

# IFF ILBM

## “ILBM” IFF Interleaved Bitmap

<b>Document Date:</b>	January 17, 1986 (CRNG data updated Oct, 1988 by Jerry Morrison) (Appendix E added and CAMG updated Oct, 1988 by Commodore-Amiga, Inc.)
<b>From:</b>	Jerry Morrison, Electronic Arts
<b>Status of Standard:</b>	Released and in use

# 1. Introduction

“EA IFF 85” is Electronic Arts' standard for interchange format files. “ILBM” is a format for a 2 dimensional raster graphics image, specifically an **InterLeaved** bitplane **BitMap** image with color map. An ILBM is an IFF “data section” or “FORM type,” which can be an IFF file or a part of one. ILBM allows simple, highly portable raster graphic storage.

[I've inserted editorial remarks (text appearing between “[” and “]” brackets) at certain points in this document to update information

about the way ILBM is currently implemented or to expand on certain points that may not be clear in the original. I've also silently corrected a few typos. — *EW*

An ILBM is an archival representation designed for three uses. First, a standalone image that specifies exactly how to display itself (resolution, size, color map, etc.). Second, an image intended to be merged into a bigger picture which has its own depth, color map, and so on. And third, an empty image with a color map selection or “palette” for a paint program. ILBM is also intended as a building block for composite IFF FORMs like “animation sequences” and “structured graphics.” Some uses of ILBM will be to preserve as much information as possible across disparate environments. Other uses will be to store data for a single program or highly cooperative programs while maintaining subtle details. So we're trying to accomplish a lot with this one format.

This memo is the IFF supplement for FORM ILBM. Section 2 defines the purpose and format of property chunks bitmap header “BMHD,” color map “CMAP,” hotspot “GRAB,” destination merge data “DEST,” sprite information “SPRT,” and Commodore Amiga viewport mode “CAMG.” Section 3 defines the standard data chunk “BODY.” These are the “standard” chunks. Section 4 defines the non-standard data chunks. Additional specialized chunks like texture pattern can be added later. The ILBM syntax is summarized in Appendix A as a regular expression and in Appendix B as a box diagram. Appendix C explains the optional run encoding scheme. Appendix D names the committee responsible for this FORM ILBM standard.

Details of the raster layout are given in part 3, “Standard Data Chunk.” Some elements are based on the Commodore Amiga

hardware but generalized for use on other computers. An alternative to ILBM would be appropriate for computers with true color data in each pixel, though the wealth of available ILBM images makes import and export important.

## Reference:

*“EA IFF 85” Standard for Interchange Format Files* describes the underlying conventions for all IFF files.

Amiga® is a registered trademark of Commodore-Amiga, Inc.

Electronic Arts™ is a trademark of Electronic Arts.

Macintosh™ is a trademark licensed to Apple Computer, Inc.

MacPaint™ is a trademark of Apple Computer, Inc.

## 2. Standard Properties

ILBM has several property chunks that act on the main data chunk. The required property “BMHD” and any optional properties must appear before any “BODY” chunk. (Since an ILBM has only one BODY chunk, any following properties would be superfluous.) Any of these properties may be shared over a LIST of FORMs ILBM by putting them in a PROP ILBM. (See the [EA IFF 85](#) document.)

[BMHD is the only essential property chunk for ILBMs used with LightWave. For broader support of different image types, you may also want to support CMAP and possibly CAMG. The other property chunks (and CAMG) have Amiga-specific semantics. It's also safe to assume that you're very unlikely to encounter PROPs.]

## BMHD

The required property “BMHD” holds a BitMapHeader as defined in the following documentation. It describes the dimensions of the image, the encoding used, and other data necessary to understand the BODY chunk to follow.

```
typedef UBYTE Masking; /* Choice of masking technique. */

#define mskNone          0
#define mskHasMask       1
#define mskHasTransparentColor 2
#define mskLasso         3

typedef UBYTE Compression; /* Choice of compression
algorithm
applied to the rows of all source and mask planes.
"cmpByteRun1"
is the byte run encoding described in Appendix C. Do not
compress
across rows! */
#define cmpNone          0
#define cmpByteRun1      1

typedef struct {
    UWORD    w, h;          /* raster width & height in pixels    */
    WORD     x, y;          /* pixel position for this image    */
    UBYTE     nPlanes;       /* # source bitplanes                */
    Masking   masking;
    Compression compression;
    UBYTE     pad1;          /* unused; ignore on read, write as 0 */
} /*
```

```

        UWORD    transparentColor; /* transparent "color number"
(sort of) */
        UBYTE    xAspect, yAspect; /* pixel aspect, a ratio width :
height */
        WORD     pageWidth, pageHeight; /* source "page" size in
pixels */
    } BitMapHeader;

