis object. The Concrete_flag can be either 0 (abstract) or 1 (concrete), depending on whether the basis image is intended for subsequent use by a Delta-PNG datastream or not. When it is abstract it must also be viewable. When Do_not_show=0 or Viewable=1, the resulting image, after the pixel samples are filled in, must be a legal PNG image. If Do_not_show=0, a viewer is expected to display it immediately, as if it were decoding a PNG datastream.

If an object already exists with the same Object_id, the contents of its object buffer are replaced with the new data.

Top-level gAMA, sRGB, cHRM, iCCP, and sPLT chunks are inherited by a BASI datastream in the same manner as by a PNG datastream.

No provision is made in this specification for storing a BASI datastream as a standalone file. A BASI datastream will normally be found as a component of a MNG datastream. Applications that need to store a BASI datastream separately should use a different file signature and filename extension, or they can wrap it in a

MNG datastream consisting of the MNG signature, the MHDR chunk, the BASI datastream, and the MEND chunk.

4.2.6 CLON Clone an object

Create a clone (a new copy) of an image, with a new Object_id. The CLON chunk contains 4, 5, 6, 7, or 16 bytes.

```
2 bytes: Source_id (nonzero unsigned integer) identifier
         of the parent object to be cloned.

2 bytes: Clone_id (nonzero unsigned integer) identifier
         to be given to the clone (new copy) to serve as
         the Object_id of the new object.

1  byte: Clone_type (unsigned integer)

       0: full clone of the object attributes set and
          the object buffer

       1: partial clone; only object attributes set
          (the location, clipping boundaries, and
          potential visibility) are copied and a link
          is made to the object buffer.

       2: renumber object (this is equivalent to
          "CLON Source_id Clone_id 1 DISC Source_id").

       This field can be omitted if the
       "Do_not_show" field is also omitted.
       If it is omitted, the Clone_type defaults to
       zero (full clone).

1 byte:  Do_not_show (unsigned integer)
     0:  Make the clone potentially visible.
     1:  Do not make the clone potentially visible.
     This field can be omitted if the Concrete_flag and
     location fields are also omitted.  When it is
     omitted, the object retains the potential
     visibility of the parent object.

1 byte:  Concrete_flag (unsigned integer)
     0:  Concrete_flag is the same as that of the parent
         image.
     1:  Make the clone "abstract" (Concrete_flag=0).
```

```
      This field can be omitted if the location fields
      are also omitted.  When it is omitted, the object
      retains the Concrete_flag of the parent object.

1  byte: Loca_delta_type (unsigned integer)

       0: location data gives X_location and
          Y_location directly.

       1: New positions are determined by adding
          the location data to the position of the
          parent object.

       This field, together with the X_location and
       Y_location fields, can be omitted.  When they
       are omitted, the clone has the same location
       as the parent object.

4 bytes: X_location or Delta_X_location
         (signed integer).

4 bytes: Y_location or Delta_Y_location
         (signed integer).
```

The `Source_id` must be an existing object identifier, and the `Clone_id` must not be an existing object identifier.

Negative values are permitted for the X and Y position. The positive directions are downward and rightward from the frame origin.

The clone is initially identical to the parent object except for the location and potential visibility. It has the same clipping boundaries as the parent object. Subsequent `DHDR`, `SHOW`, `CLON`, `CLIP`, `MOVE`, `PAST`, and `DISC` chunks can use the `Clone_id` to identify it. If the parent object is not a viewable image, neither is the clone.

Subsequent chunks can modify, show, or discard a full clone or modify its potential visibility, location and clipping boundaries without affecting the parent object, or they can modify, show, or discard the parent object or modify its object attribute set without affecting the clone.

The `Concrete_flag` byte must be zero when the `Clone_type` byte is nonzero.

If an object has partial clones, and the data in the object buffer of a parent object or any of its partial clones is modified, the parent object and all of its partial clones are changed. Decoders must take care that when the parent object or any partial clone is discarded, the object buffer is not discarded until the last remaining one of them is discarded. Only the location, potential visibility, and clipping boundaries can be changed independently for each partial clone.

4.2.7 DHDR, Delta-PNG chunks, IEND

A Delta-PNG (PNG-Delta) datastream.

See Chapter 6, The Delta-PNG Format, below, for the format of the Delta-PNG datastream. Any chunks between DHDR and IEND are written and decoded according to the Delta-PNG format. The Object_id of the Delta-PNG DHDR chunk must point to an existing parent object. The resulting image is immediately displayed if its Do_not_show=0. The parent object must be concrete (i.e., it must have Concrete_flag=1).

4.2.8 PAST Paste an image into another

Paste an image or images identified by Source_id, or part of it, into an existing abstract image identified by Destination_id.

The PAST chunk contains a 2-byte Destination_id and 9 bytes giving a "target location", plus one or more 30-byte source data sequences.

```
2 bytes: Destination_id (unsigned integer)

1 byte:  Target_delta_type (unsigned integer)
     0:  Target_x and Target_y are given directly
     1:  Target_x and Target_y are deltas from
         their previous values in a PAST chunk
         with the same Destination_id
     2:  Target_x and Target_y are deltas from
         their previous values in the immediately
         preceding PAST chunk regardless of its
         Destination_id

4 bytes: Target_x (signed integer), measured
         rightward from the left edge of the
         destination image

4 bytes: Target_y (signed integer), measured
         downward from the top edge of the
         destination image

2 bytes: Source_id (unsigned nonzero integer)
         an image to be pasted in.

1 byte: Composition_mode (unsigned integer)
     0: composite_over
     1: replace
     2: composite_under
```

```
1 byte: Orientation (unsigned integer)
        The source image, which contains a
        graphic symbol "p" in these examples,
        is flipped to another orientation.

     0: same as source image (p->p)

     2: flipped left-right, then up-down (p->d)

     4: flipped left-right (p->q)

     6: flipped up-down (p->b)

     8: tiled with source image, to fill the
        clipping boundaries.  The upper left
        corner of the assembly is positioned
        according to the prescribed offsets.

           p -> pppp
                pppp

1 byte:  Offset_delta_type (unsigned integer)
     0:  offsets are measured from the {0,0} pixel
         in the destination image
     1:  offsets are measured from the
         {Target_x,Target_y} pixel in the destination
         image

4 bytes: Xoffset or delta xoffset (signed integer)
4 bytes: Xoffset or delta yoffset (signed integer)

1 byte:  Boundary_delta_type (unsigned integer)
     0:  boundaries are measured from the {0,0} pixel
         in the destination image
     1:  boundaries are measured from the
         {Target_x,Target_y} pixel in the destination
         image

4 bytes: Left_pb or delta Left_pb (signed integer)
4 bytes: Right_pb or delta Right_pb (signed integer)
4 bytes: Top_pb or delta Top_pb (signed integer)
4 bytes: Bottom_pb or delta Bottom_pb (signed integer)
etc.
```

The destination image must have the "abstract" property (`Concrete_flag=0`). When `Destination_id=0`, the resulting image is "write-only" and therefore only "composite-over" (`Composition_mode=0`) operations are permitted.

The source images can be "abstract" or "concrete" and have any `Color_type` and `Sample_depth`. They

must have the "viewable" property. The number of source images is `((chunk_length-11)/30)`.

The `Xoffset` and `Yoffset` distances and the clipping boundaries are measured, in pixels, positive rightward and downward from either the {`0,0`} pixel of the destination image or the {`Target_x, Target_y`} position in the destination image. They do not necessarily have to fall within the destination image. Only those pixels of the source image that fall within the destination image and also within the specified clipping boundaries will be copied into the destination image. Note that the coordinate system for offsets and clipping is with respect to the upper lefthand corner of the destination image, which is not necessarily the same coordinate system used by the `MOVE` and `CLIP` chunks. If the source image has been flipped or rotated, `Xoffset` and `Yoffset` give the location of its new upper left hand corner. When it is tiled, the offsets give the location of the upper left hand corner of the assembly of tiles.

When `Composition_mode=0`, any non-opaque pixels in the source image are combined with those of the destination image. If the destination pixel is also non-opaque, the resulting pixel will be non-opaque.

When `Composition_mode=1`, all pixels simply replace those in the destination image. This mode can be used to make a transparent hole in an opaque image.

When `Composition_mode=2`, any non-opaque pixels in the destination image are combined with those of the source image. If the source pixel is also non-opaque, the resulting pixel will be non-opaque.

The order of composition is the same as the order that the `Source_id's` appear in the list (but a decoder can do the composition in any order it pleases, or all at once, provided that the resulting destination image is the same as if it had actually performed each composition in the specified order). Decoders must be careful when the destination image equals the source image–the pixels to be drawn are the ones that existed before the drawing operation began.

The `MOVE` or `CLIP` information associated with the `Destination_id` and the `Source_id's` is not used in the `PAST` operation (but if a decoder is simultaneously updating and displaying the `Destination_id`, the `MOVE` and `CLIP` for the `Destination_id` is used in the display operation).

4.2.9 `DISC` Discard objects

The `DISC` chunk can be used to inform the decoder that it can discard the object data associated with the associated object identifiers. Whether the decoder actually discards the data or not, it must not use it after encountering the `DISC` chunk.

The chunk contains a sequence of zero or more two-byte object identifiers. The number of objects to be discarded is the the chunk's data length, divided by two.

```
2 bytes: Discard_id (nonzero unsigned integer) identifier
         of object that can be discarded.  All
         information pertaining to the corresponding
         object can be discarded.
etc.
```

If the DISC chunk is empty, all objects except those preceding the SAVE chunk (i.e., the "frozen" objects) can be discarded. If a SAVE chunk has not been encountered, all objects can be discarded. Note that each appearance of a SEEK chunk in the datastream implies an empty DISC chunk.

When an object is discarded, any location, potential visibility, and clipping boundary data associated with it is also discarded.

It is not an error to include an Object_id in the Discard_id list, when no such object has been stored, or when the object has already been discarded.

It is an error to name explicitly any "frozen" object in the DISC list.

When the object is a partial clone or is the source of a partial clone that has not been discarded, only the object attribute set (location, potential visibility, and clipping boundaries) can be discarded. The data in the object buffer must be retained until the last remaining partial clone is discarded.

4.3 Critical MNG image displaying chunks

4.3.1 BACK Background

The BACK chunk suggests a background color against which transparent, clipped, or less-than-full-frame images can be displayed.

```
2 bytes: Red_background (unsigned integer).

2 bytes: Green_background (unsigned integer).

2 bytes: Blue_background (unsigned integer).

1 byte: Mandatory_background (unsigned integer)
     0: background color and background image are
        advisory; applications can use them if they
        choose to.
     1: background color is mandatory; applications
        must use it.  Background image is advisory.
     2: background image is mandatory; applications
        must use it.  Background color is advisory.
     3: background color and background image are
        both mandatory; applications must use them.
     This byte can be omitted if the
     Background_image_id is also omitted; if so, the
     Background color is advisory.
```

```
2 bytes: Background_image_id (unsigned nonzero integer)
        Object_id of an image that is to be displayed
        as the background of each frame.  If it is
        not full-frame, the remainder of the frame
        is filled with the background color.  This
        field can be omitted; if so, no background
        image is defined, and the background image
        from a previous BACK chunk becomes undefined.

1 byte: Background_tiling (unsigned integer)
     0: do not tile the background.
     1: tile the background with the background
        image.  This field can be omitted; if so,
        background_tile=0.
```

Viewers are expected to composite every frame in the MNG datastream against a fresh copy of the background. The images and the background are both clipped to the frame boundaries given in the FRAM chunk. The background image (or tiled assembly) is also clipped to its own boundaries and located like any other image. When the background image is used for tiling, the upper left tile is located according to the background image's location attributes and the entire assembly is clipped according to its clipping attributes. Viewers might actually follow some other procedure, but the final appearance of each frame must be the same as if they had filled the area within the frame boundaries with the background color, then displayed the background image, and then displayed the foreground image (or images), without delay.

It is not an error to specify a Background_image_id when such an image does not exist or ceases to exist for some reason. Viewers must be prepared to fall back to using the background color in this event. They also must be prepared for the contents, location, and clipping boundaries of the background image to change, just like any other object, if it has not been "frozen".

The three BACK components are always interpreted in the current color space as defined by any top-level gAMA, cHRM, iCCP, sRGB chunks that have appeared prior to the BACK chunk in the MNG datastream. If no such chunks appear, the color space is unknown.

The colorspace of the background image, if one is used, is determined in the same manner as the colorspace of any other image.

When the BACK chunk appears between FRAM chunks, it applies to the upcoming frame, not to the current one. When Framing_mode=1, it takes effect immediately prior to the next IHDR, JHDR, DHDR, PAST, or SHOW chunk in the datastream.

Multiple instances of the BACK chunk are permitted in a MNG datastream.

The BACK chunk can be omitted. If a background is required and the BACK chunk is omitted, then the viewer must supply its own background.

In practice, most applications that use MNG as part of a larger composition should ignore the BACK data if Mandatory_background=0 and the application already has its own background definition. This will frequently be the case in World Wide Web pages, to achieve nonrectangular transparent animations displayed against the page's background.

4.3.2 FRAM Delimit a frame

The FRAM chunk marks the beginning of a group of chunks that describe a frame. It also can set parameters that govern how the decoder is to behave when it encounters a FRAM chunk.

