MNG (Multiple-image Network Graphics) Format Version 0.97

For list of authors, see Credits (Chapter 18).

Status of this Memo

This document is a specification by the PNG development group.

It has been approved by a vote of the group, and future technical changes will require formal approval by a vote of the group. It is the intent of the group to maintain backward compatibility if possible. We will, however, correct any technical deficiencies discovered in the course of developing "beta" implementations.

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

- mpng-list@ccrc.wustl.edu
- png-group@w3.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

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

Changes from sixty-ninth MNG draft (mng-0.96-19990717)

- Revised wording about frame count and layer count in MNG-VLC datastreams.
- Clarified that the BACK chunk components are always written as though for an RGBA PNG with 16-bit sample depth.
- Added a statement that MNG editors that create a series of PNG or JNG files are expected to execute loops, including the TERM-created loop, only "iteration_min" times.
- Clarified that show_mode 6 and 7 causes the cycle to jump back to the beginning when the end of the range is reached. An example of how to create forward and back-and-forth cycles with SHOW has been inserted.
- Fixed some typos.

Abstract

This document defines the MNG (Multiple-image Network Graphics) format. It also defines the MNG-LC (Low Complexity), MNG-VLC (Very Low Complexity), and JNG (JPEG Network Graphics) formats, which are proper subsets of MNG.

MNG is a multiple-image member of the PNG (Portable Network Graphics) format family that can contain animations (slide shows) or complex still frames, comprised of multiple PNG or JNG single-image datastreams.

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

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

MNG (but not MNG-LC or MNG-VLC) can also store images in a highly compressible "Delta-PNG" format, defined herein.

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

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

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

Table of Contents

1 Introduction **7**

2 Terminology **10**

3 Objects **13**
3.1 Embedded objects . . . 13
3.2 Object attributes . . . 14
3.3 Object buffers . . . 15

4 MNG Chunks **17**
4.1 Critical MNG control chunks . . . 17
4.1.1 MHDR MNG datastream header . . . 17
4.1.2 MEND End of MNG datastream . . . 20
4.1.3 LOOP, ENDL Define a loop . . . 20
4.2 Critical MNG image defining chunks . . . 23
4.2.1 DEFI Define an object . . . 23
4.2.2 PLTE and tRNS Global palette . . . 25
4.2.3 IHDR, PNG chunks, IEND . . . 25
4.2.4 JHDR, JNG chunks, IEND . . . 26
4.2.5 BASI, PNG chunks, IEND . . . 27
4.2.6 CLON Clone an object . . . 28
4.2.7 DHDR, Delta-PNG chunks, IEND . . . 30
4.2.8 PAST Paste an image into another . . . 31
4.2.9 DISC Discard objects . . . 33
4.3 Critical MNG image displaying chunks . . . 34
4.3.1 BACK Background . . . 34
4.3.2 FRAM Frame definitions . . . 36
4.3.3 MOVE New image location . . . 42
4.3.4 CLIP Object clipping boundaries . . . 43
4.3.5 SHOW Show images . . . 44
4.3.6 TERM Termination action . . . 47
4.4 SAVE and SEEK chunks . . . 48
4.4.1 SAVE Save information . . . 48
4.4.2 SEEK Seek point . . . 51
4.5 Ancillary MNG chunks . . . 52
4.5.1 eXPI Export image . . . 52
4.5.2 fPRI Frame priority . . . 53
4.5.3 nEED Resources needed . . . 54
4.5.4 pHYg Physical pixel size (global) . . . 54
4.6 Ancillary PNG chunks . . . 55

5 The JPEG Network Graphics (JNG) Format **56**
5.1 Critical JNG chunks . . . 56
5.1.1 JHDR JNG header . . . 57
5.1.2 JDAT JNG image data . . . 58
5.1.3 IDAT JNG alpha data . . . 61
5.1.4 IEND End of JNG datastream . . . 61
5.1.5 JSEP 8-bit/12-bit image separator . . . 61
5.2 Ancillary JNG chunks . . . 61

6 The Delta-PNG Format **62**
6.1 Delta-PNG critical chunks . . . 63
6.1.1 DHDR Delta-PNG datastream header . . . 63
6.1.2 IEND End of Delta-PNG datastream . . . 67
6.1.3 PROM Promotion of parent object . . . 67
6.1.4 IHDR PNG image header . . . 69
6.1.5 IPNG Incomplete PNG . . . 70
6.1.6 PLTE and tRNS . . . 70
6.1.7 PPLT Partial palette . . . 70
6.1.8 JHDR JNG image header . . . 71
6.1.9 IJNG Incomplete JNG . . . 72
6.1.10 DROP Drop chunks . . . 72
6.1.11 DBYK Drop chunks by keyword . . . 73
6.1.12 ORDR Ordering restrictions . . . 73
6.2 Ancillary Delta-PNG chunks . . . 74
6.2.1 gAMA, cHRM, iCCP, sRGB Color space chunks . . . 74
6.2.2 oFFs and pHYs . . . 74
6.2.3 Other ancillary PNG chunks . . . 74
6.3 Chunk ordering requirements . . . 74

7 Chunk Copying Rules **75**

8 Minimum Requirements for MNG-Compliant Viewers **76**
8.1 Required MNG chunk support . . . 77
8.2 Required PNG chunk support . . . 78
8.3 Required JNG chunk support . . . 79
8.4 Required Delta-PNG chunk support . . . 79

9 Recommendations for Encoders **80**
9.1 Use a common color space . . . 80
9.2 Use the right framing mode . . . 80
9.3 Immediate frame sync point . . . 80
9.4 Embedded images in LOOPs . . . 81
9.5 Including optional index in SAVE chunk . . . 81
9.6 Interleaving JDAT and IDAT chunks . . . 81

10 Recommendations for Decoders **81**
10.1 ENDL without matching LOOP . . . 81
10.2 Note on compositing . . . 81
10.3 Retaining object data . . . 82
10.4 Decoder handling of fatal errors . . . 83
10.5 Decoder handling of interlaced images . . . 83
10.6 Decoder handling of palettes . . . 84
10.7 Behavior of single-frame viewers . . . 84
10.8 Clipping . . . 84

11 Recommendations for Editors **85**
11.1 Editing datastreams with optional index . . . 85
11.2 Handling LOOP and TERM chunks . . . 85

12 Miscellaneous Topics **86**
12.1 File name extension . . . 86
12.2 Internet media type . . . 86
12.3 Uniform Resource Identifier (URI) . . . 86

13 Rationale **88**

14 Revision History **89**
14.1 Version 0.97 . . . 89
14.2 Version 0.96 . . . 89
14.3 Version 0.95 . . . 90

15 References **90**

16 Security Considerations **91**

17 Appendix: Examples **92**
17.1 Example 1: A single image . . . 92
17.2 Example 2: A very simple movie . . . 92
17.3 Example 3: A simple slideshow . . . 94
17.4 Example 4: A more storage-efficient slideshow . . . 94
17.5 Example 5: A simple movie . . . 95
17.6 Example 6: A single composite frame . . . 96
17.7 Example 7: A movie with sprites . . . 97
17.8 Example 8: A movie with an animated sprite . . . 98
17.9 Example 9: “Fading in” a transparent image . . . 99
17.10 Example 10: Storing three-dimensional images . . . 100
17.11 Example 11: Tiling . . . 101
17.12 Example 12: Scrolling . . . 102
17.13 Example 13: Cycling animations . . . 103

17.14 Example 14: Converting a GIF animation 104
17.15 Example 15: Converting a simple GIF animation 105
17.16 Example 16: Counting layers and frames 106
17.17 Example 17: Storing an icon library 107

18 Credits **108**

1 Introduction

This specification defines the format of a MNG (Multiple-image Network Graphics) format. It also defines low-complexity and very-low-complexity versions (MNG-LC and MNG-VLC), which are proper subsets of MNG.

Note: This specification depends on the PNG (Portable Network Graphics) [PNG] and the JPEG (Joint Photographic Experts Group) specifications. The PNG specification is available at the PNG web site,

`http://www.cdrom.com/pub/png/`

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

The embedded images can be PNG, JNG, or Delta-PNG datastreams. MNG-LC and MNG-VLC datastreams do not contain JNG datastreams, but MNG-LC and MNG-VLC applications can be enhanced to recognize and process those as well.

A typical MNG datastream consists of:

- The 8-byte MNG signature.
- The MHDR chunk.
- Frame definitions. A frame is one or more subframes, the last of which has a nonzero interframe delay, composited against whatever was already on the display,
- Subframe definitions. A subframe contains one or more layers, together with clipping information and an interframe delay.
- Layer definitions. A layer is:
  - An embedded image.
  - An image that is generated in response to a SHOW or CLON directive.
  - The background.
- LOOP-ENDL chunks.
- 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.

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

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

MNG is pronounced "Ming."

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

The first eight bytes of a MNG datastream are

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

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

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

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

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

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

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

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

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

- It is unencumbered by patents.
- It is streamable.
- It has excellent, lossless compression.
- It stores up to four channels (red, green, blue, alpha), with up to 16 bits per channel.
- It provides transparency and an alpha channel.

- It provides platform-independent rendition of colors by inclusion of gamma and chromaticity information.
- It provides early detection of common file transmission errors and robust detection of file corruption.
- Single-image GIF files can be losslessly converted to PNG.
- It is complementary to JPEG and does not attempt to replace JPEG for lossy storage of images (however, MNG can accommodate JPEG-encoded images that are encoded in the PNG-like JNG format that is defined herein).

In addition:

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

2 Terminology

See also the glossary in the PNG specification.

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

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

child, or child image
: An image produced by applying an image delta to a parent object.

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

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

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

frame
: A layout of zero or more layers, with zero interframe delay time between them, followed by a specified nonzero delay time or by the MEND chunk, that is to be displayed as a still picture or as part of an animation. An animation would ideally appear to a perfect observer (with an inhumanly fast visual system) as a sequence of still pictures. A group of MNG images, with zero interframe delay time between them, and also followed by a zero interframe delay time, makes up a "subframe" (see below) instead of a "frame". The final subframe in a segment completes a frame, regardless of its interframe delay. In MNG-VLC datastreams, each layer (except for the background layer) is a frame, unless the framing rate (from the MHDR ticks-per-second field) is zero. When the framing rate is zero, the entire datastream describes a single frame.

  When the layers of a frame do not cover the entire area defined by the width and height fields from the MHDR chunk, the layers are composited over the previous frame to obtain the new frame.

frame origin
: The upper left corner of the output device (frame buffer, screen, window, page, etc.) where the

pixels are to be displayed. This is the {0,0} position for the purpose of defining frame clipping boundaries, image locations, and image clipping boundaries. Note that in a windowing system, the frame origin might be moved offscreen, but the locations in DEFI, MOVE, and CLIP chunks would still be measured from this offscreen origin. In MNG-VLC, all images must be placed with the image's upper left corner at the frame origin.

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

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

image N or object N
Shorthand for "the object with the object attribute set pointed to by `'object_id=N'`". In MNG-LC and MNG-VLC, only image 0 is permitted.

interframe delay
The amount of time a frame should be visible when an animation is played. In reality, it takes a nonzero amount of time to display a frame. No matter which moment is picked as the "start" of the frame, the interframe delay measures the time to the "start" of the next frame. There is no interframe delay prior to the implicit background layer at the beginning of the animation nor after the final frame.

layer
A layer can be

- A visible embedded image, located with respect to the frame boundaries and clipped with respect to the subframe clipping boundaries and the image's own clipping boundaries.
- A stored image that is displayed in response to a SHOW or CLON chunk directive, located and clipped.
- The background that is displayed before the first image in the entire datastream is displayed. Its contents can be defined by the application or by the BACK chunk.
- The background image, clipped, located, and displayed against a solid rectangle filled with the background color and clipped to the subframe boundaries at the beginning of a subframe, when the framing mode is 3 or 4.

Note that a layer can be completely empty if the image is entirely outside the clipping boundaries.

A layer can be thought of as a transparent rectangle with the same dimensions as the frame, with an image composited into it, or it can be thought of as a rectangle having the same dimensions (possibly zero) and location as those of the object after it has been located and clipped.

An embedded visible PNG or JNG datastream generates a single layer, even though it might be interlaced or progressive. If the background consists of both a background color and a background image, these are combined into a single layer.

MNG-LC
A low-complexity version of MNG in which only image 0 is permitted, loops can all be run once, JNG and Delta-PNG are not present, and certain MNG chunks are not present. Bit 0 of the MHDR simplicity profile must be 1, indicating that the profile is valid, and all other bits of the profile except possibly for bits 2 and 3 must be 0, indicating that "complex MNG features", JNG, and Delta-PNG are not present. Bits 2 and 3 can be 0 or 1, depending on whether "simple MNG features" and transparency are absent or present. MNG-LC is a proper subset of MNG.

MNG-VLC
A very-low-complexity version of MNG in which only image 0 is permitted, loops can all be run once, JNG and Delta-PNG are not present, and certain MNG chunks are not present. Bit 0 of the MHDR simplicity profile must be 1, indicating that the profile is valid, and all other bits of the profile except possibly for bit 3 must be 0, indicating that "simple MNG features", "complex MNG features", JNG, and Delta-PNG are not present. Bit 3 can be 0 or 1, to indicate whether transparency is absent or present. MNG-VLC is a proper subset of MNG.

object, object_id
An image or a nonviewable basis object. The `object_id` is an unsigned sixteen-bit number that serves as the identifier of a set of object attributes. In MNG-LC and MNG-VLC only object 0 is permitted.

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

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

parent, parent object, or parent image
An object to which a delta is applied.

pixel sample depth and alpha sample depth
The sample depth used for decoding IDAT data in Delta-PNG datastreams. They are not necessarily the same as the sample depth of the object, which is called "sample depth" or "object sample depth" in this document.

potentially visible image
A potentially visible image is

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

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

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

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

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

subframe
A group of one or more layers that are to be displayed as a part of a still frame or as part of one frame of an animation. Subframes are defined in MNG for convenience in applying frame parameters to a subset of the layers making up a complete frame.

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

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

3 Objects

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

An "image" is a viewable object.

3.1 Embedded objects

An embedded object is:

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

3.2 Object attributes

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

Object exists.
An object comes into existence when

- a DEFI chunk creates it.
- a CLON chunk creates it.
- (Object 0 always exists).

Object does not exist.
An object ceases to exist when it does not have the "frozen" attribute and

- it is the subject of a DISC chunk.
- an empty DISC chunk appears.
- a SEEK chunk appears.
- the MEND chunk appears (or the IEND chunk appears in a simple PNG or JNG file).
- a new embedded object with the same `object_id` replaces it without an intervening DEFI chunk. In this case, the new object inherits the object attribute set from the previous object with the same `object_id`.

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

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

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

Object is viewable.

An object is viewable if it has a viewable object buffer. It is nonviewable if it has a nonviewable object buffer or if its object buffer has not yet been created or has been destroyed. Any attempt to display a nonviewable object must be ignored and not treated as an error.

A nonviewable object becomes viewable immediately when the decoder receives a viewable object buffer or when an image delta makes it viewable, and if the object is potentially visible it can be displayed "on-the-fly" while the object buffer is being decoded or updated. Note that object 0 is only viewable while its embedded image is being decoded and displayed on-the-fly, after which it becomes nonviewable again because no object buffer is ever created for object 0.

Location.

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

Clipping boundaries.

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

3.3 Object buffers

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

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

Object 0 conceptually never has an object buffer. Decoding applications can create one for their own convenience, but such an object buffer must never be made available to the rest of the MNG datastream or be considered viewable after it has been processed.

Object buffer is viewable.

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

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

Format of data in the object buffer.

The data format can be:

- A concrete PNG or JNG object. A concrete object must be stored by the decoder in a form that retains the complete object description, sufficient to regenerate the original object description or its equivalent without loss. Its pixels have a publicly known representation, and it uses a publicly known color encoding.
  - In the case of a PNG image, the object also carries data from other known PNG chunks that are present. This means that the decoder must store sufficient information to make it possible to restore exactly the original decoded and unfiltered pixel samples as they existed prior to any gamma correction (but not the original compressed datastream or line-by-line filter selections and "zlib" compression flags), and data from the IHDR and PLTE chunks and any additional recognized PNG chunks such as gAMA, cHRM, and tRNS that the application plans to use. The sample depth, color type, filter method, compression method, and interlacing method of the image must be retained, and if the object has been modified by a Delta-PNG, the "pixel sample depth" and "alpha sample depth" must also be retained for use in decoding subsequent Delta-PNG datastreams.
  - In the case of a JNG image, the object also carries data from other known JNG chunks that are present. This means that the decoder must store sufficient information to make it possible to restore exactly the original JPEG datastream and decoded alpha channel as they existed in the original JNG file, and data from the JHDR chunk and any additional recognized JNG chunks such as gAMA and cHRM that the application plans to use. As with PNG objects, when the object has been modified by a Delta-PNG, the "alpha sample depth" must also be retained for use in decoding subsequent Delta-PNG datastreams.
  - A decoder that recreates PNG or JNG files from a series of PNG, JNG, and Delta-PNG datastreams will also have to store the contents of any unknown chunks that it finds, in case they turn out to be safe to copy (see DROP (Paragraph 6.1.10), DBYK (Paragraph 6.1.11), and ORDR (Paragraph 6.1.12), below).
- An abstract image. An abstract image can be stored by the decoder in any form that is convenient, such as an X Window System "pixmap", even though that form might not have sufficient resolution for exact, lossless conversion. In the case of a PNG image, the pixels could be stored after the gamma and chromaticity corrections have been made, and the sample depth could be the same as the display hardware, even though it is smaller than the original sample depth. Similarly, a JNG image could be stored in the same form, after the pixels have been decoded, converted to RGB form, and gamma-corrected. It is always safe, however, to store an abstract image as though it were concrete, if decoders do not wish to take advantage of the distinction between abstract and concrete objects.

