A Quick Introduction to IFF

Jerry Morrison, Electronic Arts
10-17-88

IFF is the Amiga-standard "Interchange File Format", designed to work across many machines.

Why IFF?

Did you ever have this happen to your picture file?

You can't load it into another paint program.
You need a converter to adopt to "ZooPaint" release 2.0 or a new hardware feature.
You must "export" and "import" to use it in a page layout program.
You can't move it to another brand of computer.

What about interchanging musical scores, digitized audio, and other data? It seems the only thing that *does* interchange well is plain ASCII text files.

It's inexcusable. And yet this is "normal" in MS-DOS.

What is IFF?

IFF, the "Interchange File Format" standard, encourages multimedia interchange between different programs and different computers. It supports long-lived, extensible data. It's great for composite files like a page layout file that includes photos, an animation file that includes music, and a library of sound effects.

IFF is a 2-level standard. The first layer is the "wrapper" or "envelope" structure for all IFF files. Technically, it's the syntax. The second layer defines particular IFF file types such as ILBM (standard raster pictures), ANIM (animation), SMUS (simple musical score), and 8SVX (8-bit sampled audio voice).

IFF is also a design idea:

programs should use interchange formats for their everyday storage

This way, users rarely need converters and import/export commands to change software releases, application programs, or hardware.

What's the trick?

File compatibility is easy to achieve if programmers let go of one notion—dumping internal data structures to disk. A program's internal data structures should really be suited to what the program does and how it works. What's "best" changes as the program evolves new functions and methods. But a disk format should be suited to storage and interchange.

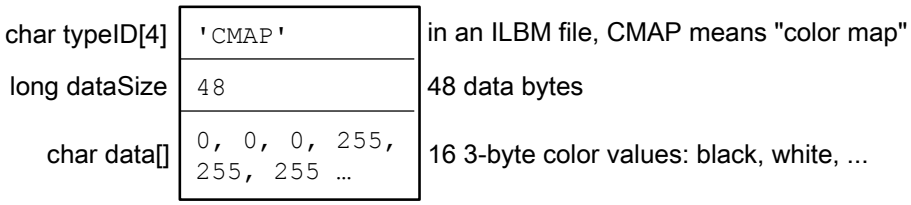
Once we design internal formats and disk formats for their own separate purposes, the rest is easy. Reading and writing become behind-the-scenes conversions. But two conversions hidden in each program is much better than a pile of conversion programs.

Does this seem strange? It's what ASCII text programs do! Text editors use line tables, piece tables, gaps, and other structures for fast editing and searching. Text generators and consumers construct and parse files incrementally. Few programs dump text files into memory. That's why the ASCII standard works so well.

Also, every file must be self-sufficient. E.g. a picture file has to include its size and number of bits/pixel.

What's an IFF file look like?

IFF is based on data blocks called "chunks". Here's an example color map chunk:



A chunk is made of a 4-character type identifier, a 32 bit data byte count, and the data bytes. It's like a Macintosh "resource" with a 32-bit size.

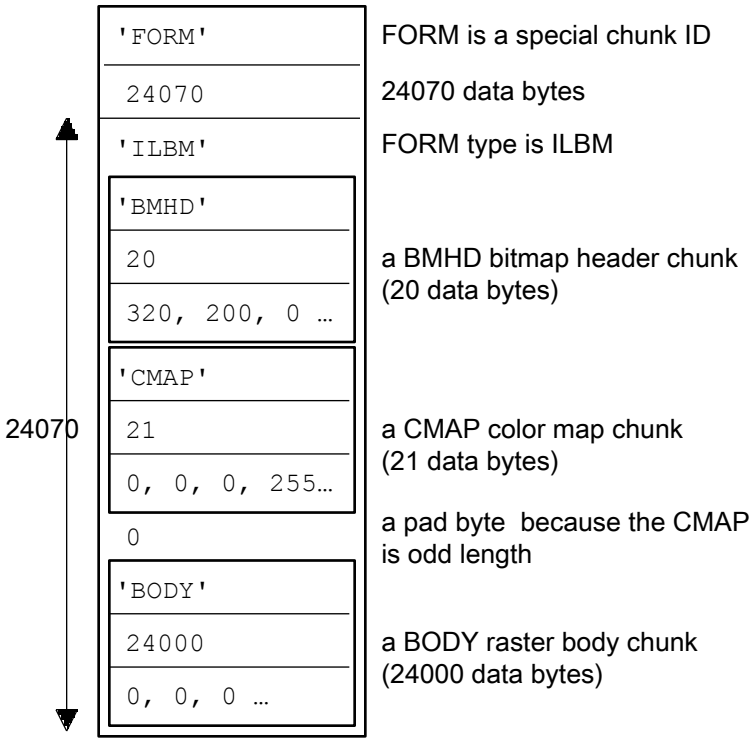
Fine points:

- Every 16- and 32-bit number is stored in 68000 byte order—highest byte first.

