# "EA IFF 85"
# STANDARD FOR INTERCHANGE FORMAT FILES

AUGUST 1, 1987

# "EA IFF 85"
# STANDARD FOR INTERCHANGE FORMAT FILES

**AUGUST 1, 1987**

# "EA IFF 85" Standard for Interchange Format Files

## 1. Introduction

### Standards are Good for Software Developers

As home computer hardware evolves to 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, moving them to a paint program for editing, 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 (e.g. 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 (texture palette for a picture editor), missing data (no color map), or minor variations (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 File Format. 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. Appendix A is included for quick reference and Appendix B names the committee responsible for this standard.

## 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.

Amiga™ is a trademark of Commodore-Amiga, Inc.

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.

Electronic Arts™ is a trademark of Electronic Arts.

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

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

"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 trademarks of Allied Corporation.

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

## 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 skip this part.

### 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. While we're 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. (In practice, pure stream I/O is adequate although random access makes it easier to write files.) It ought to be possible to read one of many objects in a file without scanning all the preceding data. 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. E.g. 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 containing other data objects with structural information like directories.

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. It 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 don't 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 strings 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, to maintain a "contents version number" so programs can detect updates, and to manage the data in "virtual memory".

## 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 data abstraction does is abstract from 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 still, 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 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.

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 requires recursion, but no lookahead 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 a kin 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 4-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'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 can'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 most convenient for the Motorola MC68000 processor [M68000 16/32-Bit Microprocessor Programmer's Reference Manual].

N.B.: Part 3 dictates the format for "primitive" data types where—and only where—used in the overall file structure and standard kinds of chunks (Cf. Chunks). The number of such occurrences 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" (see below) 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. 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. 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 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. They could potentially include signed and unsigned 8, 16, and 32 bit integers but the standard only uses the following:

```
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. But 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 Amiga DOS (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.

Data section type IDs (aka FORM types) are restriced IDs. (Cf. Data Sections.) Since they may be stored in filename extensions (Cf. Single Purpose Files) 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". EA will "register" new FORM type IDs and format descriptions as they're devised, but collisions will be improbable so there will be no pressure on this "clearinghouse" process. Appendix A has a list of currently defined IDs.

Sometimes it's necessary to make data format changes that aren't backward compatible. Since IDs are used to denote data formats in IFF, new IDs are chosen to denote revised formats. Since programs won't

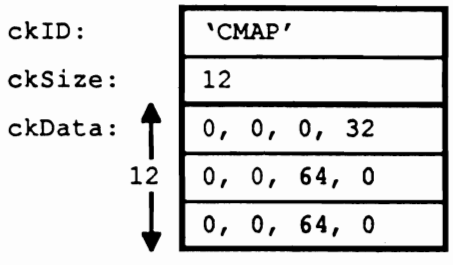
read chunks whose IDs they don't recognize (see Chunks, below), the new IDs keep old programs from stumbling over new data. The conventional way to chose a "revision" ID is to increment the last character if it's a digit or else change the last character to a digit. E.g. first and second revisions of 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;
   LONG   ckSize;                  /* sizeof(ckData) */
   UBYTE  ckData[/* ckSize */];
   } Chunk;
```

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




The fixed header part means "Here's a type `ckID` chunk with `ckSize` bytes of data."

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 23 chunk IDs. Appendix A has a list of predefined 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 not 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`, i.e. 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, LF denotes a forced line break (paragraph or line terminator). Other control characters are not used. (Cf. Characters.)

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.

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 be the 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 NUL byte (ASCII value 0).

### Data Properties

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 special case 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". 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 index 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 ref's originator. Following the reference would expect the file to be on some mounted volume. In a network environment, a file ref could name a server, too.

Issue: How can we express operating-system independent file refs?

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. 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'll 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). (LISTs and CATs are discussed in part 5, below.) 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 FormType (or 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'll 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 cancelled by a later setting of the same property has effect only on data chunks in between. E.g. in the sequence:

```
prop1 = x   (propN = value)*   prop1 = y
```

where the `propN`s are not `prop1`, the setting `prop1 = x` has no effect.

The following universal chunk IDs are reserved inside any FORM: "LIST", "FORM", "PROP", "CAT ", "    ", "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9". (Cf. Chunks. Cf. Group LIST. Cf. Group PROP.) For clarity, these universal chunk names may not be FORM type IDs, either.

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 like a multi-frame animation sequence from 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, although it'll be the rare program that's fancy enough to do that. Such editors are not 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 should 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 a good example.

Anyone can pick a new FORM type ID but should reserve it with Electronic Arts at their earliest convenience. [Issue: EA contact person? Hand this off to another organization?] 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, 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 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 overriden 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 it 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. 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 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 your users to import and export the nested FORMs. The import and export functions may move data through a separate file or a clipboard.

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

## 5. LISTs, CATs, and Shared Properties

Data often needs to be grouped together like 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.

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. Where in C you could write:

```
TEXT_FONT text_font = Courier;          /* program's global default    */

File(); {
   TEXT_FONT text_font = TimesRoman;    /* shared setting              */

      {
      TEXT_FONT text_font = Helvetica; /* local setting               */
      Print("Hello ");                 /* uses font Helvetica         */
      }

      {
      Print("there.");                 /* uses font TimesRoman        */
      }
   }
```

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're 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 fake 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.

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. This ensures robust data transfer. The public domain IFF reader/writer subroutine package does this 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!

Thre 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 requestor. 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

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". 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, and 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 pad byte after every odd-length chunk.
- 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.
- Simpler 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 an animation file may contain still pictures.

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" (see "Support Software" in Appendix A), 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.
- Four 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 length count, and (4) make a preliminary pass to compute the length count then stream write the data.
- 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 don't edit and replace the nested FORMs and don't add or remove them. That could make the containing structure inconsistent. You may write a new file containing items you copied (or copied and modified) from another IFF file, but don't copy structural parts you don't understand.
- You must adhere to the syntax descriptions in Appendex A. E.g. PROPs may only appear inside LISTs.

# 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 `ckSize` LONG 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]".

## Defined Chunk IDs

This is a table of currently defined chunk IDs. We may also borrow some Macintosh IDs and data formats.

Group chunk IDs
FORM, LIST, PROP, CAT.

Future revision group chunk IDs
FOR1 ... FOR9, LIS1 ... LIS9, CAT1 ... CAT9.

FORM type IDs
(The above group chunk IDs may not be used for FORM type IDs.)
(Lower case letters and punctuation marks are forbidden in FORM type IDs.)
8SVX 8-bit sampled sound voice, ANBM animated bitmap, FNTR raster font, FNTV vector font, FTXT formatted text, GSCR general-use musical score, ILBM interleaved raster bitmap image, PDEF Deluxe Print page definition, PICS Macintosh picture, PLBM (obsolete), USCR Uhuru Sound Software musical score, UVOX Uhuru Sound Software Macintosh voice, SMUS simple musical score, VDEO Deluxe Video Construction Set video.

Data chunk IDs
" ", TEXT, PICT.

PROP LIST property IDs
OPGM, OCPU, OCMP, OSN, UNAM.

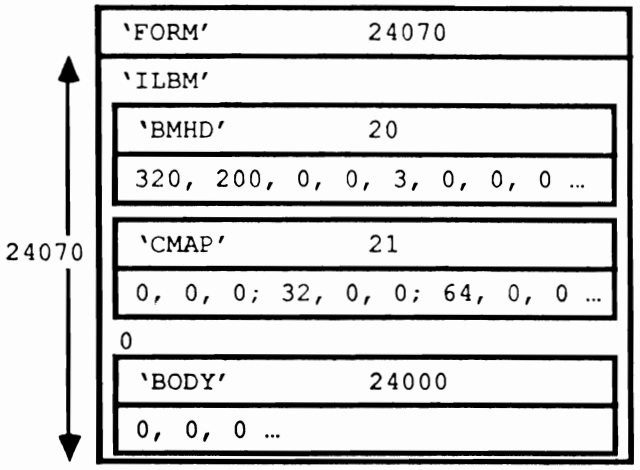
## Support Software

These public domain C source programs are available for use in building IFF-compatible programs:

| | |
|---|---|
| IFF.H, IFFR.C, IFFW.C | IFF reader and writer package. These modules handle many of the details of reliably reading and writing IFF files. |
| IFFCheck.C | This handy utility program scans an IFF file, checks that the contents are well formed, and prints an outline of the chunks. |
| Packer.H, Packer.C, UnPacker.C | Run encoder and decoder used for ILBM files. |
| ILBM.H, ILBMR.C, ILBMW.C | Reader and writer support routines for raster image FORM ILBM. ILBMR calls IFFR and UnPacker. ILBMW calls IFFW and Packer. |
| ShowILBM.C | Example caller of IFFR and ILBMR modules. This Commodore-Amiga program reads and displays a FORM ILBM. |
| Raw2ILBM.C | Example ILBM writer program. As a demonstration, it reads a raw raster image file and writes the image as a FORM ILBM file. |

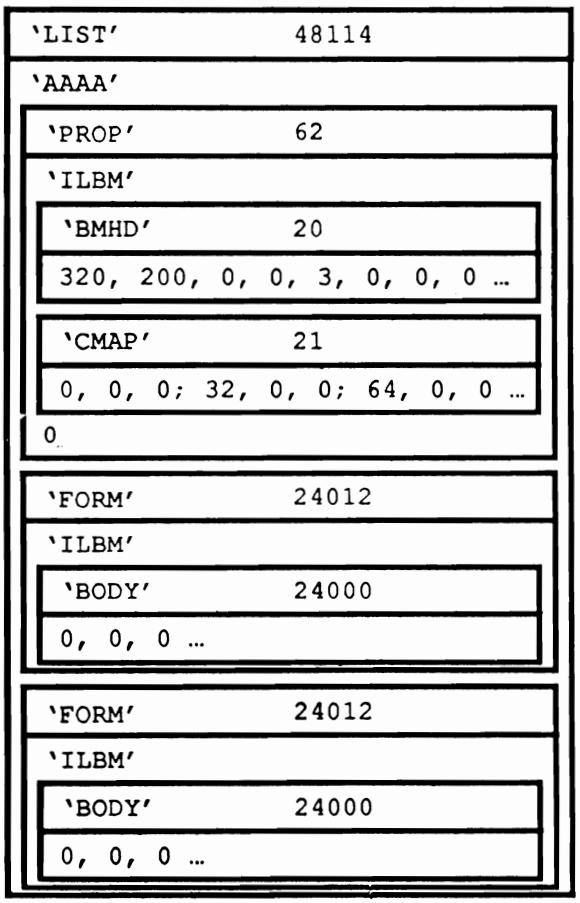
## Example Diagrams

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  2
.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
```

## Appendix B. 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

# "ILBM" IFF Interleaved Bitmap

**Date:** January 17, 1986
**From:** Jerry Morrison, Electronic Arts
**Status:** Released and in use

## 1. Introduction

"EA IFF 85" is Electronic Arts' standard for interchange format files. "ILBM" is a format for a 2 dimensional raster graphics image, specifically an InterLeaved bitplane BitMap image with color map. An ILBM is an IFF "data section" or "FORM type", which can be an IFF file or a part of one. (See the IFF reference.)

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

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

Details of the raster layout are given in part 3, "Standard Data Chunk". Some elements are based on the Commodore Amiga hardware but generalized for use on other computers. An alternative to ILBM would be appropriate for computers with true color data in each pixel.

## Reference:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

## 2. Standard Properties

The required property "BMHD" and any optional properties must appear before any "BODY" chunk. (Since an ILBM has only one BODY chunk, any following properties are superfluous.) Any of these properties may be shared over a LIST of FORMs IBLM by putting them in a PROP ILBM. (See the "EA IFF 85" memo.)

### BMHD

The required property "BMHD" holds a BitMapHeader as defined in these C declarations and following documentation. It describes the dimensions and encoding of the image, including data necessary to understand the BODY chunk to follow.

```
typedef UBYTE Masking;          /* Choice of masking technique. */
#define mskNone                 0
#define mskHasMask              1
#define mskHasTransparentColor  2
#define mskLasso                3

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

typedef struct {
   UWORD w, h;                    /* raster width & height in pixels       */
   WORD  x, y;                    /* pixel position for this image         */
   UBYTE nPlanes;                 /* # source bitplanes                    */
   Masking     masking;
   Compression compression;
   UBYTE pad1;                    /* unused; for consistency, put 0 here   */
   UWORD transparentColor;        /* transparent "color number" (sort of)  */
   UBYTE xAspect, yAspect;        /* pixel aspect, a ratio width : height  */
   WORD  pageWidth, pageHeight;   /* source "page" size in pixels          */
   } BitMapHeader;
```

Fields are filed in the order shown. The UBYTE fields are byte-packed.

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

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

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

The field `masking` indicates what kind of masking is to be used for this image. The value `mskNone` designates an opaque rectangular image. The value `mskHasMask` means that a mask plane is interleaved with the bitplanes in the BODY chunk (see below). The value `mskHasTransparentColor` indicates that pixels in the source planes matching `transparentColor` are to be considered "transparent". (Actually, `transparentColor` isn't a "color number" since it's matched with numbers formed by the source bitmap

rather than the possibly deeper destination bitmap. Note that having a transparent color implies ignoring one of the color registers. See CMAP, below.) The value `mskLasso` indicates the reader may construct a mask by lassoing the image as in MacPaint™. To do this, put a 1 pixel border of `transparentColor` around the image rectangle. Then do a seed fill from this border. Filled pixels are to be transparent.

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

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

The field `pad1` is a pad byte and must be set to 0 for consistency. This field could get used in the future.

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

The pixel aspect ratio is stored as a ratio in the two fields `xAspect` and `yAspect`. This may be used by programs to compensate for different aspects or to help interpret the fields `w`, `h`, `x`, `y`, `pageWidth`, and `pageHeight`, which are in units of pixels. The fraction `xAspect/yAspect` represents a pixel's width/height. It's recommended that your programs store proper fractions in BitMapHeaders, but aspect ratios can always be correctly compared with the the test

```
xAspect•yDesiredAspect = yAspect•xDesiredAspect
```

Typical values for aspect ratio are width : height = 10 : 11 (Amiga 320 x 200 display) and 1 : 1 (Macintosh™).

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

## CMAP

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

A CMAP chunk contains a `ColorMap` array as defined below. (These typedefs assume a C compiler that implements packed arrays of 3-byte elements.)

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

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

The color components red, green, and blue represent fractional intensity values in the range 0 through 255 256ths. White is (255, 255, 255) and black is (0, 0, 0). If your machine has less color resolution, use the high order bits. Shift each field right on reading (or left on writing) and assign it to (from) a field in a local packed format like `Color4`, below. This achieves automatic conversion of images across environments with different color resolutions. On reading an ILBM, use defaults if the color map is absent or has fewer color registers than you need. Ignore any extra color registers.

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

```
typedef struct {
   unsigned pad1 :4, red :4, green :4, blue :4;
   } Color4;                                  /* Amiga RAM format. Not filed. */
```

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

## GRAB

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

```
typedef struct {
   WORD x, y;          /* relative coordinates (pixels) */
   } Point2D;
```

## DEST

The optional property "DEST" is a way to say how to scatter zero or more source bitplanes into a deeper destination image. Some readers may ignore DEST.

The contents of a DEST chunk is DestMerge structure:

```
typedef struct {
   UBYTE depth;       /* # bitplanes in the original source                */
   UBYTE pad1;        /* unused; for consistency put 0 here                */
   UWORD planePick;   /* how to scatter source bitplanes into destination  */
   UWORD planeOnOff;  /* default bitplane data for planePick               */
   UWORD planeMask;   /* selects which bitplanes to store into             */
   } DestMerge;
```

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

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

## SPRT

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

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

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

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

### CAMG

A "CAMG" chunk is specifically for the Commodore Amiga computer. It stores a LONG "viewport mode". This lets you specify Amiga display modes like "dual playfield" and "hold and modify".

## 3. Standard Data Chunk

### Raster Layout

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

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

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

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

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

### BODY

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

The BitMapHeader, in a BMHD property chunk, specifies the raster's dimensions `w`, `h`, and `nPlanes`. It also holds the `masking` field which indicates if there is a mask plane and the `compression` field which indicates the compression algorithm used. This information is needed to interpret the BODY chunk, so the BMHD chunk must appear first. While reading an ILBM's BODY, a program may convert the image to another size by filling (with `transparentColor`) or clipping.

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

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

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

## 4. Nonstandard Data Chunk

The following data chunk was defined after various programs began using FORM ILBM so it's a "nonstandard" chunk. That means there's some slight chance of name collisions.

### CRNG

A "CRNG" chunk contains "color register range" information. It's used by Electronic Arts' Deluxe Paint program to identify a contiguous range of color registers for a "shade range" and color cycling. There can be zero or more CRNG chunks in an ILBM, but all should appear before the BODY chunk. Deluxe Paint normally writes 4 CRNG chunks in an ILBM when the user asks it to "Save Picture".

```
typedef struct {
   WORD  pad1;          /* reserved for future use; store 0 here          */
   WORD  rate;          /* color cycle rate                               */
   WORD  active;        /* nonzero means cycle the colors                 */
   UBYTE low, high;     /* lower and upper color registers selected       */
   } CRange;
```

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

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

The field `rate` determines the speed at which the colors will step when color cycling is on. The units are such that a rate of 60 steps per second is represented as $2^{14} = 16384$. Slower rates can be obtained by linear scaling: for 30 steps/second, rate = 8192; for 1 step/second, rate = 16384/60 ≈ 273.

### CCRT

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

```
typedef struct {
   WORD  direction;     /* 0 = don't cycle. 1 = cycle forwards (1, 2, 3).
                         * -1 = cycle backwards (3, 2, 1)                 */
   UBYTE start, end;    /* lower and upper color registers selected       */
   LONG  seconds;       /* # seconds between changing colors              */
   LONG  microseconds;  /* # microseconds between changing colors         */
   WORD  pad;           /* reserved for future use; store 0 here          */
   } CycleInfo;
```

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

A CCRT chunk expresses the color cycling rate as a number of seconds plus a number of microseconds.

## Appendix A. ILBM Regular Expression

Here's a regular expression summary of the FORM ILBM syntax. This could be an IFF file or a part of one.

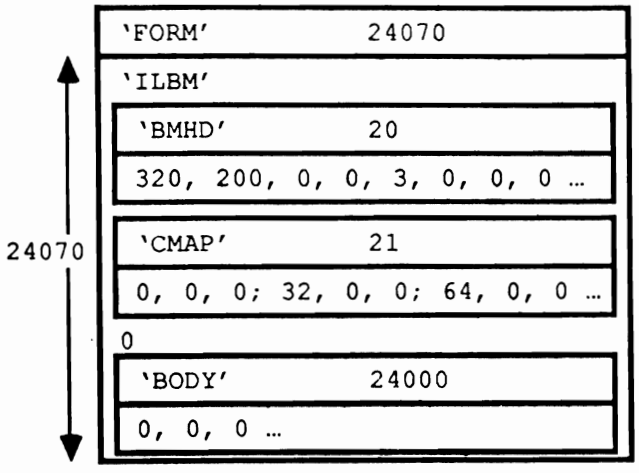
```
ILBM ::= "FORM" #{ "ILBM" BMHD [CMAP] [GRAB] [DEST] [SPRT] [CAMG]
                   CRNG* [BODY]       }

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

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

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

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

## Appendix B. ILBM Box Diagram

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




The "0" after the CMAP chunk is a pad byte.

## Appendix C. ByteRun1 Run Encoding

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

```
UnPacker:
   LOOP until produced the desired number of bytes
     Read the next source byte into n
     SELECT n FROM
        [0..127]    => copy the next n+1 bytes literally
        [-1..-127]  => replicate the next byte -n+1 times
        -128        => noop
        ENDCASE;
     ENDLOOP;
```

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

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

## Appendix D. Standards Committee

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

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Dan Silva, Electronic Arts
Barry Walsh, Commodore-Amiga

# "FTXT" IFF Formatted Text

**Date:** November 15, 1985
**From:** Steve Shaw and Jerry Morrison, Electronic Arts and Bob "Kodiak" Burns, Commodore-Amiga
**Status:** Draft 2.6

DRAFT DRAFT DRAFT DRAFT DRAFT

## 1. Introduction

This memo is the IFF supplement for FORM FTXT. An FTXT is an IFF "data section" or "FORM type"—which can be an IFF file or a part of one—containing a stream of text plus optional formatting information."EA IFF 85" is Electronic Arts' standard for interchange format files. (See the IFF reference.)

An FTXT is an archival and interchange representation designed for three uses. The simplest use is for a "console device" or "glass teletype" (the minimal 2-D text layout means): a stream of "graphic" ("printable") characters plus positioning characters "space" ("SP") and line terminator ("LF"). This is not intended for cursor movements on a screen although it does not conflict with standard cursor-moving characters. The second use is text that has explicit formatting information (or "looks") such as font family and size, typeface, etc. The third use is as the lowest layer of a structured document that also has "inherited" styles to implicitly control character looks. For that use, FORMs FTXT would be embedded within a future document FORM type. The beauty of FTXT is that these three uses are interchangeable, that is, a program written for one purpose can read and write the others' files. So a word processor does not have to write a separate plain text file to communicate with other programs.

