

IFF File Format Summary

Also Known As:

Interchange File Format, ILM, ILBM, LBM, Amiga Paint

Type	Bitmap
Colors	1- to 24-bit
Compression	RLE, uncompressed
Maximum Image Size	64Kx64K pixels
Multiple Images Per File	Yes
Numerical Format	Big-endian
Originator	Electronic Arts, Inc., Commodore-Amiga, Inc.
Platform	Amiga, MS-DOS, others
Supporting Applications	Too numerous to list
See Also	Microsoft RIFF , GIF , PNG

Usage

On platforms other than the Amiga, IFF is used mostly for storing image and sound data. On the Amiga almost any type of data may be found in an IFF file. The file extension usually indicates the type of data stored in an IFF file.

Comment

IFF is an older data file format found on most every system. Its versatility has not been greatly utilized outside of the Amiga platform.

[Vendor specifications](#) are available for this format.

[Sample images](#) are available for this format.

IFF (Interchange File Format) is a general purpose data storage format that can associate and store multiple types of data. IFF is portable and has many well-defined extensions that support still-picture, sound, music, video, and textual data. Because of this extensibility, IFF has fathered a family of special purpose file formats all based on IFF's simple data structure.

Contents:

[File Organization](#)

[File Details](#)

[For Further Information](#)

IFF is most often associated with the Commodore-Amiga computer and originated on that system. IFF is fully supported by the Amiga operating system and is used for storing virtually every type of data found in the Amiga's filesystem. Initialization files, documents, temporary data, and data exported from the clipboard may all be stored using the IFF format.

The most common IFF family member is ILBM, or InterLeaved BitMap. ILBM files are the standard image file format for the Commodore-Amiga computer and are the type of IFF files with which most graphics people are familiar.

IFF files are common in the MS-DOS and UNIX environment as well and usually have the file extension .IFF or .LBM. Electronic Arts' DeluxePaint program is generally credited with making IFF known to the MS-DOS community. For a time IFF was a widely accepted 24-bit format under MS-DOS, but it was eventually replaced first by TIFF and TGA, and then by JFIF.

IFF faces compatibility problems when the occasional program fails to write IFF file data using the big-endian byte order. This prevents most programs from reading these IFF files. Other problems created by bad IFF file writers include writing planar image data improperly and failing to use only linefeeds to terminate lines of text. Unfortunately, some people (those who are all too willing to shoot the messenger), have blamed IFF, rather than bad software, for these problems.

Today IFF is a widely used format that is supported by most graphics programs found on MS-DOS, MS Windows, Macintosh, UNIX, and Amiga systems. The format basically remained unchanged since its specification was released in 1985, but many extensions to the format have been created and documented by a great many software developers, making IFF one of the most utilized data file formats of today.

File Organization

IFF files are constructed entirely of chunks. A chunk is a data structure containing a 4-byte ID, a 4-byte size value, and possibly a block of data. Each chunk is the same, simple structure and differs only in the data it contains. You can think of a chunk as an envelope or wrapper that identifies a collection of data.

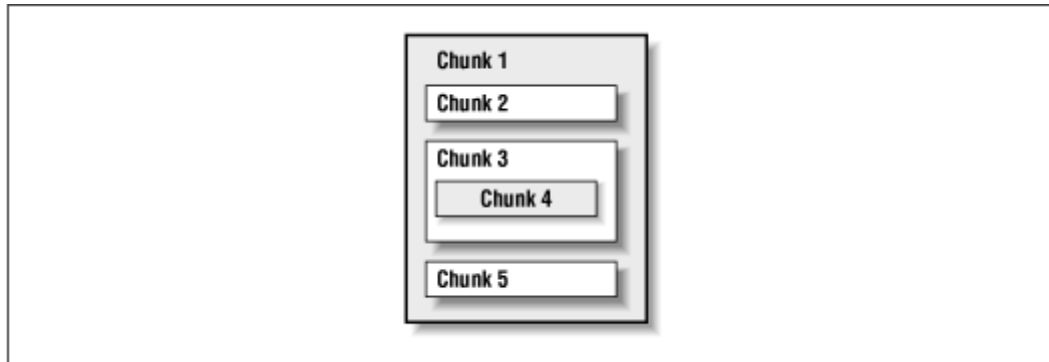
The data stored in a chunk can be anything: graphics, sound, animations, text from a word processor, or a collection of 3D objects. Any kind of data can be stored in a chunk, including a chunk itself.

