"EA IFF 85" Standard for Interchange Format Files

Document Date: January 14, 1985 (Updated Oct, 1988 Commodore-Amiga, Inc.)
From: Jerry Morrison, Electronic Arts
Status of Standard: Released to the public domain, and in use

1. Introduction

Standards are Good for Software Developers

As home computer hardware evolves into better and better media machines, the demand increases for higher quality, more detailed data. Data development gets more expensive, requires more expertise and better tools, and has to be shared across projects. Think about several ports of a product on one CD-ROM with 500M Bytes of common data!

Development tools need standard interchange file formats. Imagine scanning in images of "player" shapes, transferring them to an image enhancement package, moving them to a paint program for touch up, then incorporating them into a game. Or writing a theme song with a Macintosh score editor and incorporating it into an Amiga game. The data must at times be transformed, clipped, filled out, and moved across machine kinds. Media projects will depend on data transfer from graphic, music, sound effect, animation, and script tools.

Standards are Good for Software Users

Customers should be able to move their own data between independently developed software products. And they should be able to buy data libraries usable across many such products. The types of data objects to exchange are open-ended and include plain and formatted text, raster and structured graphics, fonts, music, sound effects, musical instrument descriptions, and animation.

The problem with expedient file formats—typically memory dumps—is that they're too provincial. By designing data for one particular use (such as a screen snapshot), they preclude future expansion (would you like a full page picture? a multi-page document?). In neglecting the possibility that other programs might read their data, they fail to save contextual information (how many bit planes? what resolution?). Ignoring that other programs might create such files, they're intolerant of extra data (a different picture editor may want to save a texture palette with the image), missing data (such as no color map), or minor variations (perhaps a smaller image). In practice, a filed representation should rarely mirror an in-memory representation. The former should be designed for longevity; the latter to optimize the manipulations of a particular program. The same filed data will be read into different memory formats by different programs.

The IFF philosophy: "A little behind-the-scenes conversion when programs read and write files is far better than NxM explicit conversion utilities for highly specialized formats".

So we need some standardization for data interchange among development tools and products. The more developers that adopt a standard, the better for all of us and our customers.

Here is "EA IFF 1985"

Here is our offering: Electronic Arts' IFF standard for Interchange Eile Eormat. The full name is "EA IFF 1985". Alternatives and justifications are included for certain choices. Public domain subroutine packages and utility programs are available to make it easy to write and use IFF-compatible programs.

Part 1 introduces the standard. Part 2 presents its requirements and background. Parts 3, 4, and 5 define the primitive data types, FORMs, and LISTs, respectively, and how to define new high level types. Part 6 specifies the top level file structure. Section 7 lists names of the group responsible for this standard. Appendix A is included for quick reference and Appendix B.

References

American National Standard Additional Control Codes for Use with ASCII, ANSI standard 3.64-1979 for an 8-bit character set. See also ISO standard 2022 and ISO/DIS standard 6429.2.

The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories. Prentice-Hall, Englewood Cliffs, NJ, 1978.

C, A Reference Manual, Samuel P. Harbison and Guy L. Steele Jr., Tartan Laboratories. Prentice-Hall, Englewood Cliffs, NJ, 1984.

Compiler Construction, An Advanced Course, edited by F. L. Bauer and J. Eickel (Springer-Verlag, 1976). This book is one of many sources for information on recursive descent parsing.

DIF Technical Specification © 1981 by Software Arts, Inc. DIF™ is the format for spreadsheet data interchange developed by Software Arts, Inc. DIF™ is a trademark of Software Arts, Inc.

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

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

M68000 16/32-Bit Microprocessor Programmer's Reference Manual © 1984, 1982, 1980, 1979 by Motorola, Inc.

PostScript Language Manual © 1984 Adobe Systems Incorporated.
PostScript™ is a trademark of Adobe Systems, Inc.
Times and Helvetica® are registered trademarks of Allied Corporation.

Inside Macintosh © 1982, 1983, 1984, 1985 Apple Computer, Inc., a programmer's reference manual.
Apple® is a trademark of Apple Computer, Inc.
MacPaint™ is a trademark of Apple Computer, Inc.
Macintosh™ is a trademark licensed to Apple Computer, Inc.

InterScript: A Proposal for a Standard for the Interchange of Editable Documents © 1984 Xerox Corporation.
Introduction to InterScript © 1985 Xerox Corporation.

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

Electronics Arts™ is a trademark of Electronic Arts.

2. Background for Designers

Part 2 is about the background, requirements, and goals for the standard. It's geared for people who want to design new types of IFF objects. People just interested in using the standard may wish to quickly scan this section.

What Do We Need?

A standard should be long on prescription and short on overhead. It should give lots of rules for designing programs and data files for synergy. But neither the programs nor the files should cost too much more than the expedient variety. Although we are looking to a future with CD-ROMs and perpendicular recording, the standard must work well on floppy disks.

For program portability, simplicity, and efficiency, formats should be designed with more than one implementation style in mind. It ought to be possible to read one of many objects in a file without scanning all the preceding data. (In practice, pure stream I/O is adequate although random access makes it easier to write files.) Some programs need to read and play out their data in real time, so we need good compromises between generality and efficiency.

As much as we need standards, they can't hold up product schedules. So we also need a kind of decentralized extensibility where any software developer can define and refine new object types without some "standards authority" in the loop. Developers must be able to extend existing formats in a forward- and backward-compatible way. A central repository for design information and example programs can help us take full advantage of the standard.

For convenience, data formats should heed the restrictions of various processors and environments. For example, word-alignment greatly helps 68000 access at insignificant cost to 8088 programs.

Other goals include the ability to share common elements over a list of objects and the ability to construct composite objects.

And finally, "Simple things should be simple and complex things should be possible".—Alan Kay.

Think Ahead

Let's think ahead and build programs that read and write files for each other and for programs yet to be designed. Build data formats to last for future computers so long as the overhead is acceptable. This extends the usefulness and life of today's programs and data.

To maximize interconnectivity, the standard file structure and the specific object formats must all be general and extensible. Think ahead when designing an object. File formats should serve many purposes and allow many programs to store and read back all the information they need; even squeeze in custom data. Then a programmer can store the available data and is encouraged to include fixed contextual details. Recipient programs can read the needed parts, skip unrecognized stuff, default missing data, and use the stored context to help transform the data as needed.

Scope

IFF addresses these needs by defining a standard file structure, some initial data object types, ways to define new types, and rules for accessing these files. We can accomplish a great deal by writing programs according to this standard, but do not expect direct compatibility with existing software. We'll need conversion programs to bridge the gap from the old world.