Text is stored in one or more "CHRS" chunks inside an FTXT. Each CHRS contains a stream of 8-bit text compatible with ISO and ANSI data interchange standards. FTXT uses just the central character set from the ISO/ANSI standards. (These two standards are henceforth called "ISO/ANSI" as in "see the ISO/ANSI reference".)

Since it's possible to extract just the text portions from future document FORM types, programs can exchange data without having to save both plain text and formatted text representations.

Character looks are stored as embedded control sequences within CHRS chunks. This document specifies which class of control sequences to use: the CSI group. This document does not yet specify their meanings, e.g. which one means "turn on italic face". Consult ISO/ANSI.

Section 2 defines the chunk types character stream "CHRS" and font specifier "FONS". These are the "standard" chunks. Specialized chunks for private or future needs can be added later. Section 3 outlines an FTXT reader program that strips a document down to plain unformatted text. Appendix A is a code table for the 8-bit ISO/ANSI character set used here. Appendix B is an example FTXT shown as a box diagram. Appendix C is a racetrack diagram of the syntax of ISO/ANSI control sequences.

## Reference:

Amiga™ is a trademark of Commodore-Amiga, Inc.

Electronic Arts™ is a trademark of Electronic Arts.

IFF: "EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

ISO/ANSI: ISO/DIS 6429.2 and ANSI X3.64-1979. International Organization for Standardization (ISO) and American National Standards Institute (ANSI) data-interchange standards. The relevant parts of these two standards documents are identical. ISO standard 2022 is also relevant.

## 2. Standard Data and Property Chunks

The main contents of a FORM FTXT is in its character stream "CHRS" chunks. Formatting property chunks may also appear. The only formatting property yet defined is "FONS", a font specifier. A FORM FTXT with no CHRS represents an empty text stream. A FORM FTXT may contain nested IFF FORMs, LISTs, or CATs, although a "stripping" reader (see section 3) will ignore them.

### Character Set

FORM FTXT uses the core of the 8-bit character set defined by the ISO/ANSI standards cited at the start of this document. (See Appendix A for a character code table.) This character set is divided into two "graphic" groups plus two "control" groups. Eight of the control characters begin ISO/ANSI standard control sequences. (See "Control Sequences", below.) Most control sequences and control characters are reserved for future use and for compatibility with ISO/ANSI. Current reader programs should skip them.

- C0 is the group of control characters in the range NUL (hex 0) through hex 1F. Of these, only LF (hex 0A) and ESC (hex 1B) are significant. ESC begins a control sequence. LF is the line terminator, meaning "go to the first horizontal position of the next line". All other C0 characters are not used. In particular, CR (hex 0D) is not recognized as a line terminator.
- G0 is the group of graphic characters in the range hex 20 through hex 7F. SP (hex 20) is the space character. DEL (hex 7F) is the delete character which is not used. The rest are the standard ASCII printable characters "!" (hex 21) through "~" (hex 7E).
- C1 is the group of extended control characters in the range hex 80 through hex 9F. Some of these begin control sequences. The control sequence starting with CSI (hex 9B) is used for FTXT formatting. All other control sequences and C1 control characters are unused.
- G1 is the group of extended graphic characters in the range NBSP (hex A0) through "ÿ" (hex FF). It is one of the alternate graphic groups proposed for ISO/ANSI standardization.

### Control Sequences

Eight of the control characters begin ISO/ANSI standard "control sequences" (or "escape sequences"). These sequences are described below and diagrammed in Appendix C.

```
G0          ::= (SP through DEL)
G1          ::= (NBSP through "ÿ")

ESC-Seq     ::= ESC (SP through "/")* ("0" through "~")
ShiftToG2   ::= SS2 G0
ShiftToG3   ::= SS3 G0
CSI-Seq     ::= CSI (SP through "?")* ("@" through "~")
DCS-Seq     ::= (DCS | OSC | PM | APC) (SP through "~" | G1)* ST
```

"ESC-Seq" is the control sequence ESC (hex 1B), followed by zero or more characters in the range SP through "/" (hex 20 through hex 2F), followed by a character in the range "0" through "~" (hex 30 through hex 7E). These sequences are reserved for future use and should be skipped by current FTXT reader programs.

SS2 (hex 8E) and SS3 (hex 8F) shift the single following G0 character into yet-to-be-defined graphic sets G2 and G3, respectively. These sequences should not be used until the character sets G2 and G3 are standardized. A reader may simply skip the SS2 or SS3 (taking the following character as a corresponding G0 character) or replace the two-character sequence with a character like "?" to mean "absent".

FTXT uses "CSI-Seq" control sequences to store character formatting (font selection by number, type face, and text size) and perhaps layout information (position and rotation). "CSI-Seq" control sequences

start with CSI (the "control sequence introducer", hex 9B). Syntactically, the sequence includes zero or more characters in the range SP through "?" (hex 20 through hex 3F) and a concluding character in the range "@" through "~" (hex 40 through hex 7E). These sequences may be skipped by a minimal FTXT reader, i.e. one that ignores formatting information.

Note: A future FTXT standardization document will explain the uses of CSI-Seq sequences for setting character face (light weight vs. medium vs. bold, italic vs. upright, height, pitch, position, and rotation). For now, consult the ISO/ANSI references.

"DCS-Seq" is the control sequences starting with DCS (hex 90), OSC (hex 9D), PM (hex 9E), or APC (hex 9F), followed by zero or more characters each of which is in the range SP through "~" (hex 20 through hex 7E) or else a G1 character, and terminated by an ST (hex 9C). These sequences are reserved for future use and should be skipped by current FTXT reader programs.

### Data Chunk CHRS

A CHRS chunk contains a sequence of 8-bit characters abiding by the ISO/ANSI standards cited at the start of this document. This includes the character set and control sequences as described above and summarized in Appendicies A and C.

A FORM FTXT may contain any number of CHRS chunks. Taken together, they represent a single stream of textual information. That is, the contents of CHRS chunks are effectively concatenated except that (1) each control sequence must be completely within a single CHRS chunk, and (2) any formatting property chunks appearing between two CHRS chunks affects the formatting of the latter chunk's text. Any formatting settings set by control sequences inside a CHRS carry over to the next CHRS in the same FORM FTXT. All formatting properties stop at the end of the FORM since IFF specifies that adjacent FORMs are independent of each other (although not independent of any properties inherited from an enclosing LIST or FORM).

### Property Chunk FONS

The optional property "FONS" holds a FontSpecifier as defined in the C declaration below. It assignes a font to a numbered "font register" so it can be referenced by number within subsequent CHRS chunks. (This function is not provided within the ISO and ANSI standards.) The font specifier gives both a name and a description for the font so the recipient program can do font substitution.

By default, CHRS text uses font 1 until it selects another font. A minimal text reader always uses font 1. If font 1 hasn't been specified, the reader may use the local system font as font 1.

```
typedef struct {
   UBYTE id;       /* 0 through 9 is a font id number referenced by an SGR
                      control sequence selective parameter of 10 through 19.
                      Other values are reserved for future standardization.   */
   UBYTE pad1;     /* reserved for future use; store 0 here                   */
   UBYTE proportional; /* proportional font? 0 = unknown, 1 = no, 2 = yes */
   UBYTE serif;        /* serif font? 0 = unknown, 1 = no, 2 = yes        */
   char name[];    /* A NUL-terminated string naming the preferred font.   */
   } FontSpecifier;
```

Fields are filed in the order shown. The UBYTE fields are byte-packed (2 per 16-bit word). The field `pad1` is reserved for future standardization. Programs should store 0 there for now.

The field `proportional` indicates if the desired font is proportional width as opposed to fixed width. The

field `serif` indicates if the desired font is serif as opposed to sans serif. [Issue: Discuss font substitution!]

### Future Properties

New optional property chunks may be defined in the future to store additional formatting information. They will be used to represent formatting not encoded in standard ISO/ANSI control sequences and for "inherited" formatting in structured documents. Text orientation might be one example.

### Positioning Units

Unless otherwise specified, position and size units used in FTXT formatting properties and control sequences are in decipoints (720 decipoints/inch). This is ANSI/ISO Positioning Unit Mode (PUM) 2. While a metric standard might be nice, decipoints allow the existing U.S.A. typographic units to be encoded easily, e.g. "12 points" is "120 decipoints".

## 3. FTXT Stripper

An FTXT reader program can read the text and ignore all formatting and structural information in a document FORM that uses FORMs FTXT for the leaf nodes. This amounts to stripping a document down to a stream of plain text. It would do this by skipping over all chunks except FTXT.CHRS (CHRS chunks found inside a FORM FTXT) and within the FTXT.CHRS chunks skipping all control characters and control sequences. (Appendix C diagrams this text scanner.) It may also read FTXT.FONS chunks to find a description for font 1.

Here's a Pascal-ish program for an FTXT stripper. Given a FORM (a document of some kind), it scans for all FTXT.CHRS chunks. This would likely be applied to the first FORM in an IFF file.

```
PROCEDURE ReadFORM4CHRS();      {Read an IFF FORM for FTXT.CHRS chunks.}
   BEGIN
   IF the FORM's subtype = "FTXT"
      THEN ReadFTXT4CHRS()
      ELSE WHILE something left to read in the FORM DO BEGIN
           read the next chunk header;
           CASE the chunk's ID OF
              "LIST", "CAT ": ReadCAT4CHRS();
              "FORM": ReadFORM4CHRS();
              OTHERWISE skip the chunk's body;
              END
           END
   END;

{Read a LIST or CAT for all FTXT.CHRS chunks.}
PROCEDURE ReadCAT4CHRS();
   BEGIN
   WHILE something left to read in the LIST or CAT DO BEGIN
      read the next chunk header;
      CASE the chunk's ID OF
         "LIST", "CAT ": ReadCAT4CHRS();
         "FORM": ReadFORM4CHRS();
         "PROP": IF we're reading a LIST AND the PROP's subtype = "FTXT"
                 THEN read the PROP for "FONS" chunks;
         OTHERWISE error--malformed IFF file;
         END
      END
   END;

PROCEDURE ReadFTXT4CHRS();    {Read a FORM FTXT for CHRS chunks.}
   BEGIN
   WHILE something left to read in the FORM FTXT DO BEGIN
      read the next chunk header;
      CASE the chunk's ID OF
         "CHRS": ReadCHRS();
         "FONS": BEGIN
            read the chunk's contents into a FontSpecifier variable;
            IF the font specifier's id = 1 THEN use this font;
            END;
         OTHERWISE skip the chunk's body;
         END
      END
   END;
```

```
{Read an FTXT.CHRS. Skip all control sequences and unused control chars.}
PROCEDURE ReadCHRS();
  BEGIN
  WHILE something left to read in the CHRS chunk DO
    CASE read the next character OF
      LF:  start a new output line;
      ESC: SkipControl([' '..'/'], ['0'..'~']);
      IN [' '..'~'], IN [NBSP..'ÿ']: output the character;
      SS2, SS3: ;  {Just handle the following G0 character directly,
                    ignoring the shift to G2 or G3.}
      CSI: SkipControl([' '..'?'], ['@'..'~']);
      DCS, OSC, PM, APC: SkipControl([' '..'~'] + [NBSP..'ÿ'], [ST]);
      END
  END;

{Skip a control sequence of the format (rSet)* (tSet), i.e. any number of
  characters in the set rSet followed by a character in the set tSet.}
PROCEDURE SkipControl(rSet, tSet);
  VAR c: CHAR;
  BEGIN
  REPEAT c := read the next character
    UNTIL c NOT IN rSet;
  IF c NOT IN tSet
    THEN put character c back into the input stream;
  END
```

The following program is an optimized version of the above routines ReadFORM4CHRS and ReadCAT4CHRS for the case where you're ignoring fonts as well as formatting. It takes advantage of certain facts of the IFF format to read a document FORM and its nested FORMs, LISTs, and CATs without a stack. In other words, it's a hack that ignores all fonts and faces to cheaply get to the plain text of the document.

```
{Cheap scan of an IFF FORM for FTXT.CHRS chunks.}
PROCEDURE ScanFORM4CHRS();
  BEGIN
  IF the document FORM's subtype = "FTXT"
    THEN ReadFTXT4CHRS()
    ELSE WHILE something left to read in the FORM DO BEGIN
         read the next chunk header;
         IF it's a group chunk (LIST, FORM, PROP, or CAT)
           THEN read its subtype ID;
         CASE the chunk's ID OF
           "LIST", "CAT ":;      {NOTE: See explanation below.*}
           "FORM": IF this FORM's subtype = "FTXT" THEN ReadFTXT4CHRS()
             ELSE;               {NOTE: See explanation below.*}
           OTHERWISE skip the chunk's body;
           END
         END
  END;
```

*Note: This implementation is subtle. After reading a group header other than FORM FTXT it just continues reading. This amounts to reading all the chunks inside that group as if they weren't nested in a group.

## Appendix A: Character Code Table

This table corresponds to the ISO/DIS 6429.2 and ANSI X3.64-1979 8-bit character set standards. Only the core character set of those standards is used in FTXT.

Two G1 characters aren't defined in the standards and are shown as dark gray entries in this table. Light gray shading denotes control characters. (DEL is a control character although it belongs to the graphic group G0.) The following five rare G1 characters are left blank in the table below due to limitations of available fonts: hex A8, D0, DE, F0, and FE.

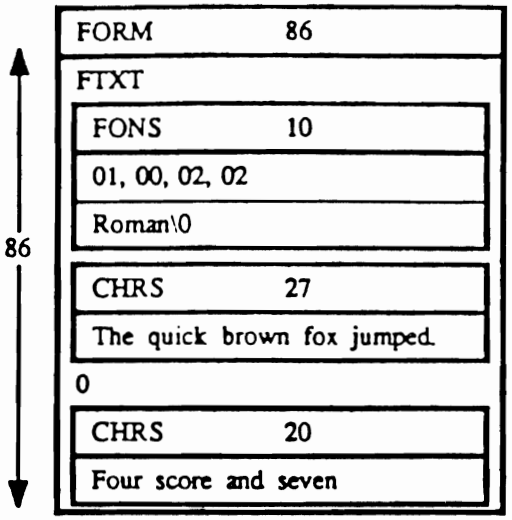
**ISO/DIS 6429.2 and ANSI X3.64-1979 Character Code Table**

| LSN ↓ / Most Significant Nibble (hex digit) → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | | SP | 0 | @ | P | ` | p | | DCS | NBSP | ° | À | | à | |
| 1 | | | ! | 1 | A | Q | a | q | | | ¡ | ± | Á | Ñ | á | ñ |
| 2 | | | " | 2 | B | R | b | r | | | ¢ | ² | Â | Ò | â | ò |
| 3 | | | # | 3 | C | S | c | s | | | £ | ³ | Ã | Ó | ã | ó |
| 4 | | | $ | 4 | D | T | d | t | | | ¤ | ´ | Ä | Ô | ä | ô |
| 5 | | | % | 5 | E | U | e | u | | | ¥ | µ | Å | Õ | å | õ |
| 6 | | | & | 6 | F | V | f | v | | | ¦ | ¶ | Æ | Ö | æ | ö |
| 7 | | | ' | 7 | G | W | g | w | | | § | · | Ç | | ç | |
| 8 | | | ( | 8 | H | X | h | x | | | | ¸ | È | Ø | è | ø |
| 9 | | | ) | 9 | I | Y | i | y | | | © | ¹ | É | Ù | é | ù |
| A | LF | | * | : | J | Z | j | z | | | ª | º | Ê | Ú | ê | ú |
| B | | ESC | + | ; | K | [ | k | { | | CSI | « | » | Ë | Û | ë | û |
| C | | | , | < | L | \ | l | \| | | ST | ¬ | 1/4 | Ì | Ü | ì | ü |
| D | CR | | - | = | M | ] | m | } | | OSC | SHY | 1/2 | Í | Ý | í | ý |
| E | | | . | > | N | ^ | n | ~ | SS2 | PM | ® | 3/4 | Î | | î | |
| F | | | / | ? | O | _ | o | DEL | SS3 | APC | ¯ | ¿ | Ï | ß | ï | ÿ |
| | Control group C0 | | Graphic group G0 | | | | | | Control group C1 | | Graphic group G1 | | | | | |

"NBSP" is a "non-breaking space"
"SHY" is a "soft hyphen"

## Appendix B. FTXT Example

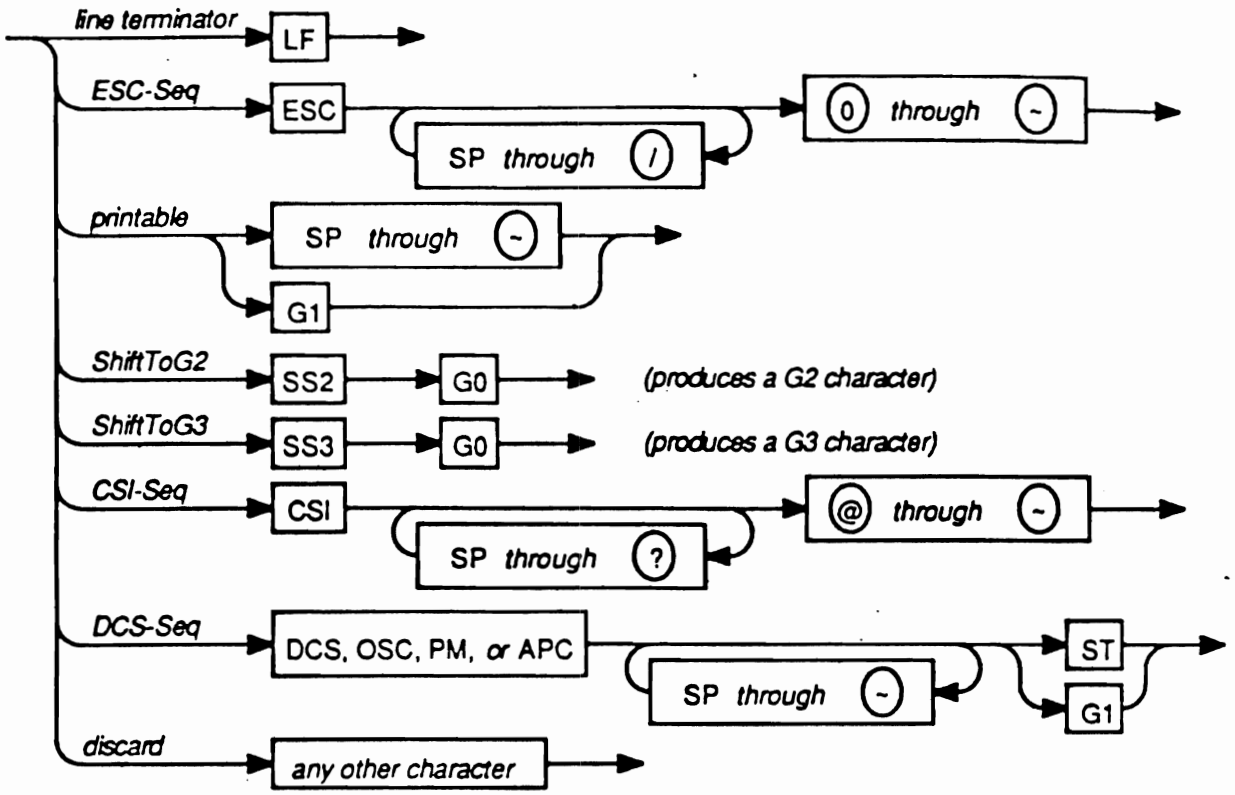
Here's a box diagram for a simple example: "The quick brown fox jumped.Four score and seven", written in a proportional serif font named "Roman".




The "0" after the first CHRS chunk is a pad byte.

## Appendix C. ISO/ANSI Control Sequences

This is a racetrack diagram of the ISO/ANSI characters and control sequences as used in FTXT CHRS chunks.




Of the various control sequences, only CSI-Seq is used for FTXT character formatting information. The others are reserved for future use and for compatibility with ISO/ANSI standards. Certain character sequences are syntactically malformed, e.g. CSI followed by a C0, C1, or G1 character. Writer programs should not generate reserved or malformed sequences and reader programs should skip them.

Consult the ISO/ANSI standards for the meaning of the CSI-Seq control sequences.

The two character set shifts SS2 and SS3 may be used when the graphic character groups G2 and G3 become standardized.

# "SMUS" IFF Simple Musical Score

**Date:** February 5, 1986
**From:** Jerry Morrison, Electronic Arts
**Status:** Adopted

## 1. Introduction

This is a reference manual for the data interchange format "SMUS", which stands for Simple MUsical Score. "EA IFF 85" is Electronic Arts' standard for interchange format files. A FORM (or "data section") such as FORM SMUS can be an IFF file or a part of one. [See "EA IFF 85" Electronic Arts Interchange File Format.]