Nesting one or more chunks within a chunk is common in IFF files. In fact, an IFF file is conceptually nothing more than a single chunk containing one or more other chunks as data. There is also no specified limit as to the number of nesting levels within a chunk.

IFF nesting is a powerful organizing principle. It offers the same sort of organizational advantages that nested directories and subdirectories do for filesystems. The down side of nesting is that it introduces a certain amount of complexity that can make IFF appear difficult to interpret.

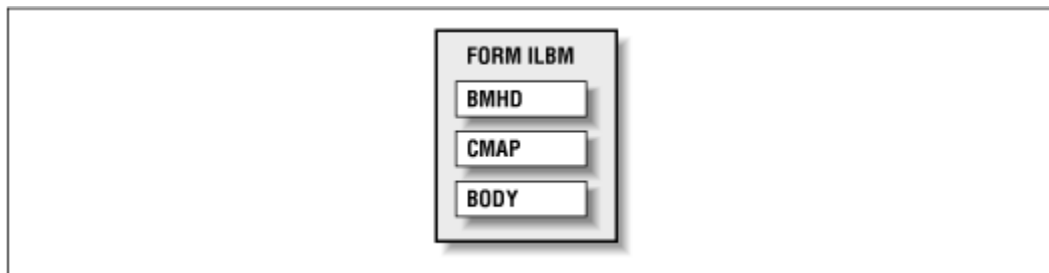
Figure IFF-1 illustrates the "chunks within a chunk" concept.

Figure IFF-1: A chunk file structure



Most IFF files contain a single chunk called a FORM chunk. This chunk stores the formatting and identification information for all chunks and data in the IFF file. All other chunks in the file are stored within the FORM chunk. The basic structure of the FORM chunk is illustrated in Figure IFF-2.

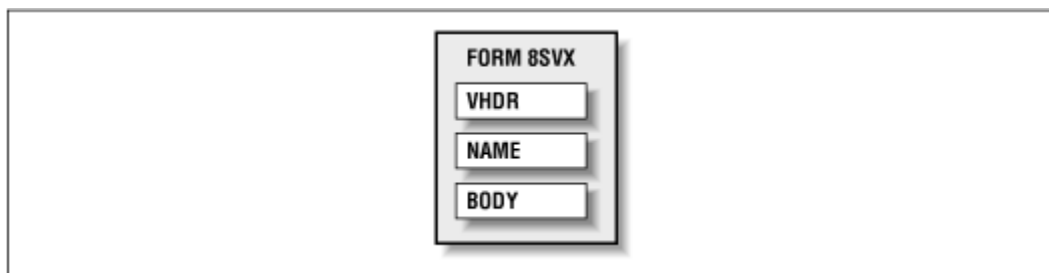
Figure IFF-2: FORM ILBM file structure



In this example, the FORM chunk is of type ILBM (InterLeaved BitMap) and contains three chunks, each of which contain data blocks that together define an image. The FORM ILBM is the most common file type for storing still-picture graphical data in an IFF file. "ILBM" is the type identifier for the FORM. It tells readers what kind of FORM chunk this is and what chunks might be expected within it. The three chunks in this example are the BMHD (BitMap Header), CMAP (Color MAP), and BODY (the actual pixels).

Another common FORM type is 8SVX (8-bit Sampled VoX, or Voice), a format for digitized sound samples. A simple FORM 8SVX is shown in Figure IFF-3.

Figure IFF-3: FORM 8SVX file structure

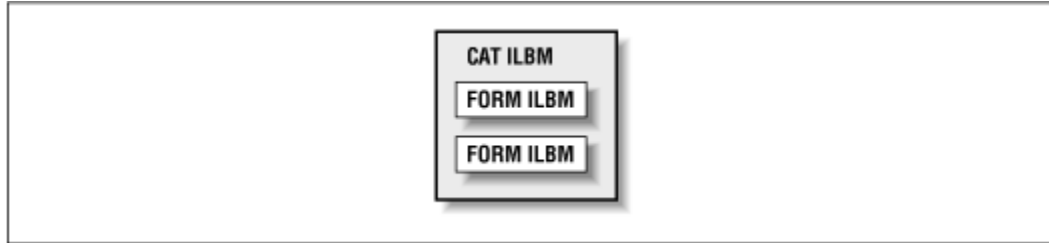


As you can see, Figures IFF-2 and IFF-3 aren't very different. The high-level structure of all IFF files is similar because it is created from the same simple chunking rules, regardless of what kind of data is stored in the files.