Frozen or not frozen.

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

A reference count.

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

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

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

4 MNG Chunks

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

4.1 Critical MNG control chunks

This section describes critical MNG control chunks that MNG-compliant decoders must recognize and process ("processing" a chunk sometimes can consist of simply recognizing it and ignoring it; some chunks have been declared to be critical only to prevent them from being relocated by MNG editors).

4.1.1 MHDR MNG datastream header

The MHDR chunk is always first in all MNG datastreams except for those that consist of a single PNG or JNG datastream with a PNG or JNG signature.

The MHDR chunk contains exactly 28 bytes:

```
Frame width:  4 bytes (unsigned integer).
Frame height: 4 bytes (unsigned integer).
Ticks per
 second:      4 bytes (unsigned integer).
Nominal layer
 count:       4 bytes (unsigned integer).
Nominal frame
 count:       4 bytes (unsigned integer).
Nominal play
 time:        4 bytes (unsigned integer).
Simplicity
 profile:     4 bytes:(unsigned integer).
                bit 0:
                       0: Presence or absence of any
                          features is unspecified.  All
                          other bits of the simplicity
                          profile must be zero (i.e, all
                          other even numbers are invalid).
```

```
            1: Presence or absence of certain
               features is specified by the
               remaining bits of the simplicity
               profile.
            (must be 1 in MNG-LC and MNG-VLC
            datastreams)
         bit 1:
            0: Simple MNG features are absent.
            1: Simple MNG features are present.
            (must be 0 in MNG-VLC datastreams)
         bit 2:
            0: Complex MNG features are absent.
            1: Complex MNG features are present.
            (must be 0 in MNG-LC and MNG-VLC
            datastreams)
         bit 3:
            0: transparency is absent or can be
               ignored.
            1: transparency is present and is
               essential (critical) for proper
               display of the images.
         bit 4:
            0: JNG is absent.
            1: JNG is present.
            (must be 0 in MNG-LC and MNG-VLC
            datastreams)
         bit 5:
            0: Delta-PNG is absent.
            1: Delta-PNG is present.
            (must be 0 in MNG-LC and MNG-VLC
            datastreams)
         bits 6-15:
            Reserved for public expansion.  Must
            be zero in this version.
         bits 16-30:
            Available for private or experimental
            expansion.  Undefined in this version
            and can be ignored.
         bit 31:
            Must be zero.
```

Decoders can ignore the "informative" frame-count, layer-count, play-time, and simplicity-profile fields.

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

The `ticks_per_second` field gives the unit used by the FRAM chunk to specify interframe delay and timeout. In MNG-VLC datastreams, it gives the framing rate. It must be nonzero if the datastream contains

an animation. When the datastream contains exactly one frame, this field should be set to zero. When this field is zero, the length of a tick is infinite, and decoders will ignore any attempt to define interframe delay, timeout, or any other variable that depends on the length of a tick. If the frames are intended to be displayed one at a time under user control, such as a slide show or a multi-page FAX, the tick length can be set to any positive number and a FRAM chunk can be used to set an infinite interframe delay and a zero timeout. Unless the user intervenes, viewers will only display the first frame in the datastream.

When `ticks_per_second` is nonzero, and there is no other information available about interframe delay, viewers should display animations at the rate of one frame per tick.

If the frame-count field contains a zero, the frame count is unspecified. If it is nonzero, it contains the number of frames that would be displayed, ignoring the fPRI chunks and the TERM chunk. If the frame count is greater than $2^{31} - 1$, encoders should write $2^{31} - 1$, representing an infinite frame count. In MNG-VLC datastreams, the frame count is the same as the number of embedded images in the datastream (or one, the background layer, if there are no embedded images).

If the layer-count field contains a zero, the layer count is unspecified. If it is nonzero, it contains the number of layers in the datastream, ignoring the fPRI chunks and the TERM chunk. If the layer count is greater than $2^{31} - 1$, encoders should write $2^{31} - 1$, representing an infinite layer count. In MNG-VLC datastreams, the layer count is the number of embedded images, plus one (for the background layer).

If the nominal-play-time field contains a zero, the nominal play time is unspecified. Otherwise, it gives the play time, in ticks, when the file is displayed ignoring the fPRI chunks and the TERM chunk. Authors who write this field should choose a value of "ticks_per_second" that will allow the nominal play time to be expressed in a four-bit integer. If the nominal play time is greater than $2^{31} - 1$ ticks, encoders should write $2^{31} - 1$, representing an infinite nominal play time. In MNG-VLC datastreams, the nominal play time is the same as the frame count, except when the ticks_per_second is zero, in which case the nominal play time is also zero.

When bit 0 of the simplicity profile is zero, the simplicity (or complexity) of the MNG datastream is unspecified, and *all* bits of the simplicity profile must be zero. The simplicity profile must be nonzero in MNG-LC and MNG-VLC datastreams.

If the simplicity profile is nonzero, it can be regarded as a 32-bit profile, with bit 0 (the least significant bit) being a "profile-validity" flag, bit 1 being a "simple MNG" flag, bit 2 being a "complex MNG" flag, bit 3 being a "transparency" flag, bit 4 being a "JNG" flag, and bit 5 being a "Delta-PNG" flag. The upper 15 bits (except for the most significant bit, which must be zero) are available for private test or experimental versions, and the remaining bits are reserved for future MNG versions, and must be zero in this version. If a bit is zero, the corresponding feature is guaranteed to be absent, and if a bit is one, the corresponding feature may be present in the MNG datastream.

When bit 1 is zero ("simple" MNG features are absent), the datastream does not contain the DEFI, FRAM, or global PLTE and tRNS chunks.

When bit 2 is zero, the datastream does not contain any "complex MNG features". These are the BASI, CLON, DHDR/IEND, PAST, DISC, MOVE, CLIP, and SHOW chunks. If the DEFI chunk is present, it only defines object 0. If the BACK chunk is present, it does not define a background image. If the LOOP chunk is

present, it has `iteration_min=1`. A MNG with a "complex MNG feature" (simplicity profile that has bit 2 set to 1) may contain at least one of these chunks. A simple decoder can display "simple" MNGs without having to store any objects or dealing with the SAVE/SEEK mechanism, and it can ignore the LOOP and ENDL chunks and execute all loops exactly once.

"Transparency is absent or can be ignored" means that the MNG or PNG tRNS chunk is not present and no PNG or JNG image has an alpha channel, or that if they are present they are not essential (or critical) for displaying the images.

A MNG-LC (i.e., a "low-complexity MNG") datastream must have a simplicity profile with bit 0 equal to 1 and all other bits except possibly for bits 1 and 3 ("simple MNG" MNG features and transparency) equal to zero. If bit 4 (JNG) is 1, the datastream is a "MNG-LC with JNG" datastream. MNG-LC decoders are allowed to reject such datastreams unless they have been enhanced with JNG capability.

A MNG-VLC (i.e., a "very low-complexity MNG") datastream must have a simplicity profile with bit 0 equal to 1 and all other bits except possibly for bit 3 (transparency) equal to 0. If bit 4 (JNG) is 1, the datastream is a "MNG-VLC with JNG" datastream. MNG-VLC decoders are allowed to reject such datastreams unless they have been enhanced with JNG capability.

Encoders that write a nonzero simplicity profile should endeavor to be accurate, so that decoders that process it will not unnecessarily reject datastreams. For example, the simplicity profile 31 indicates that JNG, critical transparency, and at least one "complex" MNG feature are all present, but Delta-PNG is not. This example would not qualify as a MNG-LC or a MNG-VLC datastream because a "complex" MNG feature might be present. If the simplicity profile promises that certain features are absent, but they are actually present in the MNG datastream, the datastream is invalid.

4.1.2 MEND End of MNG datastream

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

4.1.3 LOOP, ENDL Define a loop

The LOOP chunk provides a "shorthand" notation that can be used to avoid having to repeat identical chunks in a MNG datastream. The LOOP chunk can be ignored by MNG-LC and MNG-VLC decoders, along with the ENDL chunk. Its contents are the first two or more of the following fields:

```
Nest level:       1 byte (unsigned integer).
Repeat count:     4 bytes (unsigned integer),
                    range [0..2^31-1].
Termination
  condition:      1 byte (unsigned integer).
                    Omit if termination_condition=0,
                    which means Deterministic, not cacheable.
                    1: Decoder discretion, not cacheable.
                    2: User discretion, not cacheable.
```

```
                    3: External signal, not cacheable.
                    4: Deterministic, cacheable.
                    5: Decoder discretion, cacheable.
                    6: User discretion, cacheable.
                    7: External signal, cacheable.
Iteration min:   4 bytes(unsigned integer).  Can be
                    omitted if termination_condition
                    is not 3 or 7.  If omitted, the default
                    value is 1.
Iteration max:   4 bytes (unsigned integer).  Can be
                    omitted if termination_condition
                    is not 3 or 7; must be omitted if
                    iteration_min is omitted; if
                    omitted, the default value is infinity.
Signal number:   4 bytes (unsigned integer).  Omit
                    if termination_condition is not 3 or 7.
Additional
  signal number: 4 bytes.  Omit if termination_condition
                    is not 3 or 7.
...etc...
```

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

Simple decoders can ignore all fields except for the `repeat_count`.

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

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

Deterministic
: This is the default behavior, when the termination condition field is omitted or has a value that is unrecognized by the decoder. The loop terminates after exactly the number of iterations specified by the iteration count. This value must be used if altering the number of repetitions would mess up the MNG datastream, but can be used merely to preserve the author's intent.

Decoder-discretion
: The number of iterations can be chosen by the decoder, and must not be less than `iteration_min` nor more than `iteration_max`. If the decoder has no reason to choose its own value, it should use the iteration count. One example of a decoder wishing to choose its own value is a real-time streaming decoder hovering at a loop while waiting for its input buffer to fill to a comfortable level.

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

External-signal
: The number of iterations must not be less than `iteration_min` nor more than `iteration_max`. The exact number can be determined by the arrival of a signal whose number matches one of the `signal_number` fields.

When the value of the termination condition flag is 4 or more, the loop is guaranteed to be "cacheable", which means that every iteration of the loop produces the same sequence of frames, and that all objects and object buffers are left in the same condition at the end of each iteration. Decoders can use this information to select a different strategy for handling the loop, such as storing the composited frames in a cache and replaying them rather than decoding them repeatedly.

The `iteration_min` and `iteration_max` can be omitted. If the condition is <deterministic> the values are not used. Otherwise, defaults of 1 and <infinity> are used. The `repeat_count`, `iteration_min`, and `iteration_max` can be any non-negative integers or <infinity>, but they must satisfy `iteration_min <= repeat_count <= iteration_max`. Infinity is represented by 0x7fffffff. If all of the loops in a MNG datastream have `iteration_min=1`, the datastream can qualify as a "simple" MNG for the purpose of setting bits 1 and 2 of the "simplicity profile" to zero, unless there are other reasons for setting them to one.

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

The `signal_number` can be omitted only if the termination condition is not <external-signal>. There can be any number of `signal_number` fields. `Signal_number = 0` is reserved to represent any input from a keyboard or pointing device, and 1–255 are reserved to represent the corresponding ASCII character, received from a keyboard or simulated keyboard, and values 256–1023 are reserved for future definition by this specification.

MNG editors that extract a series of PNG or JNG files from a MNG datastream are expected to execute the loop only `iteration_min` times, when the termination condition is not <deterministic>.

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

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

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

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

The SAVE and SEEK chunks are not permitted inside a LOOP-ENDL pair. To rerun an entire datastream that includes these chunks, use the TERM chunk instead. See below (Paragraph 4.3.6).

4.2 Critical MNG image defining chunks

4.2.1 DEFI Define an object

The DEFI chunk sets the default object attribute set (object_id, potential_visibility, concrete_flag, location, and clipping boundaries) for any subsequent images that are defined with IHDR-IEND, BASI-IEND, or JHDR-IEND datastreams.

Bit 1 of the MHDR simplicity profile can be used to promise that the DEFI chunk is not present.

The DEFI chunk contains 2, 3, 4, 12, or 28 bytes:

```
Object id:      2 bytes (unsigned integer) identifier
                  to be given to the objects that
                  follow the DEFI chunk.
                  This field must be zero in MNG-LC
                  files.

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

Concrete flag: 1 byte (unsigned integer)
                  0:  Make the objects "abstract" (image
                      cannot be the source for a
                      Delta-PNG)
                  1:  Make the objects "concrete" (object
                      can be the source for a Delta-PNG).
                  This field can be omitted if the
                  location and clipping boundary fields
                  are also omitted.
                  MNG-LC decoders can ignore this flag.

X_location:     4 bytes (signed integer).
                  The X_location and Y_location fields can
                  be omitted if the clipping boundaries
```

```
                    are also omitted.

    Y_location:     4 bytes (signed integer).

    Left_cb:        4 bytes (signed integer).  Left clipping
                      boundary.  The left_cb, right_cb,
                      top_cb, and bottom_cb fields can be
                      omitted as a group.

    Right_cb:       4 bytes (signed integer).

    Top_cb:         4 bytes (signed integer).

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

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

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

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

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

```
      Object_id = 0
   Do_not_show = 0
Concrete_flag = 0
   X_location = 0
   Y_location = 0
      Left_cb = 0
     Right_cb = frame_width
       Top_cb = 0
    Bottom_cb = frame_height
```

These default values are also used to fill any fields that were omitted from the DEFI chunk, when an object with the same `object_id` has not been previously defined or a DISC or SEEK chunk has caused it to be discarded.

The object attributes for all unfrozen objects, including object 0, become undefined when a SEEK chunk is encountered. Encoders must ensure that the first DEFI chunk for every object_id after any SEEK chunk does not omit any fields that were set to values other than these defaults in the previous DEFI chunk for that

object_id, because simple decoders are allowed to ignore the SEEK chunk. Decoders that jump or skip to SEEK chunks must assume these defaults for any omitted fields in the first DEFI chunk in a segment for each object_id.

If `object_id` is an identifier that already exists when a DEFI chunk appears, the object attribute set (except for the pointer to the object buffer) is immediately replaced, but the contents of the object buffer do not change until an IHDR, JHDR, or BASI chunk is encountered; then the contents of the object buffer previously associated with the identifier are replaced with the new embedded object data.

Note that if the object has partial clones, the object buffer of the clones is naturally affected because it is shared, but the object attribute sets of the clones are not affected.

4.2.2 PLTE and tRNS Global palette

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

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

If a PNG datastream is present that does not contain an empty PLTE chunk, neither the global PLTE nor the global tRNS data is inherited by that datastream.

If the global PLTE chunk is not present, each indexed-color PNG in the datastream must supply its own PLTE (and tRNS, if it has transparency) chunks.

4.2.3 IHDR, PNG chunks, IEND

A PNG (Portable Network Graphics) datastream.

See the PNG specification [PNG] and the Extensions to the PNG Specification document [PNG-EXT] for the format of the PNG chunks.

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

If a global PLTE chunk appears in the top-level MNG datastream, the PNG datastream can have an empty PLTE chunk, to direct that the global PLTE and tRNS data be used. If an empty PLTE chunk is not present, the data is not inherited. MNG applications that recreate PNG files must write the global PLTE chunk rather than the empty one in the output PNG file, along with the global tRNS data if it is present. The global tRNS data can be subsequently overridden by a tRNS chunk in the PNG datastream. It is an error for the PNG datastream to contain an empty PLTE chunk when the global PLTE chunk is not present or has been nullified.

If the PNG sRGB, gAMA, iCCP, or cHRM chunks appear in the top-level MNG datastream (and have not been nullified), but none of them appear in the PNG datastream, then the values are inherited from the top level as though the chunks had actually appeared in the PNG datastream. Data from such chunks appearing in the PNG datastream take preference over the inherited values. If any one of these chunks, or any future chunk

that defines the color space, appears in the PNG datastream, none of them is inherited. MNG applications that recreate PNG files must write these chunks, if they are inherited, in the output PNG files. If the sRGB chunk is present in a MNG datastream, it need not be accompanied in the MNG datastream by gAMA and cHRM chunks, despite the recommendation in the PNG specification. Any MNG viewer that processes the gAMA chunk must also recognize and process the sRGB chunk. It can treat it as if it were a gAMA chunk containing the value .45455 and it can ignore its "intent" field. If the sRGB chunk is present in the MNG datastream, editors that write PNG datastreams should add the gAMA and cHRM chunks to the PNG datastream, even though they are not present in the MNG datastream.

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

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.

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

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

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

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

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

4.2.4 JHDR, JNG chunks, IEND

A JNG (JPEG Network Graphics) datastream.

See the JNG specification below (Chapter 5) for the format of the JNG datastream.

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

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

MNG-LC and MNG-VLC applications are not expected to process JNG datastreams unless they have been enhanced with JNG capability.

4.2.5 BASI, PNG chunks, IEND

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

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

```
Width:              4 bytes (unsigned integer).
Height:             4 bytes (unsigned integer).
Sample_depth:       1 byte (unsigned integer).
Color_type:         1 byte (unsigned integer).
Compression_method: 1 byte (unsigned integer).
Filter_type:        1 byte (unsigned integer).
Interlace_type:     1 byte (unsigned integer).
Red_sample or
  gray_sample:      2 bytes (unsigned integer).
Green_sample:       2 bytes (unsigned integer).
Blue_sample:        2 bytes (unsigned integer).
Alpha_sample:       2 bytes (unsigned integer).
Viewable:           1 byte (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 $2^{\text{sample_depth}}$, whose first entry contains the given {`red_sample, green_sample, blue_sample`} triple, and whose remaining entries are filled with zeroes. If the `viewable` field is omitted, the object is not viewable.