An empty FRAM chunk is just a frame delimiter. A nonempty one is a frame delimiter, and it also changes FRAM parameters, either for the upcoming frame or until reset. When the FRAM chunk is not empty, it contains a Framing_mode byte, an optional name string, a zero-byte separator, plus four 1-byte fields plus a variable number of optional fields.

```
1 byte:  Framing_mode.

     0:  Don't change Framing_mode.

     1:  Each image is an individual frame; the
         frame area background is restored
         and an Interframe delay occurs after
         each image.  This is the default
         Framing_mode.

     2:  The group of images appearing prior to
         the next "FRAM" chunk form a composite
         frame. Images are caused to appear with
         explicit or implied "SHOW" chunks.  The
         level, or stacking order, of each image
         is given by its order of appearance in the
         datastream.  The frame area background is
         restored prior to displaying the first
         image, and no Interframe delay occurs
         between the images.

     3:  The group of "n" images appearing prior to
         the next "FRAM" chunk define "n" frames as
         they accumulate, and an Interframe delay
         occurs after each image is defined or
         appears in a "SHOW" chunk.  The frame area
         background is only restored prior to
         displaying the first image.  Images appear
         one by one and remain visible after being
         explicitly caused to appear.  The level,
         or stacking order, of each image is given
         by its order of appearance in the datastream.
```

```
          4: This is the same as Framing_mode=1, except
             the frame area background is not restored
             prior to displaying each image.  Use this
             mode to avoid unnecessary clearing of the
             display when the first image covers the
             entire frame area, when the sequence
             consists of a series of full-frame opaque
             images, and when a subsequent frame is
             created by overlaying a smaller, opaque
             image over the existing frame.

          5: This is the same as Framing_mode=2, except
             the frame area background is not restored
             prior to displaying the frame.

    n bytes: Frame_name (Latin-1 Text).  Can be
         omitted; if so, the frame is nameless.

    1 byte:  zero-byte separator. Must be omitted
         if all remaining fields are also omitted.

    1 byte:  Change_interframe_delay
         0:  no.
         1:  yes, for the next frame only.
         2:  yes, also reset default.

         This field and the next three must be omitted
         as a group if no frame parameters other than
         the Framing_mode are changed.

    1 byte:  Change_sync_timeout_and_termination.
         0:  no.
         1:  deterministic, for the next frame only.
         2:  deterministic, also reset default.
         3:  decoder-discretion, for the next frame only.
         4:  decoder-discretion, also reset default.
         5:  user-discretion, for the next frame only.
         6:  user-discretion, also reset default.
         7:  external-signal, for the next frame only.
         8:  external-signal, also reset default.

    1 byte:  Change_frame_clipping_boundaries.
         0:  no.
         1:  yes, for the next frame only.
         2:  yes, also reset default.
```

```
1 byte:  Change_sync_id_list.
     0:  no.
     1:  yes, for this frame only.
     2:  yes, also reset default list.

4 bytes: Interframe_delay (unsigned integer) (Must
         be omitted if Change_interframe_delay=0).
         The range is [0..2^31-1] ticks.

4 bytes: Sync_timeout (unsigned integer) (omit if
         Change_Sync_timeout=0). The range is
         [0..2^31-1].  The value 2^31-1 (0x7fffffff)
         ticks represents an infinite timeout period.

1  byte: Fb_delta_type (unsigned integer).
      0: frame clipping boundary values are given
         directly.
      1: frame clipping boundaries are determined by
         adding the FRAM data to their previous
         values.
      This and the following four fields must be
      omitted if Change_Frame_clipping_boundaries=0.

4 bytes: Left_fb or Delta_Left_fb (signed integer).

4 bytes: Right_fb or Delta_Right_fb (signed integer).

4 bytes: Top_fb or Delta_Top_fb (signed integer).

4 bytes: Bottom_fb or Delta_Bottom_fb (signed
         integer).

4 bytes: Sync_id (unsigned integer) (omit if
         Change_Sync_id_list=0 or if the new list is
         empty; repeat until all Sync_id's have been
         listed).  The range is [0..2^31-1].
```

When the FRAM parameters are changed, the new parameters affect the frame that is about to be defined, not the one that is being terminated by the FRAM chunk.

Framing_modes:

Framing_mode=1.

When Framing_mode=1, each image that appears is a separate frame. FRAM chunks need not appear to separate them.

The following events trigger the display of a frame:

- Decoding a IHDR-IEND sequence at the MNG level, when it defines a potentially visible image.

- Decoding a JHDR-IEND sequence at the MNG level, when it defines a potentially visible image.
- Decoding a DHDR-IEND sequence, when it defines a potentially visible image.
- Decoding a BASI-IEND sequence, when it defines a potentially visible image.
- Decoding a PAST chunk, when it defines a potentially visible image.
- Decoding a SHOW chunk, when it directs that a potentially visible image be displayed. When the SHOW chunk requests that several images be displayed, each one in turn triggers a separate frame.

Any of these events triggers the display of a frame, even if the visible image is outside the clipping boundaries and no pixels are actually changed.

For example (assuming that objects 1 through 5 are all viewable objects), the sequence

```
FRAM 1
SHOW 1 5
```

will result in the display of five frames, each containing one of the images against the background according to its location and CLIP data.

Framing_mode=2.

When Framing_mode=2, viewers are expected to display all of the images at once, if possible, or as fast as can be managed, clearing the frame and restoring the background before displaying the first image but without clearing the display and restoring the background between images. Clearing the frame is also triggered by the appearance of the next FRAM chunk, which does cause the portion of the display inside the frame clipping boundaries to be cleared and restored, if nothing has been drawn. A frame boundary also occurs when a SEEK chunk or the MEND chunk appears.

For example, the sequence

```
FRAM 2
SHOW 1 5   (shows images 1, 2, 3, 4, and 5)
FRAM
```

will result in a single frame containing five images, each displayed according to its location and CLIP data.

When images in a frame overlap, viewers are expected to composite the later images against the partially completed frame that includes all earlier images.

This Framing_mode is fundamentally declarative; it describes the elements that go into an individual frame. It is up to the decoder to work out an efficient way of making the screen match the desired composition. Simple decoders can handle it as if it were procedural, compositing the images into the frame in the order that they appear, but efficient decoders might do something different, as long as the final appearance of the frame is the same.

Framing_mode=3.

When Framing_mode=3, a frame boundary occurs after each image appears, but only clearing the frame area and restoring the background between frames when the next FRAM chunk is encountered.

Frames are triggered by the same events that trigger a frame when Framing_mode=1. A frame is also triggered by the appearance of the next FRAM chunk, which does cause the portion of the display inside the frame clipping boundaries to be cleared and restored, if nothing else has triggered a frame. A frame boundary also occurs when a SEEK chunk or the MEND chunk appears.

When multiple images are caused to appear with a single SHOW chunk, a frame boundary occurs after each one. For example, the sequence

```
FRAM 3
SHOW 1 5   (shows images 1, 2, 3, 4, and 5)
FRAM
```

would result in five frames being displayed, the first with only image 1, the second with images 1 and 2, etc., and the fifth frame with all five images visible. The resulting output is exactly equivalent to what would have been accomplished (but perhaps less efficiently) with

```
FRAM 2
SHOW 1
FRAM
SHOW 1 2   (shows images 1 and 2)
FRAM
SHOW 1 3   (shows images 1, 2, and 3)
FRAM
SHOW 1 4   (shows images 1, 2, 3, and 4)
FRAM
SHOW 1 5   (shows images 1, 2, 3, 4, and 5)
FRAM
```

Framing_mode=4.

When Framing_mode=4, a frame boundary occurs after each image appears, without clearing the frame area and restoring the background between frames. Otherwise, Framing_mode=4 is identical to Framing_mode=1. Frames are triggered by the same events that trigger a frame when Framing_mode=1. A frame is also triggered by the appearance of the next FRAM chunk, but this does not cause any portion of the display to be cleared. Decoders should, however, clear the frame area when the very first FRAM chunk in the datastream is found, or the very first FRAM chunk after a SEEK chunk is found, even when Framing_mode=4.

Framing_mode=5.

When Framing_mode=5, a frame boundary occurs only when the next FRAM chunk appears, without clearing the frame area and restoring the background. Otherwise, Framing_mode=5 is identical to Framing_mode=2 A frame boundary also occurs when a SEEK chunk or the MEND chunk appears.

The Framing_mode also affects the way decoders handle the pHYs chunk (see Paragraph 4.5.5, below).

The Frame_name must conform to the same formatting rules as those for a SEEK keyword: 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 (unless the Frame_name is omitted) and no more than 79 characters

in the keyword. Keywords are case-sensitive. There is no null byte within the keyword. No specific use for the Frame_name is specified in this document, except that it can be included in the optional index that can appear in the SAVE chunk. Applications can use this field for such purposes as constructing an external list of frames in the datastream. The Frame_name only applies to the upcoming frame; subsequent frames are unnamed unless they also have their own Frame_name field.

The Interframe_delay value is the desired minimum time to elapse from the beginning of displaying one frame until the beginning of displaying the next frame.

The Sync_timeout field can be a number or <infinity>.

The termination condition given in the Change_Sync_timeout_and_termination field specifies how much longer, after the normal Interframe delay has elapsed, the frame will endure. It can take the following values:

deterministic

The frame endures no longer than the normal Interframe delay. Even though this is the default, a streaming encoder talking to a real-time decoder might write a FRAM with a termination condition of "deterministic" to force the display to be updated while the encoder decides its next move.

decoder-discretion

The decoder can lengthen the duration of the frame, but by no more than the timeout. A streaming decoder could take the opportunity to wait for its input buffer to fill to a comfortable level.

user-discretion

The decoder should wait for permission from the user (e.g., via a keypress) before proceeding, but must wait no longer than the timeout. If the decoder cannot interact with the user, this condition degenerates into "decoder-discretion".

external-signal

The decoder should wait for the arrival of a signal whose number matches a Sync_id, but must wait no longer than the timeout.

The Sync_id list can be omitted if the termination condition is not "external-signal".

When the Sync_id list is changed, the number of Sync_id entries is determined by the remaining length of the chunk data, divided by four. This number can be zero, which either inactivates the existing Sync_id list for one frame or deletes it.

The initial values of the FRAM parameters are

```
Framing_mode      = 1
Frame_name        = <empty string>
Interframe+delay  = 0
Left_fb           = 0
Right_fb          = Frame_width
Top_fb            = 0
Bottom_fb         = Frame_height
termination       = deterministic
Sync_timeout      = 0x7fffffff (infinite)
Sync_id           = <empty list>
```

The MOVE chunk can be used to specify the placement of each image within the frame. The CLIP chunk can be used to specify clipping boundaries for each image. The frame boundaries are only used for clipping. Even when the left and top frame boundaries are nonzero, the image locations are measured with respect to the {0,0} position in the display area. If the images are transparent or do not cover the entire area defined by the frame clippng boundaries, they are composited against the background defined by the BACK chunk, or against an application-defined background, if the BACK chunk is not present or is not recognized by the decoder. The background, as well as the images, is clipped to the frame clipping boundaries. Any pixels outside the frame clipping boundaries remain unchanged.

The Frame_duration field gives the duration of display, which is the minimum time that must elapse from the beginning of displaying one frame until the beginning of displaying the next (or between images, when Framing_mode=1). It is measured in "ticks" using the tick length determined from ticks_per_second defined in the MHDR chunk.

A viewer does not actually have to follow the procedure of erasing the screen, redisplaying the background, and recompositing the images against it, but what is displayed when the frame is complete must be the same as if it had. It is sufficient to redraw the parts of the display that change from one frame to the next.

The Sync_id list provides a point at which the processor must wait for all pending processes to reach the synchronization point having the same Sync_id before resuming, perhaps because of a need to synchronize a sound datastream (not defined in this specification) with the display, to synchronize stereo images, and the like. When the period defined by the sum of the Frame_duration and the Sync_timeout fields elapses, processing can resume even though the processor has not received an indication that other processes have reached the synchronization point.

Note that the synchronization point does not occur immediately, but at the end of the frame that follows the FRAM chunk. If it is necessary to establish a synchronization point immediately, this can be done by using two consecutive FRAM chunks, the first setting a temporary Frame_duration=0, Sync_timeout, and Sync_id, and the second establishing the synchronization point:

```
FRAM 4 0 1 1 0 1 0000 Sync_timeout Sync_id
FRAM 0 name
```

The identifier Sync_id=0 is reserved to represent synchronization with a user input from a keyboard or pointing device. The Sync_id values 1-255 are reserved to represent the corresponding ASCII letter, received from the keyboard (or a simulated keyboard), and values 256-1023 are reserved for future definition

by this specification. If multiple channels (not defined in this specification) are not present, viewers can ignore other values appearing in the Sync_id list.

4.3.3 MOVE New image location

New location of an existing object or objects (replacing or incrementing the location given in the DEFI chunk).

The MOVE chunk gives the position, measured downward and to the right of the frame origin, in pixels, where the named object or group of objects is to be located.

The chunk's contents are:

```
2 bytes: First_object (nonzero unsigned integer).

2 bytes: Last_object (nonzero unsigned integer).

1  byte: Loca_delta_type (unsigned integer).
      0: MOVE data gives X_location and
         Y_location directly.
      1: New locations are determined by adding
         the MOVE data to the location of the
         parent object.

4 bytes: X_location or Delta_X_location
         (signed integer).

4 bytes: Y_location or Delta_Y_location
         (signed integer).
```

The new location applies to a single object, if First_object=Last_object, or to a group of consecutive Object_ids, if they are different. Negative values are permitted for the X and Y location. The positive directions are downward and rightward from the frame origin. The MOVE chunk can specify an image placement that is partially or wholly outside the display boundaries. In such cases, the resulting image must be clipped to fit within its clipping boundaries, or not displayed at all if it falls entirely outside its clipping boundaries. The clipping boundaries are determined as described in the specification for the CLIP chunk Paragraph 4.3.4, below.

It is not an error for the MOVE chunk to name an object that has not previously been defined. In such cases, nothing is done to the nonexistent object.

When an Object_id is discarded, its object attribute set, which includes the MOVE data, is also discarded.

4.3.4 CLIP Object clipping boundaries

This chunk gives the new boundaries (replacing or incrementing those from the DEFI chunk) to which an existing object or group of objects must be clipped for display. It contains the following 21 bytes:

```
2 bytes: First_object (nonzero unsigned integer).

2 bytes: Last_object (nonzero unsigned integer).

1  byte: Clip_delta_type (unsigned integer).
      0: CLIP data gives boundary values directly.
      1: CLIP boundaries are determined by adding
         the CLIP data to their previous values
         for this Object_id.

4 bytes: Left_cb or Delta_Left_cb (signed integer).

4 bytes: Right_cb or Delta_Right_cb (signed integer).

4 bytes: Top_cb or Delta_Top_cb (signed integer).

4 bytes: Bottom_cb or Delta_Bottom_cb (signed
         integer).
```

The new clipping boundaries apply to a single object, if `First_object=Last_object`, or to a group of consecutive objects, if they are different.

The clipping boundaries are expressed in pixels, measured rightward and downward from the frame origin.

The left and top clipping boundaries are inclusive and the right and bottom clipping boundaries are exclusive, i.e., the pixel located at {x,y} is only displayed if the pixel falls within the physical limits of the display hardware and all of the following are true:

```
0       <= x < Frame_width       (from the MHDR chunk)
0       <= y < Frame_height
Left_fb <= x < Right_fb          (from the FRAM chunk)
Top_fb  <= y < Bottom_fb
Left_cb <= x < Right_cb          (from the CLIP chunk)
Top_cb  <= y < Bottom_cb
```

It is not an error for the `CLIP` chunk to name an `object` that has not previously been defined. In such cases, nothing is done to the nonexistent object.

When an `Object_id` is discarded, its object attribute set, which includes the `CLIP` data, is also discarded.