SMUS is a practical data format for uses like moving limited scores between programs and storing theme songs for game programs. The format should be geared for easy read-in and playback. So FORM SMUS uses the compact time encoding of Common Music Notation (half notes, dotted quarter rests, etc.). The SMUS format should also be structurally simple. So it has no provisions for fancy notational information needed by graphical score editors or the more general timing (overlapping notes, etc.) and continuous data (pitch bends, etc.) needed by performance-oriented MIDI recorders and sequencers.

A SMUS score can say which "instruments" are supposed play which notes. But the score is independent of whatever output device and driver software is used to perform the notes. The score can contain device- and driver-dependent instrument data, but this is just a cache. As long as a SMUS file stays in one environment, the embedded instrument data is very convenient. When you move a SMUS file between programs or hardware configurations, the contents of this cache usually become useless.

Like all IFF formats, SMUS is a filed or "archive" format. It is completely independent of score representations in working memory, editing operations, user interface, display graphics, computation hardware, and sound hardware. Like all IFF formats, SMUS is extensible.

SMUS is not an end-all musical score format. Other formats may be more appropriate for certain uses. (We'd like to design an general-use IFF score format "GSCR". FORM GSCR would encode fancy notational data and performance data. There would be a SMUS to/from GSCR converter.)

Section 2 gives important background information. Section 3 details the SMUS components by defining the required property score header "SHDR", the optional text properties name "NAME", copyright "(c) ", and author "AUTH", optional text annotation "ANNO", the optional instrument specifier "INS1", and the track data chunk "TRAK". Section 4 defines some chunks for particular programs to store private information. These are "standard" chunks; specialized chunks for future needs can be added later. Appendix A is a quick-reference summary. Appendix B is an example box diagram. Appendix C names the committee responsible for this standard.

**Update:** This standard has been revised since the draft versions. The "INST" chunk type was revised to form the "INS1" chunk type. Also, several SEvent types and a few text chunk types have been added.

**Note:** This is a MacWrite™ 4.5 document. If you strip it down to a text file, you'll lose pictures, significant formatting information like superscripts, and characters like "©". Don't do it.

## References:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

"8SVX" IFF 8-Bit Sampled Voice documents a data format for sampled instruments.



MIDI: Musical Instrument Digital Interface Specification 1.0, International MIDI Association, 1983.



SSSP: See various articles on Structured Sound Synthesis Project in Foundations of Computer Music.

## 2. Background

Here's some background information on score representation in general and design choices for SMUS.

First, we'll borrow some terminology from the Structured Sound Synthesis Project. [See the SSSP reference.] A "musical note" is one kind of *scheduled event*. It's properties include an *event duration*, an *event delay*, and a *timbre object*. The *event duration* tells the scheduler how long the note should last. The *event delay* tells how long after starting this note to wait before starting the next event. The *timbre object* selects sound driver data for the note; an "instrument" or "timbre". A "rest" is a sort of a null event. Its only property is an event delay.

### Classical Event Durations

SMUS is geared for "classical" scores, not free-form performances. So its event durations are classical (whole note, dotted quarter rest, etc.). It can tie notes together to build a "note event" with an unusual event duration.

The set of useful classical durations is very small. So SMUS needs only a handful of bits to encode an event duration. This is very compact. It's also very easy to display in Common Music Notation (CMN).

### Tracks

The events in a SMUS score are grouped into parallel "tracks". Each track is a linear stream of events.

Why use tracks? Tracks serve 4 functions:

1. Tracks make it possible to encode event delays very compactly. A "classical" score has chorded notes and sequential notes; no overlapping notes. That is, each event begins either simultaneous with or immediately following the previous event in that track. So each event delay is either 0 or the same as the event's duration. This binary distinction requires only one bit of storage.

2. Tracks represent the "voice tracks" in Common Music Notation. CMN organizes a score in parallel staves, with one or two "voice tracks" per staff. So one or two SMUS tracks represents a CMN staff.

3. Tracks are a good match to available sound hardware. We can use "instrument settings" in a track to store the timbre assignments for that track's notes. The instrument setting may change over the track.

   Furthermore, tracks can help to allocate notes among available output channels or performance devices or tape recorder "tracks". Tracks can also help to adapt polyphonic data to monophonic output channels.

4. Tracks are a good match to simple sound software. Each track is a place to hold state settings like "dynamic mark *pp* ", "time signature 3/4", "mute this track", etc., just as it's a context for instrument settings. This is a lot like a text stream with running "font" and "face" properties (attributes). Running state is usually more compact than, say, storing an instrument setting in every note event. It's also a useful way to organize "attributes" of notes. With "running track state" we can define new note attributes in an upward- and backward-compatible way.

   Running track state can be expanded (run decoded) while loading a track into memory or while playing the track. The runtime track state must be reinitialized every time the score is played.

*Separated vs. interleaved tracks.* Multi-track data could be stored either as separate event streams or

interleaved into one stream. To interleave the streams, each event has to carry a "track number" attribute.

If we were designing an editable score format, we might interleave the streams so that nearby events are stored nearby. This helps when searching the data, especially if you can't fit the entire score into memory at once. But it takes extra storage for the track numbers and may take extra work to manipulate the interleaved tracks.

The musical score format FORM SMUS is intended for simple loading and playback of small scores that fit entirely in main memory. So we chose to store its tracks separately.

There can be up to 255 tracks in a FORM SMUS. Each track is stored as a TRAK chunk. The count of tracks (the number of TRAK chunks) is recorded in the SHDR chunk at the beginning of the FORM SMUS. The TRAK chunks appear in numerical order 1, 2, 3, .... This is also priority order, most important track first. A player program that can handle up to N parallel tracks should read the first N tracks and ignore any others.

The different tracks in a score may have different lengths. This is true both of storage length and of playback duration.

### Instrument Registers

*Instrument reference.* In SSSP, each note event points to a "timbre object" which supplies the "instrument" (the sound driver data) for that note. FORM SMUS stores these pointers as a "current instrument setting" for each track. It's just a run encoded version of the same information. SSSP uses a symbol table to hold all the pointers to "timbre object". SMUS uses INS1 chunks for the same purpose. They name the score's instruments.

The actual instrument data to use depends on the playback environment, but we want the score to be independent of environment. Different playback environments have different audio output hardware and different sound driver software. And there are channel allocation issues like how many output channels there are, which ones are polyphonic, and which I/O ports they're connected to. If you use MIDI to control the instruments, you get into issues of what kind of device is listening to each MIDI channel and what each of its preset sounds like. If you use computer-based instruments, you need driver-specific data like waveform tables and oscillator parameters.

We just want to put some orchestration in the score. If the score wants a "piano", we let the playback program to find a "piano".

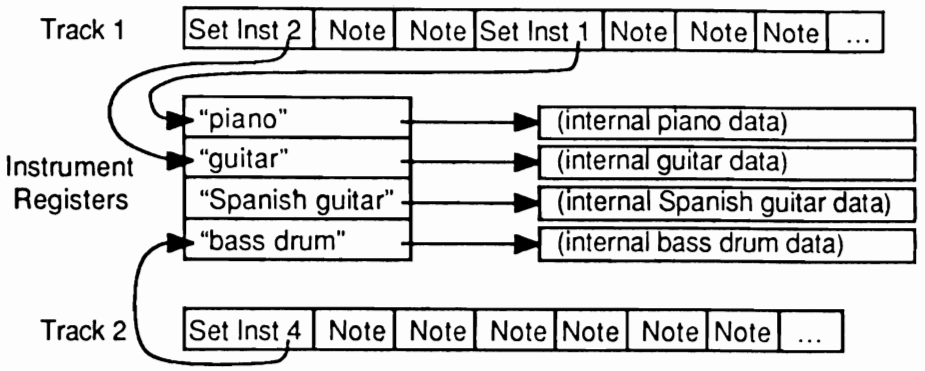
*Instrument reference by name.* A reference from a SMUS score to actual instrument data is normally by name. The score simply names the instrument, for instance "tubular bells". It's up to the player program to find suitable instrument data for its output devices. (More on locating instruments below.)

*Instrument reference by MIDI channel and preset.* A SMUS score can also ask for a specific MIDI channel number and preset number. MIDI programs may honor these specific requests. But these channel allocations can become obsolete or the score may be played without MIDI hardware. In such cases, the player program should fall back to instrument reference by name.

*Instrument reference via instrument register.* Each reference from a SMUS track to an instrument is via an "instrument register". Each track selects an instrument register which in turn points to the specific instrument data.

Each score has an array of instrument registers. Each track has a "current instrument setting", which is simply an index number into this array. This is like setting a raster image's pixel to a specific color number (a reference to a color value through a "color register") or setting a text character to a specific font number (a

reference to a font through a "font register"). This is diagrammed below.




*Locating instrument data by name.* "INS1" chunks in a SMUS score name the instruments to use for that score. The player program uses these names to locate instrument data.

To locate instrument data, the player performs these steps:

For each instrument register, check for a suitable instrument with the right name...
{"Suitable" means usable with an available output device and driver.}
{Use case independent name comparisons.}

1. Initialize the instrument register to point to a built-in default instrument.
{Every player program must have default instruments. Simple programs stop here. For fancier programs, the default instruments are a backstop in case the search fails.}

2. Check any instrument FORMs embedded in the FORM SMUS. (This is an "instrument cache".)

3. Else check the default instruments.

4. Else search the local "instrument library". (The library might simply be a disk directory.)

5. If all else fails, display the desired instrument name and ask the user to pick an available one.

This algorithm can be implemented to varying degrees of fanciness. It's ok to stop searching after step 1, 2, 3, or 4. If exact instrument name matches fail, it's ok to try approximate matches. E.g. search for any kind of "guitar" if you can't find a "Spanish guitar". In any case, a player only has to search for instruments while loading a score.

When the embedded instruments are suitable, they save the program from asking the user to insert the "right" disk in a drive and searching that disk for the "right" instrument. But it's just a cache. In practice, we rarely move scores between environments so the cache often works. When the score is moved, embedded instruments must be discarded (a cache miss) and other instrument data used.

Be careful to distinguish an instrument's name from its filename—the contents name vs. container name. A musical instrument FORM should contain a NAME chunk that says what instrument it really is. Its filename, on the other hand, is a handle used to locate the FORM. Filenames are affected by external factors like drives, directories, and filename character and length limits. Instrument names are not.

Issue: Consider instrument naming conventions for consistency. Consider a naming convention that aids approximate matches. E.g. we could accept "guitar, bass1" if we didn't find "guitar, bass". Failing that, we could accept "guitar" or any name starting with "guitar".

*Set instrument events.* If the player implements the set-instrument score event, each track can change instrument numbers while playing. That is, it can switch between the loaded instruments.

*Initial instrument settings.* Each time a score is played, every tracks' running state information must be initialized. Specifically, each track's instrument number should be initialized to its track number. Track 1 to instrument 1, etc. It's as if each track began with a set-instrument event.

In this way, programs that don't implement the set-instrument event still assign an instrument to each track. The INS1 chunks imply these initial instrument settings.

### MIDI Instruments

As mentioned above, A SMUS score can also ask for MIDI instruments. This is done by putting the MIDI channel and preset numbers in an INS1 chunk with the instrument name. Some programs will honor these requests while others will just find instruments by name.

MIDI Recorder and sequencer programs may simply transcribe the MIDI channel and preset commands in a recording session. For this purpose, set-MIDI-channel and set-MIDI-preset events can be embedded in a SMUS score's tracks. Most programs should ignore these events. An editor program that wants to exchange scores with such programs should recognize these events. It should let the user change them to the more general set-instrument events.

## 3. Standard Data and Property Chunks

A FORM SMUS contains a required property "SHDR" followed by any number of parallel "track" data chunks "TRAK". Optional property chunks such as "NAME", copyright "(c) ", and instrument reference "INS1" may also appear. Any of the properties may be shared over a LIST of FORMs SMUS by putting them in a PROP SMUS. [See the IFF reference.]

### Required Property SHDR

The required property "SHDR" holds an SScoreHeader as defined in these C declarations and following documentation. An SHDR specifies global information for the score. It must appear before the TRAKs in a FORM SMUS.

```
#define ID_SMUS MakeID('S', 'M', 'U', 'S')
#define ID_SHDR MakeID('S', 'H', 'D', 'R')

typedef struct {
   UWORD tempo;                /* tempo, 128ths quarter note/minute      */
   UBYTE volume;               /* overall playback volume 0 through 127  */
   UBYTE ctTrack;              /* count of tracks in the score           */
   } SScoreHeader;
```

[Implementation details. In the C struct definitions in this memo, fields are filed in the order shown. A UBYTE field is packed into an 8-bit byte. Programs should set all "pad" fields to 0. MakeID is a C macro defined in the main IFF document and in the source file IFF.h.]

The field `tempo` gives the nominal tempo for all tracks in the score. It is expressed in 128ths of a quarter note per minute, i.e. 1 represents 1 quarter note per 128 minutes while 12800 represents 100 quarter notes per minute. You may think of this as a fixed point fraction with a 9-bit integer part and a 7-bit fractional part (to the right of the point). A course-tempoed program may simply shift `tempo` right by 7 bits to get a whole number of quarter notes per minute. The `tempo` field can store tempi in the range 0 up to 512. The playback program may adjust this tempo, perhaps under user control.

Actually, this global `tempo` could actually be just an initial tempo if there are any "set tempo" SEvents inside the score (see TRAK, below). Or the global tempo could be scaled by "scale tempo" SEvents inside the score. These are potential extensions that can safely be ignored by current programs. [See More SEvents To Be Defined, below.]

The field `volume` gives an overall nominal playback volume for all tracks in the score. The range of `volume` values 0 through 127 is like a MIDI key velocity value. The playback program may adjust this volume, perhaps under direction of a user "volume control".

Actually, this global volume level could be scaled by dynamic-mark SEvents inside the score (see TRAK, below).

The field `ctTrack` holds the count of tracks, i.e. the number of TRAK chunks in the FORM SMUS (see below). This information helps the reader prepare for the following data.

A playback program will typically load the score and call a driver routine PlayScore(tracks, tempo, volume), supplying the `tempo` and `volume` from the SHDR chunk.

### Optional Text Chunks NAME, (c), AUTH, ANNO

Several text chunks may be included in a FORM SMUS to keep ancillary information.

The optional property "NAME" names the musical score, for instance "Fugue in C".

The optional property "(c) " holds a copyright notice for the score. The chunk ID "(c) " serves the function of the copyright characters "© ". E.g. a "(c) " chunk containing "1986 Electronic Arts" means "© 1986 Electronic Arts".

The optional property "AUTH" holds the name of the score's author.

The chunk types "NAME", "(c) ", and "AUTH" are property chunks. Putting more than one NAME (or other) property in a FORM is redundant. Just the last NAME counts. A property should be shorter than 256 characters. Properties can appear in a PROP SMUS to share them over a LIST of FORMs SMUS.

The optional data chunk "ANNO" holds any text annotations typed in by the author.

An ANNO chunk is not a property chunk, so you can put more than one in a FORM SMUS. You can make ANNO chunks any length up to $2^{31} - 1$ characters, but 32767 is a practical limit. Since they're not properties, ANNO chunks don't belong in a PROP SMUS. That means they can't be shared over a LIST of FORMs SMUS.

Syntactically, each of these chunks contains an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F), just like a standard "TEXT" chunk. [See "Strings, String Chunks, and String Properties" in "EA IFF 85" Electronic Arts Interchange File Format.] The chunk's `ckSize` field holds the count of characters.

```
#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the musical score's name.              */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.         */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the name of the score's author.        */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.             */
```

Remember to store a 0 pad byte after any odd-length chunk.

### Optional Property INS1

The "INS1" chunks in a FORM SMUS identify the instruments to use for this score. A program can ignore INS1 chunks and stick with its built-in default instrument assignments. Or it can use them to locate instrument data. [See "Instrument Registers" in section 2, above.]

```
#define ID_INS1 MakeID('I', 'N', 'S', '1')

/* Values for the RefInstrument field "type".                           */
#define INS1_Name  0          /* just use the name; ignore data1, data2 */
#define INS1_MIDI  1          /* <data1, data2> = MIDI <channel, preset> */
```

```
typedef struct {
   UBYTE register;          /* set this instrument register number    */
   UBYTE type;              /* instrument reference type              */
   UBYTE data1, data2;      /* depends on the "type" field            */
   CHAR  name[];            /* instrument name                        */
   } RefInstrument;
```

An INS1 chunk names the instrument for instrument register number `register`. The `register` field can range from 0 through 255. In practice, most scores will need only a few instrument registers.

The `name` field gives a text name for the instrument. The string length can be determined from the `ckSize` of the INS1 chunk. The string is simply an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F).

Besides the instrument name, an INS1 chunk has two data numbers to help locate an instrument. The use of these data numbers is controlled by the `type` field. A value `type = INS1_Name` means just find an instrument by name. In this case, `data1` and `data2` should just be set to 0. A value `type = INS1_MIDI` means look for an instrument on MIDI channel # `data1`, preset # `data2`. Programs and computers without MIDI outputs will just ignore the MIDI data. They'll always look for the named instrument. Other values of the `type` field are reserved for future standardization.

See section 2, above, for the algorithm for locating instrument data by name.

### Obsolete Property INST

The chunk type "INST" is obsolete in SMUS. It was revised to form the "INS1" chunk.

### Data Chunk TRAK

The main contents of a score is stored in one or more TRAK chunks representing parallel "tracks". One TRAK chunk per track.

The contents of a TRAK chunk is an array of 16-bit "events" such as "note", "rest", and "set instrument". Events are really commands to a simple scheduler, stored in time order. The tracks can be polyphonic, that is, they can contain chorded "note" events.

Each event is stored as an "SEvent" record. ("SEvent" means "simple musical event".) Each SEvent has an 8-bit type field called an "sID" and 8 bits of type-dependent data. This is like a machine language instruction with an 8-bit opcode and an 8-bit operand.

This format is extensible since new event types can be defined in the future. The "note" and "rest" events are the only ones that every program must understand. *We will carefully design any new event types so that programs can safely skip over unrecognized events in a score.*

Caution: SID codes must be allocated by a central clearinghouse to avoid conflicts.

Here are the C type definitions for TRAK and SEvent and the currently defined sID values. Afterward are details on each SEvent.

```
#define ID_TRAK MakeID('T', 'R', 'A', 'K')

/* TRAK chunk contains an SEvent[].                                  */
```

```
/* SEvent: Simple musical event.                                    */
typedef struct {
   UBYTE sID;                  /* SEvent type code                  */
   UBYTE data;                 /* sID-dependent data                */
   } SEvent;

/* SEvent type codes "sID".                                         */
#define SID_FirstNote     0
#define SID_LastNote    127   /* sIDs in the range SID_FirstNote through
                               * SID_LastNote (sign bit = 0) are notes. The
                               * sID is the MIDI tone number (pitch).   */
#define SID_Rest        128   /* a rest (same data format as a note).   */

#define SID_Instrument  129   /* set instrument number for this track.  */
#define SID_TimeSig     130   /* set time signature for this track.     */
#define SID_KeySig      131   /* set key signature for this track.      */
#define SID_Dynamic     132   /* set volume for this track.             */
#define SID_MIDI_Chnl   133   /* set MIDI channel number (sequencers)   */
#define SID_MIDI_Preset 134   /* set MIDI preset number (sequencers)    */

/* SID values 144 through 159: reserved for Instant Music SEvents.     */

/* Remaining sID values up through 254: reserved for future
 * standardization.                                                    */

#define SID_Mark        255   /* sID reserved for an end-mark in RAM.   */
```

## Note and Rest SEvents

The note and rest SEvents `SID_FirstNote` through `SID_Rest` have the following structure overlaid onto the SEvent structure:

```
typedef struct {
   UBYTE    tone;              /* MIDI tone number 0 to 127; 128 = rest  */
   unsigned chord    :1,       /* 1 = a chorded note                     */
            tieOut   :1,       /* 1 = tied to the next note or chord     */
            nTuplet  :2,       /* 0 = none, 1 = triplet, 2 = quintuplet,
                                * 3 = septuplet                         */
            dot      :1,       /* dotted note; multiply duration by 3/2  */
            division :3;       /* basic note duration is 2^-division: 0 = whole
                                * note, 1 = half note, 2 = quarter note, ...
                                * 7 = 128th note                        */
   } SNote;
```

[Implementation details. Unsigned ":n" fields are packed into n bits in the order shown, most significant bit to least significant bit. An SNote fits into 16 bits like any other SEvent. Warning: Some compilers don't implement bit-packed fields properly. E.g. Lattice 68000 C pads a group of bit fields out to a LONG, which would make SNote take 5-bytes! In that situation, use the bit-field constants defined below.]

The SNote structure describes one "note" or "rest" in a track. The field `SNote.tone`, which is overlaid with the `SEvent.sID` field, indicates the MIDI tone number (pitch) in the range 0 through 127. A value of 128 indicates a rest.