IFF is geared for computers that readily process information in 8-bit bytes. It assumes a "physical layer" of data storage and transmission that reliably maintains "files" as sequences of 8-bit bytes. The standard treats a "file" as a container of data bytes and is independent of how to find a file and whether it has a byte count.

This standard does not by itself implement a clipboard for cutting and pasting data between programs. A clipboard needs software to mediate access, and provide a notification mechanism so updates and requests for data can be detected.

Data Abstraction

The basic problem is *how to represent information* in a way that's program-independent, compiler- independent, machine-independent, and device-independent.

The computer science approach is "data abstraction", also known as "objects", "actors", and "abstract data types". A data abstraction has a "concrete representation" (its storage format), an "abstract representation" (its capabilities and uses), and access procedures that isolate all the calling software from the concrete representation. Only the access procedures touch the data storage. Hiding mutable details behind an interface is called "information hiding". What is hidden are the non-portable details of implementing the object, namely the selected storage representation and algorithms for manipulating it.

The power of this approach is modularity. By adjusting the access procedures we can extend and restructure the data without impacting the interface or its callers. Conversely, we can extend and restructure the interface and callers without making existing data obsolete. It's great for interchange!

But we seem to need the opposite: fixed file formats for all programs to access. Actually, we could file data abstractions ("filed objects") by storing the data and access procedures together. We'd have to encode the access procedures in a standard machine-independent programming language á la PostScript. Even with this, the interface can't evolve freely since we can't update all copies of the access procedures. So we'll have to design our abstract representations for limited evolution and occasional revolution (conversion).

In any case, today's microcomputers can't practically store true data abstractions. They can do the next best thing: store arbitrary types of data in "data chunks", each with a type identifier and a length count. The type identifier is a reference by name to the access procedures (any local implementation). The length count enables storage-level object operations like "copy" and "skip to next" independent of object type or contents.

Chunk writing is straightforward. Chunk reading requires a trivial parser to scan each chunk and dispatch to the proper access/conversion procedure. Reading chunks nested inside other chunks may require recursion, but no look ahead or backup.

That's the main idea of IFF. There are, of course, a few other details…

Previous Work

Where our needs are similar, we borrow from existing standards.

Our basic need to move data between independently developed programs is similar to that addressed by the Apple Macintosh desk scrap or "clipboard" [Inside Macintosh chapter "Scrap Manager"]. The Scrap Manager works closely with the Resource Manager, a handy filer and swapper for data objects (text strings, dialog window templates, pictures, fonts…) including types yet to be designed [Inside Macintosh chapter "Resource Manager"]. The Resource Manager is akin to Smalltalk's object swapper.

We will probably write a Macintosh desk accessory that converts IFF files to and from the Macintosh clipboard for quick and easy interchange with programs like MacPaint and Resource Mover.

Macintosh uses a simple and elegant scheme of four-character "identifiers" to identify resource types, clipboard format types, file types, and file creator programs. Alternatives are unique ID numbers assigned by a central authority or by hierarchical authorities, unique ID numbers generated by algorithm, other fixed length character strings, and variable length strings. Character string identifiers double as readable signposts in data files and programs. The choice of 4 characters is a good tradeoff between storage space, fetch/compare/store time, and name space size. We'll honor Apple's designers by adopting this scheme.

"PICT" is a good example of a standard structured graphics format (including raster images) and its many uses [Inside Macintosh chapter "QuickDraw"]. Macintosh provides QuickDraw routines in ROM to create, manipulate, and display PICTs. Any application can create a PICT by simply asking QuickDraw to record a sequence of drawing commands. Since it's just as easy to ask QuickDraw to render a PICT to a screen or a printer, it's very effective to pass them between programs, say from an illustrator to a word processor. An important feature is the

ability to store "comments" in a PICT which QuickDraw will ignore. (Actually, it passes them to your optional custom "comment handler".)

PostScript, Adobe System's print file standard, is a more general way to represent any print image (which is a specification for putting marks on paper) [PostScript Language Manual]. In fact, PostScript is a full-fledged programming language. To interpret a PostScript program is to render a document on a raster output device. The language is defined in layers: a lexical layer of identifiers, constants, and operators; a layer of reverse polish semantics including scope rules and a way to define new subroutines; and a printing-specific layer of built-in identifiers and operators for rendering graphic images. It is clearly a powerful (Turing equivalent) image definition language. PICT and a subset of PostScript are candidates for structured graphics standards.

A PostScript document can be printed on any raster output device (including a display) but cannot generally be edited. That's because the original flexibility and constraints have been discarded. Besides, a PostScript program may use arbitrary computation to supply parameters like placement and size to each operator. A QuickDraw PICT, in comparison, is a more restricted format of graphic primitives parameterized by constants. So a PICT can be edited at the level of the primitives, e.g. move or thicken a line. It cannot be edited at the higher level of, say, the bar chart data which generated the picture.

PostScript has another limitation: Not all kinds of data amount to marks on paper. A musical instrument description is one example. PostScript is just not geared for such uses.

"DIF" is another example of data being stored in a general format usable by future programs [DIF Technical Specification]. DIF is a format for spreadsheet data interchange. DIF and PostScript are both expressed in plain ASCII text files. This is very handy for printing, debugging, experimenting, and transmitting across modems. It can have substantial cost in compaction and read/write work, depending on use. We won't store IFF files this way but we could define an ASCII alternate representation with a converter program.

InterScript is Xerox' standard for interchange of editable documents [Introduction to InterScript]. It approaches a harder problem: How to represent editable word processor documents that may contain formatted text, pictures, cross-references like figure numbers, and even highly specialized objects like mathematical equations? InterScript aims to define one standard representation for each kind of information. Each InterScript-compatible editor is supposed to preserve the objects it doesn't understand and even maintain nested cross-references. So a simple word processor would let you edit the text of a fancy document without discarding the equations or disrupting the equation numbers.

Our task is similarly to store high level information and preserve as much content as practical while moving it between programs. But we need to span a larger universe of data types and cannot expect to centrally define them all. Fortunately, we don't need to make programs preserve information that they don't understand. And for better or worse, we don't have to tackle general-purpose cross-references yet.

3. Primitive Data Types

Atomic components such as integers and characters that are interpretable directly by the CPU are specified in one format for all processors. We chose a format that's the same as used by the Motorola MC68000 processor [M68000 16/32-Bit Microprocessor Programmer's Reference Manual]. The high byte and high word of a number are stored *first*.

N.B.: Part 3 dictates the format for "primitive" data types where—and only where—used in the overall file structure. The number of such occurrences of dictated formats will be small enough that the costs of conversion, storage, and management of processor-specific files would far exceed the costs of conversion during I/O by "foreign" programs. A particular data chunk may be specified with a different format for its internal primitive types or with processor or environment specific variants if necessary to optimize local usage. Since that hurts data interchange, it's not recommended. (Cf. Designing New Data Sections, in Part 4.).