4.3.5 `SHOW` Show images

The `SHOW` chunk is used to change the potential visibility of one or more previously-defined objects and to direct that they be displayed. It contains 2, 4, or 5 bytes, or it can be empty.

```
2 bytes: First_image (nonzero unsigned integer).
```

```
2 bytes: Last_image  (nonzero unsigned integer).
     This field can be omitted if the Show_mode
     byte is also omitted.  If so, decoders must
     assume the default values, "Show_mode=0"
     and "Last_image=First_image."

1 byte:  Show_mode (unsigned integer)
     0:  Make the images potentially visible
         and display them (set "Do_not_show=0").
     1:  Make the images invisible (set
         "Do_not_show=1").
     2:  Don't change "Do_not_show" flag;
         display those that are potentially
         visible.
     3:  Mark images "potentially visible"
         ("Do_not_show=0") but do not display
         them.
     4:  Toggle "Do_not_show" flag; display
         any that are potentially visible after
         toggling.
     5:  Toggle "Do_not_show" flag but do
         not display even if potentially visible
         after toggling.
     6:  Step through the images in the given
         range, making the next image potentially
         visible (set "Do_not_show=0) and display
         it.  Set "Do_not_show=1" for all other
         images in the range.  Cycle back to the
         beginning of the range when reaching the
         end of the range.  Perform one step for
         each SHOW chunk (in reverse order if
         Last_image < First_image).
     7.  Make the next image in the range (cycle)
         potentially visible ("Do_not_show-0")
         but do not display it.  Set
         "Do_not_show=1" for the rest of the
         images in the range.

     This field can be omitted.  If so, decoders
     must assume the default, "Show_mode=0".
```

The decoder processes the objects (or images) named in the SHOW chunk in the order First_image through Last_image, and resets the Do_not_show flag for each of the objects. If Show_mode is even-valued, it also displays the images if they are potentially visible and are viewable images.

When the SHOW chunk is empty, the decoder displays all existing potentially visible images, without changing their Do_not_show status. The empty SHOW chunk is equivalent to SHOW 1 65535 2.

If First_image > Last_image then the images are processed in reverse order.

When Show_mode is odd-valued, nothing is displayed unless a subsequent SHOW chunk with an even-valued Show_mode appears.

Interactions with Framing_mode

When Framing_mode != 2, or when the FRAM chunk is not present, each visible image that is displayed triggers a separate frame. When Show_mode is odd, or when Show_mode=2 or 4 or is empty and no image is visible, no frame is generated.

When Show_mode=1, 4, 5, 6, or 7, images can be made invisible. This is not permitted when Framing_mode != 1 in the FRAM chunk and the images have already appeared in the frame, because simple viewers will have already drawn them and have no way to make them invisible again without redrawing the entire frame.

When Show_mode=6 or 7, the decoder must make the next image in the "cycle" visible. To do this, it must examine the Do_not_show flag for each image in the range First_image through Last_image, and make the next one (the one with the next higher value of Image_id that exists and is "viewable") after the first visible one it finds visible and the rest invisible. If the first visible one was Last_id, or none were visible, it must make First_image visible. When First_image > Last_image, the cycle is reversed, and the "next" image is the one with the next lower value of Image_id. These modes are useful for manipulating a group of sequential images that represent different views of an animated icon. See Example 8 Chapter 16, below.

When Show_mode=0, 2, 4, or 6, an instance of each visible image will be displayed at the location specified by the DEFI, CLON, or MOVE chunk and clipped according to the boundaries specified by the CLIP and FRAM chunks. When the MOVE or CLON chunk is used in the delta form, which will frequently be the case, each image must be displaced from its previous position by the values given in the MOVE or CLON chunk.

Any of the following sequences would cause the image identified by Object_id=6 in a composite frame to blink:

```
LOOP 0 0 10
FRAM 2
SHOW 1 10     # Show images 1 thru 10
FRAM
SHOW 1 5      # Show images 1 thru 5
SHOW 7 10     # Show images 7 thru 10
ENDL

LOOP 0 0 10
FRAM 2
SHOW 1 5      # Show images 1 thru 5
SHOW 6 6 4    # Toggle potential visibility of image 6
SHOW 7 10     # and show it; show images 7 thru 10
FRAM
ENDL
```

```
LOOP 0 0 10
FRAM 2
SHOW 6 6 5   # Toggle potential visibility of image 6
SHOW 1 10 2  # Show potentially visible images in 1
FRAM         # through 10
ENDL
```

It is not necessary to follow an IHDR-IEND, JHDR-IEND, BASI-IEND, or DHDR-IEND sequence or PAST chunk with a SHOW chunk to display the resulting image, if it was already caused to appear by Do_not_show=0 in the DEFI chunk that introduced the image. Similarly, the CLON chunk need not be followed by a SHOW chunk, if Do_not_show=0 in the CLON chunk.

It is not an error for the SHOW chunk to name an object that has not previously been defined. In such cases, nothing is done to the nonexistent object.

If a nonviewable object is named, its Do_not_show flag is changed but it is not displayed even if the SHOW chunk requests it.

4.4 SAVE and SEEK chunks

The SAVE chunk marks a point in the datastream at which objects are "frozen" and other chunk information is "saved". The SEEK chunk marks positions in the MNG datastream where a restart is possible, and where the decoder must restore the "saved" information, if they have jumped or skipped to a SEEK point. They only need to restore information that they will use, e.g., a viewer that processes gAMA and global PLTE but ignores iCCP and sPLT need only restore the value of gamma and the global PLTE data from the prologue segment and not the values of the iCCP and sPLT data.

Simple decoders that only read MNG datastreams sequentially can safely ignore the SAVE and SEEK chunks, although it is recommended that, for efficient use of memory, they at least mark existing objects as "frozen" when the SAVE chunk is processed and discard all "nonfrozen" objects whenever the SEEK or empty DISC chunk is processed.

4.4.1 SAVE Save information

The SAVE chunk marks a point in the datastream at which objects are "frozen" and other chunk information is "saved"; a decoder skipping or jumping to a SEEK chunk must restore the "saved" chunk information if it has been redefined or discarded. In addition, the SAVE chunk can contain an optional index to the MNG datastream.

The SAVE chunk can be empty, or it can contain an index consisting of the following:

```
1 byte: Offset_size (unsigned integer)
     4: offsets and nominal start times are
        expressed as 32-bit integers
     8: offsets and nominal start times are
        expressed as 64-bit integers
```

plus zero or more of the following index entries:

```
1 byte: Entry_type (unsigned integer)
     0: segment with nominal start time and
        nominal frame number
     1: segment
     2: frame
     3: exported image
4 or 8 bytes: Offset (unsigned integer) Omit if
        entry_type > 1, set equal to zero
        if the offset is unknown.
4 or 8 bytes: Nominal_start_time (unsigned
        integer) start time of the segment,
        measured in ticks from the beginning of
        the animation, assuming that all prior
        segments were played as intended on
        an ideal player, ignoring any "fPRI"
        chunks.  Omit if entry_type > 0.
4 bytes: Nominal_Frame_number (unsigned
        integer) sequence number of the first
        frame in the segment, assuming that all
        prior segments were played as intended
        on an ideal player, ignoring any "fPRI"
        chunks; the first frame of the first
        segment being frame 0. Omit if
        entry_type > 0.
n bytes: Name (Latin-1 text) Omit for unnamed
        segments.
1 byte: zero-byte separator (must be omitted from
        the last entry)
```

The SAVE chunk must be present when the SEEK chunk is present. It appears after the set of chunks that define information that must be retained for the remainder of the datastream. These chunks, collectively referred to as the prologue segment, are no different from chunks in other segments. They can be chunks that define objects, or they can be chunks that define other information such as gAMA, cHRM, and sPLT. If any chunks appear between the SAVE chunk and the first SEEK chunk, these chunks also form a part of the prologue segment, but their contents are lost when the SEEK chunk appears.

Only one instance of the SAVE chunk is permitted in a MNG datastream. It is not allowed anywhere after the first SEEK chunk.

It is not permitted, at any point beyond the SAVE chunk, to modify or discard any object that was defined ahead of the SAVE chunk.

An object appearing ahead of the SAVE chunk can be the subject of a CLON chunk. If the clone is a partial clone, modifying the information in its object buffer is not permitted, because this would also modify the object buffer that the original object points to. Its object attributes can be modified, however, because the clone has its own set of them.

A chunk like gAMA that overwrites a single current value is permitted after the SAVE chunk, even if the chunk has appeared ahead of the SAVE chunk. Decoders that can jump or skip to SEEK chunks are responsible for saving a copy of the chunk data (in any convenient form) when the SAVE chunk is encountered and restoring it when skipping or jumping to a SEEK chunk. If no instance of the chunk appeared ahead of the SAVE chunk, such a decoder must restore the chunk data to its original "unknown" condition when it skips or jumps to a SEEK chunk.

It is the *encoder's* responsibility, if it changes or discards any "saved" data, to restore it to its "saved" condition (or nullifying it, if it was unknown) prior to the end of the segment. This makes it safe for simple decoders to ignore the SAVE/SEEK mechanism.

Known chunks in this category include DEFI, FRAM, BACK, PLTE, cHRM, eXPI, fPRI, gAMA, iCCP, pHYs, sRGB, and tERm.

In the case of chunks like sPLT that can occur multiple times, with different "purpose" fields, additional instances of the chunk are permitted after the SAVE chunk, but not with the same keyword as any instances that occurred ahead of the SAVE chunk. The decoder is required to forget such additional instances when it skips or jumps to a SEEK chunk, but it must retain those instances that were defined prior to the SAVE chunk. Encoders are required to nullify such additional instances prior to the end of the segment. Known chunks in this category include only sPLT.

If the optional index is present, every segment, whether named or not, except for the prologue segment, must be listed in it. All entries must appear in the index in the same order that they appear in the MNG datastream. Only named images or frames are permitted, and it is not an error to omit any or all named images or frames from the index.

Offsets are calculated from the first byte of the MNG 8-byte signature, which has offset=0. This is true even if the MNG datastream happens to be embedded in some other file and the signature bytes are not actually present.

Applications with direct access to the datastream can use the index to find segments, frames, and exported images quickly. After processing the prologue segment, they can jump directly to any segment and then process the remaining datastream until the desired frame, image, or time is found. Applications that have only streaming access to the datastream can still use the index to decide whether to decode the chunks in a segment or to skip over them.

Only one instance of the SAVE chunk is permitted in a MNG datastream, and, if the SEEK chunk is present, the SAVE chunk must be present, prior to the first SEEK chunk. The only chunks not allowed ahead of the SAVE chunk are the SEEK chunk and the MEND chunk.

4.4.2 SEEK Seek point

The SEEK chunk marks positions in the MNG datastream where a restart is possible, and where the decoder must restore certain information to the condition that existed when the SAVE chunk was processed.

The SEEK can be empty, or it can contain a keyword.

```
n bytes:  keyword (Latin-1 string).
```

The keyword is optional. The keyword must follow the format of a tEXt keyword: 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 keyword. There is no null byte terminator within the keyword, nor is there a separate null byte terminator. Keywords are case-sensitive. Use caution when printing or displaying keywords (Refer to Security considerations, Chapter 15, below). No specific use for the keyword is specified in this document, but applications can use the keyword for such purposes as constructing a menu of SEEK points for a slide-show viewer. It can be included in the optional index that can appear in the SAVE chunk.

Applications must not use any information preceding the SEEK chunk, except for

- data appearing in the MHDR chunk
- anything appearing ahead of the SAVE chunk.

When the SEEK chunk is encountered, the decoder can discard any objects appearing after the SAVE chunk, as though an empty DISC chunk were present.

In addition to providing a mechanism for skipping frames or backspacing over frames, the SEEK chunk provides a means of dealing with a corrupted datastream. The viewer would abandon processing and simply look for the next SEEK chunk before resuming. Note that looking for a PNG IHDR chunk would not be sufficient because the PNG datastream might be inside a loop or a Delta-PNG datastream, or it might need data from preceding MOVE or CLIP chunks.

When the SEEK chunk is encountered, a decoder must restore the information that it saved when it processed the SAVE chunk.

Multiple instances of the SEEK chunk are permitted. The SEEK chunk must not appear prior to the SAVE chunk. The SAVE chunk must also be present if the SEEK chunk is present.

4.5 Ancillary MNG chunks

This section describes ancillary MNG chunks. MNG-compliant decoders are not required to recognize and process them.

4.5.1 eXPI Export image

The eXPI chunk takes a snapshot of an object (either concrete or abstract), associates the name with that snapshot, and makes the name available to the "outside world" (like a scripting language).

The chunk contains an object identifier and a name:

```
2 bytes: Snapshot_id (unsigned nonzero integer)
n bytes: Snapshot_name (Latin-1 text)
```

Note that the Snapshot_name is associated with the snapshot, not with the Snapshot_id nor its future contents; discarding the image identified by Snapshot_id will not affect the snapshot. The Snapshot_name means nothing inside the scope of the MNG specification, except that it can be included in the optional index that can appear in the SAVE chunk.. If two eXPI chunks use the same name, it is the outside world's problem (and the outside world's prerogative to regard it as an error). A decoder that knows of no "outside world" can simply ignore the eXPI chunk. This chunk could be used in MNG datastreams that define libraries of related images, rather than animations.

Names beginning with the word "thumbnail" are reserved for snapshot images that are intended to make good icons for the MNG. Thumbnail images are regular PNG or Delta-PNG images, but they would normally have smaller dimensions and fewer colors than the MNG frames. They can be defined with the potential visibility field set to "invisible" if they are not intended to be shown as a part of the regular display.

The Snapshot_name string must follow the format of a tEXt keyword: 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 keyword. Keywords are case-sensitive. There is no null byte terminator within the Snapshot_name string, nor is there a separate null byte terminator.

Multiple instances of the eXPI chunk are permitted in a MNG datastream.

4.5.2 fPRI Frame priority

The fPRI chunk allows authors to assign a priority to a portion of the MNG datastream. Decoders can decide whether or not to decode and process that part of the datastream based on its "priority" compared to some measure of "cost."

The fPRI chunk contains two bytes:

```
1 byte: fPRI_delta_type (unsigned integer).
     0: Priority is given directly.
     1: Priority is determined by adding
        the fPRI data to the previous
        value, modulo 256.

1 byte: Priority or Delta_priority (signed
        integer). Value to be assigned to
        subsequent frames until another fPRI
        chunk is reached.
```

While 256 distinct values of Priority are possible, it is recommended that only the values 0 (low priority), 128 (medium priority), and 255 (high priority) be used. Viewers that can only display a single image can look for one with Priority=255 and stop after displaying it. If the datastream contains a large number of frames and includes periodic "initial" frames that do not contain Delta-PNG datastreams, each "initial" frame could be preceded by a fPRI with Priority=128 and followed by one with Priority=0, and the best representative initial frame could be preceded by a fPRI chunk with Priority=255. Then single-image

viewers would just display the representative frame, slow viewers would display just the "initial" frames, and fast viewers would display everything.

If a viewer has established a nonzero "cost," it must skip any portion of the datastream whose priority is less than that "cost." The "cost" must be established prior to processing the proloque segment, and it cannot be changed unless the prologue segment is processed again according to the new "cost."

The SAVE, SEEK, and MEND chunks always have Priority=255; decoders must look for these chunks in addition to the fPRI chunk while skipping a low-priority portion of the datastream.

It is not permissible for a portion of the datastream to depend on any portion of the datastream having a lower value, because a decoder might have skipped the lower value portion. Use of the fPRI chunk is illustrated in Examples 5 and 9.