The fields `nTuplet`, `dot`, and `division` together give the duration of the note or rest. The `division`

gives the basic duration: whole note, half note, etc. The `dot` indicates if the note or rest is dotted. A dotted note is 3/2 as long as an undotted note. The value `nTuplet` (0 through 3) tells if this note or rest is part of an N-tuplet of order 1 (normal), 3, 5, or 7; an N-tuplet of order `(2 * nTuplet + 1)`. A triplet note is 2/3 as long as a normal note, while a quintuplet is 4/5 as long and a septuplet is 6/7 as long.

Putting these three fields together, the duration of the note or rest is

$$2^{-division} * \{1,\ 3/2\} * \{1,\ 2/3,\ 4/5,\ 6/7\}$$

These three fields are contiguous so you can easily convert to your local duration encoding by using the combined 6 bits as an index into a mapping table.

The field `chord` indicates if the note is chorded with the following note (which is supposed to have the same duration). A group of notes may be chorded together by setting the `chord` bit of all but the last one. (In the terminology of SSSP and GSCR, setting the chord bit to 1 makes the "entry delay" 0.) A monophonic-track player can simply ignore any SNote event whose `chord` bit is set, either by discarding it when reading the track or by skipping it when playing the track.

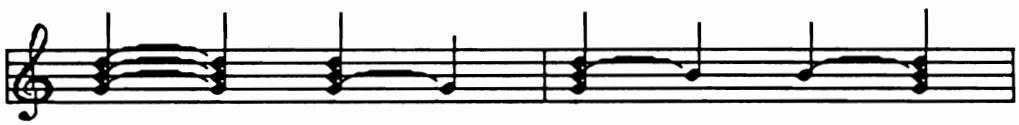
Programs that create polyphonic tracks are expected to store the most important note of each chord last, which is the note with the 0 `chord` bit. This way, monophonic programs will play the most important note of the chord. The most important note might be the chord's root note or its melody note.

If the field `tieOut` is set, the note is tied to the following note in the track if the following note has the same pitch. A group of tied notes is played as a single note whose duration is the sum of the component durations. Actually, the tie mechanism ties a group of one or more chorded notes to another group of one or more chorded notes. Every note in a tied chord should have its `tieOut` bit set.

Of course, the `chord` and `tieOut` fields don't apply to `SID_Rest` SEvents.

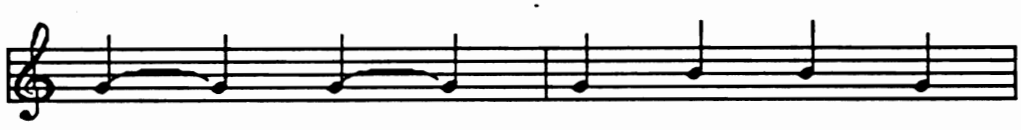
Programs should be robust enough to ignore an unresolved tie, i.e. a note whose `tieOut` bit is set but isn't followed by a note of the same pitch. If that's true, monophonic-track programs can simply ignore chorded notes even in the presense of ties. That is, tied chords pose no extra problems.

The following diagram shows some combinations of notes and chords tied to notes and chords. The text below the staff has a column for each SNote SEvent to show the pitch, `chord` bit, and `tieOut` bit.



```
pitch:  D B G  D B G D B G   G     D B G   B     B     D B G
chord:  c c    c c   c c             c c                 c c
tieOut: t t t        t t t           t t t         t
```

If you read the above track into a monophonic-track program, it'll strip out the chorded notes and ignore unresolved ties. You'll end up with:



```
pitch:  G      G      G      G      G      B      B      G
chord:
tieOut: t             t            (t)           (t)
```

A rest event (`sID = SID_Rest`) has the same `SEvent.data` field as a note. It tells the duration of the rest. The `chord` and `tieOut` fields of rest events are ignored.

Within a TRAK chunk, note and rest events appear in time order.

Instead of the bit-packed structure SNote, it might be easier to assemble `data` values by or-ing constants and to disassemble them by masking and shifting. In that case, use the following definitions.

```
#define noteChord   (1<<7)          /* note is chorded to next note      */

#define noteTieOut  (1<<6)          /* tied to next note/chord           */

#define noteNShift  4               /* shift count for nTuplet field     */
#define noteN3      (1<<noteNShift) /* note is a triplet                 */
#define noteN5      (2<<noteNShift) /* note is a quintuplet              */
#define noteN7      (3<<noteNShift) /* note is a septuplet               */
#define noteNMask   noteN7          /* bit mask for the nTuplet field    */

#define noteDot     (1<<3)          /* note is dotted                    */

#define noteD1      0               /* whole note division               */
#define noteD2      1               /* half note division                */
#define noteD4      2               /* quarter note division             */
#define noteD8      3               /* eighth note division              */
#define noteD16     4               /* sixteenth note division           */
#define noteD32     5               /* thirty-secondth note division     */
#define noteD64     6               /* sixty-fourth note division        */
#define noteD128    7               /* 1/128 note division               */
#define noteDMask   noteD128        /* bit mask for the division field   */

#define noteDurMask 0x3F            /* mask for combined duration fields */
```

**Note:** The remaining SEvent types are optional. A writer program doesn't have to generate them. A reader program can safely ignore them.

## Set Instrument SEvent

One of the running state variables of every track is an instrument number. An instrument number is the array index of an "instrument register", which in turn points to an instrument. (See "Instrument Registers", in section 2.) This is like a color number in a bitmap; a reference to a color through a "color register".

The initial setting for each track's instrument number is the track number. Track 1 is set to instrument 1, etc. Each time the score is played, every track's instrument number should be reset to the track number.

The SEvent `SID_Instrument` changes the instrument number for a track, that is, which instrument plays the following notes. Its `SEvent.data` field is an instrument register number in the range 0 through 255. If a program doesn't implement the `SID_Instrument` event, each track is fixed to one instrument.

## Set Time Signature SEvent

The SEvent `SID_TimeSig` sets the time signature for the track. A "time signature" SEvent has the following structure overlaid on the SEvent structure:

```
typedef struct {
   UBYTE    type;              /* = SID_TimeSig                              */
   unsigned timeNSig :5,       /* time sig. "numerator" is timeNSig + 1      */
            timeDSig :3;       /* time sig. "denominator" is 2^timeDSig:
                                * 0 = whole note, 1 = half note, 2 = quarter
                                * note, ... 7 = 128th note                    */
   } STimeSig;
```

[Implementation details. Unsigned ":n" fields are packed into n bits in the order shown, most significant bit to least significant bit. An STimeSig fits into 16 bits like any other SEvent. Warning: Some compilers don't implement bit-packed fields properly. E.g. Lattice C pads a group of bit fields out to a LONG, which would make an STimeSig take 5-bytes! In that situation, use the bit-field constants defined below.]

The field `type` contains the value `SID_TimeSig`, indicating that this SEvent is a "time signature" event. The field `timeNSig` indicates the time signature "numerator" is `timeNSig + 1`, that is, 1 through 32 beats per measure. The field `timeDSig` indicates the time signature "denominator" is $2^{\texttt{timeDSig}}$, that is each "beat" is a $2^{-\texttt{timeDSig}}$ note (see SNote `division`, above). So 4/4 time is expressed as `timeNSig` = 3, `timeDSig` = 2.

The default time signature is 4/4 time.

Beware that the time signature has no effect on the score's playback. Tempo is uniformly expressed in quarter notes per minute, independent of time signature. (Quarter notes per minute equals beats per minute only if `timeDSig` = 2, n/4 time). Nonetheless, any program that has time signatures should put them at the beginning of each TRAK when creating a FORM SMUS because music editors need them.

Instead of the bit-packed structure STimeSig, it might be easier to assemble `data` values by or-ing constants and to disassemble them by masking and shifting. In that case, use the following definitions.

```
#define timeNMask  0xF8          /* bit mask for the timeNSig field   */
#define timeNShift 3             /* shift count for  timeNSig field   */

#define timeDMask  0x07          /* bit mask for the timeDSig field   */
```

## Key Signature SEvent

An SEvent `SID_KeySig` sets the key signature for the track. Its `data` field is a UBYTE number encoding a major key:

| data | key | music notation | data | key | music notation |
|---|---|---|---|---|---|
| 0 | C maj | | | | |
| 1 | G | # | 8 | F | b |
| 2 | D | ## | 9 | Bb | bb |
| 3 | A | ### | 10 | Eb | bbb |
| 4 | E | #### | 11 | Ab | bbbb |
| 5 | B | ##### | 12 | Db | bbbbb |
| 6 | F# | ###### | 13 | Gb | bbbbbb |
| 7 | C# | ####### | 14 | Cb | bbbbbbb |

A `SID_KeySig` SEvent changes the key for the following notes in that track. C major is the default key in every track before the first `SID_KeySig` SEvent.

### Dynamic Mark SEvent

An SEvent `SID_Dynamic` represents a dynamic mark like *ppp* and *fff* in Common Music Notation. Its `data` field is a MIDI key velocity number 0 through 127. This sets a "volume control" for following notes in the track. This "track volume control" is scaled by the overall score `volume` in the SHDR chunk.

The default dynamic level is 127 (full volume).

### Set MIDI Channel SEvent

The SEvent `SID_MIDI_Chnl` is for recorder programs to record the set-MIDI-channel low level event. The `data` byte contains a MIDI channel number. Other programs should use instrument registers instead.

### Set MIDI Preset SEvent

The SEvent `SID_MIDI_Preset` is for recorder programs to record the set-MIDI-preset low level event. The `data` byte contains a MIDI preset number. Other programs should use instrument registers instead.

### Instant Music Private SEvents

Sixteen SEvents are used for private data for the Instant Music program. SID values 144 through 159 are reserved for this purpose. Other programs should skip over these SEvents.

### End-Mark SEvent

The SEvent type `SID_Mark` is reserved for an end marker in working memory. *This event is never stored in a file.* It may be useful if you decide to use the filed TRAK format intact in working memory.

### More SEvents To Be Defined

More SEvents can be defined in the future. The sID codes 133 through 143 and 160 through 254 are reserved for future needs. Caution: sID codes must be allocated by a central "clearinghouse" to avoid conflicts. When this SMUS standard passes the "draft" state, Commodore-Amiga will be in charge of this activity.

The following SEvent types are under consideration and should not yet be used.

Issue: A "change tempo" SEvent changes tempo during a score. Changing the tempo affects all tracks, not just the track containing the change tempo event.

One possibility is a "scale tempo" SEvent `SID_ScaleTempo` that rescales the global tempo:

```
currentTempo := globalTempo * (data + 1) / 128
```

This can scale the global tempo (in the SHDR) anywhere from x1/128 to x2 in roughly 1% increments.

An alternative is two events `SID_SetHTempo` and `SID_SetLTempo`. `SID_SetHTempo` gives the high byte and `SID_SetLTempo` gives the low byte of a new tempo setting, in 128ths quarter note/minute. `SetHTempo` automatically sets the low byte to 0, so the `SetLTempo` event isn't needed for course settings. In this scheme, the SHDR's `tempo` is simply a starting tempo.

An advantage of `SID_ScaleTempo` is that the playback program can just alter the global tempo to adjust the overall performance time and still easily implement tempo variations during the score. But the "set tempo" SEvent may be simpler to generate.

Issue: The events `SID_BeginRepeat` and `SID_EndRepeat` define a repeat span for one track. The span of events between a `BeginRepeat` and an `EndRepeat` is played twice. The `SEvent.data` field in the `BeginRepeat` event could give an iteration count,1 through 255 times or 0 for "repeat forever".

Repeat spans can be nested. All repeat spans automatically end at the end of the track.

An event `SID_Ending` begins a section like "first ending" or "second ending". The `SEvent.data` field gives the ending number. This `SID_Ending` event only applies to the innermost repeat group. (Consider generalizing it.)

A more general alternative is a "subtrack" or "subscore" event. A "subtrack" event is essentially a "subroutine call" to another series of SEvents. This is a nice way to encode all the possible variations of repeats, first endings, codas, and such.

To define a subtrack, we must demark its start and end. One possibility is to define a relative brach-to-subtrack event `SID_BSR` and a return-from-subtrack event `SID_RTS`. The 8-bit `data` field in the `SID_BSR` event can reach as far as 512 SEvents. A second possibility is to call a subtrack by index number, with an IFF chunk outside the TRAK defining the start and end of all subtracks. This is very general since a portion of one subtrack can be used as another subtrack. It also models the tape recording practice of first "laying down a track" and then selecting portions of it to play and repeat. To embody the music theory idea of playing a sequence like "ABBA", just compose the "main" track entirely of subtrack events. A third possibility is to use a numbered subtrack chunk "STRK" for each subroutine.

## 4. Private Chunks

As in any IFF FORM, there can be private chunks in a FORM SMUS that are designed for one particular program to store its private information. All IFF reader programs skip over unrecognized chunks, so the presense of private chunks can't hurt.

Instant Music stores some global score information in a chunk of ID "IRev".

## Appendix A. Quick Reference

### Type Definitions

Here's a collection of the C type definitions in this memo. In the "struct" type definitions, fields are filed in the order shown. A UBYTE field is packed into an 8-bit byte. Programs should set all "pad" fields to 0.

```
#define ID_SMUS MakeID('S', 'M', 'U', 'S')
#define ID_SHDR MakeID('S', 'H', 'D', 'R')

typedef struct {
   UWORD tempo;                /* tempo, 128ths quarter note/minute      */
   UBYTE volume;               /* overall playback volume 0 through 127  */
   UBYTE ctTrack;              /* count of tracks in the score           */
   } SScoreHeader;


#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the musical score's name.               */


#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.          */


#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the name of the score's author.         */


#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.              */


#define ID_INS1 MakeID('I', 'N', 'S', '1')

/* Values for the RefInstrument field "type".                            */
#define INS1_Name 0            /* just use the name; ignore data1, data2 */
#define INS1_MIDI 1            /* <data1, data2> = MIDI <channel, preset> */

typedef struct {
   UBYTE register;             /* set this instrument register number    */
   UBYTE type;                 /* instrument reference type              */
   UBYTE data1, data2;         /* depends on the "type" field            */
   CHAR  name[];               /* instrument name                        */
   } RefInstrument;


#define ID_TRAK MakeID('T', 'R', 'A', 'K')
/* TRAK chunk contains an SEvent[].                                      */

/* SEvent: Simple musical event.                                         */
typedef struct {
   UBYTE sID;                  /* SEvent type code                       */
   UBYTE data;                 /* sID-dependent data                     */
   } SEvent;
```

```
/* SEvent type codes "sID".                                          */
#define SID_FirstNote     0
#define SID_LastNote    127   /* sIDs in the range SID_FirstNote through
                               * SID_LastNote (sign bit = 0) are notes. The
                               * sID is the MIDI tone number (pitch).    */
#define SID_Rest        128   /* a rest (same data format as a note).    */

#define SID_Instrument  129   /* set instrument number for this track.   */
#define SID_TimeSig     130   /* set time signature for this track.      */
#define SID_KeySig      131   /* set key signature for this track.       */
#define SID_Dynamic     132   /* set volume for this track.              */
#define SID_MIDI_Chnl   133   /* set MIDI channel number (sequencers)    */
#define SID_MIDI_Preset 134   /* set MIDI preset number (sequencers)     */

/* SID values 144 through 159: reserved for Instant Music SEvents.      */

/* Remaining sID values up through 254: reserved for future
 * standardization.                                                     */

#define SID_Mark        255   /* sID reserved for an end-mark in RAM.    */


/* SID_FirstNote..SID_LastNote, SID_Rest SEvents                        */
typedef struct {
  UBYTE    tone;              /* MIDI tone number 0 to 127; 128 = rest   */
  unsigned chord    :1,       /* 1 = a chorded note                      */
           tieOut   :1,       /* 1 = tied to the next note or chord      */
           nTuplet  :2,       /* 0 = none, 1 = triplet, 2 = quintuplet,
                               * 3 = septuplet                           */
           dot      :1,       /* dotted note; multiply duration by 3/2   */
           division :3;       /* basic note duration is 2-division: 0 = whole
                               * note, 1 = half note, 2 = quarter note, …
                               * 7 = 128th note                          */
  } SNote;

#define noteChord  (1<<7)          /* note is chorded to next note       */

#define noteTieOut (1<<6)          /* tied to next note/chord            */

#define noteNShift 4               /* shift count for nTuplet field      */
#define noteN3     (1<<noteNShift) /* note is a triplet                  */
#define noteN5     (2<<noteNShift) /* note is a quintuplet               */
#define noteN7     (3<<noteNShift) /* note is a septuplet                */
#define noteNMask  noteN7          /* bit mask for the nTuplet field     */

#define noteDot    (1<<3)          /* note is dotted                     */

#define noteD1     0               /* whole note division                */
#define noteD2     1               /* half note division                 */
#define noteD4     2               /* quarter note division              */
#define noteD8     3               /* eighth note division               */
#define noteD16    4               /* sixteenth note division            */
#define noteD32    5               /* thirty-secondth note division      */
#define noteD64    6               /* sixty-fourth note division         */
```

```
#define noteD128    7                   /* 1/128 note division              */
#define noteDMask   noteD128            /* bit mask for the division field  */

#define noteDurMask 0x3F                /* mask for combined duration fields */


/* SID_Instrument SEvent                                                    */
/* "data" value is an instrument register number 0 through 255.            */


/* SID_TimeSig SEvent                                                       */
typedef struct {
  UBYTE    type;                /* = SID_TimeSig                           */
  unsigned timeNSig :5,         /* time sig. "numerator" is timeNSig + 1   */
           timeDSig :3;         /* time sig. "denominator" is 2^timeDSig:
                                 * 0 = whole note, 1 = half note, 2 = quarter
                                 * note, ... 7 = 128th note                */
  } STimeSig;

#define timeNMask   0xF8        /* bit mask for the timeNSig field         */
#define timeNShift  3           /* shift count for  timeNSig field         */

#define timeDMask   0x07        /* bit mask for the timeDSig field         */


/* SID_KeySig SEvent                                                        */
/* "data" value 0 = Cmaj; 1 through 7 = G,D,A,E,B,F#,C#;
 * 8 through 14 = F,Bb,Eb,Ab,Db,Gb,Cb.                                     */


/* SID_Dynamic SEvent                                                       */
/* "data" value is a MIDI key velocity 0..127.                             */
```

### SMUS Regular Expression

Here's a regular expression summary of the FORM SMUS syntax. This could be an IFF file or part of one.

```
SMUS      ::= "FORM" #{ "SMUS" SHDR [NAME] [Copyright] [AUTH] [IRev]
                        ANNO* INS1*  TRAK*  InstrForm* }

SHDR      ::= "SHDR" #{ SScoreHeader  }
NAME      ::= "NAME" #{ CHAR*         } [0]
Copyright ::= "(c) " #{ CHAR*         } [0]
AUTH      ::= "AUTH" #{ CHAR*         } [0]
IRev      ::= "IRev" #{ ...           }

ANNO      ::= "ANNO" #{ CHAR*         } [0]
INS1      ::= "INS1" #{ RefInstrument } [0]

TRAK      ::= "TRAK" #{ SEvent*       }

InstrForm ::= "FORM" #{ ...           }
```

The token "#" represents a `ckSize` LONG count of the following {braced} data bytes. Literal items are

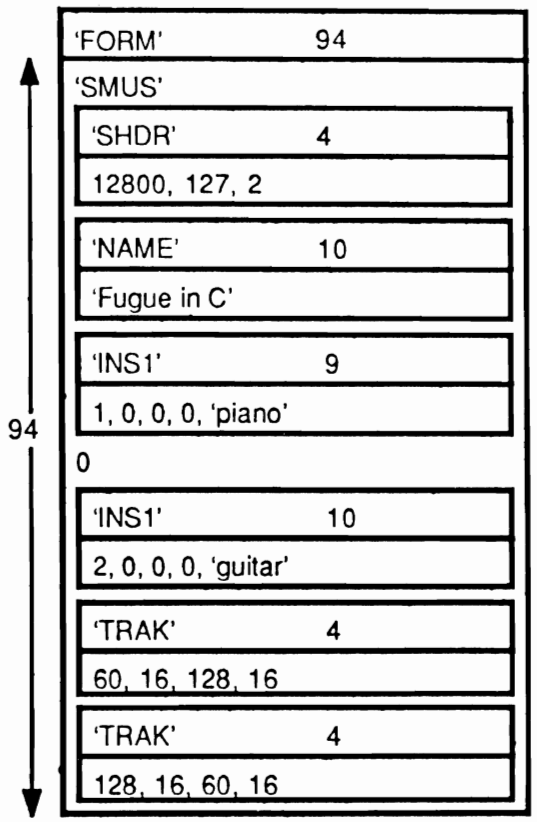
shown in "quotes", [square bracket items] are optional, and "*" means 0 or more replications. A sometimes-needed pad byte is shown as "[0]".