The color and alpha samples are written as four sixteen-bit samples regardless of the `color_type` and `sample_depth`. When the `sample_depth` is less than sixteen, the least significant bits are used and the remaining bits must be zero filled. When `color_type=3`, the least significant byte of each color sample is used and the upper byte must be zero. When `color_type=0` or `color_type=4`, the green and blue samples must be present but must be ignored by decoders.

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

The permitted deviations from the PNG format are:

- Multiple instances of some chunks can be present even though the PNG specification allows only one. The subsequent Delta-PNG that uses this as the parent object must select only one, through the DBYK

or similar mechanism.

- The IDAT chunk can be omitted or there can be a single empty IDAT chunk. If so, all of the pixels are filled with the given color and alpha samples. A subsequent Delta-PNG that uses this as the parent object can supply the IDAT chunk or chunks.
- The PLTE chunk can be omitted or incomplete even when the `color_type=3`. If so, the subsequent Delta-PNG that uses this as the parent object can supply a complete PLTE chunk, if the single-entry palette that is generated is not desired.

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

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

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

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

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

4.2.6 CLON Clone an object

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

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

Clone_id:    2 bytes (nonzero unsigned integer).
               Identifier of the child object that is
               created.

Clone_type:  1 byte (unsigned integer).
```

```
                  0: Full clone of the object attributes
                     set and the object buffer.

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

                  2: Renumber object (this is equivalent to
                     "CLON source_id clone_id 1
                      DISC source_id").

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

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

Concrete
  flag:        1 byte (unsigned integer).
                  0:  Concrete_flag is the same as
                      that of the parent object.
                  1:  Make the clone "abstract"
                      (concrete_flag=0).

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

Loca
 delta_type: 1 byte (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.

    X_location or delta
      X_location:4 bytes (signed integer).

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

The `source_id` must be an existing object identifier, and the `clone_id` must not be an existing object identifier.

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

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

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

The `concrete_flag` byte must be zero or omitted when the `clone_type` byte is nonzero.

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

If `viewable=1` and `do_not_show=0`, the resulting image is displayed immediately.

4.2.7 DHDR, Delta-PNG chunks, IEND

A Delta-PNG datastream.

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

4.2.8 PAST Paste an image into another

Paste an image or images identified by source_id, or part of it, into an existing abstract image identified by destination_id.

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

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

Target
  delta_type:   1 byte (unsigned integer).
                  0:  Target_x and target_y are given
                      directly.
                  1:  Target_x and target_y are deltas
                      from their previous values in a
                      PAST chunk with the same
                      destination_id.
                  2:  Target_x and target_y are deltas
                      from their previous values in the
                      immediately preceding PAST chunk
                      regardless of its destination_id.

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

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

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

Composition
  mode:           1 byte (unsigned integer).
                     0:  Composite_over.
                     1:  Replace.
                     2:  Composite_under.

Orientation:      1 byte (unsigned integer).
                    The source image is flipped to
                    another orientation.

                    0:  Same as source image.
                    2:  Flipped left-right, then up-down.
                    4:  Flipped left-right.
                    6:  Flipped up-down.
                    8:  Tiled with source image, to fill
```

```
                          the clipping boundaries.  The
                          upper left corner of the assembly
                          is positioned according to the
                          prescribed offsets.

    Offset
     delta_type:      1 byte (unsigned integer).
                        0: Offsets are measured from the
                           {0,0} pixel in the destination
                           image.
                        1: Offsets are measured from the
                           {target_x,target_y} pixel in the
                           destination image.

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

    Boundary
      delta_type:      1 byte (unsigned integer).
                        0: Boundaries are measured from
                           the {0,0} pixel in the
                           destination image.
                        1: Boundaries are measured from the
                           {target_x,target_y} pixel in the
                           destination image.

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

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

The source images can be "abstract" or "concrete" and have any color_type and sample_depth. They must have the "viewable" property. The number of source images is ((chunk_length-11)/30).

The x_offset and y_offset distances and the clipping boundaries are measured, in pixels, positive rightward and downward from either the {0,0} pixel of the destination image or the {target_x, target_y} position in the destination image. They do not necessarily have to fall within the destination image. Only those pixels of the source image that fall within the destination image and also within the specified clipping

boundaries will be copied into the destination image. Note that the coordinate system for offsets and clipping is with respect to the upper lefthand corner of the destination image, which is not necessarily the same coordinate system used by the MOVE and CLIP chunks. If the source image has been flipped or rotated, `X_offset` and `Y_offset` give the location of its new upper left hand corner. When it is tiled, the offsets give the location of the upper left hand corner of the assembly of tiles.

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

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

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

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

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

4.2.9 DISC Discard objects

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

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

```
Discard_id: 2 bytes (nonzero unsigned integer).
...etc...
```

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

If the DISC chunk is not empty, the listed objects can be discarded.

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

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

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

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

4.3 Critical MNG image displaying chunks

4.3.1 BACK Background

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

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

Green_background: 2 bytes (unsigned integer).

Blue_background:  2 bytes (unsigned integer).

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

Background
  image_id:        2 bytes (unsigned nonzero integer).
                     Object_id of an image that is to be
                     displayed as the background of each
                     subframe.  If the image is not
                     full-frame, the remainder of the area
                     inside the subframe boundary is filled
                     with the background color.  This field
                     can be omitted; if so, no background
```

```
                    image is defined, and the background
                    image_id from a previous BACK chunk
                    becomes undefined.  This byte must be
                    omitted in MNG-LC and MNG-VLC
                    datastreams.

  Background
    tiling:       1 byte (unsigned integer).
                    0: Do not tile the background.
                    1: Tile the background with the
                       background image.  This field
                       can be omitted; if so, do not
                       tile the background.
                    This byte must be omitted in
                    MNG-LC and MNG-VLC datastreams.
```

Viewers are expected to composite the first layer in the MNG, MNG-LC, or MNG-VLC datastream against a background layer that fills the entire frame. The BACK chunk provides a background that the viewer can use for this purpose. If it is not "mandatory" the viewer can choose another background if it wishes. If the BACK chunk is not present, or if it has been clipped to less than full frame, the viewer must provide its own background for the first subframe. Each subframe after the first must be composited over the layers that precede it.

Viewers are expected, however, to composite every subframe in the MNG datastream against a fresh copy of the background, when the framing mode given in the FRAM chunk is 3 or 4. The images and the background are both clipped to the frame boundaries given in the FRAM chunk. Anything outside these boundaries is inherited from the previous subframe.

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

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

The three BACK components are always written as though for an RGBA PNG with 16-bit sample depth. For example, a mid-level gray background could be specified with (0x9999, 0x9999, 0x9999). The background color is interpreted in the current color space as defined by any top-level gAMA, cHRM, iCCP, sRGB chunks that have appeared prior to the BACK chunk in the MNG datastream. If no such chunks appear, the color space is unknown.

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

When the BACK chunk appears between FRAM chunks, it applies to the upcoming frame, not to the current one. When the framing_mode is 3, it takes effect immediately prior to the next IHDR, JHDR, BASI, DHDR, PAST, or SHOW chunk in the datastream.

For the purpose of counting layers, when the background consists of both a background color and a background image, these are considered to generate a single layer and there is no delay between displaying the background color and the background image.

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

The BACK chunk can be omitted. If a background is needed and the BACK chunk is omitted, then the viewer must supply its own background. For the purpose of counting layers, such a viewer-supplied background layer is counted the same as a background supplied by the BACK chunk.

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

4.3.2 FRAM Frame definitions

The FRAM chunk provides information that a decoder needs for generating layers, subframes, and frames. The FRAM parameters govern how the decoder is to behave when it encounters a FRAM chunk, an embedded image, or a SHOW chunk. The FRAM chunk also delimits subframes.

Bit 1 of the MHDR simplicity profile can be used to promise that the FRAM chunk is not present.

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

```
Framing mode:  1 byte.

                0:  Don't change framing mode.

                1:  No background layer is inserted.
                    The interframe delay occurs before
                    each image (except the very first
                    in the datastream) is displayed.

                2:  No background layer is inserted.
                    The interframe delay occurs before
                    each "FRAM" chunk.  No delay occurs
                    before or between the images.

                3:  The background layer is displayed
                    before each image is displayed.
```

```
                    The interframe delay occurs before
                    displaying the background, and no
                    delay occurs between the background
                    and the image.

                4:  The background layer is inserted
                    after each "FRAM" chunk.  The
                    interframe delay occurs before
                    the background is displayed, and no
                    delay occurs before or between the
                    images.

Subframe name: 0 or more bytes (Latin-1 Text).  Can be
               omitted; if so, the subframe is nameless.

Separator:     1 byte:  (null).  Must be omitted if all
               remaining fields are also omitted.

Change interframe
    delay:     1 byte.

                0:  No.
                1:  Yes, for the next subframe only.
                2:  Yes, also reset default.

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

Change timeout and
  termination: 1 byte
                0:  No.
                1:  Deterministic, for the next subframe
                    only.
                2:  Deterministic, also reset default.
                3:  Decoder-discretion, for the next
                    subframe only.
                4:  Decoder-discretion, also reset
                    default.
                5:  User-discretion, for the next subframe
                    only.
                6:  User-discretion, also reset default.
                7:  External-signal, for the next subframe
                    only.
                8:  External-signal, also reset default.

Change subframe clipping
 boundaries:   1 byte.
                0:  No.
```

```
                  1:  Yes, for the next subframe only.
                  2:  Yes, also reset default.

Change sync id
  list:         1 byte.
                  0:  No.
                  1:  Yes, for the next subframe only.
                  2:  Yes, also reset default list.

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

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

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

Left_fb or delta
  left_fb:      4 bytes (signed integer).

Right_fb or delta
  right_fb:     4 bytes (signed integer).

Top_fb or delta
  top_fb:       4 bytes (signed integer).

Bottom_fb or delta
  bottom_fb:    4 bytes (signed integer).

Sync id:        4 bytes (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 subframe that is about to be defined, not the one that is being terminated by the FRAM chunk.

Framing modes:

The framing mode provides information to the decoder that it uses whenever it is about to display an image, and when it is processing the *next* FRAM chunk.

The following events trigger the display of an image:

- Decoding a IHDR-IEND sequence at the MNG level, when it defines a potentially visible image.
- Decoding a JHDR-IEND sequence at the MNG level, when it defines a potentially visible image.
- Decoding a DHDR-IEND sequence, when it defines a potentially visible image.
- Decoding a BASI-IEND sequence, when it defines a potentially visible image.
- Decoding a CLON chunk, when it defines a potentially visible image.
- Decoding a PAST chunk, when its destination is a potentially visible image.
- Decoding a SHOW chunk, when it directs that a potentially visible image be displayed. When the SHOW chunk directs that several images be displayed, each one in turn generates a separate layer.
- Also, decoding the next FRAM chunk, when the framing mode is 3 or 4 and none of the above were found between the FRAM chunks.

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

When a decoder is ready to perform a display update, it must check the framing mode, to decide whether it should restore the background (framing modes 3 and 4) or not (framing modes 1 and 2), and whether it needs to wait for the interframe delay to elapse before continuing (framing modes 1 and 3) or not (framing modes 2 and 4).

When the interframe delay is zero, viewers are not required actually to update the display but can continue to process the remainder of the frame and composite the next image over the existing frame before displaying anything. The final result must appear the same as if each image had been displayed in turn with no delay.

Regardless of the framing mode, encoders must insert a background layer, with a zero delay, ahead of the first image layer in the datastream, even when the BACK chunk is not present or has been clipped to less than full-frame. This layer must be included in the layer count but not in the frame count.

Also, viewers that jump or skip to a segment must insert a background layer, with a zero delay, ahead of the segment, even when the BACK chunk is not present in the prologue segment. Such layers are *not* included in either the layer count or the frame count.

Framing mode 1.

When `framing_mode=1`, the decoder must wait until the interframe delay has elapsed before displaying each image. The background layer is inserted, with no interframe delay, before the first

image in the datastream. If the interframe delay is nonzero, the first frame in the datastream consists of two layers and the rest consist of a single layer.

Framing mode 2.
Framing mode 2 is the same as framing mode 1, except that the interframe delay occurs between subframes rather than between images.

In the usual case, the interframe delay is nonzero, so each subframe is a composite frame. When `framing_mode=2`, viewers are expected to display all of the images in a frame at once, if possible, or as fast as can be managed, without clearing the display or restoring the background. The next FRAM chunk delimits the subframe. A subframe also ends when a SEEK chunk or the MEND chunk appears, and the final frame ends when the MEND chunk appears.

Framing mode 3.
When `framing_mode=3`, a background layer is displayed before each image layer is displayed, without a delay between the background and the image. The decoder must wait until the interframe delay has elapsed before displaying the background. If the interframe delay is nonzero, each frame consists of two layers. A subframe boundary occurs after each image appears. Otherwise, `framing_mode=3` is identical to `framing_mode=1`.

Framing mode 4.
When `framing_mode=4`, the background layer is displayed before each frame, i.e., after each FRAM chunk, with no interframe delay before each image. The decoder must wait until the interframe delay has elapsed before displaying the background layer. Otherwise, `framing_mode=4` is identical to `framing_mode=2`.

The subframe name must conform to the same formatting rules as those for a PNG 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 (unless the subframe name is omitted) and no more than 79 characters in the keyword. Keywords are case-sensitive. There is no null byte within the keyword. No specific use for the subframe name is specified in this document, except that it can be included in the optional index that can appear in the SAVE chunk. Applications can use this field for such purposes as constructing an external list of subframes in the datastream. The subframe name only applies to the upcoming subframe; subsequent subframes are unnamed unless they also have their own `frame_name` field. It is recommended that the same name not appear in any other FRAM chunk or in any SEEK or eXPI chunk. Subframe names should not begin with the case-insensitive strings “CLOCK(”, “FRAME(”, or “FRAMES(”, which are reserved for use in URI queries and fragments.

The interframe delay value is the desired minimum time to elapse from the beginning of displaying one frame until the beginning of displaying the next frame. When the interframe delay is nonzero, which will probably be the usual case, subframes are frames. When it is zero, a frame consists of any number of consecutive subframes, until a nonzero delay subframe is encountered and completed. Decoders are not obligated to display such subframes individually; they can composite them offscreen and only display the complete frame.

There is no interframe delay before the first layer (the implicit background layer) in the datastream nor after the final frame, regardless of the framing mode.

The timeout field can be a number or <infinity>. Infinity can be represented by 0x7fffffff.

The termination condition given in the `change_timeout_and_termination` field specifies how much longer, after the normal interframe delay has elapsed, the frame will endure, and whether the interframe delay can be shortened. It can take the following values:

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

decoder-discretion
: If the interframe delay is nonzero, the decoder can lengthen the duration of the frame, but by no more than the timeout. A streaming decoder could take the opportunity to wait for its input buffer to fill to a comfortable level. When the timeout is zero, the decoder can shorten the duration of the frame to less than the interframe delay.

user-discretion
: If the interframe delay is nonzero, after the interframe delay has expired, 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". When the timeout is zero, the user can shorten the duration of the frame to less than the interframe delay.

external-signal
: If the interframe delay is nonzero, after the interframe delay has expired, the decoder should wait for the arrival of a signal whose number matches a `sync_id`, but must wait no longer than the timeout. When the timeout is zero, the arrival of a signal can shorten the duration of the frame to less than the interframe delay.