  An Intel CPU must reverse the 2- or 4-byte sequence of each number. This applies to chunk `dataSize` fields and to numbers inside chunk data. It does not affect character strings and byte data because you can't reverse a 1-byte sequence. But it does affect the 32-bit math used in IFF's MakeID macro. The standard does allow CPU-specific byte ordering within a chunk's data, but the practice is discouraged.
- Every 16- and 32-bit number is stored on an even address.
- Every odd-length chunk must be followed by a 0 pad byte. This pad byte is <u>not</u> counted in `dataSize`.
- An ID is made of 4 ASCII characters in the range " " (space, hex 20) through "~" (tilde, hex 7E). Leading spaces are not permitted.
- IDs are compared using a quick 32-bit equality test. Case matters.

A chunk typically holds a C struct (a Pascal record) or an array. For example, an 'ILBM' picture has a 'BMHD' bitmap header chunk (a structure) and a 'BODY' raster body chunk (an array).

To construct an IFF file, just put a file type ID (like 'ILBM') into a wrapper chunk called a 'FORM' (think "FILE"). Inside that wrapper place chunks one after another (with pad bytes as needed). The chunk size, rounded up to an even number, always tells you how many more bytes you need to skip over to get to the next chunk.



A FORM always contains one 4-character FORM type ID (a file type, in this case 'ILBM') followed by any number of data chunks. In this example, the FORM type is 'ILBM', which stands for "InterLeaved BitMap". (ILBM is an IFF standard for bitplane raster pictures.) This example has 3 chunks. Note the pad byte after the odd length chunk.

Within FORMs ILBM, 'BMHD' identifies a bitmap header chunk, 'CMAP' a color map, and 'BODY' a raster body. In general, the chunk IDs in a FORM are local to the FORM type ID. The exceptions are the 4 global chunk IDs 'FORM', 'LIST', 'CAT ', and 'PROP'. (A FORM may contain other FORM chunks. E.g. an animation FORM might contain picture FORMs and sound FORMs.)

How to read an IFF file?

Given the C subroutine "GetChunkHeader()":

```
/* Skip any remaining bytes of the current chunk, skip any pad byte, and
   read the next chunk header. Returns the chunk ID or END_MARK. */
ID GetChunkHeader();
```

we read the chunks in a FORM ILBM with a loop like this:

```
do
  switch (id = GetChunkHeader())
    {
    case 'CMAP': ProcessCMAP(); break;
    case 'BMHD': ProcessBMHD(); break;
    case 'BODY': ProcessBODY(); break;
    /* default: just ignore the chunk */
    }
  until (id == END_MARK);
```

This loop processes each chunk by dispatching to a routine that reads the specific type of chunk data. We don't assume a particular order of chunks. This is a simple parser. Note that even if you have fully processed a chunk, you should respect its chunk size, even if the size is larger than you expected.

This sample ignores important details like I/O errors. There are also higher-level errors to check, e.g. if we hit END_MARK without reading a BODY, we didn't get a picture.

Every IFF file is a 'FORM', 'LIST', or 'CAT ' chunk. You can recognize an IFF file by those first 4 bytes. ('FORM' is far and away the most common. We'll get to LIST and CAT below.) If the file contains a FORM, dispatch on the FORM type ID to a chunk-reader loop like the one above.

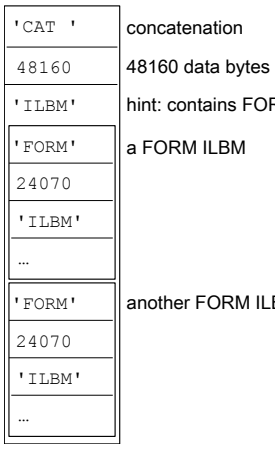
File extensibility

IFF files are extensible and forward/backward compatible:

- Chunk contents should be designed for compatibility across environments and for longevity.
- The standards team for a FORM type can extend one of its chunks that contains a structure by appending new, optional structure fields.
- Anyone can define new FORM types as well as new chunk types within a FORM type. Storing private chunks within a FORM is ok, but be sure to register your activities with Commodore-Amiga Technical Support.
- A chunk can be superseded by a new chunk type, e.g. to store more bits per RGB color register. New programs can output the old chunk (for backward compatibility) along with the new chunk.
- If you must change data in an incompatible way, change the chunk ID or the FORM type ID.

Advanced Topics: CAT, LIST, and PROP

Sometimes you want to put several "files" into one, such as a picture library. This is what CAT is for. It "concatenates" FORM and LIST chunks.



This example CAT holds two ILBMs. It can be shown outline-style:

```
CAT ILBM
..FORM ILBM          \
....BMHD             |  a complete FORM ILBM picture
....CMAP             |
....BODY             /
..FORM ILBM
....BMHD
....CMAP
....BODY
```

Sometimes you want to share the same color map across many pictures. LIST and PROP do this:

```
LIST ILBM
..PROP ILBM          default properties for FORMs ILBM
....CMAP             an ILBM CMAP chunk (there could be a BMHD chunk here, too)
..FORM ILBM
....BMHD             (there could be a CMAP here to override the default)
....BODY
..FORM ILBM
....BMHD             (there could be a CMAP here to override the default)
....BODY
```

A LIST holds PROPs and FORMs (and occasionally LISTs and CATs). A PROP ILBM contains default data (in the above example, just one CMAP chunk) for all FORMs ILBM in the LIST. Any FORM may override the PROP-defined default with its own CMAP. All PROPs must appear at the beginning of a LIST. Each FORM type standardizes (among other things) which of its chunks are "property chunks" (may appear in PROPs) and which are "data chunks" (may not appear in PROPs).