PNG-GROUP-DRAFT
MNG
Expires: 25 Jul 1997

PNG Development Group
mpng-list@dworkin.wustl.edu
25 Jan 1997

MNG (Multiple-image Network Graphics) Format Version 19970125

Draft 34, file: draft-mng-19970125.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 thirty-third MNG draft (draft-mng-19970121)

- Added "APLY" chunk, and a requirement that bare PND datastreams be stored for later use.
- Removed references to "max_image_width" (left over from earlier MHDR chunk design).
- Revised Example 8 to illustrate the use of PAST to turn an animated sprite around.
- Revised Example 9 to illustrate APLY chunk.
- Rewrote terminology section and other sections to speak of "objects" rather than "images".
- Speak of "viewable" instead of "displayable" and "potential visibility" instead of "visibility".

- Added a chapter on "objects"
- Added optional "viewable" flag to the BASI chunk.
- Fixed bugs in BACK, DEFI, and CLON chunks in the examples.
- Changed the defaults for omitted visibility and location in the CLON chunk (they now retain the values from the parent rather than being reset to zero)
- Changed "section" to "segment" throughout (when speaking of SEEK segments, but not when speaking of sections of this document).
- Added FRAM termination conditions from DOH-3.
- Changed recommended behavior for encoders encountering an unmatched ENDL, from "ignore the ENDL" to a more conservative "terminate all inner loops and resume after SEEK".

Abstract

This document presents the *[proposed]* 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 "PND" format, 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.

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 PND datastream.

A PND datastream defines an image in terms of a parent PNG or PND 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 format uses the same chunk structure that is defined in the PNG specification, and shares other features of the PNG format. Any valid PNG 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, converting GIF animations to MNG format.

Contents

1 Introduction 5
2 Terminology 7

3 Objects . . . 9
3.1 Embedded Objects . . . 10
3.2 Object attributes . . . 10
3.3 Object buffers . . . 11
4 MNG chunks . . . 12
4.1 Critical MNG control chunks . . . 12
4.1.1 MHDR MNG datastream header . . . 13
4.1.2 MEND End of MNG datastream . . . 13
4.1.3 SAVE Save information . . . 13
4.1.4 SEEK Seek point . . . 14
4.1.5 LOOP, ENDL Define a loop . . . 15
4.1.6 FRAM Delimit a frame . . . 16
4.1.7 SHOW Show images . . . 22
4.1.8 BACK Background . . . 24
4.2 Critical MNG image defining chunks . . . 25
4.2.1 DEFI Define an object . . . 25
4.2.2 IHDR, PNG chunks, IEND . . . 26
4.2.3 BASI, PNG chunks, IEND . . . 27
4.2.4 DHDR, PND chunks, DEND . . . 28
4.2.5 APLY Apply an image delta . . . 29
4.2.6 CLON Clone an image . . . 29
4.2.7 PAST Paste an image into an another . . . 31
4.2.8 DISC Discard objects . . . 33
4.2.9 MOVE New image location . . . 33
4.2.10 CLIP Object clipping boundaries . . . 34
4.3 Ancillary MNG chunks . . . 35
4.3.1 eXPI Export image . . . 35
4.3.2 fPRI Frame priority . . . 36
4.3.3 nEED Resources needed . . . 36
4.3.4 tERm Termination . . . 37
4.3.5 pHYs Physical pixel size . . . 38
4.3.6 PNG ancillary chunks . . . 38
5 The PND format . . . 39
5.1 PND critical chunks . . . 40
5.1.1 DHDR PND datastream header . . . 40
Entire image replacement . . . 41
Block pixel addition . . . 41
Block alpha addition . . . 42
Block pixel replacement . . . 42
Block alpha replacement . . . 42
No change to pixel data . . . 43
5.1.2 DEND End of PND datastream . . . 43
5.1.3 PROM Promotion of parent object . . . 43

5.1.4 IHDR/IPNG, PNG chunks, IEND 44
IHDR 44
IPNG Incomplete PNG 44
PLTE and tRNS 45
IPLT Incomplete palette 45
gAMA, cHRM, sRGB Color encoding chunks 45
oFFs and pHYs 45
Chunk ordering requirements 45
IEND 45
5.1.5 DROP Drop chunks 46
5.1.6 DBYK Drop chunks by keyword 46
5.1.7 ORDR Ordering restrictions 46
6 Chunk copying rules 47
7 Recommendations for encoders 48
7.1 Embedded images in LOOPs 48
8 Recommendations for decoders 48
8.1 ENDL without matching LOOP 48
8.2 Note on Compositing 48
8.3 Retaining object data 49
8.4 Decoder handling of fatal errors 50
8.5 Decoder handling of interlaced images 50
8.6 Decoder handling of palettes 50
9 Recommendations for editors 51
10 Miscellaneous Topics 51
10.1 File name extension 51
10.2 Internet media type 51
10.3 Uniform Resource Locator (URL) 51
11 References 51
12 Security considerations 52
13 Appendix: Examples 53
13.1 Example 1: A single image 53
13.2 Example 2: Very simple movie 53
13.3 Example 3: Simple slideshow 54
13.4 Example 4: A more storage-efficient slideshow 54
13.5 Example 5: Simple movie 55
13.6 Example 6: Single composite frame 56
13.7 Example 7: Movie with sprites 57
13.8 Example 8: Movie with an animated sprite 58
13.9 Example 9: “Fading in” a transparent image 60
13.10 Example 10: Storing three-dimensional images 61
13.11 Example 11: Tiling 61
13.12 Example 12: Scrolling 63
13.13 Example 13: Converting a GIF animation to MNG 64

14 Credits 65

1 Introduction

This *[proposed]* specification defines the format of the MNG (Multiple-image Network Graphics) datastream.