Viewers that care about the priority must assume Priority=255 for any portion of the MNG datastream that is processed prior to the first fPRI chunk.

Multiple instances of the fPRI chunk are permitted.

4.5.3 nEED Resources needed

The nEED chunk can be used to specify needed resources, to provide a quick exit path for viewers that are not capable of displaying the MNG datastream.

The nEED chunk contains a list of keywords that the decoder must recognize. Keywords are typically private critical chunk names.

```
n bytes: Keyword
1  byte: null separator
etc.
```

The nEED chunk should be placed early in the MNG datastream, preferably immediately after the MHDR chunk.

The keywords are typically 4-character private critical chunk names, but they could be any string that a decoder is required to recognize. No critical chunks defined in this specification or in the PNG specification should be named in a nEED chunk, because MNG-compliant decoders are required to recognize all of them, whether they appear in a nEED chunk or not. The purpose of the nEED chunk is only to identify requirements that are above and beyond the requirements of this document and of the PNG specification.

Each keyword string must follow the format of a tEXt keyword: 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 keyword. Keywords are case-sensitive. There is no null byte terminator within the keyword. A null separator byte must appear after each keyword in the nEED chunk except for the last one.

Decoders that do not recognize a chunk name or keyword in the list should abandon the MNG datastream or request user intervention. The normal security precautions should be taken when displaying the keywords.

-[During the draft phase of the development of this specification, to specify that the datastream is written in accordance with a certain draft version of MNG, you can use "nEEDdraft nn".]

4.5.4 `tERm` Termination

The `tERm` chunk suggests how the end of the MNG datastream should be handled, when a `MEND` chunk is found. It contains either a single byte or ten bytes:

```
1 byte: Termination_action (unsigned integer)
     0: Show the last frame indefinitely.
     1: Cease displaying anything.
     2: Discard all "nonfrozen" objects and repeat
        the animation starting immediately after
        the SAVE chunk (or immediately after the
        MHDR chunk, if a SAVE chunk is not present).
     3: Repeat the animation starting immediately
        after the last SEEK chunk in the datastream.
        If no SEEK chunk is present, repeat the
        animation starting immediately after the
        SAVE chunk (or immediately after the MHDR
        chunk, if a SAVE chunk is not present).
     4: Reset the decoder to its initial state,
        retaining only the MHDR data, and repeat
        the entire animation starting immediately
        after the MHDR chunk.
1 byte: Action_after_iterations
     0: Show the last frame indefinitely after
        Iteration_max iterations have been done.
     1: Cease displaying anything.
        This and the remaining fields must be
        omitted if Termination_action<=1, and
        must be present otherwise.
4 bytes: Delay (unsigned integer) delay, in ticks,
        before repeating the animation.
4 bytes: Iteration_max (unsigned integer) maximum
        number of times to repeat the animation.
```

The loop created by processing a `tERm` is always be treated by the decoder as if it were a <user-discretion> loop, with `Iteration_min=1`.

4.5.5 `pHYs` Physical pixel size

The MNG `pHYs` chunk is identical in syntax to the PNG `pHYs` chunk, but, when `Framing_mode != 1` and `Framing_mode != 4` from the `FRAM` chunk, it applies to complete MNG frames and not to the individual images within them. When `Framing_mode=1` or `Framing_mode=4`, the MNG `pHYs` chunk provides a default value if the subsequent PNG datastream does not contain one.

The MNG top-level pHYs chunk can be nullified by a subsequent empty pHYs chunk appearing in the MNG top level.

4.5.6 PNG ancillary chunks

The namespace for MNG chunk names is separate from that of PNG. Only those PNG chunks named in this paragraph are also defined at the MNG top level. They have exactly the same syntax and semantics as when they appear in a PNG datastream:

- tEXt
- tIME Same format as in PNG. Can appear at most once in the prologue segment (before the first SEEK chunk), and at most once per segment (between two consecutive SEEK chunks). In the prologue it indicates the last time any part of the MNG was modified. In a regular segment (between SEEK chunks), it indicates the last time that segment was modified.
- zTXt

A MNG editor that writes PNG datastreams should not include the top-level tEXt, tIME, and zTXt chunks in the generated PNG datastreams.

The following PNG chunks are also defined at the MNG top level. They provide default values to be used in case they are not provided in subsequent PNG datastreams. Any of these chunks can be nullified by the appearance of a subsequent empty chunk with the same chunk name. Such empty chunks are not legal PNG or JNG chunks and must only appear in the MNG top level.

- cHRM
- gAMA
- iCCP
- sRGB

A MNG editor that writes PNG or JNG datastreams is expected to include the top-level cHRM, gAMA, iCCP, and sRGB chunks in the generated PNG or JNG datastreams, if the embedded image does not contain its own chunks that define the color space.

The following PNG chunks are also defined at the MNG top level. They provide values that take precedence over those that might be provided in subsequent PNG or JNG datastreams and provide values to be used when they are not provided in subsequent PNG or JNG datastreams:

- sPLT (this chunk also takes precedence over the PLTE chunk in a subsequent PNG datastream when the PLTE and hIST chunks are being used as a suggested palette (i.e., Color_type != 3). This chunk can appear for any color type. There can be multiple sPLT chunks in a MNG datastream. If a palette_name is repeated, the previous palette having the same palette_name is replaced. It is not permitted, at the MNG top level, to redefine a palette after the SAVE chunk with the same palette_name as one that appears ahead of the SAVE chunk. It is permitted, however, to define

and redefine other palettes with other palette_name fields. A single empty sPLT chunk can be used to nullify all sPLT chunks that have been previously defined in the MNG top level, except for those that appeared ahead of the SAVE chunk, when the SAVE chunk has been read.

When a decoder needs to chose between a suggested palette defined at the MNG level and a suggested palette defined in the PNG datastream (either with the sPLT chunk, or with the PLTE/hIST chunks for grayscale or truecolor images), it should give preference to the palette from the MNG level, to avoid spurious frame-to-frame color changes.

MNG editors that write PNG datastreams should ignore the sPLT and pHYs data from the MNG level and simply copy any sPLT and pHYs chunks appearing within the PNG datastreams.

5 The JPEG Network Graphics (JNG) format

JNG (JPEG Network Graphics) is the lossy sub-format for MNG objects.

Note: This specification depends on the PNG Portable Network Graphics specification [PNG]. The PNG specification is available at the PNG home page,

<URL:http://www.cdrom.com/pub/png/>

A JNG datastream consists of a header chunk (JHDR), JDAT chunks that contain a complete JPEG datastream, and optionally, IDAT chunks that contain a PNG-encoded grayscale image that is to be used as an alpha mask. Such a mask must have the same dimensions as the image itself. The JDAT and IDAT chunks can be interleaved. Some of the PNG ancillary chunks are also recognized in PNJ datastreams.

While JNG is primarily intended for use as a sub-format within MNG, a single-image JNG datastream can be written in a standalone file. If so, the first eight bytes of a JNG datastream are

139 80 78 74 13 10 26 10

(decimal) which is similar to the PNG signature with "\213 J N G" instead of "\211 P N G" in bytes 0-3.

JNG is pronounced "Jing."

5.1 JNG critical chunks

This section specifies the critical chunks that are defined in the JNG format.

5.1.1 JHDR JNG header

The format of the JHDR chunk has not yet been fully defined. Tentatively, it will contain

```
4 bytes:  Width (unsigned integer, range 0..65535)
4 bytes:  Height (unsigned integer, range 0..65535)
1 byte:   Color_type
           8: gray (Y)
          10: color (YCbCr)
          12: gray-alpha (Y-alpha)
          14: color-alpha (YCbCr-alpha)
1 byte:   JDAT_Sample_depth
           8: 8 bits
1 byte:   JDAT_compression_method
           8: ISO-10918-1 Huffman-coded baseline JPEG
1 byte:   JDAT_interlace_method
           0: sequential JPEG, single scan
           [8: progressive JPEG]
1 byte:   IDAT_sample_depth
           0, 1, 2, 4, 8, or 16
1 byte:   IDAT_compression_method
           0: zlib DEFLATE
1 byte:   IDAT_filter_method
           0: adaptive (see PNG spec)
1 byte:   IDAT_interlace_method
           0: not interlaced
```

The Width, Height, JDAT_sample_depth, JDAT_compression_method, and JDAT_interlace_method fields are redundant because equivalent information is also embedded in the `JDAT` datastream. They appear in the `JHDR` chunk for convenience. Their values must be identical to their equivalents embedded in the `JDAT` chunk. We use four bytes in the Width and Height fields for similarity to MNG and PNG, and to leave room for future expansion, even though two bytes would have been sufficient.

When the `Color_type` is 8 or 10 (no alpha channel), the last four bytes, which describe the `IDAT` data, must be set to zero. The `IDAT_Sample_depth` must be nonzero when the alpha channel is present.

5.1.2 `JDAT` JNG image data

A JNG datastream must contain one or more `JDAT` chunks, whose data, when concatenated, forms a complete JNG baseline JPEG datastream.

A JNG baseline JPEG is a baseline JPEG as defined by JPEG Part 1 (ISO IS 10918-1), using only JFIF-compatible component interpretations, with a few additional restrictions that reflect limitations of many existing JPEG implementations.

- Baseline JPEG restrictions

  A baseline JPEG according to Part 1 is DCT-based (lossy) sequential JPEG, using Huffman entropy encoding, with the following further restrictions:

  - Only 8-bit sample precision

  - Only 8-bit quantization table precision
  - Huffman code tables can have table numbers 0 and 1 only

  The SOF marker type for baseline JPEG is SOF0.

  JDAT datastreams must always follow "interchange JPEG" rules: all necessary quantization and Huffman tables must be included in the datastream, no tables can be omitted.

- JFIF-compatible restrictions

  The image data is always stored left-to-right, top-to-bottom, ie, only the default SPIFF orientation is permissible.

  The encoded data shall have one of the two colorspace interpretations allowed by the JFIF specification:

  - Grayscale: a single component representing luminance, ranging from 0 for black to 255 for white [(or 0 to 4095 when dealing with 12-bit data)]. This component shall have JPEG component identifier 1.
  - YCbCr: three components representing luminance, chroma blue, and chroma red, in that order. The components shall be assigned JPEG component identifiers 1, 2, 3 respectively. YCbCr is defined as a linear transformation from RGB color space:

    ```
    Y = Luma_red * R + Luma_green * G + Luma_blue * B
    Cb = (B - Y) / (2 - 2*Luma_blue)  + Half_scale
    Cr = (R - Y) / (2 - 2*Luma_red)   + Half_scale
    ```

    By convention, the luminance coefficients are always those defined by CCIR Recommendation 601-1:

    ```
    Luma_red   = 0.299
    Luma_green = 0.587
    Luma_blue  = 0.114
    ```

    The constant Half_scale is 128 [when dealing with 8-bit data, 2048 for 12-bit data]. With these equations, Y, Cb, and Cr all have the same range as R, G, and B: 0 to 255 [for 8-bit data, 0 to 4095 for 12-bit data].
    The JFIF convention for YCbCr differs from typical digital television practice in that no headroom/footroom is reserved: the coefficient values range over the full available 8 [or 12] bits.

    Intercomponent sample alignment shall be such that the first (upper leftmost) samples of each component share a common upper left corner position. This again differs from common digital TV practice, in which the first samples share a common center position. The JFIF convention is simpler to visualize: subsampled chroma samples always cover an integral number of luminance sample positions, whereas with co-centered alignment, chroma samples only partially overlap some luminance samples.

- Additional JNG restrictions

  JNG imposes three additional restrictions not found in the text of either JPEG Part 1 or the JFIF specification:

  - The sampling factors for YCbCr images must be one of these sets:
    * 1h1v,1h1v,1h1v (also called 4:4:4 or 1x1 sampling)
    * 2h1v,1h1v,1h1v (also called 4:2:2 or 2x1 sampling)
    * 2h2v,1h1v,1h1v (also called 4:2:0 or 2x2 sampling)
    * *-[1x2 sampling is starting to be seen around the net now, thanks to lossless rotation of 2x1 datastreams. Should this be called out as a required or at least highly recommended option?]*

    In other words, the chroma components may be downsampled 2:1 horizontally relative to luminance, and optionally also 2:1 vertically; or they may be left full size. These three sampling ratios are the only ones supported by a wide spectrum of implementations.

    For grayscale images, the sampling factors are irrelevant according to a strict reading of JPEG Part 1. Hence decoder authors should accept any sampling factors for grayscale. However, we recommend that encoders always emit sampling factors 1h1v for grayscale, since some decoders have been observed to malfunction when presented with other sampling factors.
  - There must be only one scan in an image: that is, YCbCr images must be fully interleaved. There is little advantage to be gained by encoding a baseline image in multiple scans, and many baseline decoders do not support multiple scans at all.
  - The DNL (Define Number of Lines) marker is prohibited. The image Height must always be specified accurately in the SOFn marker and in the `JHDR` chunk.

- Recommended progressive JPEG subset [if progressive JPEG is allowed]

  For JNG progressive JPEG datastreams, the JPEG process is progressive Huffman coding (SOF marker type SOF2) rather than baseline (SOF0). All JNG-compliant decoders must support full progression, including both spectral-selection and successive-approximation modes, with any sequence of scan progression parameters allowed by the JPEG Part 1 standard.

  Otherwise, all the restrictions listed above apply, except these:

  - Multiple-scan support is obviously required for progressive JPEG.
  - Huffman table numbers up to 3 (the full-JPEG limit) may be used, since the baseline two-table limit is unlikely to be needed by any decoder that can handle progressive JPEG.

  We require full progression support since relatively little code savings can be achieved by subsetting the JPEG progression features. In particular, successive approximation offers significant gains in the visual quality of early scans. Omitting successive-approximation support from a decoder does not save nearly enough code to justify restricting JNG progressive encoders to spectral selection only.

  No particular progressive scan sequence is specified or recommended by this specification. Not enough experience has been gained with progressive JPEG to warrant making such a recommendation. To allow for future experimentation with scan sequences, decoders are expected to handle any JPEG-legal sequence. Again, the code savings that might be had by making restrictive assumptions are too small to justify a limitation.

`JDAT` chunks are like PNG `IDAT` chunks in that there may be multiple `JDAT` chunks, the data from which are concatenated to form a single datastream that can be sent to the decompressor. No chunks are permitted

among the sequence of JDAT chunks, except for interleaved IDAT chunks. The ordering requirements of other ancillary chunks are the same with respect to JDAT as they are in PNG with respect to the IDAT chunk.

5.1.3 IDAT JNG alpha data

This chunk is exactly like the IDAT chunk occurring in a PNG grayscale image, except that it is interpreted as an alpha mask to be applied to the image data from the JDAT chunks. The alpha channel, if present, can have sample depths 1, 2, 4, 8, or 16?. The IDAT chunks can be interleaved with the JDAT chunks. No other chunk type can appear among the sequence of IDAT and JDAT chunks. No other chunk type can between the sequences of IDAT and JDAT chunks when they are not interleaved. The samples in the IDAT must be presented in noninterlaced order, left to right, top to bottom. As in PNG, zero means fully transparent and 2^IDAT_Sample_depth-1 means fully opaque.