IFF files that contain a single FORM chunk are by far the most common. In fact, if you confine yourself to the most widely used FORM types such as ILBM, you may never encounter any other IFF structure. But group structures do exist that allow IFF writers to collect multiple FORMs into a single file.

A CAT chunk is used to append or "concatenate" two or more FORM chunks together in a single IFF file. Figure IFF-4 shows a CAT ILBM file that contains two FORM ILBM chunks.

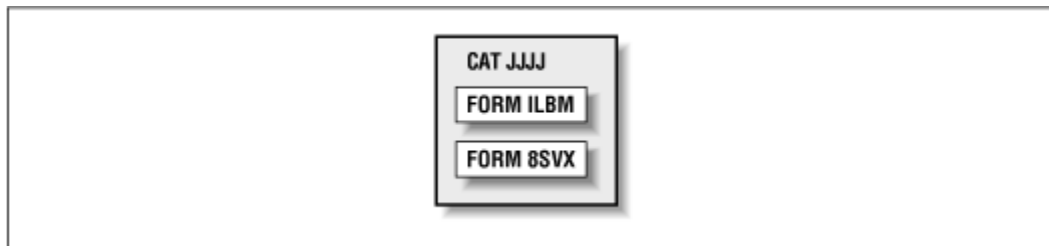
Figure IFF-4: CAT ILBM file structure



In this figure, the CAT chunk contains a single ILBM chunk that contains an ILBM type ID that identifies the type of data stored in each FORM chunk. These two FORM chunks have the same format they would have if they were stored in separate IFF files.

All of the FORM chunks in a CAT chunk need not store the same type of data. Figure IFF-5 illustrates a CAT chunk that stores a FORM ILBM chunk and a FORM 8SVX chunk.

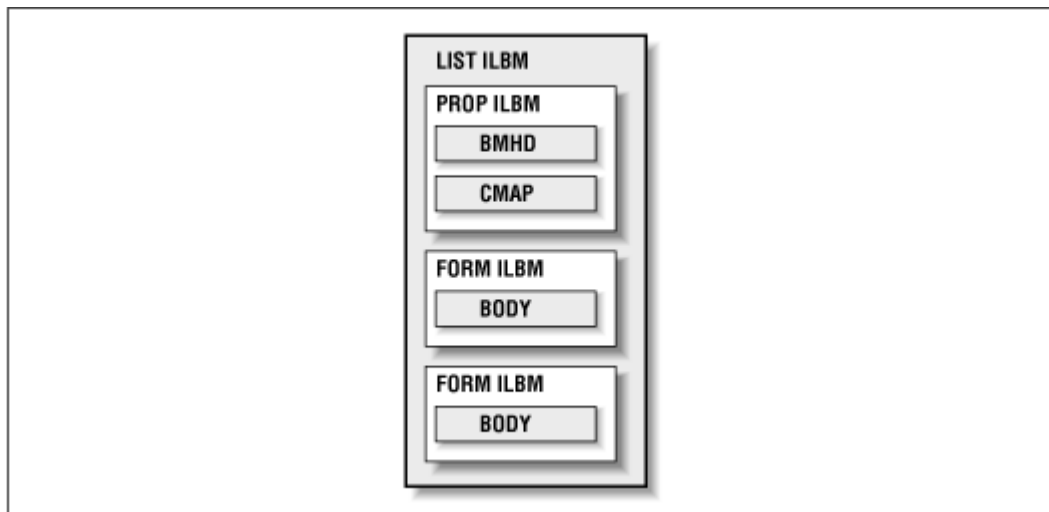
Figure IFF-5: CAT J J J J file structure



When the FORM chunks in a CAT chunk do not all store the same type of data, a contents type identifier of "J J J J" is used to indicate that the CAT contains FORMs of more than one type, or that the IFF writer did not care what type(s) of FORM might be in the file. CAT type IDs are often referred to as "hints," because each FORM unambiguously identifies what it contains.

LIST chunks also allow the storage of multiple data objects within a single IFF file but add the ability to group data objects together and have them share common data by the use of the PROP (property) chunk. Figure IFF-6 illustrates a LIST ILBM file that contains two images that share a common bitmap header and color map by using a PROP chunk.