```

Fields are filed in the order shown. The UBYTE fields are byte-packed (the C compiler must not add pad bytes to the structure).

The fields *w* and *h* indicate the size of the image rectangle in pixels. Each row of the image is stored in an integral number of 16 bit words. The number of words per row is  $\text{words} = ((w+15)/16)$  or  $\text{Ceiling}(w/16)$ . The fields *x* and *y* indicate the desired position of this image within the destination picture. Some reader programs may ignore *x* and *y*. A safe default for writing an ILBM is  $(x, y) = (0, 0)$ .

The number of source bitplanes in the BODY chunk (see below) is stored in *nPlanes*. An ILBM with a CMAP but no BODY and  $nPlanes = 0$  is the recommended way to store a color map.

Note: Color numbers are color map index values formed by pixels in the destination bitmap, which may be deeper than *nPlanes* if a DEST chunk calls for merging the image into a deeper image.

The field *masking* indicates what kind of masking is to be used for this image. The value *mskNone* designates an opaque rectangular image. The value *mskHasMask* means that a mask plane is interleaved with the bitplanes in the BODY chunk (see below).

[These two are usually the only masking options you'll encounter.]

The value *mskHasTransparentColor* indicates that pixels in the

source planes matching `transparentColor` are to be considered “transparent.” (Actually, `transparentColor` isn't a “color number” since it's matched with numbers formed by the source bitmap rather than the possibly deeper destination bitmap. Note that having a transparent color implies ignoring one of the color registers. See CMAP, below.) The value `mskLasso` indicates the reader may construct a mask by lassoing the image as in MacPaint. To do this, put a 1 pixel border of `transparentColor` around the image rectangle. Then do a seed fill from this border. Filled pixels are to be transparent.

Issue: Include in an appendix an algorithm for converting a transparent color to a mask plane, and maybe a lasso algorithm.

A code indicating the kind of data compression used is stored in `compression`. Beware that using data compression makes your data unreadable by programs that don't implement the matching decompression algorithm. So we'll employ as few compression encodings as possible. The run encoding `byteRun1` is documented in Appendix C, below.

The field `pad1` is a pad byte reserved for future use. It must be set to 0 for consistency.

The `transparentColor` specifies which bit pattern means “transparent.” This only applies if masking is `mskHasTransparentColor` or `mskLasso`. Otherwise, `transparentColor` should be 0 (see above).

The pixel aspect ratio is stored as a ratio in the two fields `xAspect` and `yAspect`. This may be used by programs to compensate for different aspects or to help interpret the fields `w`, `h`, `x`, `y`, `pageWidth`, and `pageHeight`, which are in units of pixels. The

fraction  $xAspect/yAspect$  represents a pixel's width/height. It's recommended that your programs store proper fractions in `BitMapHeaders`, but aspect ratios can always be correctly compared with the test

$$xAspect * yDesiredAspect = yAspect * xDesiredAspect$$

Typical values for aspect ratio are width : height = 10 : 11 for an Amiga 320 × 200 display and 1 : 1 for a Macintosh™.

The size in pixels of the source “page” (any raster device) is stored in `pageWidth` and `pageHeight`, e.g. (320, 200) for a low resolution Amiga display. This information might be used to scale an image or to automatically set the display format to suit the image. Note that the image can be larger than the page.

## CMAP

The optional (but encouraged) property “CMAP” stores color map data as triplets of red, green, and blue intensity values. The  $n$  color map entries (“color registers”) are stored in the order 0 through  $n-1$ , totaling  $3n$  bytes. Thus  $n$  is the `ckSize/3`. Normally,  $n$  would equal  $2^{nPlanes}$ .