5.1.4 IEND End of JNG datastream

The JNG IEND chunk is identical to its counterpart in PNG. Its data length is zero, and it serves to mark the end of the JNG datastream.

5.2 JNG ancillary chunks

Some PNG ancillary chunks can also appear in JNG datastreams, and are used for the same purposes as described in the PNG specification:

If the bKGD chunk is present, it must be written as if it were written for a PNG datastream with Sample_depth=8. It has one 2-byte entry for grayscale JNGs and three 2-byte entries for color JNGs. The first (most significant) byte of each entry must be 0.

The following chunks have exactly the same meaning and have the same format as given in the PNG specification: cHRM, gAMA, iCCP, sRGB, pHYs, oFFs, tEXt, tIME, and zTXt.

The PNG PLTE, hIST, pCAL, sBIT, and tRNS chunks are not defined in JNG [Would JNG have any use for the sPLT chunk?].

When cHRM, gAMA, iCCP, or sRGB are present, they provide information about the colorspace of the decoded JDAT image, and they have no effect on the decoded alpha samples from the IDAT chunks.

The chunk copying and ordering rules for JNG are the same as those in PNG, except for the fact that JDAT and IDAT chunks can be interleaved.

6 The Delta-PNG format

A Delta-PNG datastream describes a single image, by giving the changes from a previous PNG (Portable Network Graphics), a JNG (JPEG Network Graphics), or another Delta-PNG image.

No provision is made in this specification for storing a Delta-PNG datastream as a standalone file. A Delta-PNG datastream will normally be found as a component of a MNG datastream. Applications that need to store a Delta-PNG datastream separately should use a different file signature and filename extension, or they can wrap it in a MNG datastream consisting of the MNG signature, the MHDR chunk, a BASI chunk with the appropriate dimensions and an IEND chunk, the Delta-PNG datastream, and the MEND chunk.

The decoder must have available a parent (decoded) object from which the original chunk data is known. The parent object can be the result of decoding a PNG, another Delta-PNG datastream, or it could have been generated by a PNG-like datastream introduced by a BASI chunk.

The child image is always of the same basic type (at present only PNG is defined) as the parent object. The child is always a viewable image even if the parent is not.

The decoder must not have modified the pixel data in the parent object by applying output transformations such as gAMA or cHRM, or by compositing the image against a background. Instead, the decoder must make available to the Delta-PNG decoder the unmodified pixel data along with the values for the gAMA, cHRM, and any other recognized chunks from the parent object datastream.

A Delta-PNG datastream consists of a DHDR and IEND enclosing other optional chunks (if there are no other chunks, the decoder simply copies the parent image, and displays it if its Do_not_show=0).

Chunk structure (length, name, CRC) and the chunk-naming system are identical to those defined in the PNG specification. Definitions of Compression_type, Filter_type, and Interlace_type are also the same as defined in the PNG specification.

6.1 Delta-PNG critical chunks

This section describes critical Delta-PNG chunks. MNG-compliant decoders must recognize and process them.

6.1.1 DHDR Delta-PNG datastream header

The DHDR chunk introduces a Delta-PNG datastream. Subsequent chunks, through the next IEND chunk, are interpreted according to the Delta-PNG format.

The DHDR chunk can contain 4, 12, or 20 bytes:

```
2 bytes: Object_id (nonzero unsigned integer).
         Identifies the parent object from which changes
         will be made.  This is also the Object_id
         of the child image, which can be used as the
         parent image for a subsequent Delta-PNG.

1  byte: Image_type.
```

```
            0: Image type is unspecified.  An IHDR or IPNG
               chunk must be present.

            1: Image type is PNG.  IHDR and IPNG can be
               omitted under certain conditions.

            2: Image type is JNG.  Delta_type must not
               be 1 or 3.

      1  byte: Delta_type.

            0: Entire image replacement.

            1: Block pixel addition, by samples, modulo
               2^Sample_depth.

            2: Block alpha addition, by samples, modulo
               2^Sample_depth.  Regardless of the color
               type of the parent image, the IDAT data
               are written as a grayscale image (color
               type 0) but the decoded samples are used
               as deltas to the alpha samples in the
               parent image.  The parent image must have
               (or be promoted to via the PROM chunk)
               color type 4 or color type 6.

            3: Block pixel replacement.

            4: Block alpha replacement.

            5: No change to pixel data.

      4 bytes: Block_width (unsigned integer).  Omit when
               Delta_type=5.

      4 bytes: Block_height (unsigned integer).  Omit when
               Delta_type=5.

      4 bytes: Block_X_location (unsigned integer),
               measured in pixels from the left edge of
               the parent object.  Omit when Delta_type=0
               or when Delta_type=5.

      4 bytes: Block_Y_location (unsigned integer),
               measured in pixels from the top edge of
               the parent object.  Omit when Delta_type=0
               or when Delta_type=5.
```

The `Object_id` must identify an existing object, and the object must be a “concrete” object, i.e., it must

have the property Concrete_flag=1.

The Image_type, whether given explicitly as 1 or 2 or implied by the presence of an IHDR, IPNG, or JHDR chunk, must be the same as that of the parent object.

When Delta_type=0, the width and height of the child image are given by the Block_width and Block_height fields.

For all other values of Delta_type, the width and height of the child image are inherited from the parent object.

When Delta_type=1, 2, 3, or 4, the Block_width and Block_height fields give the size of the block of pixels to be modified or replaced, and Block_X_location and Block_Y_location give its location with respect to the top left corner of the parent object. The block must fall entirely within the parent object.

Entire image replacement

When Delta_type=0 in the DHDR chunk, the pixel data in the IDAT chunks represent a completely new image, with dimensions given by the Block_width and Block_height fields of the DHDR chunk. Data from chunks other than IDAT or JDAT can be inherited from the parent object.

Block pixel addition

When Delta_type=1 in the DHDR chunk, the pixel data in the IDAT chunks represent deltas from the pixel data in a parent object known to the decoder.

The compression method, filter method, and interlace method need not be the same as those of the parent object.

The IDAT chunk data contains a filtered and perhaps interlaced set of delta pixel samples. The delta samples are presented in the order specified by interlace method, filtered according to the filter method and compressed according to the compression method given in the IHDR chunk. The actual pixel values are calculated using the method defined in the Delta_type field of the DHDR chunk. Only the pixels in the block defined by the block location and dimensions given in the DHDR chunk are changed. The size of the IDAT data must correspond exactly to this rectangle.

An encoder calculates the new sample values from the samples in the parent object and those in the child image by subtracting the parent object samples from the child image samples, modulo 2^Sample_depth. When decoding the IDAT chunk, the child image bytes are obtained by adding the delta bytes to the parent object bytes, modulo 2^Sample_depth. This is similar in operation to the PNG SUB filter, except that it works by samples instead of by bytes.

When Color_type=3, the deltas are differences between index values, not between color samples.

Block alpha addition

When Delta_type=2 in the DHDR chunk, the pixel data in the IDAT chunks represent deltas from the alpha data in a parent object known to the decoder. The color samples are not changed, and the updated alpha samples are calculated in the same manner as the updated pixel samples are calculated when Delta_type=1.

The Sample_depth of the data must match that of the parent object, and color type is 0 (grayscale), regardless of the Color_type of the parent object. The parent object must have an alpha channel or must have been promoted to a type that has an alpha channel. The compression method, filter method, and interlace method need not be the same.

Block pixel replacement

When Delta_type=3 in the DHDR chunk, the pixel data in the IDAT chunks represent replacement values for the pixel samples in the rectangle given by the block location and dimension fields in the DHDR chunk.

The Sample_depth and Color_type of the data must match that of the parent object. The compression method, filter method, and interlace method need not be the same.

Block alpha replacement

When Delta_type=4 in the DHDR chunk, the pixel data in the IDAT chunks represent replacement values of the alpha samples in the rectangle given by the block location and dimension fields in the DHDR chunk. The Sample_depth of the data must match that of the parent object, and color type is 0 (grayscale), regardless of the Color_type of the parent object. The parent object must have an alpha channel or must have been promoted to a type that has an alpha channel. The compression method, filter method, and interlace method need not be the same.

No change to pixel data

When Delta_type=5 in the DHDR chunk, there is no change to the pixel data, and it is an error for IDAT or JDAT to appear.

6.1.2 IEND End of Delta-PNG datastream

End of Delta-PNG datastream. An IEND chunk must be present for each DHDR chunk in a MNG datastream.

The IEND chunk is empty.

6.1.3 PROM Promotion of parent object

This chunk is used to "promote" a parent object to a higher bit depth or to add an alpha channel, before making changes to it.

```
1 byte: new Color_type.
1 byte: new Sample_depth.
1 byte: fill_method
     0: left-bit-replication
     1: zero fill
```

When a decoder encounters the PROM chunk, it must promote the pixel data. The cases are:

G -> GA (`Color_type 0 -> 4`)

Don't change the gray values. Set all the alpha values to fully opaque, except for pixels marked transparent by cheap transparency–set their alpha values to fully transparent. Discard the cheap transparency information (the PNG `tRNS` chunk data).

RGB -> RBGA (`Color_type 2 -> 6`)

Don't change the RGB values. Convert the `tRNS` chunk data to alpha values as in the G -> GA promotion.

G -> RGB (`Color_type 0 -> 2`)

Set R, G, and B equal to the gray value. Apply the same operation to the cheap transparency data (if any).

GA -> RGBA (`Color_type 4 -> 6`)

Set R, G, and B equal to the gray value. Don't change the alpha values.

G -> RGBA (`Color_type 0 -> 6`)

Set R, G, and B equal to the gray value. Handle transparency as in the G -> GA promotion.

indexed -> RGB (`Color_type 3 -> 2`)

Set R, G, and B according to the palette entry corresponding to the index. Discard the cheap transparency information (if any).

indexed -> RGBA (`Color_type 3 -> 6`)

Set R, G, and B as in indexed -> RGB. Set the alpha value according to the cheap transparency information (if any). Discard the cheap transparency information.

JNG-G -> JNG-GA (`JNG Color_type Gray -> GA`)

Don't change the gray values. Set all the alpha values to fully opaque. The given sample depth is the new sample depth for the alpha channel.

JNG-C -> JNG-CA (`JNG Color_type Color -> CA`)

Don't change the color values. Set all the alpha values to fully opaque. The given sample depth is the new sample depth for the alpha channel.

No change in `Color_type`

Only the sample depth is changed. The new sample depth must be larger than the old one.

If the sample depth has been changed, the sample values must be widened. The decoder must use left-bit-replication or zero-fill according to the specified `fill_method` to fill the additional bits of each sample. If cheap transparency information is present in a grayscale or truecolor object, its sample values must also be widened in the same manner. If the image type is JNG, then only the alpha channel is affected, if one is present. If the `Color_type` has been promoted from indexed, the original bit depth is always considered to be 8. See the PNG specification [PNG] for further information on these filling methods.

If the basis object contains data from the PNG bKGD chunk, this data must be promoted as well. If a grayscale object is being promoted to a truecolor object, the background RGB samples are set equal to the grayscale background sample. If the bit depth has been changed, the background samples are widened in accordance with the specified fill_method. If the basis object is a JNG, the bKGD chunk is not affected.

If the basis object contains data from the PNG sBIT chunk, this data must also be promoted. If a grayscale object is being promoted to a truecolor object, the new RGB bytes are set equal to the grayscale byte. When an alpha channel is added, the alpha byte is set equal to the sample depth of the basis image. If the sample depth has been changed, the sBIT bytes do not change.

The PROM chunk is not permitted to "demote" a parent object to an object with a lesser bit depth or from one with an alpha channel to one without an alpha channel.

The PROM chunk must appear ahead of the IHDR chunk, if IHDR is present, and ahead of any chunks that would have followed IHDR, if IHDR is omitted.

6.1.4 IHDR

Inside a Delta-PNG datastream, the IHDR chunk introduces an incomplete PNG (Portable Network Graphics) datastream. The parent object must be a PNG or PNG-based Delta-PNG. The datastream can be introduced by a complete PNG IHDR chunk or by an IPNG chunk, which is empty.

If the IHDR chunk is present, its Width, Height, Sample_depth, and Color_type fields are ignored. The values for these parameters are inherited from the parent object or from the PROM chunk.

The Compression_method, Interlace_type, and Filter_type fields, if different from those of the parent object, are used in decoding any subsequent IDAT chunks, and the new values will be inherited by any subsequent image that uses this object as its parent.

See the PNG specification for the format of the PNG chunks. The PNG datastream must contain at least IHDR and IEND (whether actually present in the datastream or omitted and included by implication, as described below) but can inherit other chunk data from the parent object. Except for IDAT, any chunks appearing between IHDR and IEND are always treated as replacements or additions and not as deltas.

6.1.5 IPNG Incomplete PNG

The IPNG chunk is empty.

The IPNG chunk can be used instead of the IHDR chunk if the IHDR chunk is not needed for resetting the value of Compression_method, Filter_type, or Interlace_type. The purpose of this chunk is to identify the beginning of the PNG datastream, so decoders can start interpreting PNG chunks instead of Delta-PNG chunks. The decoder must treat this datastream as though the IHDR chunk were present in the location occupied by the IPNG chunk.

The IHDR chunk can also be omitted when Image_type=1 and the PNG stream begins with a PLTE chunk, a PPLT chunk, or an IDAT chunk. In this case, no IPNG chunk is required, either. The decoder must treat

this datastream as though the IHDR chunk were present, immediately preceding the first PNG chunk. If the first PNG chunk is neither a PLTE chunk, a PPLT chunk, nor an IDAT chunk, then either the IPNG or IHDR must be present to introduce the PNG datastream.

6.1.6 PLTE and tRNS

If the PLTE chunk is present, it need not have the same length as that inherited from the parent object, but it must contain the complete palette needed in the child image. If it is shorter than the palette of the parent object, decoders can discard the remaining entries and the child image must not refer to them. Decoders can also truncate any tRNS data inherited from an indexed-color parent object. If the new palette is longer than the parent palette, and a new tRNS chunk is not present in an indexed-color image, the tRNS data must be extended with opaque entries. The new palette must not be longer than the object's Sample_depth would allow, and must not have more than 256 entries.

When processing the tRNS chunk, if Color_type=3 and PLTE is not supplied, then the number of allowable entries is determined from the number of PLTE entries in the parent object. A tRNS chunk appearing in a Delta-PNG datastream is always treated as a complete replacement for the tRNS chunk data in the parent object. All entries beyond those actually supplied are overwritten with the "opaque" value (255).