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

<URL:http://www.wco.com/~png/%gt;

A MNG datastream describes a sequence of single frames, each of which can be composed of one or more image appearances.

An image appearance is

- An embedded image whose "potential_visibility=0".
- A directive by a "SHOW" chunk to display a potentially visible image.

An image is a viewable object.

An object is is the result of decoding

- A PNG "IHDR-IEND" datastream as defined in the PNG specification.
- A PND "DHDR-IEND" datastream (A PNG-Delta, as defined herein).
- A MNG "BASI-IEND" datastream.
- A MNG "PAST" chunk.
- A MNG "CLON" chunk.

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.
- "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 manipulation of 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 1-4.

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. At some future date, support for a lossy image format such as the proposed PNP (Portable Network Photo) format might be added. PNP is under discussion by pnp-list@dworkin.wustl.edu.

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 6, 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 datastream, with a PNG 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 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.

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 the datastream.
- It provides "restart" points at which an animation can be resumed in case of data loss or corruption.
- 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.
- An "eXPI" chunk allows authors to mark one or more images for export outside the scope of MNG, where they can be used for icons or similar purposes.
- A series of PNG images can be losslessly converted to MNG and back to a series of equivalent PNG images, even when the delta format is used to store them in the MNG.
- Multiple-image GIF files can be losslessly converted to MNG.
- It is complementary to MPEG and does not attempt to replace MPEG for lossy storage of video.

2 Terminology

See also the glossary in the PNG specification.

image, image_id

A viewable object. The image_id is an unsigned sixteen-bit number that serves as a pointer to a set of object attributes.

object, object_id

An image, an image delta, or a nonviewable basis object. The object_id is an unsigned sixteen-bit number that serves as a pointer to 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.

color encoding

File gamma and chromaticity values, or an sRGB rendering intent, 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 image also carries data from other known PNG chunks that are present.

abstract image or object

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

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.

parent, parent object, or parent image

An object to which a delta is applied.

child, or child image

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

embedded object or image

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

image N or object N

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

frame

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

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.

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.

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).

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).

frozen object

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

viewable image

A stored object that is capable of being made visible. An image is viewable while an image delta and 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.

potentially visible image

- An not-yet-defined viewable object that is "marked" for on-the-fly display as the embedded image that defines it is being decoded.
- An existing viewable object that is "marked" for being made visible by subesequent "`SHOW`" chunks.

3 Objects

An "`object`", which is identified by an "`object_id`", is an image, an image delta, or a nonviewable basis object. The "`object_id`" is an unsigned sixteen-bit number that serves as a pointer to a set of object attributes.

An image is a viewable object, and its "object_id" is also called an "image_id" in this specification.

3.1 Embedded Objects

An embedded object is

- A PNG datastream ("IHDR", PNG chunks, "IEND").
- A BASI datastream ("BASI", PNG chunks, "IEND").
- A PND datastream ("DHDR", PND/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 (32k by number_of_attributes) array in which to store them, but real applications will probably 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.
- 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.

Frozen or not frozen.

All objects are initially "not frozen". Any objects in existence 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 "potential_visibility" byte of the "DEFI" or "CLON" chunk that introduced it. The "potential visibility" of viewable objects can be changed by means of 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 means of 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, 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 aa nonviewable one cannot. Any attempt to do so must be ignored.

- A PNG datastream always has the "viewable" attribute.
- A PND datastream always has the "not viewable" attribute when it is stored rather than immediately applied to a parent object.
- 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 an embedded or stored PND 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

- Concrete PNG or PND.
  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 or a PND image delta, 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".

  - In the case of a PND image delta, the object also carries the data from the "DHDR" chunk that tells the decoder how to apply the data to a parent object.
  - Applications that recreate PNG files from a series of PNG and PND 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 Paragraph 5.1.5, DROP, Paragraph 5.1.6, DBYK, and Paragraph 5.1.7, 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. It is always safe 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.

- At some future time, image data in a lossy format might be allowed.

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 "object_id" 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 PND 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: max_frame_width  (unsigned integer).
         Maximum width of any frame to be
         displayed.

4 bytes: max_frame_height (unsigned integer).
         Maximum width of any frame to be
         displayed.