Figure IFF-6: LIST ILBM file structure



File Details

This section describes the details of IFF chunks.

Chunks

All IFF files are composed of very simple data structures called *chunks*. A chunk may be anywhere from four to eight gigabytes in size and is represented by the following data structure:

```
typedef struct _Chunk
{
    char    ChunkId[4];

    DWORD   Size;

    BYTE    Data[];
} CHUNK;
```

ChunkId is the ASCII identifier of the chunk. Identifiers are always alphanumeric characters, and they are right-padded with spaces if they contain fewer than four characters. FORM Type IDs may only use uppercase characters, and Chunk IDs may be mixed case.

Size is the number of data bytes stored in the Data field. This value does not include the presence of a padding byte that may follow the data. If the chunk contains no data, then this value is 0.

Data is the actual chunk data. The number of bytes of data stored in this field is indicated by the value in the Size field.

We can use the _Chunk structure to show the nested chunks within a FORM ILBM file:

```
typedef struct _ChunkFORM_ILBM
{
    char    ChunkId[4];          /* "FORM" */
    DWORD   Size;                /* FORM size (size of file minus 8) */
    /* Start of FORM chunk's data */
    char    TypeID[4];           /* "ILBM" */
    struct _ChunkBMHD
    {
        char    ChunkId[4];      /* "BMHD" */
        DWORD   Size;            /* Size of Data */
        BMHD    Data;            /* Bitmap header data */
    }
    struct _ChunkCMAP
    {
        char    ChunkId[4];      /* "CMAP" */
        DWORD   Size;            /* Size of Data */
        CMAP    Data;            /* Color map data */
    }
    struct _ChunkBODY
    {
        char    ChunkId[4];      /* "BODY" */
        DWORD   Size;            /* Size of Data */
        BODY    Data[];          /* Image data */
    }
}
```

This is the C code version of Figure IFF-2. The file contains a single FORM chunk, which like all chunks begins with a 4-character chunk ID and a 4-byte size. The data for the FORM chunk begins with a 4-character "FORM Type Identifier," or type ID, that identifies the kind of object stored in the FORM. For ILBMs the type ID is just "ILBM". The type ID is followed by a collection of chunks that describe the ILBM, including the bitmap header, color map, and pixel bits.

(Note that we're illustrating the structure of a FORM ILBM using C code here, but you shouldn't actually use a structure like this in your IFF code. The contents of a FORM chunk vary too much to be captured by a single C structure. Nor should you assume that these are the only chunks in a FORM ILBM, or that they will occur in this particular order.)

Chunks must always begin on an even byte boundary. If a chunk contains an odd number of data bytes, then the chunk that follows would improperly begin on an odd-byte boundary. To preserve alignment, a padding byte will be added between the odd-length data field of a chunk and the Chunk ID of the next chunk. This padding byte always has a value of zero and is not considered to be part of the chunk data. If the Size field of a chunk contains an odd value, then you should assume that a padding byte is present.

Parsing an IFF file is a process of reading chunk identifiers, using the data in known chunks and skipping over unknown chunks. The chunk identifiers are therefore crucial in determining whether the IFF file reader recognizes the format of the data stored in a chunk.

Chunk Content Identifiers

The first four bytes of every chunk identifies the format of the chunk's data. These bytes are called the Chunk Content Identifier, or Chunk ID for short. Chunk IDs are made of ASCII characters in the range 0x20 to 0x7E (" " to "~"). The restriction to uppercase and no punctuation applies only to FORM type IDs. Spaces are used only to pad out IDs that are fewer than four printable characters in length.

Each ID represents a specific format of data. If an IFF reader does not recognize the ID of a chunk, then the reader cannot know the format of the chunk's data, and therefore should skip over the chunk.

Five primary chunk IDs are reserved by the IFF specification. They are "FORM", "LIST", "PROP", "CAT ", and " ". Note that the CAT chunk contains a single padding space, and the ID of the Filler Chunk is all spaces.

Chunk IDs can also be used to indicate the revision level of a chunk. For example, the revision IDs for the FORM chunk are "FOR1", "FOR2", "FOR3", and so on. For the CAT chunk, the revisions are "CAT1", "CAT2", "CAT3", and so on. The FORM, LIST, and CAT chunks each have nine revisions reserved by the IFF-85 specification, bringing the total number of reserved chunk IDs to 32.