6.1.7 PPLT Incomplete palette

If it is desired only to overwrite or add palette entries, the PPLT chunk can be used. This might be useful for palette-animation applications. This chunk can also be used to overwrite or add entries to the transparency (alpha) data from the parent's tRNS chunk.

The PPLT chunk contains a Delta_type byte and one or more groups of palette entries:

```
1 byte: PPLT_delta_type
        0: values are replacement RGB samples
        1: values are delta RGB samples
        2: values are replacement alphas
        3: values are delta transparencies
        4: values are replacement RGBA samples
        5: values are delta RGBA samples
1 byte: First_index, first group
1 byte: Last_index, first group
1, 3, or 4 bytes: first set of samples
etc.
1, 3, or 4 bytes: last set of samples
1 byte: First_index, second group
etc.
```

The Last_index must be equal to or greater than First_index. The groups are not required to appear in ascending order. If any index of any group is beyond the end of the parent object's palette, the palette

and tRNS data must be extended just as if a longer complete PLTE chunk had appeared. If there are gaps in the resulting extended palette, the colors must be filled with {0,0,0} and the alphas filled with 255. If alpha samples are supplied (PPLT_delta_type > 1) and no tRNS data is present in the parent object, a tRNS chunk must be created in the child object as though a complete tRNS chunk had appeared. The new palette must not be longer than the object's Sample_depth would allow.

When PPLT_delta_type=0, the values are replacements for the existing samples in the palette.

When PPLT_delta_type=1, the values are added to the existing samples (modulo 256) to obtain the new samples.

If the new entry is beyond the range of the original palette, the values are simply appended, regardless of the contents of PPLT_delta_type

The PPLT chunk replaces the Draft 33 IPLT chunk. To achieve the effect of IPLT, use

```
PPLT 0 0 n-1 r g b r g b ...
```

in place of

```
IPLT  r g b r g b ...
```

6.1.8 JHDR

Inside a Delta-PNG datastream, the JHDR chunk introduces an incomplete JNG (JPEG Network Graphics) datastream. The parent object must be a JNG or JNG-based Delta-PNG. The datastream is introduced by a complete JNG JHDR chunk.

If the JHDR chunk is present, its Width, Height, JDAT_Sample_depth, and JDAT_color_type, JDAT_Filter_type, and JDAT_interlace_type fields are ignored. The values for these parameters are inherited from the parent object.

The IDAT_compression_method, IDAT_interlace_type, and IDAT_Filter_type fields, if different from those of the parent object, are used in decoding any subsequent IDAT chunks, and the new values will be inherited by any subsequent image that uses this object as its parent.

See the JNG specification above for the format of the JNG chunks. The PNG datastream must contain at least JHDR and IEND but can inherit other chunk data from the parent object. Except for IDAT, any chunks appearing between IHDR and IEND are always treated as replacements or additions and not as deltas.

-[is it worthwhile defining an IJNG chunk to perform a similar function to IPNG for JNG, when the JHDR data isn't needed?]

6.1.9 DROP Drop chunks

All chunks in the parent object with the specified name are inhibited from being copied into the child image.

```
4 bytes: Chunk name (ASCII text)
etc.
```

If multiple names appear in the DROP chunk, it is shorthand for multiple DROP chunks.

6.1.10 DBYK Drop chunks by keyword

```
4 bytes: Chunk name (ASCII text)

1  byte: Polarity (unsigned integer)
      0: only
      1: all-but

Keywords (null-separated Latin-1 text strings)
```

The chunk name must be the name of a chunk whose data begins with a null-terminated text string. Some parent object chunks with the specified chunk name are inhibited from being copied into the child image. If polarity is <only> then any parent chunk whose keyword appears in the keywords list is inhibited. If polarity is <all-but> then any parent object chunk whose keyword does not appear in the keywords list is inhibited.

The format of the keyword is the same as that specified for the parent chunk. Comparisons of keywords in the parent chunk and the DBYK chunk are case sensitive.

Use caution when printing or displaying keywords (Refer to Security considerations, Chapter 15, below).

6.1.11 ORDR Ordering restrictions

The ORDR chunk informs the applier of the Delta-PNG of the ordering restrictions for ancillary chunks. It contains one or more 5-byte sequences:

```
4 bytes: Chunk_name
1 byte:  Order_type
     0: anywhere
     1: after IDAT and/or JDAT
     2: before IDAT and/or JDAT
     3: before IDAT, but not before PLTE
     4: before IDAT, but not after PLTE
etc.
```

Critical chunk names must not appear in the ORDR chunk. The applier needs to know everything about them anyway.

If a chunk name appears in the ORDR chunk, it is a promise that any chunk of that name appearing in the parent object which is not inhibited by DROP/DBYK will not be broken by this Delta-PNG, and therefore the applier must copy it into the child image at a location compatible with its ordering restrictions.

If any ancillary chunk appears in the parent object, and it is not inhibited, and its name does not appear in the ORDR chunk, then the applier should copy it into the child only if it knows the chunk well enough to be sure that it is consistent with the changes made by the Delta-PNG, and knows where it can be placed in the child. Those conditions are always true of safe-to-copy chunks.

If any critical chunk defined in neither this specification nor the PNG specification appears in the parent object or in the Delta-PNG, it is a fatal error unless the applier knows how to handle it. The specification of the critical chunk can include provisions for this scenario.

6.1.12 Delta-PNG ancillary chunks

This section describes ancillary Delta-PNG chunks. MNG-compliant decoders should recognize and process them, but are not required to.

6.1.13 gAMA, cHRM, iCCP, sRGB Colorspace chunks

A gAMA, cHRM, iCCP, sRGB or similar chunk existing in the parent object would not affect the pixel data in a concrete object inherited by this Delta-PNG datastream because they are not used in decoding the pixel data. Applications are responsible for ensuring that the pixel values that are inherited from the parent object are the raw pixel data that existed prior to any transformations that were applied while displaying the parent image. These color transformations are applied to the resulting pixel data for display purposes.

6.1.14 oFFs and pHYs

MNG viewers must ignore oFFs and pHYs chunks that appear inside a PNG stream, when Framing_mode != 1 and Framing_mode != 4 from the MNG FRAM chunk. When Framing_mode=1 or Framing_mode=4 and when the FRAM chunk is not present, viewers must treat these chunks in the same manner as they would treat them in single-image PNG files. MNG editors are expected to treat them as unknown chunks that will be handled as described in Paragraph 6.1.11, below.

6.2 Chunk ordering requirements

The PNG specification places ordering requirements on many chunks with respect to the PLTE and IDAT chunks. If neither of these two chunks is present, and the ORDR chunk is not present, known chunks (always including all standard chunks described in the PNG specification) are considered to have appeared in their proper order with respect to the critical chunks. Unknown chunks are ordered as described in Paragraph 6.1.11, above.

7 Chunk copying rules

The chunk copying rules for MNG are the same as those in PNG, except that a MNG editor is not permitted to move unknown chunks across any of the following chunks:

- SAVE
- SEEK
- IHDR
- JHDR
- IEND
- DHDR
- BASI
- PAST
- SHOW

The copy-safe status of an unknown chunk is determined from the chunk name, just as in PNG. If bit 5 of the first byte of the name is 0 (Normally corresponding to an uppercase ASCII letter), the unknown chunk is critical and cannot be processed or copied. If it is 1 (usually corresponding to a lowercase ASCII letter), the unknown chunk is ancillary and its copy-safe status is determined by bit 5 of the fourth byte of the name, 0 meaning copy-unsafe and 1 meaning copy-safe.

If an editor makes changes to the MNG datastream that render unknown chunks unsafe-to-copy, this does not affect the copy-safe status of any chunks beyond the next SEEK chunk or prior to the previous one. However, if it makes such changes prior the the SAVE chunk, this affects the copy-safe status of all top-level unknown chunks in the entire MNG datastream.

As in PNG, unsafe-to-copy ancillary chunks in the top-level MNG datastream can have ordering rules only with respect to critical chunks. Safe-to-copy ancillary chunks in the top-level MNG stream can have ordering rules only with respect to the SAVE, SEEK, SHOW, and PAST chunks, IHDR–IEND, DHDR–IEND, JHDR–IEND, and BASI–IEND sequences, or with respect to any other critical "header-end" sequence that might be defined in the future that could contain IDAT or similar chunks.

The copying rules for unknown chunks inside IHDR–IEND, BASI–IEND, DHDR–IEND, and JHDR–IEND sequences are governed by the PNG and JNG specifications, and any changes inside such sequences have no effect on top-level chunks.

The copy-safe status of chunks inside a DHDR–IEND sequence depends on the copy-safe status of its parent object.

8 Minimum requirements for MNG-compliant viewers

This section specifies the minimum level of support that is expected of MNG-compliant decoders, and provides recomendations for viewers that will support slightly more than the minimum requirements. All critical chunks must be recognized, but some of them can be ignored after they have been read and recognized. Ancillary chunks can be ignored, and do not even have to be recognized.

Anything less than this level of support would require subsetting.

8.1 Required PNG/JNG chunk support

IHDR, JHDR, PLTE, IDAT, JDAT, IEND

The PNG and JNG critical chunks must be fully supported. All values of `Color_types`, `Bit_depth`, `Compression_method`, `Filter_method` and `Interlace_method` must be supported (interlacing, as in PNG, need not necessarily be displayed on-the-fly; the image can be displayed after it is fully decoded). The alpha-channel must be supported, at least to the degree that fully opaque pixels are opaque and fully transparent ones are transparent. It is recommended that alpha be fully supported.

tRNS

The PNG `tRNS` chunk, while it is an ancillary chunk, must be supported in MNG-compliant viewers, at least to the degree that fully opaque pixels are opaque and fully transparent ones are transparent. It is recommended that alpha data from the `tRNS` chunk be fully supported in the same manner as alpha data from an RGBA image or a JNG with an alpha channel contained in `IDAT` chunks.

Other PNG/JNG ancillary chunks

Ancillary chunks other than PNG `tRNS` can be ignored, and do not even have to be recognized.

Color management

It is highly recommended that decoders support at least the `gAMA` chunk to allow platform-independent color rendering.

8.2 Required MNG chunk support

MHDR

`Ticks_per_second` must be supported by animation viewers.

MEND

The `MEND` chunk must be recognized but does not require any processing.

LOOP, ENDL

The `Repeat_count` must be supported. `Nest_level` should be used as a sanity check but is not required.

DEFI, CLON

Must be fully supported. All objects can be treated as "concrete" if the decoder does not wish to take advantage of the distinction between "abstract" and "concrete".

DISC

Need not be supported, but support is recommended to conserve memory.

BASI, BACK, PAST

Must be fully supported.

FRAM

`Framing_mode` and clipping parameters must be supported. `Interframe_delay` must be supported except by single-frame viewers. `Sync_id` and `Sync_timeout` data can be ignored.

MOVE, CLIP, SHOW

Must be fully supported.

SAVE and SEEK

Can be ignored. It is recommended that objects be marked "frozen" when the `SAVE` chunk is processed, so that unneeded objects can be discarded when the `SEEK` chunk or an empty `DISC` chunk is processed. Other information need only be "saved" and "restored" when the viewer is able to skip or jump to random `SEEK` chunk locations. The optional index can be ignored. Slide-show controllers may wish to support `SAVE` and `SEEK` fully.

8.3 Required Delta-PNG chunk support

DHDR, PROM, IHDR, JHDR, JDAT, IDAT, IPNG, PLTE, tRNS, IEND, PPLT

Must be fully supported.

DROP, DBYK, ORDR

Can be recognized and ignored. These are only of concern to MNG editors and to MNG viewers that handle private chunks or chunks that can be selected by keyword, such as `pCAL` and `iCCP`. If you decide to support such chunks, then you will also have to support these three chunks.

Ancillary chunks

Ancillary chunks appearing in Delta-PNG datastreams must be treated in the same manner as if they appeared in a PNG or JNG datastream. See the recommendations, above. Note that the PNG `tRNS` chunk must be supported, despite its being an ancillary chunk.

9 Recommendations for encoders

9.1 Embedded images in LOOPs

Embedded images should not be enclosed in loops unless absolutely necessary. It is better to store them ahead of time and then use SHOW chunks inside the loops. Otherwise, decoders will be forced to repeatedly decode them. See Examples 2, 8, 11, and 12 in Chapter 16, below.

9.2 Including optional index in SAVE chunk

Authors of MNG files that are intended for transmission over a network should consider whether it is more economical for the client to rebuild the index from scratch than it is to transmit it. Web pages that are likely to be downloaded over slow lines, and whose clients are unlikely to use the index anyway, generally should have empty SAVE chunks. No information is lost by deleting the index, because the MNG datastream contains all of the information needed to build the index. If an application does build an index, and the file is going to be kept as a local file, the application should replace the empty SAVE chunk with one containing the index. See Paragraph 4.4.1, above.

10 Recommendations for decoders

10.1 ENDL without matching LOOP

If a decoder reads an ENDL chunk for which the matching LOOP chunk is missing, or has been skipped for some reason, any active loops with a higher nest_level should be terminated, and processing can resume after the next SEEK chunk. See above.

10.2 Note on compositing

The PNG specification gives a good explanation of how to composite a partially transparent image over an opaque image, but things get more complicated when both images are partially transparent.

Pixels in PNG images are represented using gamma-encoded RGB (or gray) samples along with a linear alpha value. Alpha processing can only be performed on linear samples. This chapter assumes that R, G, B, and A values have all been converted to real numbers in the range [0..1], and that any gamma encoding has been undone.

For a top pixel {Rt,Gt,Bt,At} and a bottom pixel {Rb,Gb,Bb,Ab}, the composite pixel {Rc,Gc,Bc,Ac} is given by:

```
Ac = 1 - (1 - At)(1 - Ab)
if (Ac != 0) then
  s = At / Ac
  t = (1 - At) Ab / Ac
else
  s = 0.0
  t = 1.0
endif
Rc = s Rt + t Rb
Gc = s Gt + t Gb
Bc = s Bt + t Bb
```

When the bottom pixel is fully opaque (Ab = 1.0), the function reduces to:

```
Ac = 1
Rc = At Rt + (1 - At) Rb
Gc = At Gt + (1 - At) Gb
Bc = At Bt + (1 - At) Bb
```

When the bottom pixel is not fully opaque, the function is much simpler if premultiplied alpha is used. A pixel that uses non-premultiplied alpha can be converted to premultiplied alpha by multiplying R, G, and B by A.

For a premultiplied top pixel {Rt,Gt,Bt,At} and a premultiplied bottom pixel {Rb,Gb,Bb,Ab}, the premultiplied composite pixel {Rc,Gc,Bc,Ac} is given by:

```
Ac = 1 - (1 - At)(1 - Ab)
Rc = Rt + (1 - At) Rb
Gc = Gt + (1 - At) Gb
Bc = Bt + (1 - At) Bb
```

As mentioned in the PNG specification, the equations become much simpler when all the alpha values are integers, and the RGB samples need not be linear in that case.