4 bytes: ticks_per_second  (unsigned integer).
         This is the unit used by the FRAM chunk to
         specify frame_duration and sync_timeout.
```

The "max_frame_width" and "max_frame_height" fields give the intended display size (measured in pixels) and provide default clipping boundaries (see Paragraph 4.2.10, below). These can 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 should 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 SAVE Save information

[This chunk could be eliminated. The first SEEK chunk would also serve as the SAVE chunk.]

The "SAVE" chunk marks a point in the datastream at which images and other chunk information are "saved"; decoders encountering a "SEEK" must restore this information if it has redefined or discarded it.

The "SAVE" chunk is empty.

It appears after the set of chunks that define information that must be retained for the remainder of the datastream. The "SAVE" chunk must be present when the "SEEK" chunk is present. These can be chunks that define images, or they can be chunks that define other information such as "gAMA", "cHRM", and "sPLT".

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 it is not permitted, because this would also modify the object buffer that the original object points to.

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 are responsible for saving a copy of the chunk data (in any convenient form) when the "SAVE" chunk is encountered and restoring it when the "SEEK" chunk is encountered. If no instance of the chunk appeared ahead of the "SEEK" chunk, the decoder must restore the chunk data to its original "unknown" condition when it encounters a "SEEK" chunk. Known chunks in this category include "DEFI", "FRAM", "BACK", "cHRM", "eXPI", "fPRI", "gAMA", "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 encounters a "SEEK" chunk, but it must retain those instances that were defined prior to the "SAVE" chunk. Known chunks in this category include only "sPLT".

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.

4.1.4 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. It need not be terminated by a null byte; if it is, the null byte will be ignored. 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. Keywords are case-sensitive. Use caution when printing or displaying keywords (Refer to Security considerations, Chapter 12, 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.

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 PND 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.

4.1.5 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

```
1  byte: start_loop_level (unsigned integer).
1  byte: must_run_loop (unsigned integer).
      0: The decoder must run the loop exactly
         repeat_count times, because execution of
         the loop modifies the properties of
         images that might be needed later.
      1: It is safe for a decoder to run the loop
         fewer than repeat_count times, provided
         that it runs the loop at least once. Every
         iteration of the loop generates an
         identical sequence of frames.
4 bytes: repeat_count (unsigned integer), range
         [0..2^31-1].
```

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 images for any PND datastreams in the loop are the images in existence prior to entering the "LOOP" chunk, but in subsequent iterations these parent images might have been modified. The "must_run_loop" field can be used to inform decoders that it is safe to reduce the number of loop iterations.

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

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.

It is the responsibility of the encoder to make sure that the assertions made by the "must_run_loop" field are true. When "must_run_loop=1", all iterations of the loop are identical, and a viewer could choose to actually execute the loop just once, while storing copies of the composited frames for redisplay, and then replaying the stored frames for the remaining "repeat_count" iterations. Also, when "must_run_loop=1", it is safe to escape from the loop upon reaching the "ENDL" chunk even though the loop has not yet been executed for "repeat_count" iterations. It is unsafe, however, to escape from the interior of a loop while it is being executed; in this event, the decoder must proceed to the next "SEEK" chunk before resuming.

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

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

When the "ENDL" chunk is encountered, the loop "repeat_count" is decremented. 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 "loop_level" must be present earlier in the MNG datastream.

The "SEEK" chunk is permitted inside a "LOOP-ENDL" pair, but only when "must_run_loop=1".

4.1.6 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, 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.
```

```
          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.

          3:  The group of "n" images appearing prior to
              the next "FRAM" chunk define "n" frames as
              they accumulate, and a frame boundary
              occurs after each image is defined or
              appears in a "SHOW" chunk.  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.

     n bytes: frame_name (Latin-1 Text).  Can be
          omitted if all other fields are also omitted.

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

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

          This field and the next three can 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:  external-signal, for the next frame only.
          2:  external-signal, also reset default.
          3:  deterministic, for the next frame only.
          4:  deterministic, also reset default.
          5:  decoder-discretion, for the next frame only.
          6:  decoder-discretion, also reset default.
          7:  user-discretion, for the next frame only.
          8:  user-discretion, also reset default.

     1 byte:  change_frame_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.

1  byte: fb_delta_type (unsigned integer).
       (omit this and the following four fields
       if change_frame_boundaries=0).
       0: frame_boundary values are given directly.
       1: frame_boundaries are determined by adding
          the FRAM data to their previous values.

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: frame_duration (unsigned integer) (omit
         if change_frame_duration=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.

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.

The "frame_name" must conform to the same formatting rules as those for a "SEEK" keyword. No specific use for the "frame_name" is specified in this document, but 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 "frame_duration" 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 .

The termination condition given in the "change_sync_timeout_and_termination" field specifies

how much longer, after the normal frame duration has elapsed, the frame will endure. It can take the following values:

deterministic

The frame endures no longer than the normal frame duration. 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, by no more than the timeout. A streaming decoder could use 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 name matches the name field, 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>
frame_duration  := 0
left_fb         := 0
right_fb        := max_frame_width
top_fb          := 0
bottom_fb       := max_frame_height
termination     := decoder-discretion
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. If the images are transparent or do not cover the entire frame, as defined by the "max_frame_width" and "max_frame_height" fields of the "MHDR" chunk, 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.

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 visible image.
- Decoding a "DHDR-DEND" sequence, when it defines a visible image.
- Decoding a "BASI-DEND" sequence, when it defines a visible image.
- Decoding a "PAST" chunk, when it defines a visible image.
- Decoding a "SHOW" chunk, when it requests that a 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, 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" information.

The "frame_duration" value is the desired minimum time to elapse from the beginning of displaying one image until the beginning of displaying the next image.

Viewers are expected to display all of the images at once, if possible, or as fast as can be managed, without clearing the display and restoring the background between images. 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.

When "framing_mode=3", a frame boundary occurs after each image appears, without clearing the display and restoring the background between frames until the next "FRAM" chunk is encountered.

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
```

If "MOVE" or "CLIP" chunks appear, they must appear before the images to which they pertain are caused to appear. Decoders can simply erase the display prior to displaying the first image, and then show each image as it is defined, without erasing the display between frames.