Alignment

All data objects larger than a byte are aligned on even byte addresses relative to the start of the file. This may require padding. Pad bytes are to be written as zeros, but don't count on that when reading.

This means that every odd-length "chunk" must be padded so that the next one will fall on an even boundary. Also, designers of structures to be stored in chunks should include pad fields where needed to align every field larger than a byte. For best efficiency, long word data should be arranged on long word (4 byte) boundaries. Zeros should be stored in all the pad bytes.

Justification: Even-alignment causes a little extra work for files that are used only on certain processors but allows 68000 programs to construct and scan the data in memory and do block I/O. Any 16 bit or greater CPU will have faster access to aligned data. You just add an occasional pad field to data structures that you're going to block read/write or else stream read/write an extra byte. And the same source code works on all processors. Unspecified alignment, on the other hand, would force 68000 programs to (dis)assemble word and long word data one byte at a time. Pretty cumbersome in a high level language. And if you don't conditionally compile that step out for other processors, you won't gain anything.

Numbers

Numeric types supported are two's complement binary integers in the format used by the MC68000 processor—high byte first, high word first—the reverse of 8088 and 6502 format.

```
UBYTE            8 bits unsigned
WORD            16 bits signed
UWORD           16 bits unsigned
LONG            32 bits signed
```

The actual type definitions depend on the CPU and the compiler. In this document, we'll express data type definitions in the C programming language. [See C, A Reference Manual.] In 68000 Lattice C:

```
typedef unsigned char   UBYTE;    /*  8 bits unsigned */
typedef short           WORD;     /* 16 bits signed   */
typedef unsigned short  UWORD;    /* 16 bits unsigned */
typedef long            LONG;     /* 32 bits signed   */
```

Characters

The following character set is assumed wherever characters are used, e.g. in text strings, IDs, and TEXT chunks (see below). Characters are encoded in 8-bit ASCII. Characters in the range NUL (hex 0) through DEL (hex 7F) are well defined by the 7-bit ASCII standard. IFF uses the graphic group " " (SP, hex 20) through "~" (hex 7E).

Most of the control character group hex 01 through hex 1F have no standard meaning in IFF. The control character LF (hex 0A) is defined as a "newline" character. It denotes an intentional line break, that is, a paragraph or line terminator. (There is no way to store an automatic line break. That is strictly a function of the margins in the environment the text is placed.) The control character ESC (hex 1B) is a reserved escape character under the rules of ANSI standard 3.64-1979 American National Standard Additional Control Codes for Use with ASCII, ISO standard 2022, and ISO/DIS standard 6429.2.

Characters in the range hex 7F through hex FF are not globally defined in IFF. They are best left reserved for future standardization. (Note that the FORM type FTXT (formatted text) defines the meaning of these characters within FTXT forms.) In particular, character values hex 7F through hex 9F are control codes while characters hex A0 through hex FF are extended graphic characters like Å, as per the ISO and ANSI standards cited above. [See the supplementary document "FTXT" IFF Formatted Text.]

Dates

A "creation date" is defined as the date and time a stream of data bytes was created. (Some systems call this a "last modified date".) Editing some data changes its creation date. Moving the data between volumes or machines does not.

The IFF standard date format will be one of those used in MS-DOS, Macintosh, or AmigaDOS (probably a 32-bit unsigned number of seconds since a reference point). Issue: Investigate these three.

Type IDs

A "type ID", "property name", "FORM type", or any other IFF identifier is a 32-bit value: the concatenation of four ASCII characters in the range " " (SP, hex 20) through "~" (hex 7E). Spaces (hex 20) should not precede printing characters; trailing spaces are ok. Control characters are forbidden.

```
typedef CHAR ID[4];
```

IDs are compared using a simple 32-bit case-dependent equality test. FORM type IDs are restricted. Since they may be stored in filename extensions lower case letters and punctuation marks are forbidden. Trailing spaces are ok.

Carefully choose those four characters when you pick a new ID. Make them mnemonic so programmers can look at an interchange format file and figure out what kind of data it contains. The name space makes it possible for developers scattered around the globe to generate ID values with minimal collisions so long as they choose specific names like "MUS4" instead of general ones like "TYPE" and "FILE".

Commodore-Amiga Technical Support has undertaken the task of maintaining the registry of FORM type IDs and format descriptions. See the IFF registry document for more information.

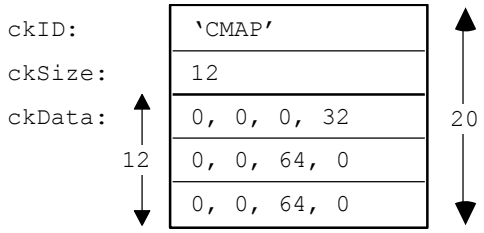
Sometimes it's necessary to make data format changes that aren't backward compatible. As much as we work for compatibility, unintended interactions can develop. Since IDs are used to denote data formats in IFF, new IDs are chosen to denote revised formats. Since programs won't read chunks whose IDs they don't recognize (see Chunks, below), the new IDs keep old programs from stumbling over new data. The conventional way to chose a "revision" ID is to increment the last character if it's a digit or else change the last character to a digit. E.g. first and second revisions of the ID "XY" would be "XY1" and "XY2". Revisions of "CMAP" would be "CMA1" and "CMA2".

Chunks

Chunks are the building blocks in the IFF structure. The form expressed as a C typedef is:

```
typedef struct {
        ID                      ckID;/* 4 character ID */
        LONG                    ckSize;     /* sizeof(ckData) */
        UBYTE                   ckData[/* ckSize */];
        } Chunk;
```

We can diagram an example chunk—a "CMAP" chunk containing 12 data bytes—like this:



That's 4 bytes of `ckID`, 4 bytes of `ckSize` and 12 data bytes. The total space used is 20 bytes.

The `ckID` identifies the format and purpose of the chunk. As a rule, a program must recognize `ckID` to interpret `ckData`. It should skip over all unrecognized chunks. The `ckID` also serves as a format version number as long as we pick new IDs to identify new formats of `ckData` (see above).

The following `ckID`s are universally reserved to identify chunks with particular IFF meanings: "LIST", "FORM", "PROP", "CAT ", and "   ". The special ID "   " (4 spaces) is a `ckID` for "filler" chunks, that is, chunks that fill space but have no meaningful contents. The IDs "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9" are reserved for future "version number" variations. All IFF-compatible software must account for these chunk IDs.