A CMAP chunk contains a `CoLorMap` array as defined below. Note that these typedefs assume a C compiler that implements packed arrays of 3-byte elements.

```
typedef struct {
    UBYTE red, green, blue;      /* color intensities 0..255 */
} ColorRegister;               /* size = 3 bytes          */

typedef ColorRegister ColorMap[n]; /* size = 3n bytes      */
```

The color components red, green, and blue represent fractional intensity values expressed in 256ths in the range 0 through 255 (e.g., 24/256). White is (255, 255, 255 — i.e., hex 0xFF, 0xFF, 0xFF) and black is (0, 0, 0). If your machine has less color resolution, use the higher order color bits when displaying by simply shifting the CMAP R, G, and B values to the right. When writing a CMAP, storage of less than 8 bits each of R, G, and B was previously accomplished by left justifying the significant bits within the stored bytes (i.e., a 4-bit per gun value of 0xF, 0xF, 0xF was stored as 0xF0, 0xF0, 0xF0). This provided correct color values when the ILBM was redisplayed on the same hardware since the zeros were shifted back out.

However, if color values stored by the above method were used as-is when redisplaying on hardware with more color resolution, diminished color could result. For example, a value of (0xF0, 0xF0, 0xF0) would be pure white on 4-bit-per-gun hardware (i.e., 0xF, 0xF, 0xF), but not quite white (0xF0, 0xF0, 0xF0) on 8-bit-per-gun hardware.

Therefore, when storing CMAP values, it is now suggested that you store full 8 bit values for R, G, and B which correctly scale your color values for eight bits. For 4-bit RGB values, this can be as simple as duplicating the 4-bit values in both the upper and lower parts of the bytes — i.e., store (0x1, 0x7, 0xF) as (0x11, 0x77, 0xFF). This will provide a more correct color rendition if the image is displayed on a device with 8 bits per gun.

When reading in a CMAP for 8-bit-per-gun display or manipulation, you may want to assume that any CMAP which has 0 values for the low bits of all guns for all registers was stored shifted rather than scaled, and provide your own scaling. Use defaults if the



The example type `Color4` represents the format of a color register in working memory of an Amiga computer, which has 4 bit video DACs. (The “:4” tells the C compiler to pack the field into 4 bits.)

Remember that every chunk must be padded to an even length, so a color map with an odd number of entries would be followed by a 0 byte, not included in the `ckSize`.

The optional property GRAB locates a “handle” or “hotspot” of the image relative to its upper left corner, e.g. when used as a mouse cursor or a “paint brush.” A GRAB chunk contains a `Point2D`.

DEST

The contents of a DEST chunk is a DestMerge structure:

2024-07-14, 0:27

```

    UBYTE pad1;    /* unused; for consistency put 0 here */
    UWORD planePick; /* how to map source planes into
destination */
    UWORD planeOnOff; /* default bitplane data for planePick */
    UWORD planeMask; /* selects which bitplanes to store into */
} DestMerge;

```

The low order depth number of bits in `planePick`, `planeOnOff`, and `planeMask` correspond one-to-one with destination bitplanes. Bit 0 with bitplane 0, etc. (Any higher order bits should be ignored.) “1” bits in `planePick` mean “put the next source bitplane into this bitplane,” so the number of “1” bits should equal `nPlanes`. “0” bits mean “put the corresponding bit from `planeOnOff` into this bitplane.” Bits in `planeMask` gate writing to the destination bitplane: “1” bits mean “write to this bitplane” while “0” bits mean “leave this bitplane alone.” The normal case (with no DEST property) is equivalent to `planePick = planeMask = 2nPlanes - 1`.

Remember that color numbers are formed by pixels in the destination bitmap (depth planes deep) not in the source bitmap (`nPlanes` planes deep).

## SPRT

The presence of an “SPRT” chunk indicates that this image is intended as a sprite. It's up to the reader program to actually make it a sprite, if even possible, and to use or overrule the sprite precedence data inside the SPRT chunk:

```

typedef UWORD SpritePrecedence; /* relative precedence, 0 is
the highest */

```

Precedence 0 is the highest, denoting a sprite that is foremost.