The "framing_mode" also affects the way decoders handle the "pHYs" chunk (see Paragraph 4.3.5, below).

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 0 0 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. If multiple channels (not defined in this specification) are not present, viewers can ignore other values appearing in the "sync_id" list.

4.1.7 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 potential
     visibility byte is also omitted.  If so,
     decoders must assume the default value,
     "last_image=first_image."

1 byte:  show_mode (unsigned integer)
     0:  Make the images visible and display
         them (set "potential_visibility=0").
     1:  Make the images invisible (set
         "potential_visibility=1").
     2:  Don't change potential_visibility;
         display those that are potentially
         visible.
     3:  Mark images "potentially visible" but
         do not display them.
     4:  Toggle potential_visibility; display
         any that are potentially visible after
         toggling.
     5:  Toggle "potential_visibility" but do
         not display even if potentially visible
         after toggling.
     6:  Make the next image in the cycle visible
         and display it.  Make all other images
         in the cycle invisible.
     7.  Make the "next" image in the cycle
         visible but do not display it.  Make
         all other images in the cycle invisible.

     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 "potential_visibility" flag for each of the objects, and, if directed, 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 "potential_visibility" status.

If "first_image > last_image" then the images are processed in reverse order. When "framing_mode=1" or when the "FRAM" chunk is not present, each image is displayed as a separate frame. When "show_mode=1, 3, 5, or 7", when "show_mode=2" and no image is visible, or when "show_mode=4" and an image becomes invisible, no frame is generated. Also, when the "SHOW" chunk is empty and there are no existing visible frames, 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=2" or "framing_mode=3" 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 examine the "potential_visibility" of each image in the range "first_image" through "last_image", and make the next one 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. These modes can be useful in manipulating moving animated objects, where a group of sequential images represent different views of the object.

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" chunk. 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.

Either of the following sequences would cause the image identified by "image_id=4" in a composite frame to blink:

```
LOOP 0 0 10
FRAM 2
SHOW 1 3     # 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", "BASI-IEND", or "DHDR-DEND" sequence or "PAST" chunk with a "SHOW" chunk to display the resulting image, if it was already caused to appear by

"visibility=0" in the "DEFI" chunk that introduced the image. Similarly, the "CLON" chunk need not be followed by a "SHOW" chunk, if "potential_visibility=0" in the "CLON" chunk.

It is not an error for the "SHOW" chunk to name an "image" that has not previously been defined. In such cases, nothing is done to the nonexistent image. If a nonviewable object is named, its potential visibility flag is changed but it is not displayed even if the "SHOW" chunk requests it.

4.1.8 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 is advisory; applications
        can use it if they choose to.
     1: background color is mandatory; applications
        must use it.
     This byte can be omitted; if so, the background
     color is advisory.
```

Viewers are expected to composite every frame in the MNG datastream, whether it be a PNG or PND datastream or a group of PNG or PND datastreams delimited by "FRAM" chunks, against a fresh copy of the background, clipped to the frame boundaries given in the "FRAM" chunk (they might actually follow some other procedure, but the final appearance of each frame must be the same as if they had).

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

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.

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, abstract_flag, location, and clipping boundaries) for any subsequent images that are defined with "IHDR-IEND" or objects that are defined with "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:  potential_visibility (unsigned integer)
     0:  Make the objects potentially visible.
     1:  Make the objects invisible.
     This field can be omitted if the abstract_flag,
     location, and clipping boundary fields
     are also omitted.  When it is omitted, the image
     is potentially visible (potential_visibility=0).

1 byte:  abstract_flag (unsigned integer)
     0:  Make the objects "abstract" (image can not
         be the source for a PND)
     1:  Make the objects "concrete" (object can be
         modified by a PND).
     This field can be omitted if location, and
     clipping boundary fields are also omitted.
     When it is omitted, the object is made "abstract"
     (abstract_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, max_frame_width, 0, max_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 "DHDR", "SHOW", "CLON", "MOVE", "CLIP", and "DISC" chunks can use this number to identify it.

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

Negative values are permitted for the X and Y location and clipping boundaries. The positive directions are downward and rightward from the upper left corner of the display.

The "DEFI" chunk values remain in effect until another "DEFI" chunk or a "SEEK" chunk appears. 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
  potential_visibility := 0
         abstract_flag := 0
            x_location := 0
            y_location := 0
               left_cb := 0
              right_cb := max_frame_width
                top_cb := 0
             bottom_cb := max_frame_height
```

If "object_id" is an identifier that already exists when an "DEFI", "IHDR" or "BASI" chunk is encountered, the object previously associated with the identifier is discarded (or disassociated with it, if the identifier was a partial clone of another object), along with any location, potential visibility, and clipping boundary data associated with it.

4.2.2 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 the PNG "sRGB", "gAMA", or "cHRM" chunks appear in the top-level MNG datastream but not 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. MNG applications that recreate PNG files must write these chunks 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.

When "framing_mode != 1" 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" (i.e. when each PNG image is an individual frame), these chunks can be treated as described in the PNG specification.

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

If "abstract_flag=1" and "image_id != 0", the decoder must store the original pixel data losslessly, because it is possible that a subsequent PND stream might want to modify it. If "abstract_flag=0", the decoder can store the pixel data in any form that it chooses.

4.2.3 BASI, PNG chunks, IEND

The "BASI" chunk introduces a basis object, which, while it may 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, and 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.