The ckSize is a logical block size—how many data bytes are in ckData. If `ckData` is an odd number of bytes long, a 0 pad byte follows which is <u>not</u> included in `ckSize`. (Cf. Alignment.) A chunk's total physical size is `ckSize` rounded up to an even number plus the size of the header. So the smallest chunk is 8 bytes long with `ckSize = 0`. For the sake of following chunks, programs must respect every chunk's ckSize as a virtual end-of-file for reading its `ckData` even if that data is malformed, e.g. if nested contents are truncated.

We can describe the syntax of a chunk as a regular expression with "`#`" representing the `ckSize`, the length of the following {braced} bytes. The "`[0]`" represents a sometimes needed pad byte. (The regular expressions in this document are collected in Appendix A along with an explanation of notation.)

```
Chunk   ::= ID #{ UBYTE* } [0]
```

One chunk output technique is to stream write a chunk header, stream write the chunk contents, then random access back to the header to fill in the size. Another technique is to make a preliminary pass over the data to compute the size, then write it out all at once.

Strings, String Chunks, and String Properties

In a string of ASCII text, linefeed (0x0A) denotes a forced line break (paragraph or line terminator). Other control characters are not used. (Cf. Characters.) For maximum compatibility with line editors, two linefeed characters are often used to indicate a paragraph boundary.

The `ckID` for a chunk that contains a string of plain, unformatted text is "TEXT". As a practical matter, a text string should probably not be longer than 32767 bytes. The standard allows up to 2^{31} - 1 bytes. The `ckID` "TEXT" is globally reserved for this use.

When used as a data property (see below), a text string chunk may be 0 to 255 characters long. Such a string is readily converted to a C string or a Pascal `STRING[255]`. The `ckID` of a property must have a unique property name, *not* "TEXT".

When used as a part of a chunk or data property, restricted C string format is normally used. That means 0 to 255 characters followed by a NULL byte (ASCII value 0).

Data Properties (advanced topic)

Data properties specify attributes for following (non-property) chunks. A data property essentially says "identifier = value", for example "XY = (10, 200)", telling something about following chunks. Properties may only appear inside data sections ("FORM" chunks, cf. Data Sections) and property sections ("PROP" chunks, cf. Group PROP).

The form of a data property is a type of Chunk. The `ckID` is a property name as well as a property type. The `ckSize` should be small since data properties are intended to be accumulated in RAM when reading a file. (256 bytes is a reasonable upper bound.) Syntactically:

```
Property      ::= Chunk
```

When designing a data object, use properties to describe context information like the size of an image, even if they don't vary in your program. Other programs will need this information.

Think of property settings as assignments to variables in a programming language. Multiple assignments are redundant and local assignments temporarily override global assignments. The order of assignments doesn't matter as long as they precede the affected chunks. (Cf. LISTs, CATs, and Shared Properties.)

Each object type (FORM type) is a local name space for property IDs. Think of a "CMAP" property in a "FORM ILBM" as the qualified ID "ILBM.CMAP". A "CMAP" inside some other type of FORM may not have the same meaning. Property IDs specified when an object type is designed (and therefore known to all clients) are called "standard" while specialized ones added later are "nonstandard".

Links

Issue: A standard mechanism for "links" or "cross references" is very desirable for things like combining images and sounds into animations. Perhaps we'll define "link" chunks within FORMs that refer to other FORMs or to specific chunks within the same and other FORMs. This needs further work. EA IFF 1985 has no standard link mechanism.

For now, it may suffice to read a list of, say, musical instruments, and then just refer to them within a musical score by sequence number.

File References

Issue: We may need a standard form for references to other files. A "file ref" could name a directory and a file in the same type of operating system as the reference's originator. Following the reference would expect the file to be on some mounted volume, or perhaps the same directory as the file that made the reference. In a network environment, a file reference could name a server, too.

Issue: How can we express operating-system independent file references?
Issue: What about a means to reference a portion of another file? Would this be a "file ref" plus a reference to a "link" within the target file?

4. Data Sections

The first thing we need of a file is to check: Does it contain IFF data and, if so, does it contain the kind of data we're looking for? So we come to the notion of a "data section".

A "data section" or IFF "FORM" is one self-contained "data object" that might be stored in a file by itself. It is one high level data object such as a picture or a sound effect, and generally contains a grouping of chunks. The IFF structure "FORM" makes it self- identifying. It could be a composite object like a musical score with nested musical instrument descriptions.

Group FORM

A data section is a chunk with `ckID` "FORM" and this arrangement:

```
FORM    ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
FormType      ::= ID
LocalChunk    ::= Property | Chunk
```

The ID "FORM" is a syntactic keyword like "struct" in C. Think of a "struct ILBM" containing a field "CMAP". If you see "FORM" you will know to expect a FORM type ID (the structure name, "ILBM" in this example) and a particular contents arrangement or "syntax" (local chunks, FORMs, LISTs, and CATs). A "FORM ILBM", in particular, might contain a local chunk "CMAP", an "ILBM.CMAP" (to use a qualified name).

So the chunk ID "FORM" indicates a data section. It implies that the chunk contains an ID and some number of nested chunks. In reading a FORM, like any other chunk, programs must respect its `ckSize` as a virtual end-of-file for reading its contents, even if they're truncated.

The FORM type is a restricted ID that may not contain lower case letters or punctuation characters. (Cf. Type IDs. Cf. Single Purpose Files.)

The type-specific information in a FORM is composed of its "local chunks": data properties and other chunks. Each FORM type is a local name space for local chunk IDs. So "CMAP" local chunks in other FORM types may be unrelated to "ILBM.CMAP". More than that, each FORM type defines semantic scope. If you know what a FORM ILBM is, you will know what an ILBM.CMAP is.

Local chunks defined when the FORM type is designed (and therefore known to all clients of this type) are called "standard" while specialized ones added later are "nonstandard".

Among the local chunks, property chunks give settings for various details like text font while the other chunks supply the essential information. This distinction is not clear cut. A property setting can be cancelled by a later setting of the same property. E.g. in the sequence:

```
prop1 = x  (Data A)  prop1 = z  prop1 = y (Data B)
```

`prop1 is = x` for Data A, and `y` for Data B. The setting `prop1 = z` has no effect.

For clarity, the universally reserved chunk IDs "LIST", "FORM", "PROP", "CAT ", "   ", "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9" may not be FORM type IDs.

Part 5, below, talks about grouping FORMs into LISTs and CATs. They let you group a bunch of FORMs but don't impose any particular meaning or constraints on the grouping. Read on.

Composite FORMs

A FORM chunk inside a FORM is a full-fledged data section. This means you can build a composite object such as a multi-frame animation sequence by nesting available picture FORMs and sound effect FORMs. You can insert additional chunks with information like frame rate and frame count.