Creating a sprite may imply other setup. E.g. a 2 plane Amiga sprite would have `transparentColor = 0`. Color registers 1, 2, and 3 in the CMAP would be stored into the correct hardware color registers for the hardware sprite number used, while CMAP color register 0 would be ignored.

## CAMG

A CAMG chunk is specifically for the Commodore Amiga computer. All Amiga-based reader and writer software should deal with CAMG. The Amiga supports many different video display modes including interlace, Extra Halfbrite, hold and modify (HAM), plus a variety of new modes under the 2.0 operating system. A CAMG chunk contains a single long word (length=4) which specifies the Amiga display mode of the picture.

Prior to 2.0, it was possible to express all available Amiga ViewModes in 16 bits of flags (Viewport->Modes or NewScreen->ViewModes). Old-style writers and readers place a 16-bit Amiga ViewModes value in the low word of the CAMG, and zeros in the high word. The following ViewMode flags should always be removed from old-style 16-bit ViewModes values when writing or reading them:

```
EXTENDED_MODE | SPRITES | VP_HIDE | GENLOCK_AUDIO  
| GENLOCK_VIDEO  
(=0x7102, mask=0x8EFD)
```

New ILBM readers and writers, should treat the full CAMG longword as a 32-bit ModeID to support new and future display modes.

New ILBM writers, when running under the 2.0 Amiga operating system, should directly store the full 32-bit return value of the graphics function `GetVPMODEID(vp)` in the CAMG longword. When running under 1.3, store a 16-bit Viewmodes value masked as described above.

ILBM readers should only mask bits out of a CAMG if the CAMG has a zero upper word (see exception below). New ILBM readers, when running under 2.0, should then treat the 32-bit CAMG value as a ModeID, and should use the graphics `ModeNotAvailable()` function to determine if the mode is available. If the mode is not available, fall back to another suitable display mode. When running under 1.3, the low word of the CAMG may generally be used to open a compatible display.

Note that one popular graphics package stores garbage in the upper word of the CAMG of brushes, and incorrect values (generally zero) in the low word. You can screen for such garbage values by testing for non-zero in the upper word of a ModeID in conjunction with the 0x00001000 bit *NOT* set in the low word.

The following code fragment demonstrates ILBM reader filtering of inappropriate bits in 16-bit CAMG values.

```
#include <graphics/view.h>
#include <graphics/displayinfo.h>
```

```
/* Knock bad bits out of old-style CAMG modes before checking
availability.
```

```
 * (some ILBM CAMG's have these bits set in old 1.3 modes, and
should not)
```

```
 * If not an extended monitor ID, or if marked as extended but
```

missing

```
* upper 16 bits, screen out inappropriate bits now.
*/
if((!(modeid & MONITOR_ID_MASK)) ||
    ((modeid & EXTENDED_MODE)&&!(modeid &
0xFFFF0000))))
    modeid &=
        (~(EXTENDED_MODE|SPRITES|GENLOCK_AUDIO|
GENLOCK_VIDEO|VP_HIDE));

/* Check for bogus CAMG like some brushes have, with junk in
* upper word and extended bit NOT set in lower word.
*/
if(((modeid & 0xFFFF0000)&&!(modeid & EXTENDED_MODE
)))
{
    /* Bad CAMG, so ignore CAMG and determine a mode based
on
    * page size or aspect
    */
    modeid = NULL;
    if(wide >= 640) modeid |= HIRES;
    if(high >= 400) modeid |= LACE;
}

/* Now, ModeNotAvailable() may be used to determine if the
mode is available.
*
* If the mode is not available, you may prompt the user for a
mode
```

- \* choice, or search the 2.0 display database for an appropriate
- \* replacement mode, or you may be able to get a relatively compatible
- \* old display mode by masking out all bits except
- \* HIRES | LACE | HAM | EXTRA\_HALFBRITE
- \*/

[The Amiga has built-in support for interpreting the bits in a CAMG. Most of them are only meaningful on an Amiga and can otherwise be safely ignored, but two bits in the low word directly affect the interpretation of the data in the BODY chunk. Readers that attempt to support all ILBMs should test for these bits so that they can correctly translate the BODY. The bits are

```
#define CAMG_HAM 0x800 /* hold and modify */
#define CAMG_EHB 0x80 /* extra halfbrite */
```

HAM (hold-and-modify) mode allows the Amiga to display 12-bit and 18-bit RGB images using only 6 or 8 bits per pixel. HAM images store pixel values in the BODY chunk as codes that are divided into a mode in the high two bits and data in the other bits. The mode bits have the following interpretation.