```
4 bytes: width (unsigned integer).
4 bytes: height (unsigned integer).
1 byte:  bit_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^bit_depth, whose first entry contains the given {red, green,

blue} triple, and whose remaining entries are filled with zeroes. If the "viewable" field is omitted, the object is not viewable.

The "BASI" a datastream contains PNG chunks, but is not necessarily a PNG datastream. It can be incomplete or it can deviate in certain ways from the PNG specification. It can serve as a parent object for a PND 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 PND 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 PND that uses this as the parent object can supply "IDAT" chunk or chunks.
- The "PLTE" chunk can be omitted or incomplete even when the color_type=3. If so, the subsequent PND that uses this as the parent object must supply a complete "PLTE" chunk.

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 a nonzero "object_id" and the location and potential visibility for the basis object. When "potential_visibility=0" or "viewable=1", the resulting image, after the pixel samples are filled in, must be a legal PNG datastream. The "abstract_flag" can be either 0 (abstract) or 1 (concrete), depending on whether the basis image is intended for subsequent use by a PND datastream or not.

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.

4.2.4 DHDR, PND chunks, DEND

A PND (PNG-Delta) datastream.

See Chapter 5, The PND Format, below, for the format of the PND datastream. Any chunks between "DHDR" and "DEND" are written and decoded according to the PND format. If the "image_id" of the PND "DHDR" chunk points to an existing parent object, the PND is immediately applied to it and the child image replaces it, and the resulting image is immediately displayed if its "potential_visibility=0". The parent object must be concrete (i.e. it must have "abstract_flag=1").

If the "image_id" of the PND "DHDR" chunk points to a nonexistant object, the PND is stored as a new nonviewable, invisible concrete object with the specified "object_id", in a form that the decoder can use later when the PND is applied to a parent object.

4.2.5 APLY Apply an image delta

The "APLY" chunk directs that a previously defined image delta be applied to a previously-defined parent object. The chunk contains 4 bytes:

```
2 bytes: object_id (unsigned integer)  Identifier
         of the parent object.  The child image
         will be given the same object_id.

2 bytes: delta_id (unsigned integer)  Identifier
         of the image delta that is to be applied
         to the parent object to produce the
         child image.
```

The parent must be a concrete object and it must not be an image delta. The child is always a viewable concrete image.

4.2.6 CLON Clone an image

Create a clone (a new copy) of an image, with a new "image_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 object data.

      1: partial clone; only the location, clipping
         boundaries, and potential visibility are
         copied and a link is made to the object data.

      2: renumber object (this is equivalent to
         "CLON old_id new_id 1 DISC old_id").
```

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

    1 byte:  potential_visibility (unsigned integer)
         0:  Make the clone potentially visible.
         1:  Make the clone invisible.
         This field can be omitted if the abstract_flag and
         location fields are also omitted.  When it is
         omitted, the object retains the potential
         visibility of the parent object.

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

         This field can be omitted if the location fields
         are also omitted.  When it is omitted, the object
         retains the abstract_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).
```

Negative values are permitted for the X and Y position. The positive directions are downward and rightward from the upper left corner of the display.

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 image. 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 "abstract_flag" byte must be zero when the "clone_type" byte is nonzero.

If an object has partial clones, and the object data in 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 data 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 PAST Paste an image into an 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 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

4 bytes: target_x (signed integer)

4 bytes: target_y (signed integer)

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)
     0: same as source image
     2: flipped left-right, then up-down
     4: flipped left-right
     6: flipped up-down
```

```
1 byte:  offset_delta_type (unsigned integer)
     0:  offsets are measured from {0,0}
     1:  offsets are measured from
         {target_x, target_y}
4 bytes: xoffset or delta xoffset (signed integer)
4 bytes: yoffset or delta yoffset (signed integer)

1 byte:  boundary_delta_type (unsigned integer)
     0:  boundaries are measured from {0,0}
     1:  boundaries are measured from
         {target_x, target_y}
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 (abstract_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". 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} position 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. If the source image has been flipped or rotated, "xoffset" and "yoffset" give the location of its new upper left hand corner.

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, 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).

4.2.8 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 object 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 and the identifier can
         be reused by a DEFI chunk.
etc.
```

If the "DISC" chunk is empty, all objects except those preceding the "SAVE" chunk 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.

The appearance of an "object_id" in the "discard_id" list, when no such object has been stored, or when the object has already been discarded, should not be treated as an error.

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.

4.2.9 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 upper left corner of the display, 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. 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 4.2.10, below.

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

4.2.10 `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 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 all of the following are true:

```
0        <= x < max_frame_width   (from the MHDR chunk)
0        <= y < max_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 "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 "CLIP" data, is also discarded.