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
Subframe name             = <empty string>
Interframe delay          = 1
Left subframe boundary    = 0
Right subframe boundary   = frame width
Top subframe boundary     = 0
Bottom subframe boundary  = frame height
Termination               = deterministic
Timeout                   = 0x7fffffff (infinite)
Sync id                   = <empty list>
```

The subframe boundaries from the FRAM chunk are only used for clipping, not for placement. The DEFI or MOVE chunk can be used to specify the placement of each image within the layer. The DEFI or CLIP chunk can be used to specify clipping boundaries for each image. Even when the left and top subframe boundaries are nonzero, the image locations are measured with respect to the {0,0} position in the display area.

If the layers are transparent or do not cover the entire area defined by the subframe clipping boundaries, they are composited against whatever already occupies the area, when the framing mode is 1 or 2. When the framing mode is 3 or 4, 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 does not define a mandatory background. The images, as well as the background, are clipped to the subframe clipping boundaries. Any pixels outside the subframe clipping boundaries remain unchanged from the previous subframe.

The `interframe_delay` field gives the duration of display, which is the minimum time that must elapse from the beginning of displaying one subframe until the beginning of displaying the next. It is measured in "ticks" using the tick length determined from `ticks_per_second` defined in the MHDR chunk. When the interframe delay is zero, it indicates that all subframes are to be combined into a single frame, until a nonzero interframe delay is specified or the MEND chunk is reached.

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 `interframe_delay` and the `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 subframe that follows the FRAM chunk.

The identifier `sync_id=0` is reserved to represent synchronization with a user input from a keyboard or pointing device. The `sync_id` values 1–255 are reserved to represent the corresponding ASCII letter, received from the keyboard (or a simulated keyboard), and values 256–1023 are reserved for future definition by this specification. If multiple channels (not defined in this specification) are not present, viewers can ignore other values appearing in the `sync_id` list.

4.3.3 MOVE New image location

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

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

The chunk's contents are:

```
First object:       2 bytes (unsigned integer).

Last object:        2 bytes (unsigned integer).

Location
  delta type:       1 byte (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.

X_location or
  delta X_location: 4 bytes (signed integer).

Y_location or
  delta Y_location: 4 bytes (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. `Last_object` must not be less than `first_object`. Negative values are permitted for the X and Y location. The positive directions are downward and rightward from the frame origin. The MOVE chunk can specify an image placement that is partially or wholly outside the display boundaries. In such cases, the resulting image must be clipped to fit within its clipping boundaries, or not displayed at all if it falls entirely outside its clipping boundaries. The clipping boundaries are determined as described in the specification for the CLIP chunk below (Paragraph 4.3.4).

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

4.3.4 CLIP Object clipping boundaries

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

```
First object:       2 bytes (unsigned integer).

Last object:        2 bytes (unsigned integer).

Clip delta type:    1 byte (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.
```

```
Left_cb or
  delta_left_cb:   4 bytes (signed integer).

Right_cb or
  delta_right_cb:  4 bytes (signed integer).

Top_cb or
  delta_top_cb:    4 bytes (signed integer).

Bottom_cb or
  delta_bottom_cb: 4 bytes (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. `Last_object` must not be less than `first_object`.

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

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

```
0        <= x < frame_width    (from the MHDR chunk)
0        <= y < frame_height
Left_fb <= x < right_fb       (from the FRAM chunk)
Top_fb  <= y < bottom_fb
Left_cb <= x < right_cb       (from the CLIP chunk)
Top_cb  <= y < bottom_cb
```

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

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

4.3.5 SHOW Show images

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

```
First image: 2 bytes (nonzero unsigned integer).

Last image:  2 bytes  (nonzero unsigned integer).
             This field can be omitted if the show_mode
             byte is also omitted.  If so, decoders must
             assume the default values, show_mode=0
             and last_image=first_image.

Show_mode:   1 byte (unsigned integer).
```

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

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

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

When the SHOW chunk is empty, the decoder displays all existing potentially visible images, without changing their `do_not_show` status. The empty SHOW chunk is equivalent to

```
SHOW 1 65535 2
```

If `last_image` < `first_image` the images are processed in reverse order.

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

Interactions with the framing mode

When `show_mode` is even-valued, each visible image that is displayed generates a separate

layer, even if it is offscreen and no pixels are actually displayed. In such cases, the layer is totally transparent. When `show_mode` is odd, or when `show_mode=2 or 4` or is empty and no image is visible, no layer is generated.

When `show_mode` is 1, 4, 5, 6, or 7, images can be made invisible. This is not permitted when the framing mode is 2 or 4 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` is 6 or 7, a single layer is generated. The decoder must make the next image in the "cycle" visible. To do this, it must examine the `do_not_show` flag for each image in the range `first_image` through `last_image`, and make the next one (the one with the next higher value of `image_id` that exists and is "viewable") after the first visible one it finds visible and the rest invisible. When `first_image > last_image`, the cycle is reversed, and the "next" image is the one with the next lower value of `image_id`. In either case, if the first visible one found was `last_id`, or none were visible, it must make `first_image` visible. These modes are useful for manipulating a group of sequential images that represent different views of an animated icon. See Example 8, below (Chapter 17). If no "viewable" object is in the specified range, an empty layer must be generated.

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

Assuming a nonzero interframe delay, any of the following sequences would cause the image identified by `object_id=6` in a composite frame to blink:

```
LOOP 0 0 10
FRAM 4        # Show background
SHOW 1 10     # Show images 1 thru 10.
FRAM          # Show background
SHOW 1 5      # Show images 1 thru 5.
SHOW 7 10     # Show images 7 thru 10.
ENDL

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

FRAM 4        # Show background
LOOP 0 0 10
SHOW 6 6 5    # Toggle potential visibility of image 6.
SHOW 1 10 2   # Show potentially visible images in 1
```

```
FRAM          # through 10.
ENDL
```

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

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

4.3.6 TERM Termination action

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

```
Termination
  action:      1 byte (unsigned integer)
                 0: Show the last frame indefinitely.
                 1: Cease displaying anything.
                 2: Show the first frame after the TERM
                    chunk.
                    If processing the fPRI chunk,
                    use a "cost" of 255.
                 3: Repeat the animation starting
                    immediately after the TERM chunk.

Action after
  iterations: 1 byte
                 0: Show the last frame indefinitely after
                    iteration_max iterations have been
                    done.
                 1: Cease displaying anything.
                 2: Show the first frame after the TERM
                    chunk.
                    If processing the fPRI chunk,
                    use a "cost" of 255.

               This and the remaining fields must be
               present if termination_action is 3, and
               must be omitted otherwise.

Delay:         4 bytes (unsigned integer).  Delay, in
               ticks, before repeating the animation.

Iteration max: 4 bytes (unsigned integer).  Maximum
               number of times to repeat the animation.
```

The loop created by processing a TERM must always be treated by the decoder as if it were a cacheable <user-discretion> loop, with `iteration_min=1`.

MNG editors that extract a series of PNG or JNG files from a MNG datastream are expected to execute the TERM loop only once, rather than emitting the files repeatedly.

The TERM chunk, if present, must appear either immediately after the MHDR chunk or immediately prior to a SEEK chunk. The TERM chunk is not considered to be a part of any segment for the purpose of determining the copy-safe status of any chunk. Only one TERM chunk is permitted in a MNG datastream.

Simple viewers and single-frame viewers can ignore the TERM chunk. It has been made critical only so MNG editors will not inadvertently relocate it.

4.4 SAVE and SEEK chunks

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

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

4.4.1 SAVE Save information

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

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

```
Offset size: 1 byte (unsigned integer).
               4: Offsets and nominal start times are
                  expressed as 32-bit integers.
               8: Offsets and nominal start times are
                  expressed as 64-bit integers.
```

plus zero or more of the following index entries:

```
Entry type:  1 byte (unsigned integer).
               0: Segment with nominal start time,
                  nominal layer number, and nominal
                  frame number.
               1: Segment.
               2: Subframe.
               3: Exported image.
Offset:      4 or 8 bytes (unsigned integer).  Omit if
               entry_type > 1, set equal to zero
               if the offset is unknown.
Nominal start
  time:      4 or 8 bytes: (unsigned integer).  Start
               time of the segment, measured in ticks
               from the beginning of the animation,
               assuming that all prior segments were
               played as intended on an ideal player,
               ignoring any fPRI chunks.  Omit if
               entry_type > 0.
Nominal layer
  number:    4 bytes (unsigned integer).  Sequence
               number of the first layer in the segment,
               assuming that all prior segments were
               played as intended on an ideal player,
               ignoring any fPRI chunks; the first
               layer of the first segment being layer 0.
               Omit if entry_type > 0.
Nominal frame
  number:    4 bytes (unsigned integer).  Sequence
               number of the first frame in the segment,
               assuming that all prior segments were
               played as intended on an ideal player,
               ignoring any fPRI chunks; the first frame
               of the first segment being frame 0.  Omit
               if entry_type > 0.
Name:        1-79 bytes (Latin-1 text).  Omit for
               unnamed segments.
Separator:   1 byte (null) (must be omitted after
               the final entry).
```

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

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

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

An object appearing ahead of the SAVE chunk can be the subject of a CLON chunk. If the clone is a partial clone, modifying 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 skipping or jumping to a SEEK chunk. If no instance of the chunk appeared ahead of the SAVE chunk, the decoder must restore the chunk data to its original "unknown" condition when it skips or jumps to a SEEK chunk.

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

Known chunks in this category include DEFI, FRAM, BACK, PLTE, cHRM, tRNS, fPRI, gAMA, iCCP, bKGD, sBIT, pHYg, pHYs, and sRGB.

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

If an entry for a segment (entry type 0 or 1) appears in the optional index, there must also be an entry for every segment, whether named or not, except for the prologue segment, that precedes it. All entries must appear in the index in the same order that they appear in the MNG datastream. There must never be a segment entry (type 0 or 1) for the prologue segment, but there can be entries for named images or subframes in the prologue, placed ahead of the first segment entry. Only named images or subframes are permitted, and it is not an error to omit any or all named images or subframes. Nor is it an error to omit a contiguous set of segments at the end of the datastream from the index.

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

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

Only one instance of the SAVE chunk is permitted in a MNG datastream. 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. The SAVE chunk must not appear inside a LOOP-ENDL pair.

4.4.2 SEEK Seek point

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

The SEEK can be empty, or it can contain a segment name.

```
Segment name: 1-79 bytes (Latin-1 string).
```

The segment name is optional. It must follow the format of a tEXt keyword: It must consist only of printable Latin-1 characters and must not have leading or trailing blanks, but can have single embedded blanks. There must be at least one and no more than 79 characters in the keyword. There is no null byte terminator within the segment name, nor is there a separate null byte terminator. Segment names are case-sensitive. Use caution when printing or displaying keywords (Refer to Security considerations, below, Chapter 16). No specific use for the segment name is specified in this document, but applications can use the segment name for such purposes as constructing a menu of SEEK points for a slide-show viewer. It can be included in the optional index that can appear in the SAVE chunk. It is recommended that the same name not appear in any other SEEK chunk or in any FRAM or eXPI chunk. Segment names should not begin with the case-insensitive strings “CLOCK(”, “FRAME(”, or “FRAMES(”, which are reserved for use in URI queries and fragments.

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

- Data appearing in the MHDR chunk.
- Anything appearing ahead of the SAVE chunk.

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

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

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

Multiple instances of the SEEK chunk are permitted. The SEEK chunk must not appear prior to the SAVE chunk. The SAVE chunk must also be present if the SEEK chunk is present. The SEEK must not appear between a LOOP chunk and its ENDL chunk.

4.5 Ancillary MNG chunks

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

4.5.1 eXPI Export image

The eXPI chunk takes a snapshot of a viewable 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 (snapshot id) and a name:

```
Snapshot id:   2 bytes (unsigned integer).
               (must be zero in MNG-LC and
               MNG-VLC datastreams)
Snapshot name: 1-79 bytes (Latin-1 text).
```

When the snapshot_id is zero, the snapshot is the first instance of an embedded image with object_id=0 following the eXPI chunk. When the snapshot_id is nonzero, the snapshot is an already-defined object with that object_id as it already exists when the eXPI chunk is encountered.

Note that the `snapshot_name` is associated with the snapshot, not with the `snapshot_id` nor its future contents; changing the image identified by `snapshot_id` will not affect the snapshot. The `snapshot_name` means nothing inside the scope of the MNG specification, except that it can be included in the optional index that can appear in the SAVE chunk. If two eXPI chunks use the same name, it is the outside world's problem (and the outside world's prerogative to regard it as an error). It is recommended, however, that the `snapshot_name` not be the same as that appearing in any other eXPI chunk or in any FRAM or SEEK chunk. 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, to allow applications to extract images by their `snapshot_id`.

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

The `snapshot_name` string must follow the format of a tEXt keyword: It must consist only of printable Latin-1 characters and must not have leading or trailing blanks, but can have single embedded blanks. There must be at least one and no more than 79 characters in the keyword. Keywords are case-sensitive. There is no null byte terminator within the `snapshot_name` string, nor is there a separate null byte terminator. Snapshot names should not begin with the case-insensitive strings "CLOCK(", "FRAME(", or "FRAMES(" which are reserved for use in URI queries and fragments.

Multiple instances of the eXPI chunk are permitted in a MNG datastream, and they need not have different values of `snapshot_id`.

4.5.2 fPRI Frame priority

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

The fPRI chunk contains two bytes:

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

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

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

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

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

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

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

Multiple instances of the fPRI chunk are permitted.

4.5.3 nEED Resources needed

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

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

```
Keyword:   1-79 bytes.
Separator: 1 byte (null).
...etc...
```

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

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

Each keyword string must follow the format of a tEXt keyword: It must consist only of printable Latin-1 characters and must not have leading or trailing blanks, but can have single embedded blanks. There must be at least one and no more than 79 characters in the keyword. Keywords are case-sensitive. There is no null byte terminator within the keyword. A null separator byte must appear after each keyword in the nEED chunk except for the last one.

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

4.5.4 pHYg Physical pixel size (global)

The MNG pHYg chunk is identical in syntax to the PNG pHYs chunk. It applies to complete full-frame MNG layers and not to the individual images within them.

Conceptually, a MNG viewer that processes the pHYg chunk will first composite each image into a full-frame layer, then apply the pHYg scaling to the layer, and finally composite the scaled layer into the subframe. MNG datastreams can include both the pHYs chunk (either at the MNG top level or within the PNG and JNG datastreams) and the pHYg chunk (only at the MNG top level), to ensure that the images are properly displayed either when displayed by a MNG viewer or when extracted into a series of individual PNG or JNG datastreams and then displayed by a PNG or JNG application. The pHYs and pHYg chunks would normally contain the same values, but this is not necessary.

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

4.6 Ancillary PNG chunks

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

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

  A MNG editor that writes PNG datastreams should not include the top-level iTXt, tEXt, tIME, and zTXt chunks in the generated PNG datastreams.
- cHRM, gAMA, iCCP, sRGB, bKGD, sBIT, pHYs

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

  In the MNG top level, all of these chunks are written as though for 16-bit RGBA PNG datastreams. Decoders are responsible for reformatting the chunk data to suit the actual bit depth and color type of the datastream that inherits them.

  A MNG editor that writes PNG or JNG datastreams is expected to include the top-level cHRM, gAMA, iCCP, and sRGB chunks in the generated PNG or JNG datastreams, if the embedded image does not contain its own chunks that define the color space. When it writes the sRGB chunk, it should write the gAMA chunk (and perhaps the cHRM chunk), in accordance with the PNG specification, even though no gAMA or cHRM chunk is present in the MNG datastream. It is also expected to write the pHYs chunk and the reformatted top-level bKGD chunk in the generated PNG or JNG datastreams, and the reformatted sBIT chunk only in generated PNG datastreams, when the datastream does not have its own bKGD, pHYs, or sBIT chunks.
- sPLT

  This PNG chunk is also defined at the MNG top level. It provides a value that takes precedence over those that might be provided in subsequent PNG or JNG datastreams and provides a value to be used when it is not provided in subsequent PNG or JNG datastreams. It also takes precedence over the PLTE chunk in a subsequent PNG datastream when the PLTE and hIST chunks are being used as a suggested palette (i.e., `color_type != 3`). This chunk can appear for any color type. There can be multiple sPLT chunks in a MNG datastream. If a `palette_name` is repeated, the previous palette having the same `palette_name` is replaced. It is not permitted, at the MNG top level, to redefine a palette after the SAVE chunk with the same `palette_name` as one that appears ahead of the SAVE chunk. It is permitted, however, to define and redefine other palettes with other `palette_name` fields. A single empty sPLT chunk can be used to nullify all sPLT chunks that have been previously defined in the

MNG top level, except for those that appeared ahead of the SAVE chunk, when the SAVE chunk has been read.

When a decoder needs to choose 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 layer-to-layer color changes.

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

5 The JPEG Network Graphics (JNG) Format

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

MNG-LC and MNG-VLC applications can choose to support JNG or not. Those that do not can check bit 4 (JNG is present/absent) of the MHDR simplicity profile to decide whether they can process the datastream.

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

```
http://www.cdrom.com/pub/png/
```

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

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

```
139 80 78 74 13 10 26 10
```

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

JNG is pronounced "Jing."

5.1 Critical JNG chunks

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

5.1.1 JHDR JNG header

The format of the JHDR chunk introduces a JNG datastream. It contains:

```
Width:      4 bytes (unsigned integer, range 0..65535).
Height:     4 bytes (unsigned integer, range 0..65535).
Color type: 1 byte
              8: Gray (Y).
             10: Color (YCbCr).
             12: Gray-alpha (Y-alpha).
             14: Color-alpha (YCbCr-alpha).

Image sample
  depth:    1 byte
              8: 8-bit samples and quantization tables.
             12: 12-bit samples and quantization
                 tables.
             20: 8-bit image followed by a 12-bit
                 image.