If an IFF reader does not recognize the FORM Type ID of a FORM, CAT or LIST chunk, it may continue reading to find any nested FORM chunks in the file that it does recognize. If the first chunk in the file does not have a Group Type ID of FORM, CAT, or LIST, then the reader should assume that it is not an IFF-format file.

The number and kind of data chunks that appear in a FORM, and the order in which they appear, is determined by the FORM type ID. Some data chunks must always be present, such as those required of a BMHD chunk in a FORM ILBM. Some chunks must appear in a specific order; for example, a CMAP chunk must always appear before its corresponding BODY chunk. But most data chunks have no restriction on the order in which they occur, or even the number of times that they may occur, in the IFF file.

The Filler Chunk (Chunk ID " ") is a special-purpose data chunk that is used only to provide alignment between chunks in a file. The data stored in a Filler Chunk is meaningless and is never used. For example, a file reader may be designed to read 1K blocks of data from a file stream. In this case, you might require that each chunk in an IFF file be aligned on the closest 1024-byte boundary. Filler chunks would be inserted between all other chunks to provide boundary alignment padding as needed.

To give you an idea of all of the possible data chunks that might be found in a single FORM Chunk, here is a list of a few chunk IDs that are associated with storing textual data:

(C) Copyright notice and license
ANNO Annotation or comment
AUTH Author name
DOC Document formatting information
FOOT Footer information of a document
HEAD Header information of a document
PAGE Page break indicator
PARA Paragraph formatting information
PDEF Deluxe Print page definition
TABS Tab positions
TEXT Text for a paragraph
VERS File version

Here is a listing of some FORM types to give you an idea of what kind of data is stored using the IFF format:

Graphical

ACBM Amiga Contiguous Bitmap (Microsoft Basic for the Amiga)
DEEP IFF Deep (24-bit color image)
DR2D 2D object standard format (vector data)
FNTR Raster font
FNTV Vector font
ILBM InterLeaved Bitmap (interleaved planar bitmap data)
PICS Macintosh picture
RGB8 24-bit color image (Impulse)
RGBN 12-bit color image (Impulse)
TDDD Turbo 3d rendering data (3D objects)
YUVN YUV image data (V-Lab)

Animation

ANBM Animated bitmap
ANIM Cel animations
SSA Super smooth animation (ProDAD)

Video

VIDEO Deluxe Video Construction Set video

Sound

8SVX 8-bit sampled voice
AIFF Audio interchange file format
SAMP Sampled sound
UVOX Uhuru Sound Software Macintosh voice

Music

GSCR General use musical score
SMUS Simple musical score
TRAK MIDI music data
USCR Uhuru Sound Software musical score

Text

FTXT Formatted text
WORD Pro-write word processing

Probably several hundred FORM types and chunk IDs have been created and (we hope) documented by software developers over the past ten years. Most Amiga software archives contain the specifications for many of these chunks. In the section below, we look at the most common IFF data type for storing graphics data, ILBM.

ILBM Chunk

The ILBM format is the most commonly used chunk format for storing graphics data in an IFF file. The only data chunk required to appear in an ILBM is a Bitmap Header chunk (BMHD). It may seem strange not to require the presence of a BODY chunk to store the image data, but an IFF colormap file is a FORM ILBM that contains only a BMHD and colormap (CMAP) chunk but no image data.

ILBMs may optionally contain a colormap (CMAP chunk), hot spot information (GRAB chunk), destination merge data (DEST chunk), or sprite information (SPRT chunk). They may specify a Commodore-Amiga viewport mode (CAMG chunk) or image data (BODY chunk). All of these chunks must appear after the BMHD chunk and before the BODY chunk. BODY must always appear last.