Actually, the order of chunks in a FORM SMUS is not as strict as this regular expression indicates. The SHDR, NAME, Copyright, AUTH, IRev, ANNO, and INS1 chunks may appear in any order, as long as they precede the TRAK chunks.

The chunk "InstrForm" represents any kind of instrument data FORM embedded in the FORM SMUS. For example, see the document "8SVX" IFF 8-Bit Sampled Voice. Of course, a recipient program will ignore an instrument FORM if it doesn't recognize that FORM type.

## Appendix B. SMUS Example

Here's a box diagram for a simple example, a SMUS with two instruments and two tracks. Each track contains 1 note event and 1 rest event.




The "0" after the first INS1 chunk is a pad byte.

## Appendix C. Standards Committee

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

Ralph Bellafatto, Cherry Lane Technologies
Geoff Brown, Uhuru Sound Software
Steve Hayes, Electronic Arts
Jerry Morrison, Electronic Arts

# "8SVX" IFF 8-Bit Sampled Voice

**Date:** February 7, 1985
**From:** Steve Hayes and Jerry Morrison, Electronic Arts
**Status:** Adopted

## 1. Introduction

This memo is the IFF supplement for FORM "8SVX". An 8SVX is an IFF "data section" or "FORM" (which can be an IFF file or a part of one) containing a digitally sampled audio voice consisting of 8-bit samples. A voice can be a one-shot sound or—with repetition and pitch scaling—a musical instrument. "EA IFF 85" is Electronic Arts' standard interchange file format. [See "EA IFF 85" Standard for Interchange Format Files.]

The 8SVX format is designed for playback hardware that uses 8-bit samples attenuated by a volume control for good overall signal-to-noise ratio. So a FORM 8SVX stores 8-bit samples and a volume level.

A similar data format (or two) will be needed for higher resolution samples (typically 12 or 16 bits). Properly converting a high resolution sample down to 8 bits requires one pass over the data to find the minimum and maximum values and a second pass to scale each sample into the range -128 through 127. So it's reasonable to store higher resolution data in a different FORM type and convert between them.

For instruments, FORM 8SVX can record a repeating waveform optionally preceded by a startup transient waveform. These two recorded signals can be pre-synthesized or sampled from an acoustic instrument. For many instruments, this representation is compact. FORM 8SVX is less practical for an instrument whose waveform changes from cycle to cycle like a plucked string, where a long sample is needed for accurate results.

FORM 8SVX can store an "envelope" or "amplitude contour" to enrichen musical notes. A future voice FORM could also store amplitude, frequency, and filter modulations.

FORM 8SVX is geared for relatively simple musical voices, where one waveform per octave is sufficient, where the waveforms for the different octaves follows a factor-of-two size rule, and where one envelope is adequate for all octaves. You could store a more general voice as a LIST containing one or more FORMs 8SVX per octave. A future voice FORM could go beyond one "one-shot" waveform and one "repeat" waveform per octave.

Section 2 defines the required property sound header "VHDR", optional properties name "NAME", copyright "(c) ", and author "AUTH", the optional annotation data chunk "ANNO", the required data chunk "BODY", and optional envelope chunks "ATAK" and "RLSE". These are the "standard" chunks. Specialized chunks for private or future needs can be added later, e.g. to hold a frequency contour or Fourier series coefficients. The 8SVX syntax is summarized in Appendix A as a regular expression and in Appendix B as an example box diagram. Appendix C explains the optional Fibonacci-delta compression algorithm.

**Caution:** The VHDR structure Voice8Header changed since draft proposal #4! The new structure is incompatible with the draft version.

**Note:** This is a MacWrite™ 4.5 document. If you strip it down to a text file, you'll lose pictures, significant formatting information like superscripts, and characters like "©". Don't do it.

## Reference:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

## 2. Standard Data and Property Chunks

FORM 8SVX stores all the waveform data in one body chunk "BODY". It stores playback parameters in the required header chunk "VHDR". "VHDR" and any optional property chunks "NAME", "(c) ", and "AUTH" must all appear before the BODY chunk. Any of these properties may be shared over a LIST of FORMs 8SVX by putting them in a PROP 8SVX. [See "EA IFF 85" Standard for Interchange Format Files.]

### Background

There are two ways to use FORM 8SVX: as a one-shot sampled sound or as a sampled musical instrument that plays "notes". Storing both kinds of sounds in the same kind of FORM makes it easy to play a one-shot sound as a (staccato) instrument or an instrument as a (one-note) sound.

A one-shot sound is a series of audio data samples with a nominal playback rate and amplitude. The recipient program can optionally adjust or modulate the amplitude and playback data rate.

For musical instruments, the idea is to store a sampled (or pre-synthesized) waveform that will be parameterized by pitch, duration, and amplitude to play each "note". The creator of the FORM 8SVX can supply a waveform per octave over a range of octaves for this purpose. The intent is to perform a pitch by selecting the closest octave's waveform and scaling the playback data rate. An optional "one-shot" waveform supplies an arbitrary startup transient, then a "repeat" waveform is iterated as long as necessary to sustain the note.

A FORM 8SVX can also store an envelope to modulate the waveform. Envelopes are mostly useful for variable-duration notes but could be used for one-shot sounds, too.

The FORM 8SVX standard has some restrictions. For example, each octave of data must be twice as long as the next higher octave. Most sound driver software and hardware imposes additional restrictions. E.g. the Amiga sound hardware requires an even number of samples in each one-shot and repeat waveform.

### Required Property VHDR

The required property "VHDR" holds a Voice8Header structure as defined in these C declarations and following documentation. This structure holds the playback parameters for the sampled waveforms in the BODY chunk. (See "Data Chunk BODY", below, for the storage layout of these waveforms.)

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;          /* A fixed-point value, 16 bits to the left of
                                the point and 16 to the right. A Fixed is a
                                number of 2^16ths, i.e. 65536ths.          */
#define Unity 0x10000L       /* Unity = Fixed 1.0 = maximum volume         */

/* sCompression: Choice of compression algorithm applied to the samples. */
#define sCmpNone       0     /* not compressed                             */
#define sCmpFibDelta   1     /* Fibonacci-delta encoding (Appendix C)      */
                             /* Can be more kinds in the future.           */

typedef struct {
  ULONG oneShotHiSamples,    /* # samples in the high octave 1-shot part */
        repeatHiSamples,     /* # samples in the high octave repeat part */
        samplesPerHiCycle;   /* # samples/cycle in high octave, else 0   */
```

```
    UWORD samplesPerSec;        /* data sampling rate                    */
    UBYTE ctOctave,             /* # octaves of waveforms                */
          sCompression;         /* data compression technique used       */
    Fixed volume;               /* playback volume from 0 to Unity (full
                                 * volume). Map this value into the output
                                 * hardware's dynamic range.             */
    } Voice8Header;
```

[Implementation details. Fields are filed in the order shown. The UBYTE fields are byte-packed (2 per 16-bit word). MakeID is a C macro defined in the main IFF document and in the source file IFF.h.]

A FORM 8SVX holds waveform data for one or more octaves, each containing a one-shot part and a repeat part. The fields `oneShotHiSamples` and `repeatHiSamples` tell the number of audio samples in the two parts of the highest frequency octave. Each successive (lower frequency) octave contains twice as many data samples in both its one-shot and repeat parts. One of these two parts can be empty across all octaves.

Note: Most audio output hardware and software has limitations. The Amiga computer's sound hardware requires that all one-shot and repeat parts have even numbers of samples. Amiga sound driver software would have to adjust an odd-sized waveform, ignore an odd-sized lowest octave, or ignore odd FORMs 8SVX altogether. Some other output devices require all sample sizes to be powers of two.

The field `samplesPerHiCycle` tells the number of samples/cycle in the highest frequency octave of data, or else 0 for "unknown". Each successive (lower frequency) octave contains twice as many samples/cycle. The `samplesPerHiCycle` value is needed to compute the data rate for a desired playback pitch.

Actually, `samplesPerHiCycle` is an average number of samples/cycle. If the one-shot part contains pitch bends, store the samples/cycle of the repeat part in `samplesPerHiCycle`. The division `repeatHiSamples/samplesPerHiCycle` should yield an integer number of cycles. (When the repeat waveform is repeated, a partial cycle would come out as a higher-frequency cycle with a "click".)

More limitations: Some Amiga music drivers require `samplesPerHiCycle` to be a power of two in order to play the FORM 8SVX as a musical instrument in tune. They may even assume `samplesPerHiCycle` is a particular power of two without checking. (If `samplesPerHiCycle` is different by a factor of two, the instrument will just be played an octave too low or high.)

The field `samplesPerSec` gives the sound sampling rate. A program may adjust this to achieve frequency shifts or vary it dynamically to achieve pitch bends and vibrato. A program that plays a FORM 8SVX as a musical instrument would ignore `samplesPerSec` and select a playback rate for each musical pitch.

The field `ctOctave` tells how many octaves of data are stored in the BODY chunk. See "Data Chunk BODY", below, for the layout of the octaves.

The field `sCompression` indicates the compression scheme, if any, that was applied to the entire set of data samples stored in the BODY chunk. This field should contain one of the values defined above. Of course, the matching decompression algorithm must be applied to the BODY data before the sound can be played. (The Fibonacci-delta encoding scheme `sCmpFibDelta` is described in Appendix C.) Note that the whole series of data samples is compressed as a unit.

The field `volume` gives an overall playback volume for the waveforms (all octaves). It lets the 8-bit data samples use the full range -128 through 127 for good signal-to-noise ratio and be attenuated on playback to the desired level. The playback program should multiply this value by a "volume control" and perhaps by a playback envelope (see ATAK and RLSE, below).

*Recording a one-shot sound.* To store a one-shot sound in a FORM 8SVX, set `oneShotHiSamples` = number of samples, `repeatHiSamples` = 0 , `samplesPerHiCycle` = 0, `samplesPerSec` = sampling rate, and `ctOctave` = 1. Scale the signal amplitude to the full sampling range -128 through 127. Set `volume` so the sound will playback at the desired volume level. If you set the `samplesPerHiCycle` field properly, the data can also be used as a musical instrument.

Experiment with data compression. If the decompressed signal sounds ok, store the compressed data in the BODY chunk and set `sCompression` to the compression code number.

*Recording a musical instrument.* To store a musical instrument in a FORM 8SVX, first record or synthesize as many octaves of data as you want to make available for playback. Set `ctOctaves` to the count of octaves. From the recorded data, excerpt an integral number of steady state cycles for the repeat part and set `repeatHiSamples` and `samplesPerHiCycle`. Either excerpt a startup transient waveform and set `oneShotHiSamples`, or else set `oneShotHiSamples` to 0. Remember, the one-shot and repeat parts of each octave must be twice as long as those of the next higher octave. Scale the signal amplitude to the full sampling range and set `volume` to adjust the instrument playback volume. If you set the `samplesPerSec` field properly, the data can also be used as a one-shot sound.

A distortion-introducing compressor like `sCmpFibDelta` is not recommended for musical instruments, but you might try it anway.

Typically, creators of FORM 8SVX record an acoustic instrument at just one frequency. Decimate (down-sample with filtering) to compute higher octaves. Interpolate to compute lower octaves.

If you sample an acoustic instrument at different octaves, you may find it hard to make the one-shot and repeat waveforms follow the factor-of-two rule for octaves. To compensate, lengthen an octave's one-shot part by appending replications of the repeating cycle or prepending zeros. (This will have minimal impact on the sound's start time.) You may be able to equalize the ratio one-shot-samples : repeat-samples across all octaves.

Note that a "one-shot sound" may be played as a "musical instrument" and vice versa. However, an instrument player depends on `samplesPerHiCycle`, and a one-shot player depends on `samplesPerSec`.

*Playing a one-shot sound.* To play any FORM 8SVX data as a one-shot sound, first select an octave if `ctOctave` > 1. (The lowest-frequency octave has the greatest resolution.) Play the one-shot samples then the repeat samples, scaled by `volume`, at a data rate of `samplesPerSec`. Of course, you may adjust the playback rate and volume. You can play out an envelope, too. (See ATAK and RLSE, below.)

*Playing a musical note.* To play a musical note using any FORM 8SVX, first select the nearest octave of data from those available. Play the one-shot waveform then cycle on the repeat waveform as long as needed to sustain the note. Scale the signal by `volume`, perhaps also by an envelope, and by a desired note volume. Select a playback data rate `s` samples/second to achieve the desired frequency (in Hz):

```
frequency = s / samplesPerHiCycle
```

for the highest frequency octave.

The idea is to select an octave and one of 12 sampling rates (assuming a 12-tone scale). If the FORM 8SVX doesn't have the right octave, you can decimate or interpolate from the available data.

When it comes to musical instruments, FORM 8SVX is geared for a simple sound driver. Such a driver uses a single table of 12 data rates to reach all notes in all octaves. That's why 8SVX requires each octave of data to have twice as many samples as the next higher octave. If you restrict `samplesPerHiCycle` to a power of two, you can use a predetermined table of data rates.

### Optional Text Chunks NAME, (c), AUTH, ANNO

Several text chunks may be included in a FORM 8SVX to keep ancillary information.

The optional property "NAME" names the voice, for instance "tubular bells".

The optional property "(c) " holds a copyright notice for the voice. The chunk ID "(c) " serves as the copyright characters "© ". E.g. a "(c) " chunk containing "1986 Electronic Arts" means "© 1986 Electronic Arts".

The optional property "AUTH" holds the name of the instrument's "author" or "creator".

The chunk types "NAME", "(c) ", and "AUTH" are property chunks. Putting more than one NAME (or other) property in a FORM is redundant. Just the last NAME counts. A property should be shorter than 256 characters. Properties can appear in a PROP 8SVX to share them over a LIST of FORMs 8SVX.

The optional data chunk "ANNO" holds any text annotations typed in by the author.

An ANNO chunk is not a property chunk, so you can put more than one in a FORM 8SVX. You can make ANNO chunks any length up to $2^{31} - 1$ characters, but 32767 is a practical limit. Since they're not properties, ANNO chunks don't belong in a PROP 8SVX. That means they can't be shared over a LIST of FORMs 8SVX.

Syntactically, each of these chunks contains an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F), just like a standard "TEXT" chunk. [See "Strings, String Chunks, and String Properties" in "EA IFF 85" Electronic Arts Interchange File Format.] The chunk's `ckSize` field holds the count of characters.

```
#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.                    */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.       */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.                   */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.           */
```

Remember to store a 0 pad byte after any odd-length chunk.

### Optional Data Chunks ATAK and RLSE

The optional data chunks ATAK and RLSE together give a piecewise-linear "envelope" or "amplitude contour". This contour may be used to modulate the sound during playback. It's especially useful for playing musical notes of variable durations. Playback programs may ignore the supplied envelope or substitute another.

```
#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')
```

```
typedef struct {
   UWORD duration;              /* segment duration in milliseconds, > 0   */
   Fixed dest;                  /* destination volume factor               */
   } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope. */
/* The envelope defines a function of time returning Fixed values. It's
 * used to scale the nominal volume specified in the Voice8Header.       */
```

To explain the meaning of the ATAK and RLSE chunks, we'll overview the envelope generation algorithm. Start at 0 volume, step through the ATAK contour, then hold at the sustain level (the last ATAK EGPoint's `dest`), and then step through the RLSE contour. Begin the release at the desired note stop time minus the total duration of the release contour (the sum of the RLSE EGPoints' `durations`). The attack contour should be cut short if the note is shorter than the release contour.

The envelope is a piecewise-linear function. The envelope generator interpolates between the EGPoints.

Remember to multiply the envelope function by the nominal voice header `volume` and by any desired note volume.

Figure 1 shows an example envelope. The attack period is described by 4 EGPoints in an ATAK chunk. The release period is described by 4 EGPoints in a RLSE chunk. The sustain period in the middle just holds the final ATAK level until it's time for the release.

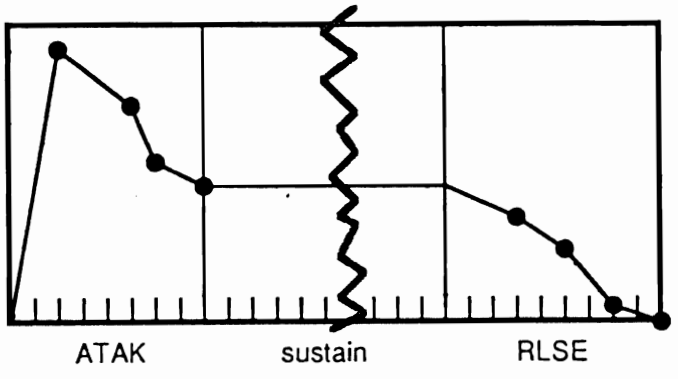



Figure 1. Amplitude contour.

Note: The number of EGPoints in an ATAK or RLSE chunk is its `ckSize / sizeof(EGPoint)`. In RAM, the playback program may terminate the array with a 0 `duration` EGPoint.

Issue: Synthesizers also provide frequency contour (pitch bend), filtering contour (wah-wah), amplitude oscillation (tremolo), frequency oscillation (vibrato), and filtering oscillation (leslie). In the future, we may define optional chunks to encode these modulations. The contours can be encoded in linear segments. The oscillations can be stored as segments with rate and depth parameters.

### Data Chunk BODY

The BODY chunk contains the audio data samples.

```
#define ID_BODY MakeID('B', 'O', 'D', 'Y')

typedef character BYTE;         /* 8 bit signed number, -128 through 127.  */
```

```
/* BODY chunk contains a BYTE[], array of audio data samples.          */
```

The BODY contains data samples grouped by octave. Within each octave are one-shot and repeat portions. Figure 2 depicts this arrangement of samples for an 8SVX where `oneShotHiSamples` = 24, `repeatHiSamples` = 16, `samplesPerHiCycle` = 8, and `ctOctave` = 3. The major divisions are octaves, the intermediate divisions separate the one-shot and repeat portions, and the minor divisions are cycles.

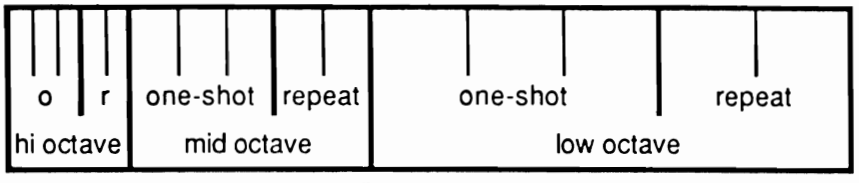



Figure 2. BODY subdivisions.

In general, the BODY has `ctOctave` octaves of data. The highest frequency octave comes first, comprising the fewest samples: `oneShotHiSamples + repeatHiSamples`. Each successive octave contains twice as many samples as the next higher octave but the same number of cycles. The lowest frequency octave comes last with the most samples: $2^{ctOctave-1}$ * (`oneShotHiSamples + repeatHiSamples`).

The number of samples in the BODY chunk is

$$(2^0 + \ldots + 2^{ctOctave-1}) * (oneShotHiSamples + repeatHiSamples)$$

Figure 3, below, looks closer at an example waveform within one octave of a different BODY chunk. In this example, `oneShotHiSamples / samplesPerHiCycle` = 2 cycles and `repeatHiSamples / samplesPerHiCycle` = 1 cycle.

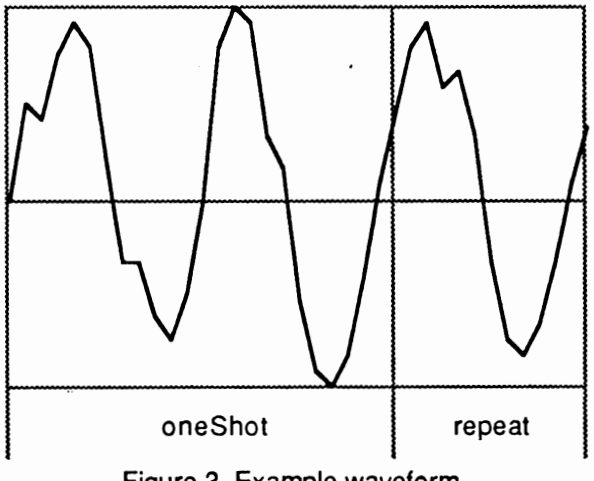



Figure 3. Example waveform.

To avoid playback "clicks", the one-shot part should begin with a small sample value, and the one-shot part should flow smoothly into the repeat part, and the end of the repeat part should flow smoothly into the beginning of the repeat part.

If the VHDR field `sCompression` ≠ `sCmpNone`, the BODY chunk is just an array of data bytes to feed through the specified decompresser function. All this stuff about sample sizes, octaves, and repeat parts applies to the decompressed data.

Be sure to follow an odd-length BODY chunk with a 0 pad byte.

## Other Chunks

Issue: In the future, we may define an optional chunk containing Fourier series coefficients for a repeating waveform. An editor for this kind of synthesized voice could modify the coefficients and regenerate the waveform.

## Appendix A. Quick Reference

### Type Definitions

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;             /* A fixed-point value, 16 bits to the left of
                                   the point and 16 to the right. A Fixed is a
                                   number of 2^16ths, i.e. 65536ths.          */
#define Unity 0x10000L          /* Unity = Fixed 1.0 = maximum volume         */

/* sCompression: Choice of compression algorithm applied to the samples. */
#define sCmpNone        0       /* not compressed                             */
#define sCmpFibDelta    1       /* Fibonacci-delta encoding (Appendix C)      */
                                /* Can be more kinds in the future.           */

typedef struct {
  ULONG oneShotHiSamples,       /* # samples in the high octave 1-shot part */
        repeatHiSamples,        /* # samples in the high octave repeat part */
        samplesPerHiCycle;      /* # samples/cycle in high octave, else 0   */
  UWORD samplesPerSec;          /* data sampling rate                       */
  UBYTE ctOctave,               /* # octaves of waveforms                   */
        sCompression;           /* data compression technique used          */
  Fixed volume;                 /* playback volume from 0 to Unity (full
                                 * volume). Map this value into the output
                                 * hardware's dynamic range.                */
  } Voice8Header;


#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.                          */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.             */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.                         */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.                 */


#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')

typedef struct {
  UWORD duration;               /* segment duration in milliseconds, > 0   */
  Fixed dest;                   /* destination volume factor               */
  } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope. */
/* The envelope defines a function of time returning Fixed values. It's
 * used to scale the nominal volume specified in the Voice8Header.       */
```

```
#define ID_BODY MakeID('B', 'O', 'D', 'Y')

typedef character BYTE;        /* 8 bit signed number, -128 through 127.  */

/* BODY chunk contains a BYTE[], array of audio data samples.             */
```

### 8SVX Regular Expression

Here's a regular expression summary of the FORM 8SVX syntax. This could be an IFF file or part of one.