Using composite FORMs, you leverage on existing programs that create and edit the component FORMs. Those editors may even look into your composite object to copy out its type of component. Such editors are <u>not</u> allowed to replace their component objects within your composite object. That's because the IFF standard lets you specify

consistency requirements for the composite FORM such as maintaining a count or a directory of the components. Only programs that are written to uphold the rules of your FORM type may create or modify such FORMs.

Therefore, in designing a program that creates composite objects, you are strongly requested to provide a facility for your users to import and export the nested FORMs. Import and export could move the data through a clipboard or a file.

Here are several existing FORM types and rules for defining new ones:

FTXT

An FTXT data section contains text with character formatting information like fonts and faces. It has no paragraph or document formatting information like margins and page headers. FORM FTXT is well matched to the text representation in Amiga's Intuition environment. See the supplemental document "FTXT" IFF Formatted Text.

ILBM

"ILBM" is an InterLeaved BitMap image with color map; a machine-independent format for raster images. FORM ILBM is the standard image file format for the Commodore-Amiga computer and is useful in other environments, too. See the supplemental document "ILBM" IFF Interleaved Bitmap.

PICS

The data chunk inside a "PICS" data section has ID "PICT" and holds a QuickDraw picture. Issue: Allow more than one PICT in a PICS? See Inside Macintosh chapter "QuickDraw" for details on PICTs and how to create and display them on the Macintosh computer.

The only standard property for PICS is "XY", an optional property that indicates the position of the PICT relative to "the big picture". The contents of an XY is a QuickDraw Point.

Note: PICT may be limited to Macintosh use, in which case there'll be another format for structured graphics in other environments.

Other Macintosh Resource Types

Some other Macintosh resource types could be adopted for use within IFF files; perhaps MWRT, ICN, ICN#, and STR#.

Issue: Consider the candidates and reserve some more IDs.

Designing New Data Sections

Supplemental documents will define additional object types. A supplement needs to specify the object's purpose, its FORM type ID, the IDs and formats of standard local chunks, and rules for generating and interpreting the data. It's a good idea to supply typedefs and an example source program that accesses the new object. See "ILBM" IFF Interleaved Bitmap for such an example.

Anyone can pick a new FORM type ID but should reserve it with Commodore-Amiga Technical Support (CATS) at their earliest convenience. While decentralized format definitions and extensions are possible in IFF, our preference is to get design consensus by committee, implement a program to read and write it, perhaps tune the format before it becomes locked in stone, and then publish the format with example code. Some organization should remain in charge of answering questions and coordinating extensions to the format.

If it becomes necessary to incompatibly revise the design of some data section, its FORM type ID will serve as a version number (Cf. Type IDs). E.g. a revised "VDEO" data section could be called "VDE1". But try to get by with compatible revisions within the existing FORM type.

In a new FORM type, the rules for primitive data types and word-alignment (Cf. Primitive Data Types) may be overridden for the contents of its local chunks—but not for the chunk structure itself—if your documentation spells out the deviations. If machine-specific type variants are needed, e.g. to store vast numbers of integers in reverse bit order, then outline the conversion algorithm and indicate the variant inside each file, perhaps via different FORM types. Needless to say, variations should be minimized.

In designing a FORM type, encapsulate all the data that other programs will need to interpret your files. E.g. a raster graphics image should specify the image size even if your program always uses 320 x 200 pixels x 3 bitplanes. Receiving programs are then empowered to append or clip the image rectangle, to add or drop bitplanes, etc. This enables a lot more compatibility.

Separate the central data (like musical notes) from more specialized information (like note beams) so simpler programs can extract the central parts during read-in. Leave room for expansion so other programs can squeeze in new kinds of information (like lyrics). And remember to keep the property chunks manageably short—let's say ≤ 256 bytes.

When designing a data object, try to strike a good tradeoff between a super-general format and a highly-specialized one. Fit the details to at least one particular need, for example a raster image might as well store pixels in the current machine's scan order. But add the kind of generality that makes the format usable with foreseeable hardware and software. E.g. use a whole byte for each red, green, and blue color value even if this year's computer has only 4-bit video DACs. Think ahead and help other programs so long as the overhead is acceptable. E.g. run compress a raster by scan line rather than as a unit so future programs can swap images by scan line to and from secondary storage.

Try to design a general purpose "least common multiple" format that encompasses the needs of many programs without getting too complicated. Be sure to leave provisions for future expansion. Let's coalesce our uses around a few such formats widely separated in the vast design space. Two factors make this flexibility and simplicity practical. First, file storage space is getting very plentiful, so compaction is not always a priority. Second, nearly any locally-performed data conversion work during file reading and writing will be cheap compared to the I/O time.

It must be ok to copy a LIST or FORM or CAT intact, e.g. to incorporate it into a composite FORM. So any kind of internal references within a FORM must be relative references. They could be relative to the start of the containing FORM, relative from the referencing chunk, or a sequence number into a collection.

With composite FORMs, you leverage on existing programs that create and edit the components. If you write a program that creates composite objects, please provide a facility for users to import and export the nested FORMs.

Finally, don't forget to specify all implied rules in detail.

5. LISTs, CATs, and Shared Properties (Advanced topics)

Data often needs to be grouped together, for example, consider a list of icons. Sometimes a trick like arranging little images into a big raster works, but generally they'll need to be structured as a first class group. The objects "LIST" and "CAT" are IFF-universal mechanisms for this purpose. Note: LIST and CAT are advanced topics the first time reader will want to skip.

Property settings sometimes need to be shared over a list of similar objects. E.g. a list of icons may share one color map. LIST provides a means called "PROP" to do this. One purpose of a LIST is to define the scope of a PROP. A "CAT", on the other hand, is simply a concatenation of objects.

Simpler programs may skip LISTs and PROPs altogether and just handle FORMs and CATs. All "fully-conforming" IFF programs also know about "CAT ", "LIST", and "PROP". Any program that reads a FORM inside a LIST must process shared PROPs to correctly interpret that FORM.

Group CAT

A CAT is just an untyped group of data objects.

Structurally, a CAT is a chunk with chunk ID "CAT " containing a "contents type" ID followed by the nested objects. The `ckSize` of each contained chunk is essentially a relative pointer to the next one.

```
CAT                                 ::= "CAT " #{ ContentsType (FORM | LIST |
                CAT)* }
ContentsType                        ::= ID    -- a hint or an "abstract data
                type" ID
```

In reading a CAT, like any other chunk, programs must respect it's `ckSize` as a virtual end-of-file for reading the nested objects even if they're malformed or truncated.