- 00 - data bits are an index into the CMAP palette
- 01 - data bits contain the blue level
- 10 - data bits contain the red level
- 11 - data bits contain the green level

Unless a pixel is color-mapped (mode 00), only one of its three RGB levels is given in its code. The other two are assumed to be the same as those for the pixel to its left. If the pixel is the first one (the leftmost) in a scanline, the hold color is assumed to be (0, 0, 0). The number of data bits is 4 for standard HAM and 6 for HAM8,

It is possible for the mode to be a single bit. `nPlanes` will then be either 5 or 7. The single bit is the low bit, while the high bit is assumed to be 0, implying that only the blue level can be modified. For obvious reasons, this is rarely if ever encountered.

There are also advanced HAM variants that rely on using multiple color tables in a single image. “Sliced HAM” or “dynamic HAM” images can be recognized by the presence of SHAM, CTBL, DYCP, or PCHG chunks. Such images aren't particularly common. Displaying them on an Amiga required programming the graphics hardware at a low level.

As described in the CMAP section, the data bits should be precision-extended when the levels are decoded to 24-bit. Regardless of the number of data bits, the maximum level should translate to 255 at 8 bits per RGB channel.

The [iff](#) SDK sample, which reads and writes IFF ILBM images, includes an `unHam()` function that shows how the BODY data for a HAM image can be translated into more conventional 24-bit RGB.

Extra-Halfbrite is another Amiga variant, now quite rare. EHBs are 64-color pictures with 32-color palettes. Colors 32 to 63 are “half-bright” versions of colors 0 to 31, computed by bit shifting the RGB levels right by one. The easiest way to read EHB images is to extend the color table to include colors 32 to 63 and then interpret the BODY data as you would for any other indexed color image.]

### 3. Standard “BODY” Data Chunk

## Raster Layout

Raster scan proceeds left-to-right (increasing X) across scan lines, then top-to-bottom (increasing Y) down columns of scan lines. The coordinate system is in units of pixels, where (0,0) is the upper left corner.

The raster is typically organized as bitplanes in memory. The corresponding bits from each plane, taken together, make up an index into the color map which gives a color value for that pixel. The first bitplane, plane 0, is the low order bit of these color indexes.

A scan line is made of one “row” from each bitplane. A row is one plane's bits for one scan line, but padded out to a word (2 byte) boundary (not necessarily the first word boundary). Within each row, successive bytes are displayed in order and the most significant bit of each byte is displayed first.

[A conventional indexed color display stores the value of a pixel in a single byte (below, left). For a pixel at (x, y), the memory offset from the start of an image  $w$  pixels wide is just  $wy + x$  (ignoring any scanline padding), and the value stored there is an index into a table of RGB color records. In an ILBM, the bits of a given pixel aren't contiguous in memory. They are instead stored in separate *bitplanes*, each of which contains a single bit from a given pixel (below, right).

	plane 0:	100 <b>0</b> 0011b	
	plane 1:	001 <b>1</b> 0010b	
	plane 2:	100 <b>0</b> 1001b	
pixel:	001 <b>11</b> 0 <b>10</b> 1b	plane 3:	110 <b>1</b> 0100b
		plane 4:	010 <b>1</b> 0111b



plane 5: 10011010b

To retrieve a pixel value (naïvely), you must read bytes at different addresses (six of them in the above example), mask off all but one bit from each of them, and string the bits together. For the pixel at  $(x, y)$ , the byte offset into each bitplane is  $(wy + x) / 8$ , and the bit is  $7 - (x \bmod 8)$ . Bitplane  $n$  contains the  $n$ -th bit of the pixel value.

The [iff](#) SDK sample includes code for efficiently encoding and extracting pixel values in bitplanes.]