4.3 Ancillary MNG chunks

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

4.3.1 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. If two "eXPI" chunks use the same name, it's 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 PND 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.

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

4.3.2 `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 three 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 PND datastreams, the "initial" frames could be preceded by a "`fPRI`" with "`priority=128`" and followed by one with "`priority=0`", and the best representative 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 could display everything.

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 can 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.

4.3.3 `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. 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, you can specify that the datastream is written in accordance with a certain draft version of MNG, you can use "nEEDdraft nn"].

4.3.4 tERm Termination

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

```
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).
```

4.3.5 `pHYs` Physical pixel size

The MNG "pHYs" chunk is identical in syntax to the PNG "pHYs" chunk, but, when "framing_mode != 1" from the "FRAM" chunk, it applies to complete MNG frames and not to the individual images within them. When "framing_mode=1", 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.

[Would this be less confusing if we used a different name for this top-level chunk, e.g. pSIz? Is there a way of eliminating the ugly interdependence on framing_mode while maintaining the functionality?]

4.3.6 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 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 "SAVE" 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 the appearance of a subsequent empty chunk with the same chunk name. Such empty chunks are not legal PNG chunks and must only appear in the MNG top level.

- "cHRM"
- "gAMA"
- "sRGB"

A MNG editor that writes PNG datastreams is expected to include the top-level "cHRM", "gAMA", and "sRGB" chunks in the generated PNG datastreams.

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

- "sPLT" (this chunk also takes precedence over the "PLTE" chunk in a subsequent PNG datastream when the "PLTE" and "hIST" chunks chunk is 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 PND format

A PND datastream describes a single image, by giving the changes from a previous PNG (Portable Network Graphics) or another PND image. The PND format might be extended at some later date to include a PNP (Portable Network Photo) datastream.

No provision is made in this specification for storing a PND datastream as a standalone file. A PND datastream will normally be found as a component of a MNG datastream. Applications that need to store a PND 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 PND 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 PND 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 PND decoder the unmodified pixel data along with the values for the "gAMA", "cHRM", and any other recognized chunks from the parent object datastream.

A PND datastream consists of a "DHDR" and "DEND" enclosing other optional chunks (if there are no other chunks, the decoder simply copies the parent image, and displays it if its "potential_visibility=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.

5.1 PND critical chunks

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

5.1.1 DHDR PND datastream header

The "DHDR" chunk introduces a PND datastream. Subsequent chunks, through the next "DEND" chunk, are interpreted according to the PND format.

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

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

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.

1  byte: delta_type.

      0: Entire image replacement.

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

      2: Block alpha addition, by samples, modulo
         2^bit_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 "image" must identify an existing object, and the object must be a "concrete" object, i.e. it must have the property "abstract_flag=1".

The image type, whether given explicitly as 1 or implied by the presence of an "IHDR" or "IPNG" 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" 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 parent object must have been derived from a PNG datastream or from a sequence of PND datastreams that depend upon a PNG datastream.

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_bit_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_bit_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 "bit_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 "bit_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 "bit_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" to appear.

5.1.2 DEND End of PND datastream

End of PND datastream. A "DEND" chunk must be present for each "DHDR" chunk in a MNG datastream.

The "DEND" chunk is empty.

5.1.3 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 bit_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.

RGB -> RBGA (color_type 2 -> 6)

Don't change the RGB values. Handle the 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 alpha values 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.

If the bit 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 the "`color_type`" has been promoted from indexed, the original bit depth is always considered to be 8. See the PNG specification [LINK] for further information on these filling methods.

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.

5.1.4 `IHDR/IPNG`, PNG chunks, `IEND`

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

`IHDR` If the "`IHDR`" chunk is present, its "`width`", "`height`", "`bit_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.

`IPNG` Incomplete PNG 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 PND 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 either a "`PLTE`" 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 nor an "IDAT" chunk, then either the "IPNG" or "IHDR" must be present to introduce the PNG datastream.

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.

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 PND 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).

IPLT Incomplete palette If it is desired to only overwrite palette entries without truncating the resulting palette, the IPLT chunk can be used. This might be useful for palette-animation applications. Its format is identical to that of the "PLTE" chunk. If it has a length "n", it overwrites the first "n/3" entries, but does not affect the palette length or the contents and validity of the remaining entries, nor does it affect the "tRNS" chunk data.

gAMA, cHRM, sRGB Color encoding chunks A "gAMA", "cHRM", "sRGB" or similar chunk existing in the parent object would not affect the pixel data inherited by this PND 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.

oFFs and pHYs MNG viewers must ignore "oFFs" and "pHYs" chunks that appear inside a PNG stream, except when "framing_mode=1" from the MNG "FRAM" chunk. datastream begins with a PNG signature instead of a MNG signature. MNG editors are expected to treat them as unknown chunks that will be handled as described in Paragraph 5.1.7, below.

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 5.1.7, below.

IEND The "IEND" chunk can be omitted, if it would be the last chunk in the PND datastream before the "DEND", or when no PNG chunks are present.

5.1.5 DROP Drop chunks

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

```
Chunk name
etc.
```

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

5.1.6 DBYK Drop chunks by keyword

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

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

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

The chunk name must be the name of a chunk that 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.

Use caution when printing or displaying keywords (Refer to Security considerations, Chapter 12, below).