10.3 Retaining object data

The decoder must retain information about each object (except for objects with `Object_id=0`) for possible redisplay with the `SHOW` chunk or for possible use as the parent object for a subsequent Delta-PNG datastream.

The following information must be retained, for each nonzero `object` that is defined and not subsequently discarded:

- The object attribute set (potential visibility, location, clipping boundary data from the `DEFI`, `MOVE`, `CLIP`, and `SHOW` chunks, and pointer to an object buffer).

- The pixel data and the values associated with other recognized PNG chunks such as PLTE and gAMA, subject to the chunk copying rules and the DROP/DBYK chunks. If the object is "abstract", the data can be stored in any convenient form. If it is "concrete", it must be stored in an object buffer in a manner that would permit the complete restoration of the original PNG or JNG file or its equivalent.
- The most recent values of Target_x and Target_y, if the object was the destination of a PAST chunk.

When the encoder knows that data in the object buffer will not be needed by subsequent frames, it can make life easier for decoders by using Object_id=0 or by using the DISC or the SEEK chunk. Abstract images rather than concrete objects should be used if the encoder knows that the data will not later be used as the parent object for a Delta-PNG.

10.4 Decoder handling of fatal errors

When a fatal error is encountered, such as a bad CRC or an unknown critical MNG chunk, MNG viewers should attempt to recover gracefully by abandoning processing of the segment and searching for a SEEK chunk. If errors occur before the SAVE chunk is reached, the viewer should probably just abandon the MNG datastream.

When an error occurs within a image datastream, such as an unknown critical PNG chunk or a missing parent object where one was required, only that image should be abandoned and the associated image should be discarded. If a bad CRC is found, indicating a corrupted datastream, the entire segment should be abandoned, as above.

MNG editors, on the other hand, should be more strict and reject any datastream with errors unless the user intervenes.

10.5 Decoder handling of interlaced images

Decoders are required to be able to interpret datastreams that contain interlaced PNG images, but are only required to display the completed frames; they are not required to display the images as they evolve. Viewers that are decoding datastreams coming in over a slow communication link might want to do that, but MNG authors should not assume that the frames will be displayed in other than their final form.

10.6 Decoder handling of palettes

When a PLTE chunk is received, it only affects the display of the PNG or Delta-PNG datastream that includes it, and any subsequent Delta-PNG datastreams that depend on it.

If PLTE or PPLT is present in a Delta-PNG datastream, the new palette is used in displaying the image defined by the Delta-PNG; if no IDAT chunk is present and the image type is PNG indexed-color, then the

resulting image is displayed using the old pixel samples as indices into the new palette, which provides a "palette animation" capability.

If a frame contains two or more images, the PLTE chunk in one image does not affect the display of the other, unless one image is a subsequent Delta-PNG without a PLTE chunk, that has been declared by the DHDR Object_id field to depend on the other.

A composite frame consisting only of indexed-color images should not be assumed to contain 256 or fewer colors, since the individual palettes do not necessarily contain the same set of colors. Encoders can supply a top-level sPLT chunk with a suggested reduced global palette, to help decoders build an appropriate palette when necessary.

10.7 Behavior of single-frame viewers

Viewers that can only display a single frame must display the first frame that they encounter. It is strongly recommended that such viewers understand the fPRI chunk Paragraph 4.5.2, above, which will enable them to find and display the best representative frame, if the encoder has identified one.

10.8 Clipping

MNG provides four types of clipping, in addition to any clipping that might be required due to the physical limitations of the display device.

Frame_width and Frame_height

The Frame_width and Frame_height are defined in the MHDR chunk and cannot be changed.

Decoders can use these parameters to establish the size of a window in which to display the MNG frames. When the Frame_width or Frame_height exceeds the physical dimensions of the display hardware, the contents of the area outside those dimensions is undefined. If a viewer chooses, it can create "scroll bars" or the like, to enable persons to pan and scroll to the offscreen portion of the frame. If this is done, then the viewer is responsible for maintaining and updating the offscreen portion of the frame.

In the case of a MNG datastream that consists of a PNG (or JNG) datastream, with the PNG (or JNG) signature, the Frame_width and Frame_height are defined by the Width and Height fields of the IHDR (or JHDR) chunk.

frame clipping boundaries

The frame clipping boundaries are optionally defined in the FRAM chunk, and cannot be changed within a frame. Viewers must, prior to displaying each frame, clear the area within the frame clipping boundaries to the background color. Viewers must not change any pixels outside the framing boundaries; encoders must be able to rely on the fact that the part of the display that is outside the frame clipping boundaries (but inside the area defined by Frame_width and Frame_height) will remain on the display from frame to frame without being explicitly redisplayed. See Example 8 below, which

displays a large background image once, and then, in each frame, only redisplays the portion of the background surrounding the moving sprite.

image clipping boundaries

The frame clipping boundaries are defined in the DEFI and CLIP chunks. They are associated with individual objects, not with the frames, and they can be changed within a frame. They are useful for exposing only a portion of an image in a frame, to achieve effects such as scrolling, panning, or gradual exposure.

PAST clipping boundaries

The clipping performed in PAST gives a fourth type that has no dependencies on the other types, since the object CLIP data is ignored and the destination object defines its own boundaries.

11 Recommendations for editors

12 Editing datastreams with optional index

Editors must recreate or delete the optional SAVE chunk index whenever they make any change that affects the offsets of chunks following the portion of the datastream that is changed. If the changes do not involve the addition, deletion, or relocation of segments, frames, and images, then it is sufficient to zero out the offsets.

The SAVE chunk is not considered to be in any MNG segment, so changing it has no effect on the copy-safe status of unknown chunks in any other part of the MNG datastream.

When the SAVE chunk is expanded to include an index, all chunks that follow will have their offsets changed by an amount equal to the change in the length of the SAVE chunk's data segment, so the offset table will have to be adjusted accordingly. If a SAVE chunk is already present with zero offsets, the correct offsets can be written without adjustment.

13 Miscellaneous topics

13.1 File name extension

On systems where file names customarily include an extension signifying file type, the extension .mng is recommended for MNG files. Lower case .mng is preferred if file names are case-sensitive. The extension .jng is recommended for JNG files.

13.2 Internet media type

When and if the MNG format becomes finalized, the MNG authors intend to register `Video/mng` as the Internet Media Type for MNG [RFC-2045, RFC-2048]. At the date of this document, the media type registration process had not been started. It is recommended that implementations also recognize the interim media type `video/x-mng`.

Although we define a standalone JNG format, we recommend that such files be used only temporarily while compiling or disassembling MNG datastreams. We do not intend to register an Internet Media Type for JNG files.

13.3 Uniform Resource Locator (URL)

-[Explanation of the use of externally accessible SEEK, FRAM, and eXPI chunk names as anchors etc., as in SRC=file.mng#segment_name and SRC=file.mng?segment_name]

See [RFC-1738], Uniform Resource Locators (URL) and the W3C SMIL recommendation.

14 References

[PNG]

Boutell, T., et. al., PNG (Portable Network Graphics Format Version 1.0), RFC 2083, <URL:ftp://ds.internic.net/rfc/rfc2083.txt> also available at <URL:ftp://swrinde.nde.swri.edu/pub/png/documents/>. This specification has also been published as a W3C Recommendation, which is available at <URL:http://www.w3.org/TR/REC-png.html>.

[PNG-EXT]

PNG Special-Purpose Public Chunks, <URL:ftp://swrinde.nde.swri.edu/pub/png/documents/>.

[RFC-2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119/BCP 14, Harvard University, March 1997.

[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-1738]

Berners-Lee, T., Masinter, L., McCahill, M., "Uniform Resource Locators (URL)", RFC 1738, CERN, Xerox Corporation, University of Minnesota, December 1994. <URL:ftp://ds.internic.net/rfc/rfc1738.txt>

15 Security considerations

Security considerations are addressed in the basic PNG specification.

An infinite or just overly long loop could give the appearance of having locked up the machine, as could an unreasonably long Interframe delay or a misplaced `Sync_id` with a long `Sync_timeout` value. Therefore a decoder should always provide a simple method for users to escape out of a loop or delay, either by canceling the MNG entirely or just proceeding on to the next `SEEK` chunk. (The `SEEK` chunk makes it safe for a viewer to resume processing after it encounters a corrupted portion of a MNG datastream.)

No known additional security concerns are raised by this format.

16 Appendix: Examples

We use the "#" character to denote commentary in these examples; such comments are not present in actual MNG datastreams.

16.1 Example 1: A single image

The simplest MNG datastream is a single-image PNG datastream. The simplest way to create a MNG from a PNG is:

```
copy file.png file.mng
```

The resulting MNG file looks like:

```
\211 P N G \r \n ^z \n  # PNG signature.
IHDR 720 468 8 0 0 0 0  # Width and Height, etc.
IDAT ...
IEND
```

16.2 Example 2: Very simple movie

This example demonstrates a very simple movie, such as might result from directly converting an animated GIF that contains a simple series of full-frame images:

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 256 300   # Width and Height.
     30        # 30 ticks per second.

tERm 2 0 120 10   # When done, repeat from SAVE 10 times

FRAM 4 0 2 0 0 0 30 # set Framing_mode=4 (because the
        #  images are opaque) and Frame_duration to 1 sec

DEFI 1 IHDR ...  IDAT ...  IEND # Eight PNG datastreams
DEFI 2 IHDR ...  IDAT ...  IEND # are read and stored as
DEFI 3 IHDR ...  IDAT ...  IEND # abstract images, and
DEFI 4 IHDR ...  IDAT ...  IEND # are displayed as they
DEFI 5 IHDR ...  IDAT ...  IEND # are read.
DEFI 6 IHDR ...  IDAT ...  IEND
DEFI 7 IHDR ...  IDAT ...  IEND
DEFI 8 IHDR ...  IDAT ...  IEND

SAVE
SHOW 1 8
MEND
```

16.3 Example 3: Simple slideshow

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 720 468   # Width and Height.
     1         # 1 tick per second.
FRAM 4 0 2 2 0  2 0 600 0 # Set Frame_duration to 0
  # Sync_timeout to 600 sec and Sync_id list to {0}.
SAVE
SEEK "Briefing to the Workforce"
IHDR ...  IDAT ...  IEND  # DEFI 0, visible, abstract
SEEK "Outline"            # is implied.
IHDR ...  IDAT ...  IEND
SEEK "Our Vision"       IHDR ...  IDAT ...  IEND
SEEK "Our Mission"      IHDR ...  IDAT ...  IEND
SEEK "Downsizing Plans" IHDR ...  IDAT ...  IEND
MEND
```

16.4 Example 4: A more storage-efficient slideshow

This slideshow gives exactly the same output as Example 3, but the storage in the datastream is more efficient (the IDAT chunks will be smaller) while the memory requirements in the decoder are larger. Image ID 1 is used to store the ornate logos and frame design that appear on every slide. The remaining smaller images only contain the text and other information that is unique to each slide; the DEFI 0 chunks give the location.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 720 468   # Width and Height.
     1         # 1 tick per second.
DEFI 1 1       # Define image 1, invisible, abstract.
IHDR ...  IDAT ...  IEND
BACK 0 0 0 2 1 # Make it the background image.
PLTE ...       # Global palette
FRAM  4 0  2 2 0 2  0 600 0 # set Frame_duration to 0
  # Sync_timeout to 600 sec and Sync_id list to {0}.
SAVE
SEEK "Briefing to the Workforce"
DEFI 0 ... IHDR ... PLTE IDAT ...  IEND # uses global PLTE
SEEK "Outline"
DEFI 0 ... IHDR ... PLTE IDAT ...  IEND # uses global PLTE
SEEK "Our Vision"
DEFI 0 ... IHDR ... PLTE IDAT ...  IEND # uses global PLTE
SEEK "Our Mission"
DEFI 0 ... IHDR ... PLTE IDAT ...  IEND # uses global PLTE
SEEK "Downsizing Plans"
DEFI 0 ... IHDR ... PLTE ... IDAT ... IEND # uses own PLTE
MEND
```

16.5 Example 5: Simple movie

This movie is still fairly simple, but it capitalizes on frame-to-frame similarities by use of Delta-PNG datastreams, and also demonstrates the use of the `fPRI` chunk.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 720 468   # Width and Height.
     30        # 30 ticks per second.
tEXtTitle\0Sample Movie
fPRI 0 128     # Default frame priority is "medium".
FRAM 1 0  2 0 0 0  3 # Set Frame_duration to 1/10 sec.
DEFI 1 0 1       # Set default image to 1 (concrete).
SAVE
SEEK "start"

IHDR 720 468 8 2 0 0 0   # DEFI 1 is implied.
IDAT ...
IEND

DHDR 1 1 1 20 30 100 220   # A PNG-delta frame.
IDAT ...       # The IDAT gives the 20x30 block
IEND           # of deltas.

DHDR 1 1 1 20 30 102 222   # Another PNG-delta frame.
IDAT ...       # This time the deltas are in a 20 x 30
IEND           # block at a slightly different location.

SEEK "frame 3"  # Ok to restart here because a
                # complete PNG frame follows.
fPRI 0 255    # This is the representative frame that
IHDR 720 468 ...# will be displayed by single-frame
IDAT ...        # viewers.
IEND
fPRI 0 128      # Return to medium frame priority.

DHDR 1 1 1 720 468 0 0     # Another PNG-delta frame.
IDAT ...   # The entire 720x468 rectangle changes
IEND       # this time.

SEEK "end"
MEND            # End of MNG datastream.
```

16.6 Example 6: Single composite frame

Here is an example single-composite-frame MNG, with thumbnails, which takes a grayscale image and draws it side-by-side with a false-color version of the same image:

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 1024 512 1 # Width, Height, ticklength.
BACK 16448 16448 52800 1 # Must use sky blue background.

DEFI 1 1     # Define invisible abstract thumbnail image.
IHDR 64 64 4 3 0 0 0
IDAT
IEND
eXPI 1 "thumbnail 1"
DEFI 1 1       # Also define a larger thumbnail.
IHDR 96 96 4 3 0 0 0
IDAT
IEND
eXPI 1 "thumbnail 2"
DISC           # Discard the thumbnail image.

FRAM 2 "Two views of the data"
DEFI 1 0 1 6 6     # Define first (bottom) image.
IHDR 500 500 16 0 .. # A 16-bit graylevel image.
gAMA 50000
IDAT ...
IEND                # End of image.

CLON 1 2 0 0 1 0 518 6 # Make a full "concrete" clone.
DHDR 2 1 5          #  Modify it (no change to pixels).
ORDR faLT 2         #  Establish chunk placement.
gAMA 100000   # Gamma value is 100000 (gamma=1.0).
tEXtComment\0The faLT chunk is described in ftp://swrinde....
faLT ...            # Apply pseudocolor to parent image.
IEND                # End of image.
DEFI 3 0 0 900 400   # Overlay near lower right-hand corner.
IHDR 101 101 2 3 ...
gAMA 50000          # We need a new gAMA because
PLTE ...            # this is not a Delta-PNG datastream.
tRNS ...            # It is transparent (maybe a logo).
IDAT ...            # Note that the color type can differ
IDAT ...            #    from that of the other images.
IEND                # End of image.