A “mask” is an optional “plane” of data the same size ( $w, h$ ) as a bitplane. It tells how to “cut out” part of the image when painting it onto another image. “One” bits in the mask mean “copy the corresponding pixel to the destination” while “zero” mask bits mean “leave this destination pixel alone.” In other words, “zero” bits designate transparent pixels.

The rows of the different bitplanes and mask are *interleaved* in the file (see below). This localizes all the information pertinent to each scan line. It makes it much easier to transform the data while reading it to adjust the image size or depth. It also makes it possible to scroll a big image by swapping rows directly from the file without the need for random-access to all the bitplanes.

## BODY

The source raster is stored in a “BODY” chunk. This one chunk holds all bitplanes and the optional mask, interleaved by row.

The BitMapHeader, in a BMHD property chunk, specifies the raster's dimensions  $w, h$ , and  $nPlanes$ . It also holds the masking field which indicates if there is a mask plane and the compression field which indicates the compression algorithm used. This

information is needed to interpret the BODY chunk, so the BMHD chunk must appear first. While reading an ILBM's BODY, a program may convert the image to another size by filling (with transparentColor) or clipping.

The BODY's content is a concatenation of scan lines. Each scan line is a concatenation of one row of data from each plane in order 0 through nPlanes-1 followed by one row from the mask (if masking = hasMask). If the BitMapHeader field compression is cmpNone, all h rows are exactly  $(w+15)/16$  words wide. Otherwise, every row is compressed according to the specified algorithm and their stored widths depend on the data compression.

Reader programs that require fewer bitplanes than appear in a particular ILBM file can combine planes or drop the high-order (later) planes. Similarly, they may add bitplanes and/or discard the mask plane.

Do *not* compress across rows, and don't forget to compress the mask just like the bitplanes. Remember to pad any BODY chunk that contains an odd number of bytes and skip the pad when reading.

## 4. Nonstandard Data Chunks

The following data chunks were defined after various programs began using FORM ILBM so they are “nonstandard” chunks.

### CRNG

A “CRNG” chunk contains “color register range” information. It's used by Electronic Arts' Deluxe Paint program to identify a contiguous range of color registers for a “shade range” and color

cycling. There can be zero or more CRNG chunks in an ILBM, but all should appear before the BODY chunk. Deluxe Paint normally writes 4 CRNG chunks in an ILBM when the user asks it to “Save Picture.”

```
typedef struct {
    WORD pad1;    /* reserved for future use; store 0 here */
    WORD rate;    /* color cycle rate */
    WORD flags;   /* see below */
    UBYTE low, high; /* lower and upper color registers selected
*/
} CRange;
```

The bits of the flags word are interpreted as follows: if the low bit is set then the cycle is “active,” and if this bit is clear it is not active. Normally, color cycling is done so that colors move to the next higher position in the cycle, with the color in the high slot moving around to the low slot. If the second bit of the flags word is set, the cycle moves in the opposite direction. As usual, the other bits of the flags word are reserved for future expansion. Here are the masks to test these bits:

```
#define RNG_ACTIVE 1
#define RNG_REVERSE 2
```

The fields `low` and `high` indicate the range of color registers (color numbers) selected by this `CRange`.

The field `active` indicates whether color cycling is on or off. Zero means off.

The field `rate` determines the speed at which the colors will step when color cycling is on. The units are such that a rate of 60 steps

per second is represented as  $2^{14} = 16384$ . Slower rates can be obtained by linear scaling: for 30 steps/second, rate = 8192; for 1 step/second, rate =  $16384 / 60$ , or 273.

*Warning!* One popular paint package always sets the RNG\_ACTIVE bit, but uses a rate of 36 (decimal) to indicate cycling is not active.

## CCRT

Commodore's Graphicraft program uses a similar chunk "CCRT" (for Color Cyling Range and Timing). This chunk contains a CycleInfo structure.

```
typedef struct {
    WORD direction;    /* 0 = don't cycle, 1 = cycle forwards, */
                      /* -1 = cycle backwards */
    UBYTE start, end;  /* lower, upper color registers selected */
    LONG seconds;      /* # seconds between changing colors
plus */
    LONG microseconds; /* # microseconds between changing
colors */
    WORD pad;          /* reserved for future use; store 0 here */
} CycleInfo;
```

This is very similar to a CRNG chunk. A program would probably only use one of these two methods of expressing color cycle data. New programs should use CRNG. You could write out both if you want to communicate this information to both Deluxe Paint and Graphicraft.