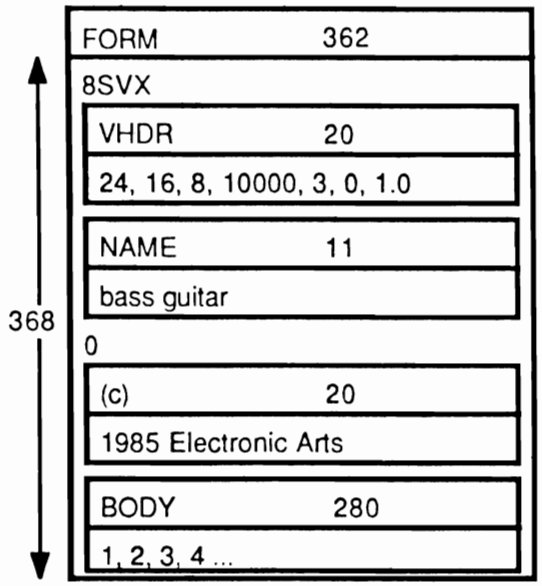
```
8SVX       ::= "FORM" #{ "8SVX" VHDR [NAME] [Copyright] [AUTH] ANNO*
                         [ATAK] [RLSE] BODY }

VHDR       ::= "VHDR" #{ Voice8Header  }
NAME       ::= "NAME" #{ CHAR*         } [0]
Copyright  ::= "(c) " #{ CHAR*         } [0]
AUTH       ::= "AUTH" #{ CHAR*         } [0]
ANNO       ::= "ANNO" #{ CHAR*         } [0]

ATAK       ::= "ATAK" #{ EGPoint*      }
RLSE       ::= "RLSE" #{ EGPoint*      }
BODY       ::= "FORM" #{ BYTE*         } [0]
```

The token "#" represents a `ckSize` LONG count of the following {braced} data bytes. E.g. a VHDR's "#" should equal `sizeof(Voice8Header)`. Literal items are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more replications. A sometimes-needed pad byte is shown as "`[0]`".

Actually, the order of chunks in a FORM 8SVX is not as strict as this regular expression indicates. The property chunks VHDR, NAME, Copyright, and AUTH may actually appear in any order as long as they all precede the BODY chunk. The optional data chunks ANNO, ATAK, and RLSE don't have to precede the BODY chunk. And of course, new kinds of chunks may appear inside a FORM 8SVX in the future.

## Appendix B. 8SVX Example

Here's a box diagram for a simple example containing the three octave BODY shown earlier in Figure 2.




The "0" after the NAME chunk is a pad byte.

## Appendix C. Fibonacci Delta Compression

This is Steve Hayes' Fibonacci Delta sound compression technique. It's like the traditional delta encoding but encodes each delta in a mere 4 bits. The compressed data is half the size of the original data plus a 2-byte overhead for the initial value. This much compression introduces some distortion, so try it out and use it with discretion.

To achieve a reasonable slew rate, this algorithm looks up each stored 4-bit value in a table of Fibonacci numbers. So very small deltas are encoded precisely while larger deltas are approximated. When it has to make approximations, the compressor should adjust all the values (forwards and backwards in time) for minimum overall distortion.