5.1.7 ORDR Ordering restrictions

ORDR informs the applier of the PND of the ordering restrictions for chunks.

```
Chunk name
Order type
 0: anywhere
 1: after IDAT
 2: before IDAT
 3: before IDAT, but not before PLTE
 4: before IDAT, but not after PLTE
```

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 PND, 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 PND, 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 PND, it is a fatal error unless the applier knows how to handle it. The specification of the critical chunk may include provisions for this scenario.

6 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"
- "IEND"
- "DHDR"
- "DEND"
- "BASI"
- "PAST"
- "SHOW"

The copy-safe status of an unknown chunk is determined from the chunk name, just as in PNG [LINK]. 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–DEND", 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", and "DHDR-DEND" sequences is governed by the PNG specification, and any changes inside such sequences have no effect on top-level chunks.

[This one is ugly.] The copy-safe status of chunks inside a "DHDR-DEND" sequence depends on the copy-safe status of its parent object.

7 Recommendations for encoders

Say something useful here.

7.1 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" and "APLY" chunks inside the loops. Otherwise, decoders will be forced to repeatedly decode them. See Examples 2, 8, 11, and 12 Chapter 13, below.

8 Recommendations for decoders

Say something useful here.

8.1 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 "loop_level" should be terminated, and processing can resume after the next "SEEK" chunk.

8.2 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)
s = At / Ac
t = (1 - At) Ab / Ac
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.

8.3 Retaining object data

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

The following information must be retained, for each nonzero "`image`" 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 file or its equivalent.

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 by using "`image_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 PND.

8.4 Decoder handling of fatal errors

When a fatal error is encountered, such as an unknown critical MNG chunk, MNG viewers should attempt to recover gracefully by abandoning processing of the frame 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.

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

8.5 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.

8.6 Decoder handling of palettes

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

If "PLTE" is present in a PND datastream, the new palette is used in displaying the image defined by the PND; 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 PND without a "PLTE" chunk, that has been declared by the "DHDR" "image" 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 reduced global palette, to help decoders build an appropriate palette when necessary.

9 Recommendations for editors

Say something useful here.

10 Miscellaneous Topics

10.1 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.

10.2 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`".

10.3 Uniform Resource Locator (URL)

[Explanation of the use of SEEK 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).

11 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/>.

[PNG-EXT]

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

[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>

12 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 inter-frame 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.

13 Appendix: Examples

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

13.1 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
```

13.2 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   # When done, repeat from SAVE indefinitely

FRAM 1 "begin" 0 2 0 0 0 30 # set 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
```

13.3 Example 3: Simple slideshow

A simple slideshow:

```
\212 M N G \r \n ^z \n  # MNG signature
MHDR 720 468   # Width and height
     1         # 1 tick per second
FRAM 1 "" 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
```

13.4 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 DHDR-DEND streams only contain deltas due to the text and other information that is unique to each slide.

```
\212 M N G \r \n ^z \n  # MNG signature
MHDR 720 468   # Width and height
     1         # 1 tick per second
DEFI 1 1 1     # define image 1, invisible, concrete
IHDR ...  IDAT ...  IEND
FRAM  1 "" 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"
CLON 1 2 DHDR 2 ...  IDAT ...  IEND SHOW 2
SEEK "Outline"
CLON 1 2 DHDR 2 ...  IDAT ...  IEND SHOW 2
SEEK "Our Vision"
CLON 1 2 DHDR 2 ...  IDAT ...  IEND SHOW 2
SEEK "Our Mission"
CLON 1 2 DHDR 2 ...  IDAT ...  IEND SHOW 2
SEEK "Downsizing Plans"
CLON 1 2 DHDR 2 ...  IDAT ...  IEND SHOW 2
MEND
```

13.5 Example 5: Simple movie

This movie is still fairly simple, but it capitalizes on frame-to-frame similarities by use of PND 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
DEND           # 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
DEND           # 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
DEND       # this time.

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

13.6 Example 6: Single composite frame

Here's 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 # Width, height
     1        # 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
DEND                 # 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 PND datastream
tRNS ...             # It's 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
```

13.7 Example 7: Movie with sprites

Here's another movie, illustrating the use of PND datastreams as sprites

```
\212 M N G \r \n ^z \n  # MNG signature
MHDR 512 512     # Start of MNG datastream
     30          # ticklength
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's 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's 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
DEND             # 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 100" #  Another frame
etc.
FRAM
etc.             # More frames
MEND             # End of MNG datastream
```

13.8 Example 8: Movie with an animated sprite

Here's another movie, illustrating 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 will run back and forth across the background ten times.

```
\212 M N G \r \n ^z \n  # MNG signature
MHDR 512 512     # Start of MNG datastream
     30          # ticklength
FRAM 2 "frame 1" 0  2 0 0 0  3 # First frame
                 # sets frame_duration=3 ticks
DEFI 1           # Background (abstract at {0,0}
IHDR 512 512 ... # It's a full-display PNG image
etc              # Chunks according to PNG spec
IEND

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 62 ...  IDAT ...  IEND # (y1=y0+64)
DEFI 12 1 0 x0 y1 # view 2 of animated part
IHDR 64 62 ...  IDAT ...  IEND
DEFI 13 1 0 x0 y1 # view 3 of animated part
IHDR 64 62 ...  IDAT ...  IEND

LOOP 0 0 10
LOOP 1 0 150
FRAM
MOVE 10 13 1 dx dy # Move animated icon {dx, dy}
SHOW  1            # Show background
SHOW 10            # Show 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
MOVE 10 13 1 -dx -dy # Move animated icon {-dx, -dy}
SHOW  1              # Show background
SHOW 10              # Show static part
SHOW 11 13 6         # Select the next view of the
ENDL 1               # animated part and show it
ENDL 0
FRAM
MEND
```

13.9 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 "APLY", "PROM", and "fPRI" chunks.