## Appendix A. ILBM Regular Expression

Here's a regular expression summary of the FORM ILBM syntax.

This could be an IFF file or a part of one.

```
ILBM ::= "FORM" #{ "ILBM" BMHD [CMAP] [GRAB] [DEST]  
[SPRT] [CAMG]  
CRNG* CCRT* [BODY] }
```

```
BMHD ::= "BMHD" #{ BitMapHeader }  
CMAP ::= "CMAP" #{ (red green blue)* } [0]  
GRAB ::= "GRAB" #{ Point2D }  
DEST ::= "DEST" #{ DestMerge }  
SPRT ::= "SPRT" #{ SpritePrecedence }  
CAMG ::= "CAMG" #{ LONG }
```

```
CRNG ::= "CRNG" #{ CRange }  
CCRT ::= "CCRT" #{ CycleInfo }  
BODY ::= "BODY" #{ UBYTE* } [0]
```

The token “#” represents a ckSize LONG count of the following braced data bytes. E.g., a BMHD’s “#” should equal `sizeof(BitMapHeader)`. Literal strings are shown in “quotes,” [square bracket items] are optional, and “\*” means 0 or more repetitions. A sometimes-needed pad byte is shown as “[0].”

The property chunks (BMHD, CMAP, GRAB, DEST, SPRT, and CAMG) and any CRNG and CCRT data chunks may actually be in any order but all must appear before the BODY chunk since ILBM readers usually stop as soon as they read the BODY. If any of the 6 property chunks are missing, default values are inherited from any shared properties (if the ILBM appears inside an IFF LIST with PROPs) or from the reader program's defaults. If any property appears more than once, the last occurrence before the BODY is

the one that counts since that's the one that modifies the BODY.

## Appendix B. ILBM Box Diagram

Here's a box diagram for a simple example: an uncompressed image 320 × 200 pixels × 3 bitplanes. The text to the right of the diagram shows the outline that would be printed by the IFFCheck utility program for this particular file.

ILBM example block diagram

The “0” after the CMAP chunk is a pad byte.

## Appendix C. IFF Hints

Hints on ILBM files from Jerry Morrison, Oct 1988. How to avoid some pitfalls when reading ILBM files:

- Don't ignore the BitMapHeader:masking field. A bitmap with a mask (such as a partially-transparent DPaint brush or a DPaint picture with a stencil) will read as garbage if you don't de-interleave the mask.
- Don't assume all images are compressed. Narrow images aren't usually run-compressed since that would actually make them longer.
- Don't assume a particular image size. You may encounter overscan pictures and PAL pictures.

Different hardware display devices have different color resolutions:

Device	R:G:B bits	maxColor
Mac SE	1	1

IBM EGA	2:2:2	3
Atari ST	3:3:3	7
Amiga	4:4:4	15
CD-I	5:5:5	31
IBM VGA	6:6:6	63
Mac II	8:8:8	255

An ILBM CMAP defines 8 bits of Red, Green and Blue (i.e., 8:8:8 bits of R:G:B). When displaying on hardware which has less color resolution, just take the high order bits. For example, to convert ILBM's 8-bit Red to the Amiga's 4-bit Red, right-shift the data by 4 bits ( $R_4 := R_8 \gg 4$ ).

To convert hardware colors to ILBM colors, the ILBM specification says just set the high bits ( $R_8 := R_4 \ll 4$ ). But you can transmit higher contrast to foreign display devices by scaling the data  $[0..maxColor]$  to the full range  $[0..255]$ . In other words,  $R_8 := (R_n \times 255) / maxColor$ . (Example #1: EGA color 1:2:3 scales to 85:170:255. Example #2: Amiga 15:7:0 scales to 255:119:0). This makes a big difference where  $maxColor$  is less than 15. In the extreme case, Mac SE white (1) should be converted to ILBM white (255), not to ILBM gray (128).