MEND                # End of MNG datastream.
```

16.7 Example 7: Movie with sprites

Here is another movie, illustrating the use of Delta-PNG datastreams as sprites

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 512 30  # Start of MNG datastream.
FRAM 2 "frame 1" 0  2 0 0 0  3 # First frame
                 # sets Frame_duration=3 ticks.
DEFI 1           # Define image 1 (abstract, LOCA 0 0).
IHDR 512 512 ... # It is a full-display PNG image.
etc              # Chunks according to PNG spec.
IEND             # SHOW 1 is implied by DEFI 1.
DEFI 2 0 1 300 200 # Define image 2, concrete.
IHDR 32 32 ...   # It is a small PNG.
gAMA 50000
IDAT ...
IEND
FRAM 0 "frame 2"   # Start new frame.
                 # New location for image 1 is still 0,0.
SHOW 1           # Display image 1 from previous frame.
MOVE 2 2 1 10 5  # New (delta) location for image 2.
SHOW 2           # Retrieve image 2 from previous frame,
CLON 2 3 0 0 1   # make a full clone of it as image 3.
     0 400 500   # Location for image 3.
DHDR 3 1 3 0 0 0 0      # Modify image 3.
tRNS ...         # Make it semitransparent.
IEND             # SHOW 3 is implied by CLON visibility.
FRAM 0 "frame 3"   # Next frame (repeat this FRAM-SHOW 1 3
                 #   sequence with different locations to
                 #   move the images around).
                 # New location for image 1 is still 0,0.
MOVE 2 2 1 10 5  # New (delta) location for image 2.
MOVE 3 3 1 5 -2  # New location for image 3.
SHOW 1 3         # Show images 1 through 3.
FRAM 0 "frame 4" #  Another frame.
etc.
FRAM 0 "frame 99"
etc.             # More frames.
MEND             # End of MNG datastream.
```

16.8 Example 8: Movie with an animated sprite

This movie illustres the use of several abstract images with Show_mode=6 to describe an animated sprite, and the PAST chunk to turn it around. The sprite runs back and forth across the background ten times. The FRAM clipping boundaries restrict the screen updates to the small region that changes, with a little "wiggle room" to make sure the disturbed part of the background gets updated.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 512 30  # Start of MNG datastream.
FRAM 2 "frame 1" 0  2 0 0 0  3 # First frame.
DEFI 1 IHDR 512 512 ... # Background PNG image.
etc ... IEND     # Chunks according to PNG spec.

DEFI 10 1 0 x0 y0 # Static part of sprite.
IHDR 64 64 ...  IDAT ...  IEND
DEFI 11 1 0 x0 y1 # View 1 of animated part.
IHDR 64 32 ...  IDAT ...  IEND # (y1=y0+64)
DEFI 12 1 0 x0 y1 # View 2 of animated part.
IHDR 64 32 ...  IDAT ...  IEND
DEFI 13 1 0 x0 y1 # View 3 of animated part.
IHDR 64 32 ...  IDAT ...  IEND

FRAM 0  0  0 0 2 0  0 x0-dx x0+64+dx y0-dy y1+32+dy
LOOP 0 0 10
LOOP 1 0 150
FRAM 0 "left-to-right" 0 0 2 0 1 dx dx dy dy
MOVE 10 13 1 dx dy # Move animated icon {dx, dy}.
SHOW  1 SHOW 10    # Show background and static part.
SHOW 11 13 6       # Select the next view of the
ENDL 1             # animated part and show it.

FRAM SHOW 1
PAST 10  0 0 0  10  1 4  0 0 0  0 0 64 64
PAST 11  0 0 0  11  1 4  0 0 0  0 0 64 32
PAST 12  0 0 0  12  1 4  0 0 0  0 0 64 32
PAST 13  0 0 0  13  1 4  0 0 0  0 0 64 32
LOOP 1 0 150
FRAM 0 "right-to-left" 0 0 2 0 1 -dx -dx -dy -dy
MOVE 10 13 1 -dx -dy # Move animated icon {-dx, -dy}
SHOW  1 SHOW 10    # Show background and static part.
SHOW 11 13 6       # Select the next view of the
ENDL 1             # animated part and show it.
ENDL 0 FRAM
MEND
```

16.9 Example 9: “Fading in” a transparent image

The opaque parts of this image will “fade in” gradually. This example also illustrates the use of the `PPLT` and `fPRI` chunks.

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 64 64 30     # Width, Height, ticklength.
BACK 52800 52800 52800 # "Browser gray" default background.

FRAM 1 0  2 0 0 0  3 #  Set Frame_duration=3 ticks.
DEFI 1 1 1  # Invisible and "concrete".
IHDR ...    # PNG header.
PLTE ...
tRNS 0      # Entries are zero for the transparent (0)
            # color and 255 for the nontransparent ones.
IDAT ...
IEND
fPRI 0 0    # Give the fade-in sequence a low priority.
CLON 1 2 1  # Make a working concrete copy of the image
            # that will be modified during the low-priority
            # part of the datastream.  It is a full clone.
DHDR 2 1 5  # No change to pixel data.
tRNS 0 0 0 0 0 0 ...  # make all pixels fully transparent
IEND
SHOW 2 2 3  # Make it visible but don't show it now.

LOOP 0 0 15
DHDR 2 1 5  # an image delta.
            # Delta-type is no change to pixels.
PPLT 1 10 3 16 16 16 16 ... # increment all alphas except
IEND                        # for entry 0 by 16.
SHOW 2
ENDL 0      # Nontransparent pixel alpha=15, 31, ... 240.

DISC 2      # Discard the working copy.
fPRI 0 255  # Give the final frame the highest value
FRAM 0 0  1 0 0 0  60 # Hold the last frame for at least
  # 60 ticks (2 sec).  Applications might show it longer..
SHOW 1      # This copy still has alpha=255 for the
            # opaque pixels and alpha=0 for the others.
MEND        # End of MNG.
```

16.10 Example 10: Storing three-dimensional images

In this example, we store a series of twenty-four 150 x 150 x 150 blocks of 8-bit voxels. Each block is stored as a composite frame with the first image being a PNG whose pixels represent the top layer of voxels, which is followed by 149 Delta-PNG images representing the rest of the layers of voxels. Only one `image` is defined, through which the parent image is passed along from PNG to Delta-PNG to Delta-PNG. This example also illustrates the use of unregistered ancillary chunks that describe the x, y, and z scales and pixel calibration.

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 150 150 1   # Width, Height, ticklength.

tEXtTitle\0Weather modeling results
tEXtComment\0The xxSC, yySC, zzSC, and ttSC chunks
 in this file are written according to the proposed
 chunk specifications version 19970203 and Sci-Vis
 chunks specification version 19970203 available at
 ftp://swrinde.nde.swri.edu/pub/png-group/documents/
xxSC kch\0 [sig\0] kilometers\0 0\0 150
yySC kch\0 [sig\0] kilometers\0 0\0 150
zzSC kch\0 [sig\0] Height (kilometers)\0 0\0 15
ttSC kch\0 [sig\0] Time (hours)\0 0\0 24
pCAL kch\0 0 255 0 2 Degrees Celsius\0 0\0 45
DEFI 1 0 1       # all images will have image = 1
SAVE             # and be visible and "concrete"
SEEK
FRAM 2      # Initial composite image
IHDR 150 150 16  # Width, Height, bit depth for top layer.
     0 0 0 0     # Color, comp, filter, interlace.
IDAT ...
IEND             # No DEFI chunk, so it is image 0.
DHDR 1 1 0       # Source=0, PNG, pixel subtraction,
     150 150 0 0 # Block is entire image.
IDAT ...     # IHDR is omitted; everything matches top.
IEND         # IEND is also omitted.
etc.         # Repeat DHDR through IEND 148 more times.
SEEK
FRAM         # End of first block.
etc.         # Repeat FRAM through SEEK 19 more times.
SEEK
MEND         # End of MNG.
```

16.11 Example 11: Tiling

Here is another composite frame, illustrating the use of the LOOP syntax to tile a large (1024 by 768) image area with a small (128 by 64) image.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 1024 768 1 # Start of MNG datastream.
FRAM 2
DEFI 1 1 0 0 -64 # Set up an offscreen "abstract" copy
IHDR 128 64 ... PLTE ... IDAT ... IEND  # of the tile.
LOOP 0 0 12   # Y loop -- make 12 rows of tiles.
MOVE 1 1 1 0 64 # Move the first copy down 64 rows.
SHOW 1        # Display it.
CLON 1 2 1    # Create a partial clone of the tile.
LOOP 1 0 7    # X loop - 7 additional columns.
MOVE 2 2 1 0 128 # Move it to the right 128 columns.
SHOW 2        # Use the second copy.
ENDL 1
ENDL 0
MEND
```

Here is a better approach, which creates a reusable tiled image by means of the `PAST` chunk.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 1024 768 1 # Start of MNG datastream.
DEFI 1 1 # Set up an offscreen "abstract" copy
IHDR 128 64 ...  PLTE ... IDAT ... IEND  # of the tile.
DEFI 2        # The abstract, visible, viewable image to
BASI 1024 768 8 2 0 0 0 0 0 0 0 1 #  be tiled. Initially
IEND          # all pixels are zero.
PAST 2 0 0 0  # Destination and target location.
  # src mod orient  offset        clipping
     1   0    8     0   0 512   0 0 1024 0 768
  #  End of PAST chunk data.
MEND
```

16.12 Example 12: Scrolling

Here is an example of scrolling a 3000-line-high image (perhaps an image of some text, but could be anything) through a 256-line-high window with an alpha-blended border.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 256 30   # Width and Height on screen.
BACK 50000 50000 50000 0 #  advisory gray background
DEFI 1 1 0 0 256  # Define image 1 but don't display now.
                  # Initially it is offscreen, just
                  # below the 512 by 256 window.
IHDR 512 3000 1 0 ... # A PNG datastream containing the
PLTE ...          # text (or whatever) to be scrolled.
IDAT ...
IEND

DEFI 2
IHDR 512 256 8 6 ... # A PNG datastream containing some kind
PLTE ...             # of alpha-blended border that is
tRNS ...             # transparent in the center.
IDAT ...
IEND

LOOP 0 0 3256
MOVE 1 1 1 0 -1 # Jack image 1 up one scanline, 3256 times.
           # It ends up just above the 512 by 256 window.
           # The border does not move.
FRAM 4  0  2 0 0 0  0 # Frame_duration = 0 ticks.
           # We use Framing_mode=4 to avoid unnecessary
           # screen clearing between frames.
SHOW 1     # show first image and continue without delay
FRAM 4  0  2 0 0 0  1 # Frame_duration = 1 tick.
SHOW 2     # composite second image over first, wait 1 tick
ENDL 0
MEND
```

Alternatively, we can declare the scrolling object to be the background:

```
[same as above down to the LOOP chunk]
BACK 50000 50000 50000 2 1 #  advisory gray background
                           #  mandatory image background
FRAM 1  0  2 0 0 0  1 # Frame_duration = 1 tick.
LOOP 0 0 3256
MOVE 1 1 1 0 -1 # Jack background up one scanline, 3256 times.
SHOW 2     # composite second image over it, wait 1 tick
ENDL 0
MEND
```

16.13 Example 13: Converting a GIF animation to MNG

Outline of a program to convert GIF animations to MNG format:

```
begin
    write "MHDR" and "BACK" chunks
    saved_images := 0
    Frame_duration := 0
    First_frame := TRUE
    if(loops>1) "write tERm 2 0 0 loops" chunk
    for subimage in gif89a file do
       if(Frame_duration != gif_duration) then
         Frame_duration := gif_duration
         write "FRAM 2 0 2 2 0 2 0 Frame_duration 0" chunk
         First_frame := FALSE
       else if(First_frame == TRUE)then
         write "FRAM 2" chunk
         First_frame := FALSE
       else
         write "FRAM" chunk
       endif
       if(X_loc == 0 AND Y_loc == 0) then
          write "DEFI saved_images 1 1" chunk
       else
          write "DEFI saved_images 1 1 X_loc Y_loc" chunk
       write "<image>"
       write "SHOW 0 saved_images" chunk
       if (gif_disposal_method == 0
          OR gif_disposal_method == 2) then
          /* (undefined or restore background) */
             write "DISC" chunk
             saved_images := 0
       else if (gif_disposal_method == 1) then
          /* (keep) */
          saved_images := saved_images + 1
       else if (gif_disposal_method == 3) then
          /* (restore previous) */
          write "DISC saved_images" chunk
       endif
    endfor
    write "FRAM" and "MEND" chunks
end
```

Where "<image>" represents a PNG or Delta-PNG containing a GIF frame converted to PNG format.

Caution: if you write such a program you might have to pay royalties in order to use it or to convey it to anyone else.

17 Credits

Contributors' names are presented in alphabetical order: [some addresses need to be updated from Greg's list]

- Mark Adler, madler@alumni.caltech.edu
- Thomas Boutell, boutell@boutell.com
- John Bowler, jbowler@acm.org
- Christian Brunschen, cb@df.lth.se
- Glen Chapman, glenc@clark.net
- Adam M. Costello, amc@cs.berkeley.edu
- Lee Daniel Crocker, lee@piclab.com
- Peter da Silva, peter@starbase.neosoft.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, alex@hermes.si
- Gerard Juyn, gjuyn@xs4all.nl
- 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
- Carl Morris, msftrncs@htcnet.com
- Owen Mortensen, ojm@csi.compuserve.com
- Josh M. Osborne, stripes@va.pubnix.com
- Keith S. Pickens, ksp@swri.edu
- Glenn Randers-Pehrson, randeg@alumni.rpi.edu
- Nancy M. Randers-Pehrson, randeg@alumni.rpi.edu

- Greg Roelofs, newt@pobox.com
- Willem van Schaik, gwillem@ntuvax.ntu.ac.sg
- Guy Schalnat
- Paul Schmidt, pschmidt@photodex.com
- Smarasderagd, smar@reptiles.org
- Thomas R. Tanner, ttehtann@argonet.co.uk
- Guido Vollbeding, guivol@esc.de
- Tim Wegner, 71320.675@compuserve.com
- Alaric B. Williams, png@abwillms.demon.co.uk

Trademarks:

- GIF is a service mark of CompuServe Incorporated.
- Macintosh is a trademark of Apple Computer, Inc.
- Windows 3.1 is a trademark of Microsoft Corporation.
- X Window System is a trademark of the Massachusetts Institute of Technology.

Author's Address

Glenn Randers-Pehrson
611 Rivershore Court
Edgewood, MD 21040-3603

Phone: (410) 278-6554

EMail: randeg@alumni.rpi.edu

End of MNG Specification. Expires 03 Mar 1999