Image compression
  method:   1 byte
              8: ISO-10918-1 Huffman-coded baseline JPEG.

Image interlace
  method:   1 byte.
              0: Sequential JPEG, single scan.
              8: Progressive JPEG.

Alpha sample
  depth:    1 byte.
              0, 1, 2, 4, 8, or 16.

Alpha compression
  method:   1 byte.
              0: PNG grayscale IDAT format.

Alpha filter
  method:   1 byte.
              0: Adaptive (see PNG spec).

Alpha interlace
  method:   1 byte.
              0: Not interlaced.
```

The width, height, image_sample_depth, image_compression_method, and image_interlace_method fields are redundant because equivalent information is also embedded in the JDAT datastream. They appear in the JHDR chunk for convenience. Their values must be identical to their equivalents embedded in the JDAT

chunk. We use four bytes in the width and height fields for similarity to MNG and PNG, and to leave room for future expansion, even though two bytes would have been sufficient.

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

5.1.2 JDAT JNG image data

A JNG datastream must contain one or more JDAT chunks, whose data, when concatenated, forms a complete JNG JPEG datastream. JNG decoders are required to read all baseline JNG JPEG and eight-bit progressive JNG JPEG datastreams. Twelve-bit capability is not required.

JDAT chunks are like PNG IDAT chunks in that there may be multiple JDAT chunks, the data from which are concatenated to form a single datastream that can be sent to the decompressor. No chunks are permitted among the sequence of JDAT chunks, except for interleaved IDAT chunks. The ordering requirements of other ancillary chunks are the same with respect to JDAT as they are in PNG with respect to the IDAT chunk.

A JNG JPEG is a baseline, extended-sequential, or progressive JPEG as defined by JPEG Part 1 [ISO/IEC-10918-1]. JNG uses only JFIF-compatible [JFIF] component interpretations, and imposes a few additional restrictions that reflect limitations of many existing JPEG implementations. In particular, only Huffman entropy coding is permitted.

Actually, a JNG may contain two separate JNG JPEG datastreams (one eight-bit and one twelve-bit), each contained in a series of JDAT chunks, and separated by a JSEP chunk (see the JSEP chunk specification below, Paragraph 5.1.5). Decoders that are unable to handle twelve-bit datastreams are allowed to display the eight-bit datastream instead, if one is present.

The core of the JNG JPEG definition is baseline JNG JPEG, which is JPEG Part 1's definition of baseline JPEG further restricted by JFIF restrictions and JNG-specific restrictions. JPEG, which is also defined in JPEG Part 1 and has JNG-specific restrictions.

- Baseline JNG JPEG restrictions

  A baseline JPEG according to JPEG Part 1 is DCT-based (lossy) sequential JPEG, using 8-bit sample precision and Huffman entropy coding, with the following further restrictions:

  - Quantization table precision must be 8 bits for baseline JPEG.
  - Huffman code tables can have table numbers 0 and 1 only.

  The SOF marker type for baseline JPEG is SOF0.

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

- JFIF-compatible restrictions

  The image data is always stored left-to-right, top-to-bottom.

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

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

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

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

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

  The constant Half_scale is 128 when dealing with eight-bit data, 2048 for twelve-bit data. With these equations, Y, Cb, and Cr all have the same range as R, G, and B: 0 to 255 for eight-bit data, 0 to 4095 for twelve-bit data.

  The JFIF convention for YCbCr differs from typical digital television practice in that no headroom/footroom is reserved: the coefficient values range over the full available 8 or 12 bits.

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

- Additional JNG restrictions

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

  - The sampling factors for YCbCr images must be one of these sets:
    * 1h1v,1h1v,1h1v (also called 4:4:4 or 1x1 sampling)
    * 2h1v,1h1v,1h1v (also called 4:2:2 or 2x1 sampling)
    * 2h2v,1h1v,1h1v (also called 4:2:0 or 2x2 sampling)
    * 1h2v,1h1v,1h1v (also called 1x2 sampling)

    In other words, the chroma components may be downsampled 2:1 or 1:2 horizontally or vertically relative to luminance, or they may be left full size. These four sampling ratios are the only ones

supported by a wide spectrum of implementations (1x2 is relatively uncommon, and is usually the result of a lossless rotation of a 2x1 sampling).

For grayscale images, the sampling factors are irrelevant according to a strict reading of JPEG Part 1. Hence decoder authors should accept any sampling factors for grayscale. However, we recommend that encoders always emit sampling factors 1h1v for grayscale, since some decoders have been observed to malfunction when presented with other sampling factors.

– There must be only one scan in an image: that is, YCbCr images must be fully interleaved. There is little advantage to be gained by encoding a baseline image in multiple scans, and many baseline decoders do not support multiple scans at all.

– The DNL (Define Number of Lines) marker is prohibited. The image height must always be specified accurately in the SOFn marker and in the JHDR chunk.

- Recommended progressive JPEG subset

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

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

  – Multiple-scan support is obviously required for progressive JPEG.

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

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

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

  When the JSEP chunk is present, both images must be progressive if one of them is progressive.

- Recommended 12-bit JPEG subset

  JNG JPEGs may optionally use 12-bit sample precision as defined in JPEG Part 1.

  For a sequential image, the SOF marker type must be SOF1 (extended sequential) not SOF0, and the baseline restriction of two Huffman tables is removed. Also, the encoder may use either 8-bit or 16-bit quantization tables. All other JNG baseline restrictions still apply. It is recommended that JNG encoders not use extended-sequential mode except to encode 12-bit data.

For a progressive image, the only difference between 8-bit and 12-bit modes is that the sample precision is 12 bits and the encoder may use either 8-bit or 16-bit quantization tables. All other JNG restrictions still apply.

5.1.3 IDAT JNG alpha data

This chunk is exactly like the IDAT chunk in a PNG grayscale image, except that it is interpreted as an alpha mask to be applied to the image data from the JDAT chunks, when `alpha_compression_method=0`. The alpha channel, if present, can have sample depths 1, 2, 4, 8, or 16. The IDAT chunks can be interleaved with the JDAT chunks (see Recommendations for Encoders: JNG interleaving below). No other chunk type can appear among the sequence of IDAT and JDAT chunks. No other chunk type can appear between the sequences of IDAT and JDAT chunks when they are not interleaved. The samples in the IDAT must be presented in noninterlaced order, left to right, top to bottom. As in PNG, zero means fully transparent and $2^{\mathrm{alpha_sample_depth}} - 1$ means fully opaque.

The IDAT chunks must precede the JSEP chunk, if the JSEP chunk is present. Minimal viewers that ignore the twelve-bit JDAT chunks must read the IDAT chunks and apply the alpha samples to the eight-bit image that is contained in the JDAT chunks that precede the JSEP chunk. Viewers that skip the eight-bit JDAT chunks must decode the IDAT chunks that precede the JSEP chunk and apply the alpha samples to the twelve-bit image that is contained in the JDAT chunks that follow the JSEP chunk.

5.1.4 IEND End of JNG datastream

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

5.1.5 JSEP 8-bit/12-bit image separator

The JSEP chunk is empty.

A JSEP chunk must appear between the JDAT chunks of an eight-bit datastream and those of a twelve-bit datastream, when `image_sample_depth=20` in the JHDR chunk. The eight-bit datastream must appear first. Both images must have the same width, height, color type, compression method, and interlace type. Viewers can choose to display one or the other image, but not both.

5.2 Ancillary JNG chunks

Some PNG ancillary chunks can also appear in JNG datastreams, and are used for the same purposes as described in the PNG specification [PNG] and the Extensions to the PNG Specification document [PNG-EXT].

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

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

The PNG PLTE, hIST, pCAL, sBIT, sPLT, tRNS, fRAc, and gIF* chunks are not defined in JNG.

When cHRM, gAMA, iCCP, or sRGB are present, they provide information about the color space of the decoded JDAT image, and they have no effect on the decoded alpha samples from the IDAT chunks. Any viewer that processes the gAMA chunk must also recognize and process the sRGB chunk. It can treat it as if it were a gAMA chunk containing the value .45455 and it can ignore its "intent" field.

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

6 The Delta-PNG Format

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

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

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

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

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

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

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

6.1 Delta-PNG critical chunks

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

6.1.1 DHDR Delta-PNG datastream header

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

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

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

Image type:   1 byte.

                0: Image type is unspecified.  An IHDR,
                   JHDR, IPNG, or IJNG chunk must be
                   present.  If JHDR or IJNG is present,
                   delta_type must not be 1, 3, 4, or 6.

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

                2: Image type is JNG.  JHDR and IJNG can
                   be omitted under certain conditions.
                   Delta_type must not be 1, 3, 4, or 6.

Delta type:   1 byte.

                0: Entire image replacement.

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

                2: Block alpha addition, by samples,
                   modulo 2^sample_depth.  Regardless of
                   the color type of the parent image,
                   the IDAT data are written as a
                   grayscale image (color type 0), but
                   the decoded samples are used as deltas
                   to the alpha samples in the parent
```

```
                     image.  The parent image must have
                     (or be promoted to via the PROM chunk)
                     a color type that has an alpha channel.

                  3: Block color addition.  Similar to
                     delta type 1 except that only the
                     color channels are updated even when
                     the parent has an alpha channel.

                  4: Block pixel replacement.

                  5: Block alpha replacement.

                  6: Block color replacement.

                  7: No change to pixel data.

Block width:  4 bytes (unsigned integer).  Omit when
                  delta_type=7.

Block height: 4 bytes (unsigned integer).  Omit when
                  delta_type=7.

Block
  X_location: 4 bytes (unsigned integer),
                  measured in pixels from the left edge
                  of the parent object.  Omit when
                  delta_type=0 or when delta_type=7.

Block
  Y_location: 4 bytes (unsigned integer),
                  measured in pixels from the top edge
                  of the parent object.  Omit when
                  delta_type=0 or when delta_type=7.
```

The `object_id` must identify an existing object, and the object must be a "concrete" object, i.e., it must have the property `concrete_flag=1`.

The `image_type`, whether given explicitly as 1 or 2 or implied by the presence of an IHDR, IPNG, JHDR, or IJNG 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--6`, the `block_width` and `block_height` fields give the size of the block of pixels to be modified or replaced, and `block_X_location` and `block_Y_location` give its location with respect to the top left corner of the parent object. The block must fall entirely within the parent object.

Entire image replacement

When `delta_type=0` in the DHDR chunk, the pixel data in the IDAT chunks represent a completely new image, with dimensions given by the `block_width` and `block_height` fields of the DHDR chunk. Data from chunks other than IDAT or JDAT can be inherited from the parent object. If the IHDR or JHDR chunk is present, all of its fields except `width` and `height` (which must be ignored by decoders) provide new values that are inherited by subsequent objects. The "pixel sample depth" and "alpha sample depth" are also reset equal to the IHDR `sample_depth` value (in the case of a JNG object, the new "alpha sample depth" is taken from the JHDR `alpha_sample_depth` field). If the IHDR or JHDR chunk is not present, the IDAT chunks are decoded according to the parent object's sample depth, and not according to the "pixel sample depth" or "alpha sample depth" which are used for decoding the IDAT chunks in subsequent Delta-PNG datastreams when `delta_type` is nonzero.

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, including the alpha channel, if the parent object has an alpha channel.

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 pixel data includes alpha samples, if the parent object has an alpha channel.

An encoder calculates the delta 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 $2^{\mathrm{sample_depth}}$. When decoding the IDAT chunk, the child image bytes are obtained by adding the delta bytes to the parent object bytes, modulo $2^{\mathrm{sample_depth}}$. This is similar in operation to the PNG SUB filter, except that it works by samples instead of by bytes.

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.

When the parent object has `color_type=3`, the deltas are differences between index values, not between color samples.

The color type must match that of the parent, except that when the parent has PNG `color_type=3`, the delta can have `color_type=0`, and vice versa, since the contents of the IDAT chunks of either color type are indistinguishable.

If the `pixel_sample_depth` does not match the `object_sample_depth`, the delta must be scaled to the `object_sample_depth` using the zero-fill method described in the PNG specification, before performing the pixel addition.

When the IHDR chunk is present, the compression method, filter method, and interlace method need not be the same as those of the parent object. The new values are used in decoding the IDAT data, and the new values are inherited by the child object.

Whenever the sample depth differs from that of the parent object, the resulting object inherits the original value from the parent. The value from the IHDR chunk is only used for decoding the IDAT data

in this and subsequent Delta-PNGs. Implicit in this is the requirement for decoders to remember in the object buffer not only the sample depth of the object but (separately) the "pixel sample depth" for use in decoding the IDAT chunks of subsequent Delta-PNG datastreams that do not contain their own IHDR chunk.

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 `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. If they are different, the child object inherits the new values, and the new values will be used in decoding the data in any subsequent IDAT chunks.

The `sample_depth` value from the IHDR chunk is interpreted as a new value of `alpha_sample_depth` and is only used for decoding the IDAT data in this and subsequent Delta-PNGs. Implicit in this is the requirement for decoders to remember in the object buffer not only the sample depth of the object but (separately) the `alpha_sample_depth` for use in decoding the IDAT chunks in any subsequent Delta-PNG datastreams.

If the `alpha_sample_depth` does not match the `object_sample_depth`, the delta must be scaled to the `object_sample_depth`, using the zero-fill method described in the PNG specification, before performing the pixel subtraction.

Block color addition

`delta_type=3` is similar to `delta_type=1` except that the alpha channel is not included in the IDAT pixels; the alpha channel is inherited from the parent object. The color type of the parent must be one that has an alpha channel (4 or 6) and the color type of the delta must be the corresponding color type (0 or 2) that does not have an alpha channel.

Block pixel replacement

When `delta_type=4` 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, including the alpha channel, if the parent object has an alpha channel.

If the `pixel_sample_depth` does not match the `object_sample_depth`, the pixel data must be linearly scaled to the `object_sample_depth` before making the replacements. This can be accomplished by using the left bit replication method described in the PNG specification, or by the right shift method in the unlikely event that the `pixel_sample_depth` is larger than the `object_sample_depth`.

The color type must match that of the parent, except for the cases mentioned for delta type 1, above.

Block alpha replacement

When `delta_type=5` in the DHDR chunk, the pixel data in the IDAT chunks represent replacement values of the alpha samples in the rectangle given by the block location and dimension

fields in the DHDR chunk. The sample depth of the data (i.e. the "alpha sample depth") need not match the sample depth of the parent object, and `color_type` is 0 (grayscale), regardless of the `color_type` of the parent object. If the sample depths differ, the samples must be scaled to the `object_sample_depth` by linear scaling, which can be achieved with the left bit replication method or right shift method described in the PNG specification, depending on whether the `alpha_sample_depth` is larger or smaller than the `object_sample_depth`. 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. If they differ, the child object inherits the new values.

Block color replacement

`delta_type=6` is similar to `delta_type=4` except that the alpha channel is not included in the IDAT pixels; the alpha channel is inherited from the parent object. The color type of the parent must be one that has an alpha channel (4 or 6) and the color type of the delta must be the corresponding color type (0 or 2) that does not have an alpha channel.

No change to pixel data

When `delta_type=7` in the DHDR chunk, there is no change to the pixel data, and it is an error for IDAT, or JDAT to appear. If the IHDR or JHDR chunk appears, the width, height, and color_type fields are ignored, the PNG sample depth (or JNG alpha_sample_depth) is used to update the `pixel_sample_depth` and `alpha_sample_depth`, and the data in the remaining fields are inherited by the child object.

Pixel sample depth, alpha sample depth

As mentioned above, the sample depth of the deltas is not necessarily the same as that of the object, when `delta_type != 0`. The decoder needs to remember the `pixel_sample_depth` and `alpha_sample_depth` to use with each object. They are initialized to the `sample_depth` value from the IHDR chunk that appears when the object is first created but can be changed by the appearance of the IHDR chunk in a Delta-PNG datastream that has `delta_type != 0`. If the object is a JNG image, they are initialized from the value of `alpha_sample_depth` from the original JHDR chunk, and can be changed by the appearance of the JHDR chunk in a Delta-PNG datastream that has `delta_type != 0`.

6.1.2 IEND End of Delta-PNG datastream

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

The IEND chunk is empty.

6.1.3 PROM Promotion of parent object

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

```
New color type:   1 byte.
New sample depth: 1 byte.
Fill method:      1 byte.
                    0: Left-bit-replication
                    1: Zero fill
```

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

G -> GA (`color_type 0 -> 4`)
Don't change the gray values. Set all the alpha values to fully opaque, except for pixels marked transparent by cheap transparency–set their alpha values to fully transparent. Discard the cheap transparency information (the PNG tRNS chunk data).

RGB -> RBGA (`color_type 2 -> 6`)
Don't change the RGB values. Convert the tRNS chunk data to alpha values as in the G -> GA promotion.

G -> RGB (`color_type 0 -> 2`)
Set R, G, and B equal to the gray value. Apply the same operation to the cheap transparency data (if any). Expand any bKGD or sBIT data.

GA -> RGBA (`color_type 4 -> 6`)
Set R, G, and B equal to the gray value. Don't change the alpha values. Expand any bKGD or sBIT data.