## CGA and EGA subtleties

IBM EGA colors in 350 scan line mode are 2:2:2 bits of R:G:B, stored in memory as xxR'G'B'RGB. That's 3 low-order bits followed by 3 high-order bits.

IBM CGA colors are 4 bits stored in a byte as xxxxIRGB. (EGA

colors in 200 scan line modes are the same as CGA colors, but stored in memory as xxxlRGB.) That's 3 high-order bits (one for each of R, G, and B) plus one low-order "Intensity" bit for all 3 components R, G, and B. Exception: IBM monitors show IRGB = 0110 as brown, which is really the EGA color R:G:B = 2:1:0, not dark yellow 2:2:0.

## 24-bit ILBMs

When storing deep images as ILBMs (e.g., images with 8 bits each of R, G, and B), the bits for each pixel represent an absolute RGB value for that pixels rather than an index into a limited color map. The order for saving the bits is critical since a deep ILBM would not contain the usual CMAP of RGB values (such a CMAP would be too large and redundant).

To interpret these "deep" ILBMs, it is necessary to have a standard order in which the bits of the R, G, and B values will be stored. A number of different orderings have already been used in deep ILBMs and a default has been chosen from among them.

The following bit ordering has been chosen as the default bit ordering for deep ILBMs.

Default standard deep ILBM bit ordering:

saved first -----> saved last

R0 R1 R2 R3 R4 R5 R6 R7 G0 G1 G2 G3 G4 G5 G6 G7 B0 B1  
B2 B3 B4 B5 B6 B7

[Recall from Section 3 that the bits representing the value at a given pixel are divided into separate bitplanes. A 24-bit RGB image uses 24 bitplanes. Also recall that images are stored in the BODY one complete scanline at a time, so one row from each of



## 8-bit Grayscale

### Grayscale ILBM bit ordering:

10 11 12 13 14 15 16 17

## 32-bit RGB plus Alpha

### 32-bit RGBA ILBM bit ordering:

saved first -----> last

R0 ... R7 G0 ... G7 B0 ... B7 A0 A1 A2 A3 A4 A5 A6 A7

]

One other existing deep bit ordering that you may encounter is the 21-bit Newtek format.

NewTek deep ILBM bit ordering:

saved first -----> saved last

R7 G7 B7 R6 G6 B6 R5 G5 B5 R4 G4 B4 R3 G3 B3 R2 G2 B2

R1 G1 B1 R0 G0 B0

[This was the RGB format used by NewTek's Digi-View video digitizer. You're unlikely to encounter this in the wild, but in addition to the 21-bit depth, you can detect these files by looking for a DGVW chunk, which stored the software's control panel settings.]

Note that you may encounter CLUT chunks in deep ILBMs. See the Third Party Specs appendix for more information on CLUT chunks.

## Appendix D. ByteRun1 Run Encoding

The run encoding scheme byteRun1 is best described by psuedo code for the decoder Unpacker (called UnPackBits in the Macintosh toolbox):

UnPacker:

LOOP until produced the desired number of bytes

Read the next source byte into n

SELECT n FROM

[0..127] => copy the next n+1 bytes literally

[-1..-127] => replicate the next byte -n+1 times

```
-128 => no operation  
ENDCASE;  
ENDLOOP;
```

In the inverse routine Packer, it's best to encode a 2 byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it's best to merge the three into one literal run. Always encode 3 byte repeats as replicate runs.

Remember that each row of each scan line of a raster is separately packed.

[Some versions of Adobe Photoshop incorrectly use the n=128 no-op as a repeat code, which breaks strictly conforming readers. To read Photoshop ILBMs, allow the use of n=128 as a repeat. This is pretty safe, since no known program writes real no-ops into their ILBMs. The reason n=128 is a no-op is historical: the Mac Packbits buffer was only 128 bytes, and a repeat code of 128 generates 129 bytes.]

## **Appendix E. Standards Committee**

The following people contributed to the design of this FORM ILBM standard:

Bob “Kodiak” Burns, Commodore-Amiga

R. J. Mical, Commodore-Amiga

Jerry Morrison, Electronic Arts

Greg Riker, Electronic Arts

Steve Shaw, Electronic Arts

Dan Silva, Electronic Arts

Barry Walsh, Commodore-Amiga