Bitmap Header (BMHD) chunk

The BMHD chunk contains information defining the metrics of the image data. This chunk is always 36 bytes in length and has the following format:

```
typedef struct _BitMapHeader
{
    char    ChunkId[4];    /* Chunk Identifier "BMHD" */
    DWORD   Size;          /* Size of chunk data in bytes */
    /* Chunk data starts here */
    WORD    Width;         /* Width of image in pixels */
    WORD    Height;        /* Height of image in pixels */
    WORD    Left;          /* X coordinate of image */
    WORD    Top;           /* Y coordinate of image */
    BYTE    Bitplanes;      /* Number of bitplanes */
    BYTE    Masking;        /* Type of masking used */
    BYTE    Compress;       /* Compression method use on image data */
    BYTE    Padding;       /* Alignment padding (always 0) */
    WORD    Transparency;   /* Transparent background color */
    BYTE    XAspectRatio;   /* Horizontal pixel size */
    BYTE    YAspectRatio;   /* Vertical pixel size */
    WORD    PageWidth;      /* Horizontal resolution of display device */
    WORD    PageHeight;     /* Vertical resolution of display device */
} BITMAPHEADERCHUNK;
```

ChunkId contains the chunk content identifier "BMHD".

Size is the number of bytes following the Size field and is always 28.

Width and Height are the width and height, respectively, of the image data in pixels.

Left and Top are the X and Y coordinates position of the upper-left corner of the image. The default values for these fields are 0 and 0.

Bitplanes is the number of bits per pixel used to store the image data.

Masking indicates the type of masking used to display the image. Valid values for this field are 0 (standard opaque rectangular image), 1 (mask data is interleaved with image data as an extra bitplane), 2 (pixels that match the value in the Transparency field are transparent), and 3 (image may be lassoed, as in MacPaint).

Compress indicates whether the image data in the BODY chunk is compressed. A value of 0 indicates no compression, and a value of 1 indicates that the data is compressed using the Packer algorithm defined by the IFF-86 ILBM specification.

Padding is used to maintain alignment padding within the BMHD structure and always contains the value zero.

Transparency is a value used with Masking to determine which pixels (if any) in an image are transparent.

XAspectRatio and YAspectRatio define the pixel aspect ratio of the image data. The aspect ratio is calculated by dividing XAspectRatio by YAspectRatio.

PageWidth and PageHeight describe the required resolution to display the image. If the image data were to be displayed at a resolution of 320x200, the values of these fields would be 320 and 200, respectively.

Color Map (CMAP) chunk

The optional CMAP chunk stores color information for the image data. CMAP data is actually an array of the following data type:

```
typedef struct _ColorMapEntry
{
    BYTE    Red;           /* Red color component (0-255) */
    BYTE    Green;         /* Green color component (0-255) */
    BYTE    Blue;          /* Blue color component (0-255) */
} COLORMAPENTRY;
```

Red, Green, and Blue store color component intensity values in the range 0 to 255 for a single color. A value of 255,255,255 is white, and 0,0,0 is black.

The CMAP chunk contains an array of COLORMAPENTRYs as data. The number of elements in this array will vary depending upon the number of colors in the image data. The CMAP chunk structure follows:

```
typedef struct _ColorMapChunk
{
    char    ChunkId[4];    /* Chunk Identifier "CMAP" */
    DWORD   Size;          /* Size of chunk data in bytes */
    /* Chunk data starts here */
    COLORMAPENTRY Map[Size/3]; /* Color map data */
} COLORMAPCHUNK;
```

ChunkId contains the chunk content identifier "CMAP".

Size is the number of bytes in the Map field.

Map is the actual color map data. It is an array of COLORMAPENTRY values. There are typically 2^{BitPlanes} entries in this array.

Body (BODY) chunk

The BODY chunk stores the actual image data as a BYTE array. The structure of the BODY chunk follows:

```
typedef struct _BodyChunk
{
    char    ChunkId[4];           /* Chunk Identifier "BODY" */
    DWORD   Size;                /* Size of chunk data in bytes */
    /* Chunk data starts here */
    BYTE    ImageData[];         /* Image data */
} BODYCHUNK;
```

ChunkId contains the chunk content identifier "BODY".

Size is the number of bytes in the ImageData field.

ImageData is the actual image data. Image data is an array of byte values and may be stored uncompressed, or compressed (using the IFF Packer encoding algorithm).

Pixel data, compressed or not, is always stored in separate bitplanes. Each scan line is made up of BMHD.BitPlanes rows of bytes. Each bitplane row encodes one bit from the pixel value. The pixel data appears as follows:

```
BODY
scan-line 0
    bitplane 0 pixel value's least significant bit
        byte 0 bits for the leftmost eight pixels
        byte 1
        .
        .
        .
        byte RowBytes - 1 see RowBytes note below
    bitplane 1
    .
    .
    .
    bitplane BMHD.BitPlanes - 1
scan-line 1
    .
    .
    .
scan-line BMHD.Height - 1
```

RowBytes is the smallest even integer greater than BMHD.Width / 8, which can be found using:

```
RowBytes = ((BMHD.Width + 15) >> 4) << 1;
```

This is equivalent to saying that each bitplane row must contain an even number of bytes. Each bitplane row (scan line) is therefore word-aligned with padding before compression, if necessary.