G -> RGBA (`color_type 0 -> 6`)
Set R, G, and B equal to the gray value. Handle transparency as in the G -> GA promotion. Expand any bKGD or sBIT data.

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). Expand any bKGD or sBIT data.

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. Expand any bKGD or sBIT data.

JNG-G -> JNG-C (`JNG color_type 8 -> 10`)
Expand the gray values to color. Expand any bKGD data.

JNG-G -> JNG-GA (`JNG color_type 8 -> 12`)
Don't change the gray values. Set all the alpha values to fully opaque. The given sample depth is the new sample depth for the alpha channel.

JNG-G -> JNG-CA (`JNG color_type 8 -> 14`)
Expand the gray values to color. Set all the alpha values to fully opaque. The given sample depth is the new sample depth for the alpha channel. Expand any bKGD data.

JNG-C -> JNG-CA (`JNG color_type 10 -> 14`)
Don't change the color values. Set all the alpha values to fully opaque. The given sample depth is the new sample depth for the alpha channel.

JNG-GA -> JNG-CA (`JNG color_type 12 -> 14`)
Expand the gray values to color. Don't change the alpha values. Expand any bKGD data.

No change in `color_type`
Only the sample depth is changed. The new sample depth must be larger than the old one.

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

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

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

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

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

6.1.4 IHDR PNG image header

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

If the IHDR chunk is present, its `width`, and `height` fields are ignored. The values for these parameters are inherited from the parent object or from the PROM chunk. different from those of the parent, they are only used for decoding the data in the IDAT chunks, and the child object inherits the original values from the parent.

The `sample_depth`, `color_type`, `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. These do not change the `sample_depth` and `color_type` of the object itself; those can only be changed by using the PROM chunk or by using `delta_type=0`.

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 and PPLT, any chunks appearing between IHDR and IEND are always treated as replacements or additions and not as deltas.

6.1.5 IPNG Incomplete PNG

The IPNG chunk is empty.

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

The IHDR chunk can also be omitted when `image_type=1` and the PNG datastream begins with a PLTE chunk, a PPLT chunk, or an IDAT chunk. In this case, no IPNG chunk is required, either. The decoder must treat this datastream as though the IHDR chunk were present, immediately preceding the first PNG chunk. If the first PNG chunk is neither a PLTE chunk, a PPLT chunk, nor an IDAT chunk, then either the IPNG or IHDR must be present to introduce the PNG datastream.

6.1.6 PLTE and tRNS

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

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

6.1.7 PPLT Partial palette

If it is desired only to overwrite or add palette entries, the PPLT chunk can be used. This might be useful for palette-animation applications. This chunk can also be used to overwrite or add entries to the transparency

(alpha) data from the parent's tRNS chunk.

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

```
PPLT delta type: 1 byte.
                   0: Values are replacement RGB samples.
                   1: Values are delta RGB samples.
                   2: Values are replacement alpha
                      samples.
                   3: Values are delta alpha samples.
                   4: Values are replacement RGBA
                      samples.
                   5: Values are delta RGBA samples.
First index,
  first group:   1 byte.
Last index,
  first group:   1 byte.
First set of
  samples:       1, 3, or 4 bytes.
...etc...
Last set of
  samples:       1, 3, or 4 bytes.
First index,
  second group:  1 byte.
...etc...
```

The `last_index` must be equal to or greater than `first_index`. The groups are not required to appear in ascending order. If any index of any group is beyond the end of the parent object's palette, the palette and tRNS data must be extended just as if a longer complete PLTE chunk had appeared. If there are gaps in the resulting extended palette, the colors must be filled with $\{0,0,0\}$ and the alphas filled with 255. If alpha samples are supplied (`PPLT_delta_type` $>$ `1`) and no tRNS data is present in the parent object, a tRNS chunk must be created in the child object as though a complete tRNS chunk had appeared. The new palette must not be longer than the object's `sample_depth` would allow.

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

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

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

6.1.8 JHDR JNG image header

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

If the JHDR chunk is present, its `width`, `height`, `image_sample_depth`, `image_color_type`, `image_filter_type`, and `image_interlace_type` fields are ignored. The values for these parameters are inherited from the parent object.

The `alpha_compression_method`, `alpha_interlace_type`, and `alpha_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. If the `alpha_sample_depth` differs, it will be used in decoding the IDAT chunk data of the Delta-PNG and subsequent Delta-PNG datastreams; but the child object itself will retain the original sample depth, and must also retain the "alpha sample depth" for use in decoding subsequent Delta-PNG datastreams. The decoded alpha samples must be scaled to the object's sample depth before the replacements or delta calculations are done.

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

6.1.9 IJNG Incomplete JNG

The IJNG chunk is empty.

The IJNG chunk can be used instead of the JHDR chunk if the JHDR chunk is not needed for resetting the value of any of the JHDR fields. The purpose of this chunk is to identify the beginning of the JNG datastream, so decoders can start interpreting JNG chunks instead of Delta-PNG chunks. The decoder must treat this datastream as though the JHDR chunk were present in the location occupied by the IJNG chunk.

The JHDR chunk can also be omitted when `image_type=2` and the JNG datastream begins with a JDAT chunk. Note: Be sure that the first JDAT chunk precedes the first IDAT chunk. In this case, no IJNG chunk is required, either. The decoder must treat this datastream as though the JHDR chunk were present, immediately preceding the first JDAT chunk. If the first JNG chunk is not a JDAT chunk, then either the IJNG or JHDR must be present to introduce the JNG datastream.

6.1.10 DROP Drop chunks

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

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

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

6.1.11 DBYK Drop chunks by keyword

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

Polarity:   1 byte (unsigned integer).
              0: Only.
              1: All-but.

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

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

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

Use caution when printing or displaying keywords (Refer to Security considerations, below, Chapter 16).

6.1.12 ORDR Ordering restrictions

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

```
Chunk name: 4 bytes (ASCII text).
Order type: 1 byte.
              0: Anywhere.
              1: After IDAT and/or JDAT.
              2: Before IDAT and/or JDAT.
              3: Before IDAT, but not before PLTE.
              4: Before IDAT, but not after PLTE.
etc.
```

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

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

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

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

6.2 Ancillary Delta-PNG chunks

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

6.2.1 gAMA, cHRM, iCCP, sRGB Color space chunks

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

6.2.2 oFFs and pHYs

MNG viewers must ignore oFFs and pHYs chunks that appear inside a PNG or JNG datastream or are inherited from the MNG top level. MNG editors are expected to treat them as if they were unknown copy-safe chunks.

6.2.3 Other ancillary PNG chunks

Any other ancillary PNG chunks that the decoder recognizes when processing a PNG datastream should also be recognized and handled when processing a delta-PNG datastream. Any chunks that it does not recognize should be processed as instructed by the ORDR, DROP, and DBYK chunks. MNG viewers are free to ignore any ancillary chunks, while MNG editors should handle them in accordance with the copying rules.

6.3 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 above (Paragraph 6.1.12).

7 Chunk Copying Rules

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

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

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

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

Changes to the MHDR chunk do not affect the copy-safe status of any other chunk.

The SAVE, SEEK, and TERM chunks are not considered to be a part of any segment. Changes to the data in the SAVE or SEEK chunks do not affect the copy-safe status of any other chunks. Adding or removing a SEEK chunk affects the copy-safe status of unknown chunks in the newly-merged or newly-separated segments. Adding, removing, or changing the TERM chunk has no effect on the copy-safe status of any chunk.

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 datastream can have ordering rules only with respect to the SAVE, SEEK, SHOW, and PAST chunks, DHDR-IEND, BASI-IEND, IHDR-IEND, JHDR-IEND sequences, or with respect to any other critical "header-end" sequence that might be defined in the future that could contain IDAT or similar chunks.

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

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

8 Minimum Requirements for MNG-Compliant Viewers

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

Anything less than this level of support requires subsetting. Applications that provide less than minimal MNG support should check the MHDR "simplicity profile" for the presence of features that they are unable to support. A specific subset, in which "complex MNG features" and JNG are absent, is called *"MNG-LC"*. In MNG-LC datastreams, bit 0 of the simplicity profile must be 1 and bits 2 and 4 must be 0. Another subset is called *"MNG-VLC"*. In MNG-VLC datastreams, "simple MNG features" are also absent, and bit 1 must therefore also be 0.

Subsets are useable when the set of MNG datastreams to be processed is known to be (or is very likely to be) limited to the feature set in MNG-LC or MNG-VLC. Limiting the feature set in a widely-deployed WWW browser to anything less than MNG with 8-bit JNG support would be highly inappropriate.

Some subsets of MNG support are listed in the following table, more or less in increasing order of complexity.

Level of support	MHDR Profile bits 31	30–6	5	4	3	2	1	0	Profile value	
MNG–VLC without transparency	0	0	0	0	0	0	0	1	1	
MNG–VLC	0	0	0	0	1	0	0	1	9	
MNG–VLC with JNG	0	0	0	1	1	0	0	1	25	
MNG–LC	0	0	0	0	1	0	1	1	11	
MNG–LC with JNG	0	0	0	1	1	0	1	1	19	
MNG without JNG and Delta–PNG	0	0	0	0	1	1	1	1	15	
MNG without Delta–PNG	0	0	0	1	1	1	1	1	31	
MNG without JNG	0	0	1	0	1	1	1	1	47	
MNG	0	0	1	1	1	1	1	1	63	(or 0)
MNG with 12–bit JNG support	0	0	1	1	1	1	1	1	63	(or 0)

One reasonable path for an application developer to follow might be to develop and test the application at each of the following levels of support in turn:

1. MNG-VLC,
2. MNG-LC,
3. MNG-LC with JNG,

4. MNG.

An equally reasonable development path might be

1. MNG-VLC with JNG,
2. MNG-LC with JNG,
3. MNG.

On the other hand, a developer working on an application for storing multi-page fax documents might have no need for more than "MNG-VLC without transparency".

We are allowing conformant decoders to skip twelve-bit JNGs because those are likely to be rarely encountered and used only for special purposes. There is no profile flag to indicate the presence or absence of 12-bit JNGs.

8.1 Required MNG chunk support

MHDR
: The `ticks_per_second` must be supported by animation viewers. The simplicity profile, frame count, layer count, and nominal play time can be ignored. Decoders that provide less than minimal support can use the simplicity profile to identify datastreams that they are incapable of processing.

MEND
: The MEND chunk must be recognized but does not require any processing other than completing the last frame.

Global PLTE and tRNS
: Must be fully supported. Bit 1 of the simplicity profile can be used to promise that these chunks are not present.

LOOP, ENDL
: The `repeat_count` must be supported. The `nest_level` should be used as a sanity check but is not required. When `iteration_min=1` either explicitly or when it is omitted and `termination_condition != 0`, the LOOP chunk and its ENDL chunk can be ignored (bit 2 of the simplicity profile can be used to promise that this is true for all loops).

DEFI, CLON
: Must be fully supported. All objects can be treated as "concrete" if the decoder does not wish to take advantage of the distinction between "abstract" and "concrete". Bit 2 of the simplicity profile can be used to promise that the CLON chunk is not present and that if the DEFI chunk is present it only defines object 0, which does not need to be stored. Bit 1 of the simplicity profile can be used to promise that the DEFI chunk is not present.

BASI, BACK, DISC, PAST
: Must be fully supported. Bit 2 of the simplicity profile can be used to promise that the DISC

and PAST chunks are not present, and that if the BACK chunk is present it does not define a background image. Bit 1 can be used to promise that the BACK chunk, if present, can be ignored.

FRAM
The `framing_mode` and clipping parameters must be supported. The `interframe_delay` must be supported except by single-frame viewers. The `sync_id` and `timeout` data can be ignored.

MOVE, CLIP, SHOW
Must be fully supported. Bit 2 of the simplicity profile can be used to promise that none of these chunks are present.

SAVE and SEEK
Partial support is required: All existing objects must be marked "frozen" when the SAVE chunk is processed, so that unneeded objects can be discarded when the SEEK chunk or an empty DISC chunk is processed. The SEEK chunk must be processed as if it were an empty DISC chunk, as a minimum. Chunk information need only be "saved" and "restored" when the viewer is able to skip or jump to random SEEK chunk locations. The optional index can be ignored. Slide-show controllers may wish to support SAVE and SEEK fully. Bit 2 of the simplicity profile can be used to promise that the SAVE and SEEK chunks can be ignored entirely (because there will be nothing to discard).

TERM
Must be recognized but can be ignored.

8.2 Required PNG chunk support

IHDR, PLTE, IDAT, IEND
All PNG critical chunks must be fully supported. All values of `color_type`, `bit_depth`, `compression_method`, `filter_method` and `interlace_method` must be supported (interlacing, as in PNG, need not necessarily be displayed on-the-fly; the image can be displayed after it is fully decoded). The alpha-channel must be supported, at least to the degree that fully opaque pixels are opaque and fully transparent ones are transparent. It is recommended that alpha be fully supported. Alpha can be ignored if bit 3 of the simplicity profile is 0.

tRNS
The PNG tRNS chunk, although it is an ancillary chunk, must be supported in MNG-compliant viewers, at least to the degree that fully opaque pixels are opaque and fully transparent ones are transparent. It is recommended that alpha data from the tRNS chunk be fully supported in the same manner as alpha data from an RGBA image or a JNG with an alpha channel contained in IDAT chunks. The tRNS chunk can be ignored if bit 3 of the simplicity profile is 0.

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

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

8.3 Required JNG chunk support

Bit 4 of the simplicity profile can be used to promise that JNG chunks are not present. Viewers that choose not to support JNG can check this bit before deciding to proceed. MNG-compliant decoders must support JNG, but MNG-LC and MNG-VLC decoders are not required to support JNG.

JHDR, JDAT, IDAT, JSEP, IEND
: All JNG critical chunks must be fully supported. All values of `color_type`, `bit_depth`, `compression_method`, `filter_method` and `interlace_method` must be supported (interlacing, as in PNG, need not necessarily be displayed on-the-fly; the image can be displayed after it is fully decoded). The alpha-channel must be supported, at least to the degree that fully opaque pixels are opaque and fully transparent ones are transparent. It is recommended that alpha be fully supported.

JNG ancillary chunks
: All JNG ancillary chunks can be ignored, and do not even have to be recognized.

JNG image sample depth
: Only `image_sample_depth=8` must be supported. The JSEP chunk must be recognized and must be used by minimal decoders to select the eight-bit version of the image, when both eight-bit and twelve-bit versions are present, as indicated by `image_sample_depth=20` in the JHDR chunk. When `image_sample_depth=12`, minimal decoders are not obligated to display anything. Such decoders can choose to display nothing or an empty rectangle of the width and height specified in the JHDR chunk. This can be done by processing the JNG as though a viewable transparent `BASI` object had appeared:

```
BASI width height 1 4 0 0 0 0 00 00 00 00 1
IEND
```

8.4 Required Delta-PNG chunk support

MNG-compliant decoders are required to support Delta-PNG, but MNG-LC and MNG-VLC decoders are not.

DHDR, PROM, IHDR, IDAT, IPNG, JHDR, JDAT, IJNG, JSEP, PLTE, tRNS, IEND, PPLT
: Must be fully supported. Bit 2 or bit 5 of the simplicity profile can be used to promise that none of these are present (because the DHDR chunk is absent). Bit 4 of the simplicity profile can be used to promise that the JHDR and JDAT chunks are not present.

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

Ancillary chunks

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

9 Recommendations for Encoders

The following recommendations do not form a part of the specification.

9.1 Use a common color space

It is a good idea to use a single color space for all of the layers in an animation, where speed and fluidity are more important than exact color rendition. This is best accomplished by defining a single color space at the top level of MNG, using and either an sRGB chunk or the gAMA and cHRM chunks and perhaps the iCCP chunk, and removing any color space chunks from the individual images after converting them to the common color space.

When the encoder converts all images to a single color space before putting them in the MNG datastream, this will allow decoders to improve the speed and consistency of the display.

For single-frame MNG datastreams, however, where decoding speed is less important and exact color rendition might be more important, it is best to leave the images in their original color space, as recommended in the PNG specification, to avoid any loss of data due to conversion, and to retain the individual color space chunks if the images have different color spaces.

9.2 Use the right framing mode

Always use framing mode 1 or 2 when all of the images are opaque. This avoids unnecessary screen clearing, which can cause flickering.

9.3 Immediate frame sync point

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 `interframe_delay=0`, `timeout`, and `sync_id`, and the second establishing the synchronization point:

```
FRAM 2 0 1 1 0 1 0000 timeout sync_id
FRAM 0 name
```

9.4 Embedded images in LOOPs

Embedded images should not be enclosed in loops unless absolutely necessary. It is better to store them ahead of time and then use SHOW chunks inside the loops. Otherwise, decoders will be forced to repeatedly decode them. See Examples 2, 8, 11, and 12, below (Chapter 17).

9.5 Including optional index in SAVE chunk

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

9.6 Interleaving JDAT and IDAT chunks

When a JNG datastream contains an alpha channel, and the file is intended for transmission over a network, it is useful to interleave the IDAT and JDAT chunks. In the case of sequential JPEG, the interleaving should be arranged so that the alpha data arrives more or less in sync with the color data for the scanlines. In the case of progressive JPEG, the alpha data should be interleaved with the first JPEG pass, so that *all* of the alpha data has arrived before beginning the second JPEG pass.