Here is the decompressor written in the C programming language.

```
/* Fibonacci delta encoding for sound data. */

BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Unpack Fibonacci-delta encoded data from n byte source buffer into 2*n byte
 * dest buffer, given initial data value x. It returns the last data value x
 * so you can call it several times to incrementally decompress the data. */
short D1Unpack(source, n, dest, x)
   BYTE source[], dest[];
   LONG n;
   BYTE x;
   {
   BYTE d;
   LONG i, lim;

   lim = n << 1;
   for (i = 0; i < lim; ++i)
      { /* Decode a data nybble; high nybble then low nybble. */
      d = source[i >> 1];          /* get a pair of nybbles */
      if (i & 1)                   /* select low or high nybble? */
         d &= 0xf;                 /* mask to get the low nybble */
      else
         d >>= 4;                  /* shift to get the high nybble */
      x += codeToDelta[d];         /* add in the decoded delta */
      dest[i] = x;                 /* store a 1-byte sample */
      }
   return(x);
   }

/* Unpack Fibonacci-delta encoded data from n byte source buffer into 2*(n-2)
 * byte dest buffer. Source buffer has a pad byte, an 8-bit initial value,
 * followed by n-2 bytes comprising 2*(n-2) 4-bit encoded samples. */
void DUnpack(source, n, dest)
   BYTE source[], dest[];
   LONG n;
   {
   D1Unpack(source + 2, n - 2, dest, source[1]);
   }
```

```
#ifndef COMPILER_H
#define COMPILER_H
/*** compiler.h ********************************************************/
/*  Steve Shaw                                                 1/29/86 */
/*  Portability file to handle compiler idiosyncrasies.                */
/*  Version: Lattice 3.03 cross-compiler for the Amiga from the IBM PC. */
/*                                                                     */
/* This software is in the public domain.                              */
/*                                                                     */
/***********************************************************************/

#ifndef EXEC_TYPES_H
#include "exec/types.h"
#endif


/* NOTE  --  NOTE  --  NOTE  --  NOTE  --  NOTE
 * Some C compilers can handle Function Declarations with Argument Types
 * (FDwAT) like this:
 *     extern LONG Seek(BPTR, LONG, LONG)
 * while others choke unless you just say
 *     extern LONG Seek()
 *
 * Comment out the #define FDwAT if you have a compiler that chokes. */


/* #define FDwAT        COMMENTED OUT BECAUSE GREENHILLS CANT TAKE IT */

#endif COMPILER_H
```



```
#ifndef GIO_H
#define GIO_H
/*---------------------------------------------------------------------*/
/* GIO.H  defs for Generic I/O Speed Up Package.              1/23/86 */
/* See GIOCall.C for an example of usage.                              */
/* Read not speeded-up yet.  Only one Write file buffered at a time.   */
/*                                                                     */
/* Note: The speed-up provided is ONLY significant for code such as IFF */
/* which does numerous small Writes and Seeks.                         */
/*                                                                     */
/* WARNING: If gio reports an error to you and you care what specific  */
/* Dos error was, you must call IoErr() BEFORE calling any other gio   */
/* functions.                                                          */
/*                                                                     */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                  */
/* This software is in the public domain.                              */
/*                                                                     */
/* This version for the Commodore-Amiga computer.                      */
/*                                                                     */
/*---------------------------------------------------------------------*/

/* Use this file interface in place of ALL Open,Close,Read,Write,Seek DOS
 * calls for an optional i/o speed-up via buffering.  You must use ONLY
 * these G routines for a file that is being buffered; e.g., call GClose
 * to Close the file, etc.
 * It is harmless though not necessary to use G routines for a file that
 * is not being buffered; e.g., GClose and Close are equivalent in that
 * case.
 * This Version only buffers one file at a time, and only for writing.
 * If you call GWriteDeclare for a second file before the first file
 * is GClosed, the first file becomes unbuffered.  This is harmless, no
 * data is lost, the first file is simply no longer speeded-up.
 */

/* Before compiling any modules that make G calls, or compiling gio.c,
 * you must set the GIO_ACTIVE flag below.
 *
 * To omit the speed-up code,
 *    #define GIO_ACTIVE 0
 *
 * To make the speed-up happen:
 * 1. #define GIO_ACTIVE 1
 * 2. link gio.o into your progrm
 * 3. GWriteDeclare(file, buffer, size)
 *    after GOpening the file and before doing
 *    any writing.
 * 4. ONLY use GRead, GWrite, GSeek, GClose -- do not use the DOS i/o
 *    routines directly.
 * 5. When done, do GClose.  Or to stop buffering without closing the
 *    file, do GWriteUndeclare(file).
 */
#define GIO_ACTIVE 0

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif
```

```
#ifndef LIBRARIES_DOS_H
#include "libraries/dos.h"
#endif

#ifndef OFFSET_BEGINNING
#define OFFSET_BEGINNING OFFSET_BEGINING
#endif

#if GIO_ACTIVE

#ifdef FDwAT  /* Compiler handles Function Declaration with Argument Types */

/* Present for completeness in the interface.
 * "openmode" is either MODE_OLDFILE to read/write an existing file, or
 * MODE_NEWFILE to write a new file.
 * RETURNs a "file" pointer to a system-supplied structure that describes
 * the open file.  This pointer is passed in to the other routines below.*/
extern BPTR GOpen(char * /*filename*/, LONG /*openmode*/);

/* NOTE: Flushes & Frees the write buffer.
 * Returns -1 on error from Write.*/
extern LONG GClose(BPTR /*file*/);

/* Read not speeded-up yet.
 * GOpen the file, then do GReads to get successive chunks of data in
 * the file.  Assumes the system can handle any number of bytes in each
 * call, regardless of any block-structure of the device being read from.
 * When done, GClose to free any system resources associated with an
 * open file.*/
extern LONG GRead(BPTR /*file*/, BYTE * /*buffer*/, LONG /*nBytes*/);

/* Writes out any data in write buffer for file.
 * NOTE WHEN have Seeked into middle of buffer:
 * GWriteFlush causes current position to be the end of the data written.
 * -1 on error from Write.*/
extern LONG GWriteFlush(BPTR /*file*/);

/* Sets up variables to describe a write buffer for the file.*/
/* If the buffer already has data in it from an outstanding GWriteDeclare,
 * then that buffer must first be flushed.
 * RETURN -1 on error from Write for that previous buffer flush.
 * See also "GWriteUndeclare".*/
extern LONG GWriteDeclare(BPTR /*file*/, BYTE * /*buffer*/, LONG /*nBytes*/);

/* ANY PROGRAM WHICH USES "GWrite" MUST USE "GSeek" rather than "Seek"
 * TO SEEK ON A FILE BEING WRITTEN WITH "GWrite".
 * "Write" with Generic speed-up.
 * -1 on error from Write.  else returns # bytes written to disk.
 * Call GOpen, then do successive GWrites with GSeeks if required,
 * then GClose when done.  (IFF does require GSeek.)*/
extern LONG GWrite(BPTR /*file*/, BYTE * /*buffer*/, LONG /*nBytes*/);

/* "Seek" with Generic speed-up, for a file being written with GWrite.*/
/* Returns what Seek returns, which appears to be the position BEFORE
 * seeking, though the documentation says it returns the NEW position.
```



```
 * In fact, the code now explicitly returns the OLD position when
 * seeking within the buffer.
 * Eventually, will support two independent files, one being read, the
 * other being written.  Or could support even more.  Designed so is safe
 * to call even for files which aren't being buffered.*/
extern LONG GSeek(BPTR /*file*/, LONG /*position*/, LONG /*mode*/);

#else /*not FDwAT*/

extern BPTR GOpen();
extern LONG GClose();
extern LONG GRead();
extern LONG GWriteFlush();
extern LONG GWriteDeclare();
extern LONG GWrite();
extern LONG GSeek();

#endif FDwAT

#else /* not GIO_ACTIVE */

#define GOpen(filename, openmode)            Open(filename, openmode)
#define GClose(file)                         Close(file)
#define GRead(file, buffer, nBytes)          Read(file, buffer, nBytes)
#define GWriteFlush(file)                    (0)
#define GWriteDeclare(file, buffer, nBytes)  (0)
#define GWrite(file, buffer, nBytes)         Write(file, buffer, nBytes)
#define GSeek(file, position, mode)          Seek(file, position, mode)

#endif GIO_ACTIVE


/* Release the buffer for that file, flushing it to disk if it has any
 * contents.  GWriteUndeclare(NULL) to release ALL buffers.
 * Currently, only one file can be buffered at a time anyway.*/
#define GWriteUndeclare(file)  GWriteDeclare(file, NULL, 0)


#endif
```

```
#ifndef IFF_H
#define IFF_H
/*-----------------------------------------------------------------------*/
/* IFF.H  defs for IFF-85 Interchange Format Files.            1/22/86 */
/*                                                                       */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                    */
/* This software is in the public domain.                                */
/*-----------------------------------------------------------------------*/

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#ifndef LIBRARIES_DOS_H
#include "libraries/dos.h"
#endif

#ifndef OFFSET_BEGINNING
#define OFFSET_BEGINNING OFFSET_BEGINING
#endif

typedef LONG IFFP;      /* Status code result from an IFF procedure */
        /* LONG, because must be type compatable with ID for GetChunkHdr.*/
        /* Note that the error codes below are not legal IDs.*/
#define IFF_OKAY    0L  /* Keep going...*/
#define END_MARK   -1L  /* As if there was a chunk at end of group.*/
#define IFF_DONE   -2L  /* clientProc returns this when it has READ enough.
                         * It means return thru all levels. File is Okay.*/
#define DOS_ERROR  -3L
#define NOT_IFF    -4L  /* not an IFF file.*/
#define NO_FILE    -5L  /* Tried to open file, DOS didn't find it.*/
#define CLIENT_ERROR -6L /* Client made invalid request, for instance, write
                         * a negative size chunk.*/
#define BAD_FORM   -7L  /* A client read proc complains about FORM semantics;
                         * e.g. valid IFF, but missing a required chunk.*/
#define SHORT_CHUNK -8L /* Client asked to IFFReadBytes more bytes than left
                         * in the chunk. Could be client bug or bad form.*/
#define BAD_IFF    -9L  /* mal-formed IFF file. [TBD] Expand this into a
                         * range of error codes.*/
#define LAST_ERROR BAD_IFF

/* This MACRO is used to RETURN immediately when a termination condition is
 * found. This is a pretty weird macro. It requires the caller to declare a
 * local "IFFP iffp" and assign it. This wouldn't work as a subroutine since
 * it returns for it's caller. */
#define CheckIFFP()    { if (iffp != IFF_OKAY) return(iffp); }


/* ---------- ID ------------------------------------------------------*/

typedef LONG ID;        /* An ID is four printable ASCII chars but
                         * stored as a LONG for efficient copy & compare.*/

/* Four-character IDentifier builder.*/
#define MakeID(a,b,c,d)   ( (LONG)(a)<<24L | (LONG)(b)<<16L | (c)<<8 | (d) )
```



```
/* Standard group IDs.  A chunk with one of these IDs contains a
   SubTypeID followed by zero or more chunks.*/
#define FORM MakeID('F','O','R','M')
#define PROP MakeID('P','R','O','P')
#define LIST MakeID('L','I','S','T')
#define CAT  MakeID('C','A','T',' ')
#define FILLER MakeID(' ',' ',' ',' ')
/* The IDs "FOR1".."FOR9", "LIS1".."LIS9", & "CAT1".."CAT9" are reserved
 * for future standardization.*/

/* Pseudo-ID used internally by chunk reader and writer.*/
#define NULL_CHUNK 0L           /* No current chunk.*/


/* ---------- Chunk ---------------------------------------------------*/

/* All chunks start with a type ID and a count of the data bytes that
   follow--the chunk's "logicl size" or "data size". If that number is odd,
   a 0 pad byte is written, too. */
typedef struct {
    ID    ckID;
    LONG  ckSize;
    } ChunkHeader;

typedef struct {
    ID    ckID;
    LONG  ckSize;
    UBYTE ckData[ 1 /*REALLY: ckSize*/ ];
    } Chunk;

/* Pass ckSize = szNotYetKnown to the writer to mean "compute the size".*/
#define szNotYetKnown 0x80000001L

/* Need to know whether a value is odd so can word-align.*/
#define IS_ODD(a)    ((a) & 1)

/* This macro rounds up to an even number. */
#define WordAlign(size)    ((size+1)&~1)

/* ALL CHUNKS MUST BE PADDED TO EVEN NUMBER OF BYTES.
 * ChunkPSize computes the total "physical size" of a padded chunk from
 * its "data size" or "logical size". */
#define ChunkPSize(dataSize)  (WordAlign(dataSize) + sizeof(ChunkHeader))

/* The Grouping chunks (LIST, FORM, PROP, & CAT) contain concatenations of
 * chunks after a subtype ID that identifies the content chunks.
 * "FORM type XXXX", "LIST of FORM type XXXX", "PROPerties associated
 * with FORM type XXXX", or "conCATenation of XXXX".*/
typedef struct {
    ID    ckID;
    LONG  ckSize;       /* this ckSize includes "grpSubID".*/
    ID    grpSubID;
    } GroupHeader;

typedef struct {
    ID    ckID;
```

```
    LONG  ckSize;
    ID    grpSubID;
    UBYTE grpData[ 1 /*REALLY: ckSize-sizeof(grpSubID)*/ ];
    } GroupChunk;


/* ---------- IFF Reader ------------------------------------------------*/

/******** Routines to support a stream-oriented IFF file reader *******
 *
 * These routines handle lots of details like error checking and skipping
 * over padding. They're also careful not to read past any containing context.
 *
 * These routines ASSUME they're the only ones reading from the file.
 * Client should check IFFP error codes. Don't press on after an error!
 * These routines try to have no side effects in the error case, except
 * partial I/O is sometimes unavoidable.
 *
 * All of these routines may return DOS_ERROR. In that case, ask DOS for the
 * specific error code.
 *
 * The overall scheme for the low level chunk reader is to open a "group read
 * context" with OpenRIFF or OpenRGroup, read the chunks with GetChunkHdr
 * (and its kin) and IFFReadBytes, and close the context with CloseRGroup.
 *
 * The overall scheme for reading an IFF file is to use ReadIFF, ReadIList,
 * and ReadICat to scan the file. See those procedures, ClientProc (below)
 * and the skeleton IFF reader. */

/* Client passes ptrs to procedures of this type to ReadIFF which call them
 * back to handle LISTs, FORMs, CATs, and PROPs.
 *
 * Use the GroupContext ptr when calling reader routines like GetChunkHdr.
 * Look inside the GroupContext ptr for your ClientFrame ptr. You'll
 * want to type cast it into a ptr to your containing struct to get your
 * private contextual data (stacked property settings). See below. */
#ifdef FDwAT
typedef IFFP ClientProc(struct _GroupContext *);
#else
typedef IFFP ClientProc();
#endif

/* Client's context for reading an IFF file or a group.
 * Client should actually make this the first component of a larger struct
 * (it's personal stack "frame") that has a field to store each "interesting"
 * property encountered.
 * Either initialize each such field to a global default or keep a boolean
 * indicating if you've read a property chunk into that field.
 * Your getList and getForm procs should allocate a new "frame" and copy the
 * parent frame's contents. The getProp procedure should store into the frame
 * allocated by getList for the containing LIST. */
typedef struct _ClientFrame {
   ClientProc *getList, *getProp, *getForm, *getCat;
    /* client's own data follows; place to stack property settings */
    } ClientFrame;
```



```
/* Our context for reading a group chunk. */
typedef struct _GroupContext {
    struct _GroupContext *parent; /* Containing group; NULL => whole file. */
    ClientFrame *clientFrame;     /* Reader data & client's context state. */
    BPTR file;          /* Byte-stream file handle. */
    LONG position;      /* The context's logical file position. */
    LONG bound;         /* File-absolute context bound
                         * or szNotYetKnown (writer only). */
    ChunkHeader ckHdr;  /* Current chunk header. ckHdr.ckSize = szNotYetKnown
                         * means we need to go back and set the size (writer only
                         * See also Pseudo-IDs, above. */
    ID subtype;         /* Group's subtype ID when reading. */
    LONG bytesSoFar;    /* # bytes read/written of current chunk's data. */
    } GroupContext;

/* Computes the number of bytes not yet read from the current chunk, given
 * a group read context gc. */
#define ChunkMoreBytes(gc)  ((gc)->ckHdr.ckSize - (gc)->bytesSoFar)


/***** Low Level IFF Chunk Reader *****/

#ifdef FDwAT

/* Given an open file, open a read context spanning the whole file.
 * This is normally only called by ReadIFF.
 * This sets new->clientFrame = clientFrame.
 * ASSUME context allocated by caller but not initialized.
 * ASSUME caller doesn't deallocate the context before calling CloseRGroup.
 * NOT_IFF ERROR if the file is too short for even a chunk header.*/
extern IFFP OpenRIFF(BPTR, GroupContext *, ClientFrame *);
                /*  file, new,            clientFrame  */

/* Open the remainder of the current chunk as a group read context.
 * This will be called just after the group's subtype ID has been read
 * (automatically by GetChunkHdr for LIST, FORM, PROP, and CAT) so the
 * remainder is a sequence of chunks.
 * This sets new->clientFrame = parent->clientFrame. The caller should repoint
 * it at a new clientFrame if opening a LIST context so it'll have a "stack
 * frame" to store PROPs for the LIST. (It's usually convenient to also
 * allocate a new Frame when you encounter FORM of the right type.)
 *
 * ASSUME new context allocated by caller but not initialized.
 * ASSUME caller doesn't deallocate the context or access the parent context
 * before calling CloseRGroup.
 * BAD_IFF ERROR if context end is odd or extends past parent. */
extern IFFP OpenRGroup(GroupContext *, GroupContext *);
                  /*  parent,         new  */

/* Close a group read context, updating its parent context.
 * After calling this, the old context may be deallocated and the parent
 * context can be accessed again. It's okay to call this particular procedure
 * after an error has occurred reading the group.
 * This always returns IFF_OKAY. */
extern IFFP CloseRGroup(GroupContext *);
                 /*  old  */
```

```
/* Skip any remaining bytes of the previous chunk and any padding, then
 * read the next chunk header into context.ckHdr.
 * If the ckID is LIST, FORM, CAT, or PROP, this automatically reads the
 * subtype ID into context->subtype.
 * Caller should dispatch on ckID (and subtype) to an appropriate handler.
 *
 * RETURNS context.ckHdr.ckID (the ID of the new chunk header); END_MARK
 * if there are no more chunks in this context; or NOT_IFF if the top level
 * file chunk isn't a FORM, LIST, or CAT; or BAD_IFF if malformed chunk, e.g.
 * ckSize is negative or too big for containing context, ckID isn't positive,
 * or we hit end-of-file.
 *
 * See also GetFChunkHdr, GetF1ChunkHdr, and GetPChunkHdr, below.*/
extern ID       GetChunkHdr(GroupContext *);
  /*  context.ckHdr.ckID    context  */

/* Read nBytes number of data bytes of current chunk. (Use OpenGroup, etc.
 * instead to read the contents of a group chunk.) You can call this several
 * times to read the data piecemeal.
 * CLIENT_ERROR if nBytes < 0. SHORT_CHUNK if nBytes > ChunkMoreBytes(context)
 * which could be due to a client bug or a chunk that's shorter than it
 * ought to be (bad form). (on either CLIENT_ERROR or SHORT_CHUNK,
 * IFFReadBytes won't read any bytes.) */
extern IFFP IFFReadBytes(GroupContext *, BYTE *, LONG);
                    /*  context,       buffer, nBytes  */


/***** IFF File Reader *****/

/* This is a noop ClientProc that you can use for a getList, getForm, getProp,
 * or getCat procedure that just skips the group. A simple reader might just
 * implement getForm, store ReadICat in the getCat field of clientFrame, and
 * use SkipGroup for the getList and getProp procs.*/
extern IFFP SkipGroup(GroupContext *);

/* IFF file reader.
 * Given an open file, allocate a group context and use it to read the FORM,
 * LIST, or CAT and it's contents. The idea is to parse the file's contents,
 * and for each FORM, LIST, CAT, or PROP encountered, call the getForm,
 * getList, getCat, or getProp procedure in clientFrame, passing the
 * GroupContext ptr.
 * This is achieved with the aid of ReadIList (which your getList should
 * call) and ReadICat (which your getCat should call, if you don't just use
 * ReadICat for your getCat). If you want to handle FORMs, LISTs, and CATs
 * nested within FORMs, the getForm procedure must dispatch to getForm,
 * getList, and getCat (it can use GetF1ChunkHdr to make this easy).
 *
 * Normal return is IFF_OKAY (if whole file scanned) or IFF_DONE (if a client
 * proc said "done" first).
 * See the skeletal getList, getForm, getCat, and getProp procedures. */
extern IFFP ReadIFF(BPTR, ClientFrame *);
              /*  file, clientFrame  */

/* IFF LIST reader.
 * Your "getList" procedure should allocate a ClientFrame, copy the parent's
```



```
 * ClientFrame, and then call this procedure to do all the work.
 *
 * Normal return is IFF_OKAY (if whole LIST scanned) or IFF_DONE (if a client
 * proc said "done" first).
 * BAD_IFF ERROR if a PROP appears after a non-PROP. */
extern IFFP ReadIList(GroupContext *, ClientFrame *);
                /*  parent,         clientFrame  */

/* IFF CAT reader.
 * Most clients can simply use this to read their CATs. If you must do extra
 * setup work, put a ptr to your getCat procedure in the clientFrame, and
 * have that procedure call ReadICat to do the detail work.
 *
 * Normal return is IFF_OKAY (if whole CAT scanned) or IFF_DONE (if a client
 * proc said "done" first).
 * BAD_IFF ERROR if a PROP appears in the CAT. */
extern IFFP ReadICat(GroupContext *);
               /*  parent  */

/* Call GetFChunkHdr instead of GetChunkHdr to read each chunk inside a FORM.
 * It just calls GetChunkHdr and returns BAD_IFF if it gets a PROP chunk. */
extern ID       GetFChunkHdr(GroupContext *);
  /*  context.ckHdr.ckID    context  */

/* GetF1ChunkHdr is like GetFChunkHdr, but it automatically dispatches to the
 * getForm, getList, and getCat procedure (and returns the result) if it
 * encounters a FORM, LIST, or CAT. */
extern ID       GetF1ChunkHdr(GroupContext *);
  /*  context.ckHdr.ckID    context  */

/* Call GetPChunkHdr instead of GetChunkHdr to read each chunk inside a PROP.
 * It just calls GetChunkHdr and returns BAD_IFF if it gets a group chunk. */
extern ID       GetPChunkHdr(GroupContext *);
  /*  context.ckHdr.ckID    context  */

#else /* not FDwAT */

extern IFFP OpenRIFF();
extern IFFP OpenRGroup();
extern IFFP CloseRGroup();
extern ID   GetChunkHdr();
extern IFFP IFFReadBytes();
extern IFFP SkipGroup();
extern IFFP ReadIFF();
extern IFFP ReadIList();
extern IFFP ReadICat();
extern ID   GetFChunkHdr();
extern ID   GetF1ChunkHdr();
extern ID   GetPChunkHdr();

#endif /* not FDwAT */

/* ---------- IFF Writer ----------------------------------------------*/

/******* Routines to support a stream-oriented IFF file writer *******
 *
```

```
 * These routines will random access back to set a chunk size value when the
 * caller doesn't know it ahead of time. They'll also do things automatically
 * like padding and error checking.
 *
 * These routines ASSUME they're the only ones writing to the file.
 * Client should check IFFP error codes. Don't press on after an error!
 * These routines try to have no side effects in the error case, except that
 * partial I/O is sometimes unavoidable.
 *
 * All of these routines may return DOS_ERROR. In that case, ask DOS for the
 * specific error code.
 *
 * The overall scheme is to open an output GroupContext via OpenWIFF or
 * OpenWGroup, call either PutCk or {PutCkHdr {IFFWriteBytes}* PutCkEnd} for
 * each chunk, then use CloseWGroup to close the GroupContext.
 *
 * To write a group (LIST, FORM, PROP, or CAT), call StartWGroup, write out
 * its chunks, then call EndWGroup. StartWGroup automatically writes the
 * group header and opens a nested context for writing the contents.
 * EndWGroup closes the nested context and completes the group chunk. */


#ifdef FDwAT

/* Given a file open for output, open a write context.
 * The "limit" arg imposes a fence or upper limit on the logical file
 * position for writing data in this context. Pass in szNotYetKnown to be
 * bounded only by disk capacity.
 * ASSUME new context structure allocated by caller but not initialized.
 * ASSUME caller doesn't deallocate the context before calling CloseWGroup.
 * The caller is only allowed to write out one FORM, LIST, or CAT in this top
 * level context (see StartWGroup and PutCkHdr).
 * CLIENT_ERROR if limit is odd.*/
extern IFFP OpenWIFF(BPTR, GroupContext *, LONG);
                /*  file, new,            limit {file position}  */

/* Start writing a group (presumably LIST, FORM, PROP, or CAT), opening a
 * nested context. The groupSize includes all nested chunks + the subtype ID.
 *
 * The subtype of a LIST or CAT is a hint at the contents' FORM type(s). Pass
 * in FILLER if it's a mixture of different kinds.
 *
 * This writes the chunk header via PutCkHdr, writes the subtype ID via
 * IFFWriteBytes, and calls OpenWGroup. The caller may then write the nested
 * chunks and finish by calling EndWGroup.
 * The OpenWGroup call sets new->clientFrame = parent->clientFrame.
 *
 * ASSUME new context structure allocated by caller but not initialized.
 * ASSUME caller doesn't deallocate the context or access the parent context
 * before calling CloseWGroup.
 * ERROR conditions: See PutCkHdr, IFFWriteBytes, OpenWGroup. */
extern IFFP StartWGroup(GroupContext *, ID, LONG, ID, GroupContext *);
                /*  parent, groupType, groupSize, subtype, new  */

/* End a group started by StartWGroup.
 * This just calls CloseWGroup and PutCkEnd.
```



```
 * ERROR conditions: See CloseWGroup and PutCkEnd. */
extern IFFP EndWGroup(GroupContext *);
                /*  old  */

/* Open the remainder of the current chunk as a group write context.
 * This is normally only called by StartWGroup.
 *
 * Any fixed limit to this group chunk or a containing context will impose
 * a limit on the new context.
 * This will be called just after the group's subtype ID has been written
 * so the remaining contents will be a sequence of chunks.
 * This sets new->clientFrame = parent->clientFrame.
 * ASSUME new context structure allocated by caller but not initialized.
 * ASSUME caller doesn't deallocate the context or access the parent context
 * before calling CloseWGroup.
 * CLIENT_ERROR if context end is odd or PutCkHdr wasn't called first. */
extern IFFP OpenWGroup(GroupContext *, GroupContext *);
                /*  parent,         new  */

/* Close a write context and update its parent context.
 * This is normally only called by EndWGroup.
 *
 * If this is a top level context (created by OpenWIFF) we'll set the file's
 * EOF (end of file) but won't close the file.
 * After calling this, the old context may be deallocated and the parent
 * context can be accessed again.
 *
 * Amiga DOS Note: There's no call to set the EOF. We just position to the
 * desired end and return. Caller must Close file at that position.
 * CLIENT_ERROR if PutCkEnd wasn't called first. */
extern IFFP CloseWGroup(GroupContext *);
                /*  old  */

/* Write a whole chunk to a GroupContext. This writes a chunk header, ckSize
 * data bytes, and (if needed) a pad byte. It also updates the GroupContext.
 * CLIENT_ERROR if ckSize == szNotYetKnown. See also PutCkHdr errors. */
extern IFFP PutCk(GroupContext *, ID,   LONG,   BYTE *);
             /*  context,        ckID, ckSize, *data  */

/* Write just a chunk header. Follow this will any number of calls to
 * IFFWriteBytes and finish with PutCkEnd.
 * If you don't yet know how big the chunk is, pass in ckSize = szNotYetKnown,
 * then PutCkEnd will set the ckSize for you later.
 * Otherwise, IFFWriteBytes and PutCkEnd will ensure that the specified
 * number of bytes get written.
 * CLIENT_ERROR if the chunk would overflow the GroupContext's bound, if
 * PutCkHdr was previously called without a matching PutCkEnd, if ckSize < 0
 * (except szNotYetKnown), if you're trying to write something other
 * than one FORM, LIST, or CAT in a top level (file level) context, or
 * if ckID <= 0 (these illegal ID values are used for error codes). */
extern IFFP PutCkHdr(GroupContext *, ID,   LONG);
             /*  context,        ckID, ckSize  */

/* Write nBytes number of data bytes for the current chunk and update
 * GroupContext.
 * CLIENT_ERROR if this would overflow the GroupContext's limit or the
```

```
 * current chunk's ckSize, or if PutCkHdr wasn't called first, or if
 * nBytes < 0. */
extern IFFP IFFWriteBytes(GroupContext *, BYTE *, LONG);