If BMHD.Masking is 1, there will be an extra bitplane row in each scan line. In other words, each scan line will contain (BMHD.BitPlanes + 1) bitplane rows. This extra bitplane forms a 1-bit mask that is to be applied to the image when it is displayed. Depending on your intentions, you'll often discard the mask plane while decoding the BODY, but you have to remember to read the mask plane if it is there, because it's not included in BMHD.BitPlanes. BMHD.Masking = 1 is becoming less common, particularly on platforms other than the Amiga.

The pixel values in a BODY can be indexes into the palette contained in a CMAP chunk, or they can be literal RGB values. If there is no CMAP and if BMHD.BitPlanes is 24, the ILBM contains a 24-bit image, and the BODY encodes pixels as literal RGB values. The bitplanes for each scan line appear in the BODY in the following order:

```
scan-line 0
    red bit 0 red least significant bit
    red bit 1
    .
    .
    .
    red bit 7
    green bit 0 green least significant bit
    .
    .
    .
    green bit 7
    blue bit 0 blue least significant bit
    .
    .
    .
    blue bit 7
scan-line 1
    red bit 0
    .
    .
    .
```

If the ILBM has no CMAP and if BMHD.BitPlanes is 8 (or occasionally, less than 8), the file contains a gray-scale image. Bitplanes are stored in the same least-to-most order of significance, and the full black-to-white range is assumed to be 0 to (2^BMHD.BitPlanes) - 1, and is usually 255. Note that if you're thinking about implementing an ILBM writer, you should include a CMAP with your gray-scale images, because most non-Amiga IFF readers will refuse to load an ILBM without a CMAP unless it is a 24-bit image.

ILBMs created for use on an Amiga may also contain pixel data in their BODY chunks that reflects display capabilities peculiar to the Amiga. Such Amiga-specific ILBMs must contain a CAMG (Comm-dore-AMiGa) chunk that identifies the Amiga display mode. CAMG data consists of a single DWORD which contains the viewmode value. The Amiga has built-in support for interpreting this value, but programs running on other platforms can safely test certain bits to identify Amiga-specific pixel data.

Hold-And-Modify (HAM) display mode

HAM (Hold-And-Modify) is a display mode that allows the Amiga to display 12-bit and 18-bit images using only 6 or 8 bits per pixel. HAM images can be identified by bit 11's being set to 1 in the CAMG chunk (CAMG mode & 0x0800 != 0).

The 8-bit HAM8 mode was introduced with the Amiga 4000 and Amiga 1200 models and provides a very good near-photographic-quality image display with only eight bitplanes of image data.

The color of any pixel in a HAM image may be any color from a standard 16-color palette, or the same as the color of the pixel to the immediate *left* with the top four bits of either the red, green, or blue components changed. On the left edge of the screen, the border color is used, which is color index zero from the 16-color palette.

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 remaining bits. Possible mode values are:

Mode Meaning

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

Unless a pixel is colormapped (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 in a scan line, the pixel to its left is assumed to be RGB(0, 0, 0).

The format of the mode and data bits in a pixel in HAM mode is:

5 4 3 2 1 0

- 0 0 *w x y z* = Use colormap value *wxyz*
- 0 1 *w x y z* = Keep color from previous pixel, but change blue upper 4 bits to *wxyz*
- 1 0 *w x y z* = Keep color from previous pixel, but change red upper 4 bits to *wxyz*
- 1 1 *w x y z* = Keep color from previous pixel, but change green upper 4 bits to *wxyz*

A HAM image cannot be directly decoded into a standard 8-bit or lower palette-based image without further color reduction. For full quality, it must be converted to at least a 12-bit color image.

The number of data bits is 4 for standard HAM and 6 for HAM8, and the corresponding BMHD.BitPlanes value will normally be 6 or 8. The data bits should be precision-extended when the levels are decoded to 24-bits, and regardless of the number of data bits, the maximum level should translate to 255 at 8 bits per RGB channel.