10 Recommendations for Decoders

10.1 ENDL without matching LOOP

If a decoder reads an ENDL chunk for which the matching LOOP chunk is missing, or has been skipped for some reason, any active loops with a higher `nest_level` should be terminated, and processing can resume after the next SEEK chunk. Simple viewers that do not process the SAVE chunk should abandon the MNG datastream. See above.

10.2 Note on compositing

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

Pixels in PNG and JNG images are represented using gamma-encoded RGB (or gray) samples along with a linear alpha value. Alpha processing can only be performed on linear samples. This chapter assumes that R,

G, B, and A values have all been converted to real numbers in the range [0..1], and that any gamma encoding has been undone.

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

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

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

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

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

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

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

As mentioned in the PNG specification, the equations become much simpler when no pixel has an alpha value other than 0.0 or 1.0, and the RGB samples need not be linear in that case.

10.3 Retaining object data

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

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

- The object attribute set (potential visibility, location, clipping boundary data from the DEFI, MOVE, CLIP, and SHOW chunks, and pointer to an object buffer).
- The pixel data and the values associated with other recognized PNG chunks such as PLTE and gAMA, subject to the chunk copying rules and the DROP/DBYK chunks. If the object is "abstract", the data can be stored in any convenient form. If it is "concrete", it must be stored in an object buffer in a manner that would permit the complete restoration of the original PNG or JNG file or its equivalent.
- The most recent values of `target_x` and `target_y`, if the object was the destination of a PAST chunk.

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

10.4 Decoder handling of fatal errors

When a fatal error is encountered, such as a bad CRC or an unknown critical MNG chunk, minimal viewers that do not implement the SAVE/SEEK mechanism should simply abandon the MNG datastream. More capable MNG viewers should attempt to recover gracefully by abandoning processing of the segment and searching for a SEEK chunk. If such errors occur before the SAVE chunk is reached, the viewer should 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 object should be discarded. If a bad CRC is found, indicating a corrupted datastream, the entire segment should be abandoned, as above.

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

10.5 Decoder handling of interlaced images

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

10.6 Decoder handling of palettes

When a PLTE chunk is received, it only affects the display of the PNG datastream that includes or inherits it. Decoders must take care that it does not retroactively affect anything that has already been decoded.

If PLTE or PPLT is present in a Delta-PNG datastream, the new palette is used in displaying the image defined by the Delta-PNG; if no IDAT chunk is present and the image type is PNG indexed-color, then the resulting image is displayed using the old pixel samples as indices into the new palette, which provides a "palette animation" capability.

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

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

10.7 Behavior of single-frame viewers

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

10.8 Clipping

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

Frame width and frame height

The `frame_width` and `frame_height` are defined in the MHDR chunk and cannot be changed by any other MNG chunk.

This is the only type of clipping available in MNG-VLC datastreams.

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

In the case of a MNG datastream that consists of a PNG or JNG datastream, with the PNG or JNG signature, the `frame_width` and `frame_height` are defined by the `width` and `height` fields of the IHDR (or JHDR) chunk.

Subframe clipping boundaries
The subframe clipping boundaries are optionally defined in the FRAM chunk, and cannot be changed within a subframe. When the framing mode is 3 or 4, viewers must, prior to displaying each subframe, clear the area within the subframe clipping boundaries to the background color, and display the background image if one has been defined, thus creating a separate layer at the beginning of each subframe. Viewers must not change any pixels outside the subframe boundaries; encoders must be able to rely on the fact that the part of the display that is outside the subframe clipping boundaries (but inside the area defined by `frame_width` and `frame_height`) will remain on the display from frame to frame without being explicitly redisplayed. See Example 8, which displays a large background image once, and then, in each frame, only redisplays the portion of the background surrounding the moving sprite.

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

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

11 Recommendations for Editors

11.1 Editing datastreams with optional index

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

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

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

11.2 Handling LOOP and TERM chunks

Editors that create a series of PNG or JNG datastreams from a MNG datastream should check the termination condition of any LOOP chunks and execute loops only `iteration_min` times. The loop created by the TERM chunk should be executed only once.

12 Miscellaneous Topics

12.1 File name extension

On systems where file names customarily include an extension signifying file type, the extension `.mng` is recommended for MNG (including MNG-LC and MNG-VLC) files. Lowercase `.mng` is preferred if file names are case-sensitive. The extension `.jng` is recommended for JNG files.

12.2 Internet media type

When and if the MNG format becomes finalized, the MNG authors intend to register `video/mng` as the Internet Media Type for MNG [RFC-2045], [RFC-2048].

At the date of this document, the media type registration process had not been started. It is recommended that implementations also recognize the interim media type `video/x-mng`.

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

12.3 Uniform Resource Identifier (URI)

Segments, subframes, and objects are externally accessible via named SEEK, eXPI, and FRAM chunk names. They can be referred to by URI, as in

```
SRC=file.mng#segment_name
SRC=file.mng#subframe_name
SRC=file.mng#snapshot_name
SRC=file.mng?segment_name#segment_name
SRC=file.mng?snapshot_name#snapshot_name
```

When the URI specializer ("#" or "?") is "#", and the fragment identifier (the string following the specializer) is the name of a segment, i.e., a named SEEK chunk, the viewer should display the animation from the beginning of the named segment up to the next segment. When it refers to a subframe or an image, i.e., a named FRAM or eXPI chunk, it should display the single frame (as it exists when the next FRAM chunk is encountered) or image that is identified by the fragment identifier. If the SAVE chunk is present and contains the optional index, this can help the client find the needed segment quickly.

When the URI specializer is "?" (server side query), the "query component" is the string following the "?" specializer and up to but not including the "#" if the "#" specializer is also present. The server should find the segment that is named in the query component or the segment that contains the subframe or image named in the query component, and it should return a datastream consisting of:

- all chunks prior to the SAVE chunk,

- an empty SAVE chunk,
- the SEEK chunk for the segment being returned,
- all chunks in the segment, and
- a MEND chunk.

If no SAVE chunk is present, the server must simply return the entire MNG datastream. Servers that are unwilling to parse the MNG datastream and are unconcerned about bandwidth can return the entire MNG datastream even when the SAVE chunk is present. Authors should defend against this behavior by including both a query and a fragment in the URI even when a segment is being requested.

The client can process this as a complete MNG datastream, either displaying the entire segment, if no fragment identifier is present, or extracting the segment, frame or image that is named in a fragment identifier and displaying it, if a fragment identifier is present (a fragment identifier must be present if a frame or image is being requested). To "extract a frame" means to decode the returned datastream through the end of the frame that contains the named subframe and to display the result as a single still image. If the named subframe does not cover the entire frame, pixels from the background and from earlier subframes must be included in the resulting composition.

A part of the MNG datastream can also be requested by timecode, as in

```
SRC=file.mng#clock(10s-20s)
SRC=file.mng#clock(0:00-0:15)
SRC=file.mng?clock(0:00-0:15)#clock(0:00-0:15)
```

or by frame number, as in

```
SRC=file.mng#frame(10)
SRC=file.mng#frames(30-60)
SRC=file.mng?frames(30-60)#frames(30-60)
```

The timecode must consist of starting and ending clock values, as defined in the W3C SMIL recommendation, separated by a hyphen (ASCII code 45).

When the URI specializer is "#", the viewer should play that part of the animation beginning and ending at the requested times, measuring from zero time at the beginning of the MNG datastream, or beginning and ending with the specified frame numbers. To do this it must start with the segment containing the requested time and decode any part of the segment up to that time, composing but not displaying the frames; this will provide the background against which the desired frames are displayed.

When the URI specializer is "?", the server can send the entire MNG datastream, or, preferably, it should construct a complete MNG file containing:

- the chunks preceding the SAVE chunk,
- the SAVE chunk itself with an optional index that gives the starting time and starting frame number of the first SEEK chunk that is sent, and

- the one or more consecutive set of segments, with their SEEK chunks, that contain the animation beginning and ending at the requested times, or frame numbers, at the proper framing rate.

If the server does not send the entire MNG datastream, and the first segment after the SAVE chunk is not sent but a later segment *is* sent, the optional index must be written even if it does not exist in the source file. The index must contain at least one "type 0" entry that gives the nominal start time and frame number for the first segment that is sent after the SAVE chunk. The offset field can be set to zero and the segment name can be omitted.

The query component should always be repeated as a fragment identifier, so clients can find the requested item in case the server sends more than what was requested.

MNG datastreams should not contain segment, subframe, or image names that begin with the case-insensitive strings "CLOCK(", "FRAME(", or "FRAMES(", which are reserved for use in URI queries and fragments.

See [RFC-2396] and the W3C SMIL recommendation at http://www.w3.org/TR/.

13 Rationale

This (incomplete as of version 0.96) section does not form a part of the specification. It provides the rationale behind some of the design decisions in MNG.

Interframe delay

Explain why the interframe delay has to be provided *before* the subframe images are defined, instead of having a simpler DELA chunk that occurs in the stream where the delay is wanted.

DHDR delta types

Some delta types are not allowed when the parent object is a JNG image. Explain why types 4 and 6 (pixel replacement and color channel replacement) are not allowed under these circumstances.

JNG alpha channel

It has been suggested that the JNG alpha data should optionally be transmitted as a grayscale JPEG. Why not?

We will be running some tests with a proposed JDAA chunk to evaluate this suggestion. But we are reluctant to add new critical chunks to the JNG specification now.

Global JPEG tables

It has been suggested that a new global MNG chunk, JTAB, be defined to hold global JPEG quantization and huffman tables that could be inherited by JNG datastreams from which these have been omitted. This has not been tested, and we are reluctant to add new critical chunks to the MNG specification now.

14 Revision History

14.1 Version 0.97

Released 28 February 2000.

Minor editorial changes only. A new example was added.

14.2 Version 0.96

Released 18 July 1999.

The changes that are not simple editorial changes were approved by votes of the PNG Development group that closed 16 July 1999 (pHYg and change to treatment of the pHYs chunk), 14 July 1999 (global bKGD and sBIT) and 25 June 1999 (change to LOOP chunk and treatment of the DEFI chunk and nonviewable objects).

- An object "comes into existence" when it is named in a DEFI chunk instead of later, when the corresponding embedded image is received. This makes it possible to MOVE or CLIP objects whose object buffer does not yet exist.
- The special treatment of the object attributes set for object 0 was eliminated.
- Any attempt to display a nonviewable object must be ignored and not treated as an error. The restriction that a nonviewable object must not be made potentially visible was removed.
- Any nonviewable object included in the list of objects to be processed by the SHOW chunk must be ignored and not treated as an error (in MNG-0.95 and earlier, the SHOW chunk would change its visibility but not display it).
- If fields are omitted from the DEFI chunk, values are inherited from a previous DEFI chunk, if one was present. In MNG-0.95, such fields assumed specified default values. In this version, the default values are only used if no prior DEFI chunk with the same object_id was present or if the prior DEFI chunk has been discarded.
- The termination_condition byte of the LOOP chunk was extended to include a "cacheable" bit.
- Revised wording of paragraph 3.3 to describe "viewable objects" as well as "viewable object buffers".

- Clarified that an image is displayed immediately if it is the subject of a CLON chunk with do_not_show==0.
- Revised Examples 6, 7, 9, 13 and 14.
- Changed "`JDAT_sample_depth`" to "`image_sample_depth`" and "`IDAT_sample_depth`" to "`alpha_sample_depth`", etc.
- Started a Rationale section.
- Started a Revision History section.
- Added the pHYg chunk and changed the meaning of the global pHYs chunk.
- Added the global bKGD and sBIT chunks.

14.3 Version 0.95

Initial public release, approved by the PNG Development Group on 11 May 1999.

15 References

[ISO/IEC-10918-1]
International Organization for Standardization and International Electrotechnical Commission, "Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines" ISO/IEC IS 10918-1, ITU-T T.81.
See also Pennebaker, William B., and Joan L. Mitchell, "JPEG : Still Image Data Compression Standard" Van Nostrand Reinhold, ISBN:0442012721, September 1992

[JFIF]
C-Cube Microsystems, "JPEG File Interchange Format, Version 1.02", September 1992.

[PNG]
Boutell, T., et. al., "PNG (Portable Network Graphics Format) Version 1.0", RFC 2083,
ftp://ftp.isi.edu/in-notes/rfc2083.txt also available at
ftp://swrinde.nde.swri.edu/pub/png/documents/. This specification has also been published as a W3C Recommendation, which is available at
http://www.w3.org/TR/REC-png.html.

See also the PNG-1.2 draft:
Randers-Pehrson, G., et. al., "PNG (Portable Network Graphics Format) Version 1.2", which is available at
ftp://swrinde.nde.swri.edu/pub/png/documents/.

[PNG-EXT]
Extensions to the PNG 1.2 Specification,
ftp://swrinde.nde.swri.edu/pub/png/documents/pngext-*.

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

[RFC-2045]
Freed, N., and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, Innosoft, First Virtual, November 1996.
ftp://ftp.isi.edu/in-notes/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.
ftp://ftp.isi.edu/in-notes/rfc2048.txt

[RFC-2396]
Berners-Lee, T., R. Fielding, U. C. Irvine, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, MIT/LCS, Xerox Corporation, University of Minnesota, August 1998.
ftp://ftp.isi.edu/in-notes/rfc2396.txt

16 Security Considerations

Security considerations are addressed in the PNG specification.

An infinite or just overly long loop could give the appearance of having locked up the machine, as could an unreasonably long interframe delay or a misplaced `sync_id` with a long `timeout` value. Therefore a decoder should always provide a simple method for users to escape out of a loop or delay, either by abandoning the MNG entirely or just proceeding 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.)

Some people may experience epileptic seizures when they are exposed to certain kinds of flashing lights or patterns that are common in everyday life. This can happen even if the person has never had any epileptic seizures. All graphics software and file formats that support animation and/or color cycling make it possible to encode effects that may induce an epileptic seizure in these individuals. It is the responsibility of authors and software publishers to issue appropriate warnings to the public in general and to animation creators in particular.

No known additional security concerns are raised by this format.

17 Appendix: Examples

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

17.1 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.
sRGB 2
IHDR 720 468 8 0 0 0 0  # Width and Height, etc.
IDAT ...
IEND
```

If `file.png` contains an sRGB chunk and also gAMA and cHRM chunks that are recommended in the PNG specification for "fallback" purposes, you can remove those gAMA and cHRM chunks from `file.mng` because any MNG viewer that processes the gAMA chunk is also required to recognize and process the sRGB chunk, so those chunks will always be ignored. Any MNG editor that converts the MNG file back to a PNG file is supposed to insert the recommended gAMA and cHRM chunks.

17.2 Example 2: A 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.
     1         # 1 tick per second.
     5 4 4     # Layers, frames, play time
     7         # Simplicity profile
DEFI 1 0 0 IHDR ... IDAT ... IEND # Four PNG datastreams
DEFI 2 0 0 IHDR ... IDAT ... IEND # are read and stored
DEFI 3 0 0 IHDR ... IDAT ... IEND # and are displayed as
DEFI 4 0 0 IHDR ... IDAT ... IEND # they are read.
SAVE  # This is needed so we can place TERM before SEEK.
TERM 3 0 120 10   # When done, repeat from TERM 10 times.
SEEK
```

```
SHOW
MEND
```

17.3 Example 3: A simple slideshow

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 720 468 1 # Width and height, 1 tick per second.
     6 5 5     # Layers, frames, play time.
     1         # Simplicity profile (MNG-VLC)
FRAM 1 0 2 2 0  2 1 600 0 # Set interframe_delay to 1,
  # 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
```

17.4 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-IEND datastreams 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 6 5 5 15 # 1 tick per second, complex, no JNG.
DEFI 1 1 1      # Define image 1, invisible, concrete.
IHDR ...  IDAT ...  IEND
FRAM  1 0  2 2 0 2  1 600 0 # set interframe_delay to 1,
  # 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
```

17.5 Example 5: A simple movie

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 720 468   # Width and height.
     30 6 5 15 # 30 ticks per second.
     47        # Delta-PNG, transparent, complex
tEXtTitle\0Sample Movie
fPRI 0 128     # Default frame priority is "medium".
FRAM 1 0  2 0 0 0  3 # Set interframe_delay to 1/10 sec.
DEFI 1 0 1       # Set default image to 1 (concrete).
SAVE
SEEK "start"

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

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

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

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

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

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

17.6 Example 6: A single composite frame

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

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 1024 512 0 # Width, height, ticks per second
     4 1 0 47   # Layers, frames, time, simplicity
BACK 16448 16448 52800 1 # Must use sky blue background.

PLTE ...     # Define global PLTE
gAMA 50000   # Define global gAMA
DEFI 1 1     # Define invisible abstract thumbnail image.
IHDR 64 64 4 3 0 0 0 PLTE IDAT ... IEND # use global PLTE
eXPI 1 "thumbnail 1"
DEFI 1 1       # Also define a larger thumbnail.
IHDR 96 96 4 3 0 0 0 PLTE IDAT ... IEND # use global PLTE
eXPI 1 "thumbnail 2"
DISC           # Discard the thumbnail image.

FRAM 4 "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.
IDAT ...
IEND                 # End of image.