```
\212 M N G \r \n ^z \n # MNG signature
MHDR 64 64         # Width, height
     30            # Tick length
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    # Make a working abstract copy of the image
            # that will be modified during the low-priority
            # part of the datastream.  It's a full clone.
DHDR 2 1 5  # no change to pixel data
PROM 6 8 0  # Promote it to RGBA
DEND
SHOW 2 2 3  # Make it visible but don't show it now.

DHDR 3 1 2 64 64 0 0 # Object 3 is an image delta
            # Delta-type is block alpha addition.
IDAT ...    # Grayscale IDAT with 0 in transparent pixels
DEND        # and 16 in opaque pixels
SHOW 2

LOOP 0 0 15
APLY 2 3    # Apply delta to image 2 and display it.
ENDL 0      # Nontransparent pixel alpha=15, 31, ... 240

DISC 2 3    # Discard the working copy and the image delta
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
```

13.10 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 PND 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 PND to PND. 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     # Width, height
     1           # Tick length (unused in this example)

tEXtTitle\0Weather modeling results
tEXtComment\0The pcAL, xsCL, ysCL, zsCA, and tsCL chunks
 in this file are written according to the PNG Sci-vis
 chunks specification version 19960921 and pCAL
 specification version 19961230 available at
 ftp://swrinde.nde.swri.edu/pub/png-group/documents/
xsCL x\0 [sig] kilometers\0 0\0 150
ysCL x\0 [sig] kilometers\0 0\0 150
zsCA x\0 [sig] Height (kilometers)\0 0\0 15
tsCL x\0 [sig] Time (hours)\0 0\0 24
pcAL x\0 [sig] 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's 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
DEND         # IEND is also omitted
etc.         # Repeat DHDR through DEND 148 more times
SEEK
FRAM         # End of first block
etc.         # Repeat FRAM through SEEK 19 more times
SEEK
MEND         # End of MNG
```

13.11 Example 11: Tiling

Here's 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 ...  # of the tile
PLTE ...
IDAT ...      # Nothing will be displayed because it's
IEND ...      # outside the 1024 by 768 composite frame
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's a better approach, which creates a reusable tiled image by pasting the small image into the corner of a large blank image, and then repeatedly pasting the growing image back into itself.

```
\212 M N G \r \n ^z \n  # MNG signature
MHDR 1024 768 1 # Start of MNG datastream
DEFI 1 1 0 0 -64 # Set up an offscreen "abstract" copy
IHDR 128 64 ...  # of the tile
PLTE ...
IDAT ...      # Nothing will be displayed because it's
IEND ...      # outside the 1024 by 768 composite frame
DEFI 2        # The abstract image to be tiled
BASI 1024 768 8 2 0 0 0 # initially all pixels are zero
IEND
PAST 2 0 0 0  # destination and target location
  # src mod orient  offset        clipping
     1   0    0     0   0    0   0 0  128 0  64
     2   0    0     0 128    0   0 0  256 0  64
     2   0    0     0 256    0   0 0  512 0  64
     2   0    0     0 512    0   0 0 1024 0  64
     2   0    0     0   0   64   0 0 1024 0 128
     2   0    0     0   0  128   0 0 1024 0 258
     2   0    0     0   0  256   0 0 1024 0 512
     2   0    0     0   0  512   0 0 1024 0 768
  #  end of PAST chunk data
MEND
```

13.12 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      # Width and height on screen
     30          # Tick length
FRAM 2 "" 0  2 0 0 0  1 # frame_duration = 1 tick
DEFI 1 1 0 0 256  # Define image 1 but don't display now
                  # Initially it's 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
FRAM
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
SHOW 1 2     # Show the two images
ENDL 0

MEND
```

13.13 Example 13: Converting a GIF animation to MNG

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

```
begin
    write "MHDR" chunk
    write "BACK" chunk
    saved_images := 0
    frame_duration := 0
    first_frame := TRUE
    if(loops>1) "write tERm 2" chunk
    for subimage in gif89a file do
       if(frame_duration != gif_duration) then
         frame_duration := gif_duration
         write "FRAM 2 x 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" chunk
    write "MEND" chunk
end
```

Where "<image>" represents a PNG or PND datastream containing the 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.

14 Credits

Contributors' names are presented in alphabetical order:

- 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
- 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, glennrp@arl.mil or 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
U.S. Army Research Laboratory
ATTN: AMSRL-WM-TD
Aberdeen Proving Ground, MD 21005-5066

Phone: (410) 278-6554

EMail: glennrp@arl.mil or randeg@alumni.rpi.edu

End of MNG Specification. Expires 25 Jul 1997