The format of the mode and data bits in a pixel in HAM8 mode is:

7 6 5 4 3 2 1 0

- 0 0 *n m w x y z* = Use color palette value *nmwxyz*
- 0 1 *n m w x y z* = Keep color from previous pixel, but change blue upper 6 bits to *nmwxyz*
- 1 0 *n m w x y z* = Keep color from previous pixel, but change red upper 6 bits to *nmwxyz*
- 1 1 *n m w x y z* = Keep color from previous pixel, but change green upper 6 bits to *nmwxyz*

HAM8 images need to be directly converted to a 24-bit image in order to retain the full image quality on non-Amiga systems.

It is possible for the mode to be a single bit; BMHD.BitPlanes 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, single mode bit images are rarely encountered.

Extra-Halfbrite (EHB) display mode

Extra-Halfbrite is another Amiga variant, now quite rare. The original Amiga models had a color palette of 32 colors (from a range of 4096) and could support up to 6 bitplanes. When 6 bitplanes were selected, one of two modes could be chosen to utilize the extra bit of data.

EHBs are 64-color (6 bitplane) images with 32-color palette entries. They can be identified by bit 7's being set to 1 in the CAMG (CAMG mode & 0x0080 != 0). Colors 32 to 63 are "half-bright" versions of colors 0 to 31.

To decode an EHB image, extend the color palette to 64 colors, and create colors 32 to 63 by copying and bit-shifting each palette color (0 to 31) right by one. The image is then decoded as it normally would be.

Image data compression

FORM ILBM files may contain image data compressed using a simple, run-length encoding algorithm called Packer. This algorithm is identical to the Macintosh PackBits algorithm and is also the algorithm used by the TIFF file format.

Packer encodes runs of identical byte values within a scan line. Encoding always stops at the end of every scan line. All byte runs are encoded as two-byte codes. The first byte is a code byte which indicates the type of compressed run and the number of pixels in the run. If the value of the code is 0 to 127 (signed bit off), the run is a literal run of pixels, and the next (code + 1) bytes are copied literally from the compressed data.

If the value of the code is -1 to -127 (signed bit on), the next byte following the code byte is read and its value is repeated (-code + 1) times. A code value of -128 is a no-op and is always ignored.

When compression is indicated, all of the image data stored in a BODY chunk is compressed, including any masking data interleaved with the image data.

For Further Information

For further information about the IFF format, see the following specifications:

Morrison, Jerry, *EA IFF 85 Standard for Interchange Format Files*, Electronic Arts, 14 January 1985.

Morrison, Jerry, *ILBM IFF Interleaved Bitmap*, Electronic Arts, 17 January 1986.

These documents are also available from many online services and BBSs. You may also obtain them directly from the creator of IFF, Electronic Arts, at:

Electronic Arts

1820 Gateway Drive

San Mateo, CA 94404

Voice: 415-571-7171

Voice: 415-572-2787

WWW: <http://www.ea.com/>

The following documents from Electronic Arts are also widely available online:

FTXT: IFF Formatted Text, Electronic Arts. IFF supplement document for a text format.

ILBM: IFF Interleaved Bitmap, Electronic Arts. IFF supplement document for a raster image format.

Commodore-Amiga previously supported the IFF format, but the company was purchased by ESCOM, a German PC manufacturer.

The following Commodore-Amiga document is widely available online:

Scheppner, Carolyn, *Introduction to Amiga IFF ILBM Files and Amiga Viewmodes*, Commodore Amiga Technical Support.

The following books also discuss the IFF file format:

Commodore Amiga, Inc., *Amiga ROM KERNEL Reference Manual: Includes and Autodocs*, Addison-Wesley, Reading, MA, 1989.

Commodore Amiga, Inc., *Amiga ROM KERNEL Reference Manual: Devices*, Third Edition, Addison-Wesley, 1991.

The *RKM: Devices* manual contains more than 200 pages devoted to IFF and includes the complete text of every official Jerry Morrison specification and a generous amount of source code, including an implementation of Packer.

The Aminet archives are the best source of IFF information and display programs for the Amiga. Aminet mirrors include:

<ftp://nic.funet.fi/pub/amiga/textonly/applications/convert>

<ftp://wuarchive.wustl.edu/pub/aminet/gfx/conv>

http://wuarchive.wustl.edu/~aminet/dirs/gfx_conf.html

Inquiries about the Aminet archives may be emailed to aminet.aminet.org.