CLON 1 2 0 1 0 0 518 6 # Make full invisible concrete clone.
SHOW 2 2 3           #  Mark it for immediate display during
                     #  the upcoming delta-PNG operation.
DHDR 2 1 7           #  Modify it (no change to pixels).
ORDR faLT 2          #  Establish chunk placement.
gAMA 100000   # Local gamma value is 100000 (gamma=1.0).
tEXtComment\0The faLT chunk is described in ftp://swrinde...
faLT ...             # Apply pseudocolor to parent image.
IEND                 # End of image.
DEFI 3 0 0 900 400   # Overlay near lower right-hand corner.
IHDR 101 101 2 3 ...
PLTE ...             # Use a local PLTE and global gAMA.
tRNS ...             # It is transparent (maybe a logo).
IDAT ...             # Note that the color type can differ
IDAT ...             # from that of the other images.
IEND                 # End of image.

MEND                 # End of MNG datastream.
```

17.7 Example 7: A movie with sprites

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 512 30 0 0 0 47  # Start of MNG datastream.
FRAM 2 "frame 1" 0  2 0 0 0  3 # First frame
                 # sets interframe_delay=3 ticks.
DEFI 1           # Define image 1 (abstract, LOCA 0 0).
IHDR 512 512 ... # It is a full-display PNG image.
etc              # Chunks according to PNG spec.
IEND             # SHOW 1 is implied by DEFI 1.
DEFI 2 0 1 300 200 # Define image 2, concrete.
IHDR 32 32 ...   # It is a small PNG.
gAMA 50000
IDAT ...
IEND
FRAM 0 "frame 2"   # Start new frame.
                 # New location for image 1 is still 0,0.
SHOW 1           # Display image 1 from previous frame.
MOVE 2 2 1 10 5  # New (delta) location for image 2.
SHOW 2           # Retrieve image 2 from previous frame,
CLON 2 3 0 1 0   # make a full clone of it as image 3.
     0 400 500   # Location for image 3.
DHDR 3 1 7 0 0 0 0 # Modify image 3 (no change to pixels).
tRNS ...         # Make it semitransparent.
IEND             # SHOW 3 is implied by CLON visibility.
FRAM 0 "frame 3" # Next frame (repeat this FRAM-SHOW 1 3
                 #   sequence with different locations to
                 #   move the images around).
                 # New location for image 1 is still 0,0.
MOVE 2 2 1 10 5  # New (delta) location for image 2.
MOVE 3 3 1 5 -2  # New location for image 3.
SHOW 1 3         # Show images 1 through 3.
FRAM 0 "frame 4" # Another frame.
etc.
FRAM 0 "frame 99"
etc.             # More frames.
MEND             # End of MNG datastream.
```

17.8 Example 8: A movie with an animated sprite

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 512 30 0 0 0 15 # Start of MNG datastream.
FRAM 2 "frame 1" 0  2 0 0 0  3 # First frame.
DEFI 1 IHDR 512 512 ... # Background PNG image.
etc ... IEND     # Chunks according to PNG spec.

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

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

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

17.9 Example 9: "Fading in" a transparent image

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

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 64 64 30 0 0 0 47 # Width, height, ticklength, ....
BACK 52800 52800 52800 # "Browser gray" default background.

FRAM 3 0  2 0 0 0  3 #  Set interframe_delay=3 ticks.  Use
            # framing mode 3 so background gets restored.
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 concrete copy of the image
            # that will be modified during the low-priority
            # part of the datastream.  It is a full clone.
DHDR 2 1 7  # No change to pixel data.
tRNS 0 0 0 0 0 0 ...  # Make all pixels fully transparent.
IEND
SHOW 2 2 3  # Make it visible but don't show it now.

LOOP 0 0 15
DHDR 2 1 7  # A Delta-PNG.
            # Delta-type 7 means no change to pixels.
PPLT 1 10 3 16 16 16 16 ... # Increment all alphas except
IEND                        # for entry 0 by 16.
SHOW 2
ENDL 0      # Nontransparent pixel alpha=15, 31, ... 240.

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

17.10 Example 10: Storing three-dimensional images

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

```
\212 M N G \r \n ^z \n # MNG signature.
MHDR 150 150 1   # Width, height, ticklength.
     0 0 0 47    # Layers, frames, time, simplicity.
tEXtTitle\0Weather modeling results
tEXtComment\0The xxSC, yySC, zzSC, and ttSC chunks
 in this file are written according to the Proposed
 chunk specifications version 19970203 and Sci-Vis
 chunks specification version 19970203 available at
 ftp://swrinde.nde.swri.edu/pub/png-group/documents/
xxSC kch\0 [sig\0] kilometers\0 0\0 150
yySC kch\0 [sig\0] kilometers\0 0\0 150
zzSC kch\0 [sig\0] Height (kilometers)\0 0\0 15
ttSC kch\0 [sig\0] Time (hours)\0 0\0 24
pCAL kch\0 0 255 0 2 Degrees Celsius\0 0\0 45
DEFI 1 0 1       # All images will have image = 1
SAVE             # and be visible and "concrete".
SEEK
FRAM 2      # Initial composite image.
IHDR 150 150 16  # Width, height, bit depth for top layer.
     0 0 0 0     # Color, comp, filter, interlace.
IDAT ...
IEND             # No DEFI chunk, so it is image 0.
DHDR 1 1 0       # Source=0, PNG, pixel addition,
     150 150 0 0 # Block is entire image.
IDAT ...     # IHDR is omitted; everything matches top.
IEND         # IEND is also omitted.
etc.         # Repeat DHDR through IEND 148 more times.
SEEK
FRAM         # End of first block.
etc.         # Repeat FRAM through SEEK 19 more times.
SEEK
MEND         # End of MNG.
```

17.11 Example 11: Tiling

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 1024 768 0 # Start of MNG datastream.
     98 1 0 15  # Layers, frames, time, simplicity.
FRAM 2
DEFI 1 1 0 0 -64 # Set up an offscreen "abstract" copy
IHDR 128 64 ... PLTE ... IDAT ... IEND  # of the tile.
LOOP 0 0 12   # Y loop -- make 12 rows of tiles.
MOVE 1 1 1 0 64 # Move the first copy down 64 rows.
SHOW 1        # Display it.
CLON 1 2 1    # Create a partial clone of the tile.
LOOP 1 0 7    # X loop - 7 additional columns.
MOVE 2 2 1 0 128 # Move it to the right 128 columns.
SHOW 2        # Use the second copy.
ENDL 1
ENDL 0
MEND
```

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 1024 768 0 # Start of MNG datastream.
     3 1 0  15  # Layers, frames, time, simplicity.
DEFI 1 1 # Set up an offscreen "abstract" copy
IHDR 128 64 ...  PLTE ... IDAT ... IEND  # of the tile.
DEFI 2        # The abstract, visible, viewable image to
BASI 1024 768 8 2 0 0 0 0 0 0 0 1 #  be tiled.  Initially
IEND          # all pixels are zero.
PAST 2 0 0 0  # Destination and target location.
  # src mod orient  offset        clipping
     1   0    8     0   0 512   0 0 1024 0 768
  #  End of PAST chunk data.
MEND
```

17.12 Example 12: Scrolling

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

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 512 256 30   # Width, height, ticks per second
     6513 3257 3257 15  # Layers, frames, time, simplicity.
BACK 50000 50000 50000 0 #  advisory gray background
DEFI 1 1 0 0 256  # Define image 1 but don't display now.
                  # Initially it is offscreen, just
                  # below the 512 by 256 window.
IHDR 512 3000 1 0 ... # A PNG datastream containing the
PLTE ...          # text (or whatever) to be scrolled.
IDAT ...
IEND

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

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

Alternatively, we can declare the scrolling object to be the background and use framing-mode 3:

```
                (Same as above down to the LOOP chunk.)
BACK 50000 50000 50000 2 1 #  Advisory gray background.
                           #  Mandatory image background.
FRAM 3  0  2 0 0 0  1 # Interframe_delay = 1 tick.
LOOP 0 0 3256
MOVE 1 1 1 0 -1 # Jack background up one scanline, 3256 times.
SHOW 2     # Composite the second image over it, wait 1 tick.
ENDL 0
MEND
```

17.13 Example 13: Cycling animations

This demonstrates the use of the SHOW chunk with `show_mode=6` to create animations that cycle through a series of ten objects.

This will cycle through the ten objects in the forward direction, 100 times, unless terminated sooner by the user or the decoder.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 400 88 30  # Width, height, ticks per second
     11 1001 1001 7  # Layers, frames, time, simplicity.
DEFI 1 ...
etc.          # Define 10 objects.
DEFI 10 ...
LOOP 0 100 6 # 100 iterations, user-discretion, cacheable
SHOW 1 10 6
ENDL 0
MEND
```

This will cycle through the ten objects, back and forth, 50 times, unless terminated sooner by the user or the decoder.

```
\212 M N G \r \n ^z \n  # MNG signature.
MHDR 400 88 6  # Width, height, ticks per second
     11 901 901 7  # Layers, frames, time, simplicity.
DEFI 1 ...
etc.          # Define 10 objects.
DEFI 10 ...
CLON 11 9 1  # Make partial clones of objects 2-9
etc.          # in reverse order, as objects 11-18.
CLON 18 2 1

LOOP 0 50 6 # 50 iterations, user-discretion, cacheable
SHOW 1 18 6
ENDL 0
MEND
```

17.14 Example 14: Converting a GIF animation

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

```
begin
    write "MHDR" chunk
    saved_images := 0; Interframe_delay := 0
    First_frame := TRUE
    write "BACK" chunk
    if(loops>1) "write TERM 3 0 0 loops" chunk
    for subimage in gif89a file do
       if(interframe_delay != gif_duration) then
         interframe_delay := gif_duration
         write "FRAM 4 0 2 2 0 2 0 interframe_delay 0" chunk
         First_frame := FALSE
       else if(First_frame == TRUE)then
         write "FRAM 4" chunk
         First_frame := FALSE
       else
         write "FRAM" chunk
       endif
       if(X_loc == 0 AND Y_loc == 0) then
          write "DEFI saved_images 1 1" chunk
       else
          write "DEFI saved_images 1 1 X_loc Y_loc" chunk
       write "<image>"
       write "SHOW 0 saved_images" chunk
       if (gif_disposal_method == 0
          OR gif_disposal_method == 2) then
          /* (undefined or restore background) */
             write "DISC" chunk
             saved_images := 0
       else if (gif_disposal_method == 1) then
          /* (keep) */
          saved_images := saved_images + 1
       else if (gif_disposal_method == 3) then
          /* (restore previous) */
          write "DISC saved_images" chunk
       endif
    endfor
    write "FRAM" and "MEND" chunks
end
```

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

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

17.15 Example 15: Converting a simple GIF animation

Outline of a program to convert simple GIF animations that do not use the "restore-to-previous" disposal method to "simple" MNG (or "MNG-LC") format:

```
begin
   write "MHDR" chunk
   Interframe_delay := 0; Previous_mode := 1
   Framing_mode := 1
   if(loops>1) "write TERM 3 0 0 loops"
   write "mandatory BACK" chunk
   for subimage in gif89a file do
      if(interframe_delay != gif_duration) then
         interframe_delay := gif_duration
         write "FRAM 0 0 2 2 0 2 0 interframe_delay 0"
      endif
      if(X_loc != 0 OR Y_loc != 0) then
         write "DEFI 0 0 0 X_loc Y_loc" chunk
      endif
      write "<image>"
      if (gif_disposal_method < 1) then
         /* (none or keep) */
         Framing_mode := 1
      else if (gif_disposal_method == 2) then
         /* (restore background) */
         write "FRAM 4 0 1 0 1 0 0 L R T B"
         Previous_mode := 4; Framing_mode := 1
      else if (gif_disposal_method == 3) then
         /* (restore previous) */
         error ("can't do gif_disposal method = previous.")
      endif
      if(Framing_mode != Previous_mode) then
         write "FRAM Framing_mode" chunk
         Previous_mode := Framing_mode
      endif
   end
   write "MEND" chunk
end
```

Where "<image>" represents a PNG datastream containing a GIF frame that has been converted to PNG format.

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

17.16 Example 16: Counting layers and frames

This demonstrates the determination of the layer-count and frame-count that should be written in the MHDR chunk. For framing_modes 1 and 2, the FRAM chunks themselves don't generate layers. For framing_modes 3 and 4, they do generate layers ("B" for background), and also generate frames if there is no embedded image with which to combine the background layer. Note that every framing_mode creates a "B" layer at the beginning.

Given the following chunk stream:

```
MHDR sRGB Fn F I I I F F I I I F F I I I MEND
```

in which

```
Fn represents a FRAM chunk with framing_mode n
F represents an empty FRAM chunk;
I represents an embedded image
```

This table shows the layer count and frame count for each of the four possible values of framing-mode:

Framing mode	Layer count		Frame count	
1	B,I,I,I, I,I,I, I,I,I	= 10	BI,I,I, I,I,I, I,I,I	= 9
2	B,I,I,I, I,I,I, I,I,I	= 10	BIII,III,III	= 3
3	3*(B, B,I, B,I, B,I)	= 21	3*(B,BI,BI,BI)	= 12
4	3*(B,B,I,I,I)	= 15	B,BIII,B,BIII,B,BIII	= 6

17.17 Example 17: Storing an icon library

Here is an example of storing a library of icons in a MNG-LC datastream. All of the icons use the same palette, transparency, and colorspace, so these are put in global chunks at the beginning. Empty PLTE chunks in the embedded images are used to import the global palette and transparency data.

```
MHDR 96 96 1 6 5 5 11  # Profile 11 is MNG-LC
sRGB 2                 # Global sRGB
PLTE ...               # Global PLTE
tRNS 0                 # Global tRNS
eXPI 0 "thumbnail"
IHDR 32 32 ... PLTE IDAT ... IEND
eXPI 0 "left arrow"
IHDR 96 96 ... PLTE IDAT ... IEND
eXPI 0 "right arrow"
IHDR 96 96 ... PLTE IDAT ... IEND
eXPI 0 "up arrow"
IHDR 96 96 ... PLTE IDAT ... IEND
eXPI 0 "down arrow"
IHDR 96 96 ... PLTE IDAT ... IEND
MEND
```

This is similar, but it uses Delta PNG datastreams to create modified versions by replacing the palette. This can be more storage-efficient, but requires a full MNG decoder because of the presence of Delta PNG datastreams.

```
MHDR 96 96 1 6 5 5 47  # Profile 47 is MNG without JNG
sRGB 2                 # Global sRGB
PLTE ...               # Global PLTE
tRNS 0                 # Global tRNS
eXPI 0 "thumbnail"
IHDR 32 32 ... PLTE IDAT ... IEND
SAVE
<index>
SEEK "left arrows"
DEFI 1
IHDR 96 96 ... PLTE IDAT ... IEND
eXPI 1 "red left arrow"
DHDR 1 1 7 PPLT ... IEND  # Change some palette entries.
eXPI 1 "blue left arrow"
SEEK "right arrows"
IHDR 96 96 ... PLTE IDAT ... IEND
eXPI 1 "red right arrow"
DHDR 1 1 7 PPLT ... IEND
eXPI 1 "blue right arrow"
MEND
```

18 Credits

Editor

- Glenn Randers-Pehrson randeg @ alum.rpi.edu

Contributors

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, oliver @ fromme.com
- Jean-loup Gailly, jloup @ gzip.org
- Chris Herborth, chrish @ qnx.com
- Alex Jakulin, jakulin @ acm.org
- Gerard Juyn, gjuyn @ xs4all.nl
- Neal Kettler, neal @ westwood.com
- 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 @ acm.org

- Josh M. Osborne, stripes @ va.pubnix.com
- Keith S. Pickens, ksp @ rice.edu
- Glenn Randers-Pehrson, randeg @ alum.rpi.edu
- Nancy M. Randers-Pehrson, randeg @ alum.rpi.edu
- Greg Roelofs, newt @ pobox.com
- Willem van Schaik, willem @ schaik.com
- Guy Schalnat, gschal @ infinet.com
- Paul Schmidt, pschmidt @ photodex.com
- Smarry Smarasderagd, smar @ reptiles.org
- Thomas R. Tanner, ttehtann @ argonet.co.uk
- Guido Vollbeding, guivol @ esc.de
- Tim Wegner, twegner @ phoenix.net
- Alaric B. Williams, alaric @ alaric-williams.com

Trademarks

- GIF is a service mark of CompuServe Incorporated.
- X Window System is a trademark of the Massachusetts Institute of Technology.

Document source

This document was built from the file `mng-master-20000228` on 29 February 2000.

Copyright Notice

Copyright © 1998, 1999, 2000 by: Glenn Randers-Pehrson

This specification is being provided by the copyright holder under the following license. By obtaining, using and/or copying this specification, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and distribute this specification for any purpose and without fee or royalty is hereby granted, provided that the full text of this **NOTICE** appears on *ALL* copies of the specification or portions thereof, including modifications, that you make.

THIS SPECIFICATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDER MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SPECIFICATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. COPYRIGHT HOLDER WILL BEAR NO LIABILITY FOR ANY USE OF THIS SPECIFICATION.

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

End of MNG Specification.