The "contents type" following the CAT's `ckSize` indicates what kind of FORMs are inside. So a CAT of ILBMs would store "ILBM" there. It's just a hint. It may be used to store an "abstract data type". A CAT could just have blank contents ID (" ") if it contains more than one kind of FORM.

CAT defines only the format of the group. The group's meaning is open to interpretation. This is like a list in LISP: the structure of cells is predefined but the meaning of the contents as, say, an association list depends on use. If you need a group with an enforced meaning (an "abstract data type" or Smalltalk "subclass"), some consistency constraints, or additional data chunks, use a composite FORM instead (Cf. Composite FORMs).

Since a CAT just means a concatenation of objects, CATs are rarely nested. Programs should really merge CATs rather than nest them.

Group LIST

A LIST defines a group very much like CAT but it also gives a scope for PROPs (see below). And unlike CATs, LISTs should not be merged without understanding their contents.

Structurally, a LIST is a chunk with `ckID` "LIST" containing a "contents type" ID, optional shared properties, and the nested contents (FORMs, LISTs, and CATs), in that order. The `ckSize` of each contained chunk is a relative pointer to the next one. A LIST is not an arbitrary linked list—the cells are simply concatenated.

```
LIST    ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
ContentsType  ::= ID
```

Group PROP

PROP chunks may appear in LISTs (not in FORMs or CATs). They supply shared properties for the FORMs in that LIST. This ability to elevate some property settings to shared status for a list of forms is useful for both indirection and compaction. E.g. a list of images with the same size and colors can share one "size" property and one "color map" property. Individual FORMs can override the shared settings.

The contents of a PROP is like a FORM with no data chunks:

```
PROP    ::= "PROP" #{ FormType Property* }
```

It means, "Here are the shared properties for FORM type <FormType>".

A LIST may have at most one PROP of a FORM type, and all the PROPs must appear before any of the FORMs or nested LISTs and CATs. You can have subsequences of FORMs sharing properties by making each subsequence a LIST.

Scoping: Think of property settings as variable bindings in nested blocks of a programming language. In C this would look like:

```
#define Roman             0
#define Helvetica         1

void main()
     {
     int font=Roman;      /* The global default */
        {
        printf("The font number is %d\n",font);
        }
        {
        int font=Helvetica;                                          /* local setting
        printf("The font number is %d\n",font);
        }
        {
        printf("The font number is %d\n",font);
        }
     }
/*
 * Sample output:         The font number is 0
 *                                                                   The font number i
 *                                                                   The font number i
 */
```

An IFF file could contain:

```
LIST {
     PROP TEXT {
        FONT {TimesRoman}              /* shared setting              */
        }

     FORM TEXT {
        FONT {Helvetica}               /* local setting               */
        CHRS {Hello }                  /* uses font Helvetica         */
        }

     FORM TEXT {
        CHRS {there.}                  /* uses font TimesRoman        */
        }
     }
```

The shared property assignments selectively override the reader's global defaults, but only for FORMs within the group. A FORM's own property assignments selectively override the global and group-supplied values. So when reading an IFF file, keep property settings on a stack. They are designed to be small enough to hold in main memory.

Shared properties are semantically equivalent to copying those properties into each of the nested FORMs right after their FORM type IDs.

Properties for LIST

Optional "properties for LIST" store the origin of the list's contents in a PROP chunk for the pseudo FORM type "LIST". They are the properties originating program "OPGM", processor family "OCPU", computer type "OCMP", computer serial number or network address "OSN ", and user name "UNAM". In our imperfect world, these could be called upon to distinguish between unintended variations of a data format or to work around bugs in particular originating/receiving program pairs. Issue: Specify the format of these properties.

A creation date could also be stored in a property, but let's ask that file creating, editing, and transporting programs maintain the correct date in the local file system. Programs that move files between machine types are expected to copy across the creation dates.

6. Standard File Structure

File Structure Overview

An IFF file is just a single chunk of type FORM, LIST, or CAT. Therefore an IFF file can be recognized by its first 4 bytes: "FORM", "LIST", or "CAT ". Any file contents after the chunk's end are to be ignored. (Some file transfer programs add garbage to the end of transferred files. This specification protects against such common damage).

The simplest IFF file would be one that does no more than encapsulate some binary data (perhaps even an old-fashioned single-purpose binary file). Here is a binary dump of such a minimal IFF example:

```
0000: 464F524D 0000001A 534E4150 43524143    FORM....SNAPCRAC
0010: 0000000D 68656C6C 6F2C776F 726C6421    ....hello,world!
0020: 0A00                                   ..
```

The first 4 bytes indicate this is a "FORM"; the most common IFF top level structure. The following 4 bytes indicate that the contents totals 26 bytes. The form type is listed as "SNAP".

Our form "SNAP" contains only one chunk at the moment; a chunk of type "CRAC". From the size ($0000000D) the amount of data must be 13 bytes. In this case, the data happens to correspond to the ASCII string "`hello, world!`<lf>". Since the number 13 is odd, a zero pad byte is added to the file. At any time new chunks could be added to form SNAP without affecting any other aspect of the file (other than the form size). It's that simple.

Since an IFF file can be a group of objects, programs that read/write single objects can communicate to an extent with programs that read/write groups. You're encouraged to write programs that handle all the objects in a LIST or CAT. A graphics editor, for example, could process a list of pictures as a multiple page document, one page at a time.

Programs should enforce IFF's syntactic rules when reading and writing files. Users should be told when a file is corrupt. This ensures robust data transfer. For minor damage, you may wish to give the user the option of using the suspect data, or cancelling. Presumably a user could read in a damaged file, then save whatever was salvaged to a valid file. The public domain IFF reader/writer subroutine package does some syntatic checks for you. A utility program "IFFCheck" is available that scans an IFF file and checks it for conformance to IFF's syntactic rules. IFFCheck also prints an outline of the chunks in the file, showing the `ckID` and `ckSize` of each. This is quite handy when building IFF programs. Example programs are also available to show details of reading and writing IFF files.

A merge program "IFFJoin" will be available that logically appends IFF files into a single CAT group. It "unwraps" each input file that is a CAT so that the combined file isn't nested CATs.

If we need to revise the IFF standard, the three anchoring IDs will be used as "version numbers". That's why IDs "FOR1" through "FOR9", "LIS1" through "LIS9", and "CAT1" through "CAT9" are reserved.

IFF formats are designed for reasonable performance with floppy disks. We achieve considerable simplicity in the formats and programs by relying on the host file system rather than defining universal grouping structures like directories for LIST contents. On huge storage systems, IFF files could be leaf nodes in a file structure like a B-tree. Let's hope the host file system implements that for us!

There are two kinds of IFF files: single purpose files and scrap files. They differ in the interpretation of multiple data objects and in the file's external type.

Single Purpose Files

A single purpose IFF file is for normal "document" and "archive" storage. This is in contrast with "scrap files" (see below) and temporary backing storage (non-interchange files).

The external file type (or filename extension, depending on the host file system) indicates the file's contents. It's generally the FORM type of the data contained, hence the restrictions on FORM type IDs.

Programmers and users may pick an "intended use" type as the filename extension to make it easy to filter for the relevant files in a filename requester. This is actually a "subclass" or "subtype" that conveniently separates files of the same FORM type that have different uses. Programs cannot demand conformity to its expected subtypes without overly restricting data interchange since they cannot know about the subtypes to be used by future programs that users will want to exchange data with.

Issue: How to generate 3-letter MS-DOS extensions from 4-letter FORM type IDs?

Most single purpose files will be a single FORM (perhaps a composite FORM like a musical score containing nested FORMs like musical instrument descriptions). If it's a LIST or a CAT, programs should skip over unrecognized objects to read the recognized ones or the first recognized one. Then a program that can read a single purpose file can read something out of a "scrap file", too.

Scrap Files (not currently used)

A "scrap file" is for maximum interconnectivity in getting data between programs; the core of a clipboard function. Scrap files may have type "IFF " or filename extension ".IFF".

A scrap file is typically a CAT containing alternate representations of the same basic information. Include as many alternatives as you can readily generate. This redundancy improves interconnectivity in situations where we can't make all programs read and write super-general formats. [Inside Macintosh chapter "Scrap Manager".] E.g. a graphically-annotated musical score might be supplemented by a stripped down 4-voice melody and by a text (the lyrics).

The originating program should write the alternate representations in order of "preference": most preferred (most comprehensive) type to least preferred (least comprehensive) type. A receiving program should either use the first appearing type that it understands or search for its own "preferred" type.

A scrap file should have at most one alternative of any type. (A LIST of same type objects is ok as one of the alternatives.) But don't count on this when reading; ignore extra sections of a type. Then a program that reads scrap files can read something out of single purpose files.

Rules for Reader Programs

Here are some notes on building programs that read IFF files. If you use the standard IFF reader module "IFFR.C", many of these rules and details will be automatically handled. (See "Support Software" in Appendix A.) We recommend that you start from the example program "ShowILBM.C". For LIST and PROP work, you should also read up on recursive descent parsers. [See, for example, Compiler Construction, An Advanced Course.]

- The standard is very flexible so many programs can exchange data. This implies a program has to scan the file and react to what's actually there in whatever order it appears. An IFF reader program is a parser.

- For interchange to really work, programs must be willing to do some conversion during read-in. If the data isn't exactly what you expect, say, the raster is smaller than those created by your program, then adjust it. Similarly, your program could crop a large picture, add or drop bitplanes, or create/discard a mask plane. The program should give up gracefully on data that it can't convert.

- If it doesn't start with "FORM", "LIST", or "CAT ", it's not an IFF-85 file.

- For any chunk you encounter, you must recognize its type ID to understand its contents.

- For any FORM chunk you encounter, you must recognize its FORM type ID to understand the contained "local chunks". Even if you don't recognize the FORM type, you can still scan it for nested FORMs, LISTs, and CATs of interest.

- Don't forget to skip the implied pad byte after every odd-length chunk, this is *not* included in the chunk count!

- Chunk types LIST, FORM, PROP, and CAT are generic groups. They always contain a subtype ID followed by chunks.

- Readers ought to handle a CAT of FORMs in a file. You may treat the FORMs like document pages to sequence through, or just use the first FORM.

- Many IFF readers completely skip LISTs. "Fully IFF-conforming" readers are those that handle LISTs, even if just to read the first FORM from a file. If you do look into a LIST, you must process shared properties (in PROP chunks) properly. The idea is to get the correct data or none at all.

- The nicest readers are willing to look into unrecognized FORMs for nested FORM types that they do recognize. For example, a musical score may contain nested instrument descriptions and and animation or desktop publishing files may contain still pictures. This extra step is highly recommended.

Note to programmers: Processing PROP chunks is not simple! You'll need some background in interpreters with stack frames. If this is foreign to you, build programs that read/write only one FORM per file. For the more intrepid programmers, the next paragraph summarizes how to process LISTs and PROPs. See the general IFF reader module "IFFR.C" and the example program "ShowILBM.C" for details.

Allocate a stack frame for every LIST and FORM you encounter and initialize it by copying the stack frame of the parent LIST or FORM. At the top level, you'll need a stack frame initialized to your program's global defaults. While reading each LIST or FORM, store all encountered properties into the current stack frame. In the example ShowILBM, each stack frame has a place for a bitmap header property ILBM.BMHD and a color map property ILBM.CMAP. When you finally get to the ILBM's BODY chunk, use the property settings accumulated in the current stack frame.

An alternate implementation would just remember PROPs encountered, forgetting each on reaching the end of its scope (the end of the containing LIST). When a FORM XXXX is encountered, scan the chunks in all remembered PROPs XXXX, in order, as if they appeared before the chunks actually in the FORM XXXX. This gets trickier if you read FORMs inside of FORMs.

Rules for Writer Programs

Here are some notes on building programs that write IFF files, which is much easier than reading them. If you use the standard IFF writer module "IFFW.C" , many of these rules and details will automatically be enforced. See the example program "Raw2ILBM.C".

- An IFF file is a single FORM, LIST, or CAT chunk.
- Any IFF-85 file must start with the 4 characters "FORM", "LIST", or "CAT ", followed by a LONG `ckSize`. There should be no data after the chunk end.
- Chunk types LIST, FORM, PROP, and CAT are generic. They always contain a subtype ID followed by chunks. These three IDs are universally reserved, as are "LIS1" through "LIS9", "FOR1" through "FOR9", "CAT1" through "CAT9", and " ".
- Don't forget to write a 0 pad byte after each odd-length chunk.
- Do not try to edit a file that you don't know how to create. Programs may look into a file and copy out nested FORMs of types that they recognize, but they should not edit and replace the nested FORMs and not add or remove them. Breaking these rules could make the containing structure inconsistent. You may write a new file containing items you copied, or copied and modified, but don't copy structural parts you don't understand.
- You must adhere to the syntax descriptions in Appendix A. E.g. PROPs may only appear inside LISTs.

There are at least four common techniques for writing an IFF group:

(1) build the data in a file mapped into virtual memory.
(2) build the data in memory blocks and use block I/O.
(3) stream write the data piecemeal and (don't forget!) random access back to set the group (or FORM) length count.
(4) make a preliminary pass to compute the length count then stream write the data.

Issue: The standard disallows "blind" chunk copying for consistency reasons. Perhaps we can define a ckID convention for chunks that are ok to replicate without knowledge of the contents. Any such chunks would need to be internally consistent, and not be bothered by changed external references.

Issue: Stream-writing an IFF FORM can be inconvenient. With random access files one can write all the chunks then go back to fix up the FORM size. With stream access, the FORM size must be calculated before the file is written. When compression is involved, this can be slow or inconvenient. Perhaps we can define an "END " chunk. The stream writer would use -1 ($FFFFFFFF) as the FORM size. The reader would follow each chunk, when the reader reaches an "END ", it would terminate the last -1 sized chunk. Certain new IFF FORMs could require that readers understand "END ".

7. Standards Committee

The following people contributed to the design of this IFF standard:

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Barry Walsh, Commodore-Amiga
Oct, 1988 revision by Bryce Nesbitt, and Carolyn Scheppner, Commodore-Amiga

Appendix A. Reference

Type Definitions

The following C typedefs describe standard IFF structures. Declarations to use in practice will vary with the CPU and compiler. For example, 68000 Lattice C produces efficient comparison code if we define ID as a "LONG". A macro "MakeID" builds these IDs at compile time.

```
/* Standard IFF types, expressed in 68000 Lattice C.           */

typedef unsigned char UBYTE;      /*  8 bits unsigned          */
typedef short WORD;               /* 16 bits signed            */
typedef unsigned short UWORD;     /* 16 bits unsigned          */
typedef long LONG;                /* 32 bits signed            */

typedef char ID[4];               /* 4 chars in ' ' through '~' */

typedef struct {
        ID                        ckID;
        LONG                      ckSize;                     /*
  sizeof(ckData)                  */
        UBYTE                     ckData[/* ckSize */];
        } Chunk;

/* ID typedef and builder for 68000 Lattice C. */
typedef LONG ID;                  /* 4 chars in ' ' through '~' */
#define MakeID(a,b,c,d) ( (a)<<24 | (b)<<16 | (c)<<8 | (d) )

/* Globally reserved IDs. */
#define ID_FORM   MakeID('F','O','R','M')
#define ID_LIST   MakeID('L','I','S','T')
#define ID_PROP   MakeID('P','R','O','P')
#define ID_CAT    MakeID('C','A','T',' ')
#define ID_FILLER MakeID(' ',' ',' ',' ')
```

Syntax Definitions

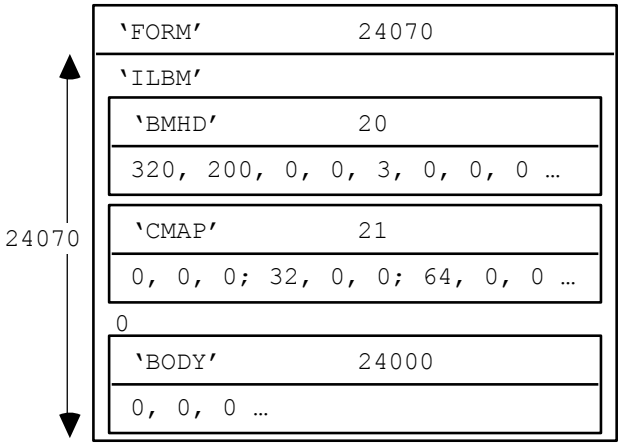
Here's a collection of the syntax definitions in this document.

```
Chunk                             ::= ID #{ UBYTE* } [0]

Property                          ::= Chunk

FORM                              ::= "FORM" #{ FormType (LocalChunk |
            FORM | LIST | CAT)* }
FormType                          ::= ID
LocalChunk                        ::= Property | Chunk

CAT                               ::= "CAT " #{ ContentsType (FORM | LIST |
            CAT)* }
ContentsType                      ::= ID    -- a hint or an "abstract data
            type" ID

LIST                              ::= "LIST" #{ ContentsType PROP* (FORM |
            LIST | CAT)* }
PROP                              ::= "PROP" #{ FormType Property* }
```

In this extended regular expression notation, the token "`#`" represents a count of the following {braced} data bytes. Literal items are shown in "quotes", [square bracketed items] are optional, and "`*`" means 0 or more instances. A sometimes-needed pad byte is shown as "`[0]`".

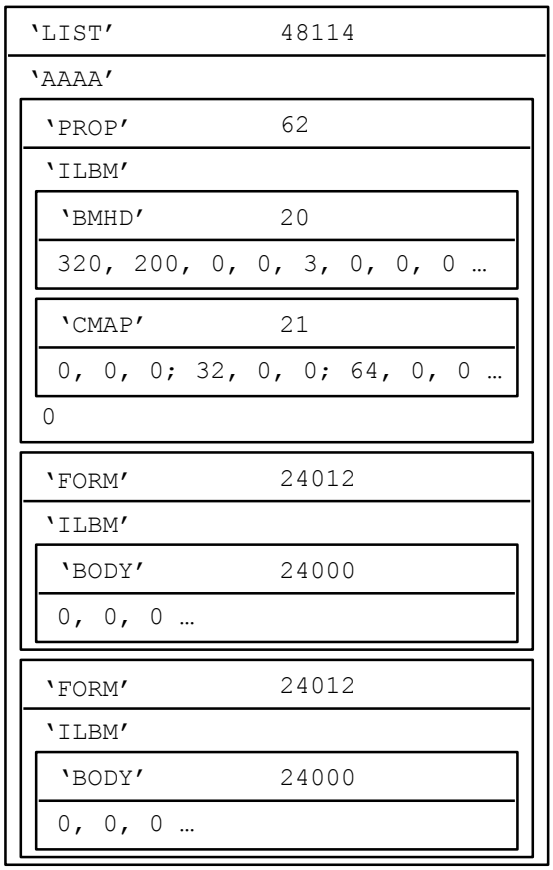
Example Diagrams

Here's a box diagram for an example IFF file, a raster image FORM ILBM. This FORM contains a bitmap header property chunk BMHD, a color map property chunk CMAP, and a raster data chunk BODY. This particular raster is 320 x 200 pixels x 3 bit planes uncompressed. The "0" after the CMAP chunk represents a zero pad byte; included since the CMAP chunk has an odd length. The text to the right of the diagram shows the outline that would be printed by the IFFCheck utility program for this particular file.




```
FORM  24070  ILBM
.BMHD  20
.CMAP  21
.BODY  24000
```

This second diagram shows a LIST of two FORMs ILBM sharing a common BMHD property and a common CMAP property. Again, the text on the right is an outline á la IFFCheck.




```
LIST  48114  AAAA
.PROP  62  ILBM
..BMHD  20
..CMAP  21
.FORM  24012  ILBM
..BODY  24000
.FORM  24012  ILBM
..BODY  24000
```