                  /*  context,       *data,  nBytes  */

/* Complete the current chunk, write a pad byte if needed, and update
 * GroupContext.
 * If current chunk's ckSize = szNotYetKnown, this goes back and sets the
 * ckSize in the file.
 * CLIENT_ERROR if PutCkHdr wasn't called first, or if client hasn't
 * written 'ckSize' number of bytes with IFFWriteBytes. */
extern IFFP PutCkEnd(GroupContext *);
            /*  context  */

#else /* not FDwAT */

extern IFFP OpenWIFF();
extern IFFP StartWGroup();
extern IFFP EndWGroup();
extern IFFP OpenWGroup();
extern IFFP CloseWGroup();
extern IFFP PutCk();
extern IFFP PutCkHdr();
extern IFFP IFFWriteBytes();
extern IFFP PutCkEnd();

#endif /* not FDwAT */

#endif IFF_H
```



```
#ifndef ILBM_H
#define ILBM_H
/*****************************************************************************
 * ILBM.H  Definitions for InterLeaved BitMap raster image.      1/23/86
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *****************************************************************************/
#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#ifndef GRAPHICS_GFX_H
#include "graphics/gfx.h"
#endif

#include "iff/iff.h"

#define ID_ILBM MakeID('I','L','B','M')
#define ID_BMHD MakeID('B','M','H','D')
#define ID_CMAP MakeID('C','M','A','P')
#define ID_GRAB MakeID('G','R','A','B')
#define ID_DEST MakeID('D','E','S','T')
#define ID_SPRT MakeID('S','P','R','T')
#define ID_CAMG MakeID('C','A','M','G')
#define ID_BODY MakeID('B','O','D','Y')

/* ---------- BitMapHeader ------------------------------------------------*/

typedef UBYTE Masking;          /* Choice of masking technique.*/
#define mskNone                 0
#define mskHasMask              1
#define mskHasTransparentColor  2
#define mskLasso                3

typedef UBYTE Compression;      /* Choice of compression algorithm applied to
     * each row of the source and mask planes. "cmpByteRun1" is the byte run
     * encoding generated by Mac's PackBits. See Packer.h . */
#define cmpNone      0
#define cmpByteRun1  1

/* Aspect ratios: The proper fraction xAspect/yAspect represents the pixel
 * aspect ratio pixel_width/pixel_height.
 *
 * For the 4 Amiga display modes:
 *    320 x 200: 10/11  (these pixels are taller than they are wide)
 *    320 x 400: 20/11
 *    640 x 200:  5/11
 *    640 x 400: 10/11          */
#define x320x200Aspect 10
#define y320x200Aspect 11
#define x320x400Aspect 20
#define y320x400Aspect 11
#define x640x200Aspect  5
```

```
#define y640x200Aspect 11
#define x640x400Aspect 10
#define y640x400Aspect 11

/* A BitMapHeader is stored in a BMHD chunk. */
typedef struct {
    UWORD w, h;                 /* raster width & height in pixels */
    WORD  x, y;                 /* position for this image */
    UBYTE nPlanes;              /* # source bitplanes */
    Masking masking;            /* masking technique */
    Compression compression;    /* compression algoithm */
    UBYTE pad1;                 /* UNUSED.  For consistency, put 0 here.*/
    UWORD transparentColor;     /* transparent "color number" */
    UBYTE xAspect, yAspect;     /* aspect ratio, a rational number x/y */
    WORD  pageWidth, pageHeight;  /* source "page" size in pixels */
    } BitMapHeader;

/* RowBytes computes the number of bytes in a row, from the width in pixels.*/
#define RowBytes(w)    (((w) + 15) >> 4 << 1)


/* ---------- ColorRegister ---------------------------------------------*/
/* A CMAP chunk is a packed array of ColorRegisters (3 bytes each). */
typedef struct {
    UBYTE red, green, blue;   /* MUST be UBYTEs so ">> 4" won't sign extend.*/
    } ColorRegister;

/* Use this constant instead of sizeof(ColorRegister). */
#define sizeofColorRegister  3

typedef WORD Color4;     /* Amiga RAM version of a color-register,
                          * with 4 bits each RGB in low 12 bits.*/

/* Maximum number of bitplanes in RAM. Current Amiga max w/dual playfield. */
#define MaxAmDepth 6

/* ---------- Point2D ---------------------------------------------------*/
/* A Point2D is stored in a GRAB chunk. */
typedef struct {
    WORD x, y;              /* coordinates (pixels) */
    } Point2D;

/* ---------- DestMerge -------------------------------------------------*/
/* A DestMerge is stored in a DEST chunk. */
typedef struct {
    UBYTE depth;            /* # bitplanes in the original source */
    UBYTE pad1;             /* UNUSED; for consistency store 0 here */
    UWORD planePick;        /* how to scatter source bitplanes into destination */
    UWORD planeOnOff;       /* default bitplane data for planePick */
    UWORD planeMask;        /* selects which bitplanes to store into */
    } DestMerge;

/* ---------- SpritePrecedence ------------------------------------------*/
/* A SpritePrecedence is stored in a SPRT chunk. */
typedef UWORD SpritePrecedence;
```



```
/* ---------- Viewport Mode ---------------------------------------------*/
/* A Commodore Amiga ViewPort->Modes is stored in a CAMG chunk. */
/* The chunk's content is declared as a LONG. */

/* ---------- CRange ----------------------------------------------------*/
/* A CRange is store in a CRNG chunk. */
typedef struct {
    WORD  pad1;         /* reserved for future use; store 0 here */
    WORD  rate;         /* color cycling rate, 16384 = 60 steps/second */
    WORD  active;       /* nonzero means color cycling is turned on */
    UBYTE low, high;    /* lower and upper color registers selected */
    } CRange;

/* ---------- ILBM Writer Support Routines ------------------------------*/

/* Note: Just call PutCk to write a BMHD, GRAB, DEST, SPRT, or CAMG
 * chunk. As below. */
#define PutBMHD(context, bmHdr)  \
    PutCk(context, ID_BMHD, sizeof(BitMapHeader), (BYTE *)bmHdr)
#define PutGRAB(context, point2D)  \
    PutCk(context, ID_GRAB, sizeof(Point2D), (BYTE *)point2D)
#define PutDEST(context, destMerge)  \
    PutCk(context, ID_DEST, sizeof(DestMerge), (BYTE *)destMerge)
#define PutSPRT(context, spritePrec)  \
    PutCk(context, ID_SPRT, sizeof(SpritePrecedence), (BYTE *)spritePrec)

#ifdef FDwAT

/* Initialize a BitMapHeader record for a full-BitMap ILBM picture.
 * This gets w, h, and nPlanes from the BitMap fields BytesPerRow, Rows, and
 * Depth. It assumes you want  w = bitmap->BytesPerRow * 8 .
 * CLIENT_ERROR if bitmap->BytesPerRow isn't even, as required by ILBM format.
 *
 * If (pageWidth, pageHeight) is (320, 200), (320, 400), (640, 200), or
 * (640, 400) this sets (xAspect, yAspect) based on those 4 Amiga display
 * modes. Otherwise, it sets them to (1, 1).
 *
 * After calling this, store directly into the BitMapHeader if you want to
 * override any settings, e.g. to make nPlanes smaller, to reduce w a little,
 * or to set a position (x, y) other than (0, 0).*/
extern IFFP InitBMHdr(BitMapHeader *, struct BitMap *,
                      /*  bmHdr,          bitmap  */
    int,     int,         int,              WORD,      WORD);
 /*  masking, compression, transparentColor, pageWidth, pageHeight */
 /*  Masking, Compression, UWORD -- are the desired types, but get
  *  compiler warnings if use them.                                  */

/* Output a CMAP chunk to an open FORM ILBM write context. */
extern IFFP PutCMAP(GroupContext *, WORD *,   UBYTE);
                    /*  context,        colorMap, depth  */

/* This procedure outputs a BitMap as an ILBM's BODY chunk with
 * bitplane and mask data. Compressed if bmHdr->compression == cmpByteRun1.
 * If the "mask" argument isn't NULL, it merges in the mask plane, too.
 * (A fancier routine could write a rectangular portion of an image.)
 * This gets Planes (bitplane ptrs) from "bitmap".
```

```
 *
 * CLIENT_ERROR if bitmap->Rows != bmHdr->h, or if
 * bitmap->BytesPerRow != RowBytes(bmHdr->w), or if
 * bitmap->Depth < bmHdr->nPlanes, or if bmHdr->nPlanes > MaxAmDepth, or if
 * bufsize < MaxPackedSize(bitmap->BytesPerRow), or if
 * bmHdr->compression > cmpByteRun1. */
extern IFFP PutBODY(
    GroupContext *, struct BitMap *, BYTE *, BitMapHeader *, BYTE *, LONG);
    /*  context,          bitmap,   mask,   bmHdr,         buffer, bufsize */

#else /*not FDwAT*/

extern IFFP InitBMHdr();
extern IFFP PutCMAP();
extern IFFP PutBODY();

#endif FDwAT

/* ---------- ILBM Reader Support Routines ----------------------------*/

/* Note: Just call IFFReadBytes to read a BMHD, GRAB, DEST, SPRT, or CAMG
 * chunk. As below. */
#define GetBMHD(context, bmHdr)   \
    IFFReadBytes(context, (BYTE *)bmHdr, sizeof(BitMapHeader))
#define GetGRAB(context, point2D)   \
    IFFReadBytes(context, (BYTE *)point2D, sizeof(Point2D))
#define GetDEST(context, destMerge)   \
    IFFReadBytes(context, (BYTE *)destMerge, sizeof(DestMerge))
#define GetSPRT(context, spritePrec)   \
    IFFReadBytes(context, (BYTE *)spritePrec, sizeof(SpritePrecedence))

/* GetBODY can handle a file with up to 16 planes plus a mask.*/
#define MaxSrcPlanes 16+1

#ifdef FDwAT

/* Input a CMAP chunk from an open FORM ILBM read context.
 * This converts to an Amiga color map: 4 bits each of red, green, blue packed
 * into a 16 bit color register.
 * pNColorRegs is passed in as a pointer to a UBYTE variable that holds
 * the number of ColorRegisters the caller has space to hold. GetCMAP sets
 * that variable to the number of color registers actually read.*/
extern IFFP GetCMAP(GroupContext *, WORD *,   UBYTE *);
                /*  context,        colorMap, pNColorRegs  */

/* GetBODY reads an ILBM's BODY into a client's bitmap, de-interleaving and
 * decompressing.
 *
 * Caller should first compare bmHdr dimensions (rowWords, h, nPlanes) with
 * bitmap dimensions, and consider reallocating the bitmap.
 * If file has more bitplanes than bitmap, this reads first few planes (low
 * order ones). If bitmap has more bitplanes, the last few are untouched.
 * This reads the MIN(bmHdr->h, bitmap->Rows) rows, discarding the bottom
 * part of the source or leaving the bottom part of the bitmap untouched.
 *
 * GetBODY returns CLIENT_ERROR if asked to perform a conversion it doesn't
```



```
 * handle. It only understands compression algorithms cmpNone and cmpByteRun1.
 * The filed row width (# words) must agree with bitmap->BytesPerRow.
 *
 * Caller should use bmHdr.w; GetBODY only uses it to compute the row width
 * in words. Pixels to the right of bmHdr.w are not defined.
 *
 * [TBD] In the future, GetBODY could clip the stored image horizontally or
 * fill (with transparentColor) untouched parts of the destination bitmap.
 *
 * GetBODY stores the mask plane, if any, in the buffer pointed to by mask.
 * If mask == NULL, GetBODY will skip any mask plane. If
 * (bmHdr.masking != mskHasMask) GetBODY just leaves the caller's mask alone.
 *
 * GetBODY needs a buffer large enough for two compressed rows.
 * It returns CLIENT_ERROR if bufsize < 2 * MaxPackedSize(bmHdr.rowWords * 2).
 *
 * GetBODY can handle a file with up to MaxSrcPlanes planes. It returns
 * CLIENT_ERROR if the file has more. (Could be due to a bum file, though.)
 * If GetBODY fails, itt might've modified the client's bitmap. Sorry.*/
extern IFFP GetBODY(
    GroupContext *, struct BitMap *, BYTE *, BitMapHeader *, BYTE *, LONG);
    /*  context,          bitmap,   mask,   bmHdr,         buffer, bufsize */

/* [TBD] Add routine(s) to create masks when reading ILBMs whose
 * masking != mskHasMask. For mskNone, create a rectangular mask. For
 * mskHasTransparentColor, create a mask from transparentColor. For mskLasso,
 * create an "auto mask" by filling transparent color from the edges. */

#else /*not FDwAT*/

extern IFFP GetCMAP();
extern IFFP GetBODY();

#endif FDwAT

#endif ILBM_H
```

```
/*** intuall.h ************************************************************/
/*  intuall.h, Include lots of Amiga-provided header files.     1/22/86 */
/*  Plus the portability file "iff/compiler.h" which should be tailored */
/*  for your compiler.                                                  */
/*                                                                      */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                   */
/* This software is in the public domain.                               */
/*                                                                      */
/* This version for the Commodore-Amiga computer.                       */
/*                                                                      */
/************************************************************************/

#include "iff/compiler.h"       /* COMPILER-DEPENDENCIES */

/* Dummy definitions because some includes below are commented out.
 * This avoids 'undefined structure' warnings when compile.
 * This is safe as long as only use POINTERS to these structures.
 */
struct Region { int dummy; };
struct VSprite { int dummy; };
struct collTable { int dummy; };
struct CopList { int dummy; };
struct UCopList { int dummy; };
struct cprlist { int dummy; };
struct copinit { int dummy; };
struct TimeVal { int dummy; };

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"

#include "exec/tasks.h"
#include "exec/devices.h"


#include "exec/interrupts.h"

#include "exec/io.h"
#include "exec/memory.h"
#include "exec/alerts.h"

/* ALWAYS INCLUDE GFX.H before any other amiga includes */

#include "graphics/gfx.h"
/*#include "hardware/blit.h"*/

/*****
#include "graphics/collide.h"
#include "graphics/copper.h"
#include "graphics/display.h"
#include "hardware/dmabits.h"
#include "graphics/gels.h"
***/
```



```
#include "graphics/clip.h"

#include "graphics/rastport.h"
#include "graphics/view.h"
#include "graphics/gfxbase.h"
/*#include "hardware/intbits.h"*/
#include "graphics/gfxmacros.h"

#include "graphics/layers.h"

#include "graphics/text.h"
#include "graphics/sprite.h"
/*#include "hardware/custom.h"*/

/*#include "libraries/dos.h"*/
/*#include "libraries/dosextens.h"*/

#include "devices/timer.h"
#include "devices/inputevent.h"
#include "devices/keymap.h"

#include "intuition/intuition.h"

/*#include "intuitionbase.h"*/
/*#include "intuinternal.h"*/
```

```
#ifndef PACKER_H
#define PACKER_H
/*-----------------------------------------------------------------------*
 * PACKER.H  typedefs for Data-Compresser.                       1/22/86
 *
 * This module implements the run compression algorithm "cmpByteRun1"; the
 * same encoding generated by Mac's PackBits.
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *-----------------------------------------------------------------------*/

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

/* This macro computes the worst case packed size of a "row" of bytes. */
#define MaxPackedSize(rowSize)  ( (rowSize) + ( ((rowSize)+127) >> 7 ) )


#ifdef FDwAT  /* Compiler handles Function Declaration with Argument Types */

/* Given POINTERS to POINTER variables, packs one row, updating the source
 * and destination pointers. Returns the size in bytes of the packed row.
 * ASSUMES destination buffer is large enough for the packed row.
 * See MaxPackedSize. */
extern LONG PackRow(BYTE **, BYTE **, LONG);
                /*  pSource, pDest,   rowSize */

/* Given POINTERS to POINTER variables, unpacks one row, updating the source
 * and destination pointers until it produces dstBytes bytes (i.e., the
 * rowSize that went into PackRow).
 * If it would exceed the source's limit srcBytes or if a run would overrun
 * the destination buffer size dstBytes, it stops and returns TRUE.
 * Otherwise, it returns FALSE (no error). */
extern BOOL UnPackRow(BYTE **, BYTE **, WORD,     WORD);
                  /*  pSource, pDest,   srcBytes, dstBytes  */

#else /* not FDwAT */

extern LONG PackRow();
extern BOOL UnPackRow();

#endif /* FDwAT */

#endif
```



```
#ifndef PUTPICT_H
#define PUTPICT_H
/** putpict.h ************************************************************ */
/* PutPict().  Given a BitMap and a color map in RAM on the Amiga,       */
/* outputs as an ILBM.  See /iff/ilbm.h & /iff/ilbmw.c.       23-Jan-86 */
/*                                                                       */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                    */
/* This software is in the public domain.                                */
/*                                                                       */
/* This version for the Commodore-Amiga computer.                        */
/*                                                                       */
/*************************************************************************/
#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#ifndef ILBM_H
#include "iff/ilbm.h"
#endif

#ifdef FDwAT

/****** IffErr ***********************************************************/
/* Returns the iff error code and resets it to zero                      */
/*************************************************************************/
extern IFFP IffErr(void);

/****** PutPict **********************************************************/
/* Put a picture into an IFF file                                        */
/* Pass in mask == NULL for no mask.                                     */
/*                                                                       */
/* Buffer should be big enough for one packed scan line                  */
/* Buffer used as temporary storage to speed-up writing.                 */
/* A large buffer, say 8KB, is useful for minimizing Write and Seek calls. */
/* (See /iff/gio.h & /iff/gio.c).                                        */
/*************************************************************************/
extern BOOL PutPict(LONG, struct BitMap *, WORD,WORD, WORD *, BYTE *, LONG);
            /* file, bm,           pageW,pageH,colorMap, buffer,bufsize */

#else /*not FDwAT*/

extern IFFP IffErr();
extern BOOL PutPict();

#endif FDwAT

#endif PUTPICT_H
```

```
#ifndef READPICT_H
#define READPICT_H
/** ReadPict.h ***************************************************************/
/*                                                                           */
/* Read an ILBM raster image file into RAM.   1/23/86.                       */
/*                                                                           */
/* By Jerry Morrison, Steve Shaw, and Steve Hayes, Electronic Arts.          */
/* This software is in the public domain.                                    */
/*                                                                           */
/* USE THIS AS AN EXAMPLE PROGRAM FOR AN IFF READER.                         */
/*                                                                           */
/* The IFF reader portion is essentially a recursive-descent parser.         */
/*****************************************************************************/

/* ILBMFrame is our "client frame" for reading FORMs ILBM in an IFF file.
 * We allocate one of these on the stack for every LIST or FORM encountered
 * in the file and use it to hold BMHD & CMAP properties. We also allocate
 * an initial one for the whole file. */
typedef struct {
    ClientFrame clientFrame;
    UBYTE foundBMHD;
    UBYTE nColorRegs;
    BitMapHeader bmHdr;
    Color4 colorMap[32 /*1<<MaxAmDepth*/ ];
    /* If you want to read any other property chunks, e.g. GRAB or CAMG, add
     * fields to this record to store them. */
    } ILBMFrame;

/** ReadPicture() ************************************************************
 *
 * Read a picture from an IFF file, given a file handle open for reading.
 * Allocates BitMap RAM by calling (*Allocator)(size).
 *
 *****************************************************************************/

typedef UBYTE *UBytePtr;

#ifdef FDwAT

typedef UBytePtr Allocator(LONG);
    /* Allocator: a memory allocation procedure which only requires a size
     * argument. (No Amiga memory flags argument.) */

extern IFFP ReadPicture(LONG, struct BitMap *, ILBMFrame *, Allocator *);
                    /*  file, bm,             iFrame,      allocator  */
    /* iFrame is the top level "client frame". */
    /* allocator is a ptr to your allocation procedure. It must always
     *   allocate in Chip memory (for bitmap data). */

    /* PS: Notice how we used two "typedef"s above to make allocator's type
     * meaningful to humans.
     * Consider the usual C style: UBYTE *(*)(), or is it (UBYTE *)(*()) ? */

#else /* not FDwAT */

typedef UBytePtr Allocator();
```



```
extern IFFP ReadPicture();

#endif

#endif READPICT_H
```

```
/** RemAlloc.h ***************************************************************/
/*  ChipAlloc(), ExtAlloc(), RemAlloc(), RemFree().                          */
/*  ALLOCators which REMember the size allocated, for simpler freeing.       */
/*                                                                           */
/* Date      Who Changes                                                     */
/* --------- --- ---------------------------------------------------------*/
/* 16-Jan-86 sss Created from DPaint/DAlloc.c                                */
/* 22-Jan-86 jhm Include Compiler.h                                          */
/* 25-Jan-86 sss Added ChipNoClearAlloc,ExtNoClearAlloc                      */
/*                                                                           */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                        */
/* This software is in the public domain.                                    */
/*                                                                           */
/* This version for the Commodore-Amiga computer.                            */
/*                                                                           */
/*****************************************************************************/
#ifndef REM_ALLOC_H
#define REM_ALLOC_H

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif


/* How these allocators work:
 * The allocator procedures get the memory from the system allocator,
 * actually allocating 4 extra bytes. We store the length of the node in
 * the first 4 bytes then return a ptr to the rest of the storage. The
 * deallocator can then find the node size and free it. */


#ifdef FDwAT

/* RemAlloc allocates a node with "size" bytes of user data.
 * Example:
 *   struct BitMap *bm;
 *   bm = (struct BitMap *)RemAlloc( sizeof(struct BitMap), ...flags... );
 */
extern UBYTE *RemAlloc(LONG, LONG);
                    /* size, flags */

/* ALLOCator that remembers size, allocates in CHIP-accessable memory.
 * Use for all data to be displayed on screen, all sound data, all data to be
 * blitted, disk buffers, or access by any other DMA channel.
 * Does clear memory being allocated.*/
extern UBYTE *ChipAlloc(LONG);
                    /* size */

/* ChipAlloc, without clearing memory.  Purpose: speed when allocate
 * large area that will be overwritten anyway.*/
extern UBYTE *ChipNoClearAlloc(LONG);

/* ALLOCator that remembers size, allocates in extended memory.
 * Does clear memory being allocated.
 * NOTICE: does NOT declare "MEMF_FAST".  This allows machines
 * lacking extended memory to allocate within chip memory,
```



```
 * assuming there is enough memory left.*/
extern UBYTE *ExtAlloc(LONG);
                    /* size */

/* ExtAlloc, without clearing memory.  Purpose: speed when allocate
 * large area that will be overwritten anyway.*/
extern UBYTE *ExtNoClearAlloc(LONG);


/* FREEs either chip or extended memory, if allocated with an allocator
 * which REMembers size allocated.
 * Safe: won't attempt to de-allocate a NULL pointer.
 * Returns NULL so caller can do
 *   p = RemFree(p);
 */
extern UBYTE *RemFree(UBYTE *);
                   /*  p  */

#else /* not FDwAT */

extern UBYTE *RemAlloc();
extern UBYTE *ChipAlloc();
extern UBYTE *ExtAlloc();
extern UBYTE *RemFree();

#endif /* FDwAT */

#endif REM_ALLOC_H
```

```
/*--------------------------------------------------------------*/
/*                                                              */
/*                        bmprintc.c                            */
/*                                                              */
/* print out a C-language representation of data for bitmap     */
/*                                                              */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.           */
/* This software is in the public domain.                       */
/*                                                              */
/* This version for the Commodore-Amiga computer.               */
/*                                                              */
/*--------------------------------------------------------------*/

#include <iff/intuall.h>
#undef NULL
#include <lattice/stdio.h>

#define NO 0
#define YES 1

static BOOL doCRLF;

PrCRLF(fp) FILE *fp; {
    if (doCRLF) fprintf(fp,"%c%c",0xD,0xA); else fprintf(fp,"\n");
    }

PrintBob(bm, fp, name)
    struct BitMap *bm;
    FILE *fp;
    UBYTE *name;
    {
    UWORD *wp;

    int p,j,nb;
    int nwords = (bm->BytesPerRow/2)*bm->Rows;

    fprintf(fp,"/*----- bitmap : w = %ld, h = %ld ------ */",
       bm->BytesPerRow*8, bm->Rows);

    PrCRLF(fp);

    for (p=0; p<bm->Depth; ++p) {
        wp = (UWORD *)bm->Planes[p];
        fprintf(fp, "/*------ plane # %ld: --------*/",p);
        PrCRLF(fp);
        fprintf(fp, "UWORD %s%c[%ld] = { ", name, (p?('0'+p):' '), nwords);
        PrCRLF(fp);
        for (j = 0 ;  ; j++) {
            for (nb = 0; ; ) {
                fprintf(fp,"  0x%lx", *wp++);
                nb += 2;
                if (nb == bm->BytesPerRow) {
                    if (j == bm->Rows-1)  goto endplane;
                    else { fprintf(fp,", "); PrCRLF(fp);  break; }
                    }
                else fprintf(fp,", ");
```



```
                }
            }
        endplane: fprintf(fp," };");
        PrCRLF(fp); PrCRLF(fp);
        }
    }

PSprite(bm, fp,name,p,dohead)
    struct BitMap *bm;
    FILE *fp;
    UBYTE *name;
    int p;
    BOOL dohead;
    {
    UWORD *wp0,*wp1;
    int j,nwords;
    int wp1 = bm->BytesPerRow/2;
    nwords =  2*bm->Rows + (dohead?4:0);
    wp0 = (UWORD *)bm->Planes[p];
    wp1 = (UWORD *)bm->Planes[p+1];
    fprintf(fp,"UWORD %s[%ld] = {", name, nwords);
    PrCRLF(fp);
    if (dohead){
        fprintf(fp,"  0x0000, 0x0000, /* VStart, VStop */");
        PrCRLF(fp);
        }
    for (j=0 ; j<bm->Rows; j++) {
        fprintf(fp,"  0x%lx, 0x%lx", *wp0, *wp1);
        if (dohead || (j!=bm->Rows-1)) {
                fprintf(fp, ",");
                PrCRLF(fp);
                }
        wp0 += wp1;
        wp1 += wp1;
        }
    if (dohead) fprintf(fp,"  0x0000, 0x0000 }; /* End of Sprite */");
    else fprintf(fp," };");
    PrCRLF(fp); PrCRLF(fp);
    }

static UBYTE one[] = "1";

PrintSprite(bm, fp, name, attach, dohdr)
    struct BitMap *bm; FILE *fp;
    UBYTE *name;
    BOOL attach, dohdr;
    {
    fprintf(fp,"/*----- Sprite format: h = %ld ------ */", bm->Rows);
    PrCRLF(fp);
    if (bm->Depth>1) {
        fprintf(fp, "/*--Sprite containing lower order two planes:   */");
        PrCRLF(fp);
        PSprite(bm,fp,name,0,dohdr);
        }
    if (attach && (bm->Depth > 3) ) {
        strcat(name,one);
```

```
        fprintf(fp, "/*--Sprite containing higher order two planes:   */");
        PrCRLF(fp);
        PSprite(bm,fp,name,2,dohdr);
        }
    }

#define BOB 0
#define SPRITE 1

BMPrintCRep(bm, fp, name, fmt)
    struct BitMap *bm;  /* Contains the image data */
    FILE *fp;           /* file we will write to */
    UBYTE *name;        /* name associated with the bitmap */
    UBYTE *fmt;         /* string of characters describing output fmt*/
    {
    BOOL attach, doHdr;
    char c;
    SHORT type;
    doCRLF = NO;
    doHdr = YES;
    type = BOB;
    attach = NO;
    while ( (c=*fmt++) != 0 )
        switch (c) {
            case 'b': type = BOB; break;
            case 's': type = SPRITE; attach = NO; break;
            case 'a': type = SPRITE; attach = YES; break;
            case 'n': doHdr = NO; break;
            case 'c': doCRLF = YES; break;
            }
    switch(type) {
        case BOB: PrintBob(bm,fp,name); break;
        case SPRITE:  PrintSprite(bm,fp,name,attach,doHdr); break;
        }
    }
```



```
/*-------------------------------------------------------------------------*/
/* GIO.C  Generic I/O Speed Up Package                          1/23/86    */
/* See GIOCall.C for an example of usage.                                  */
/* Read not speeded-up yet.  Only one Write file buffered at a time.       */
/* Note: The speed-up provided is ONLY significant for code such as IFF    */
/* which does numerous small Writes and Seeks.                             */
/*                                                                         */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                      */
/* This software is in the public domain.                                  */
/*                                                                         */
/* This version for the Commodore-Amiga computer.                          */
/*                                                                         */
/*-------------------------------------------------------------------------*/
#include "iff/gio.h"    /* See comments here for explanation.*/

#if GIO_ACTIVE

#define local static


local BPTR wFile      = NULL;
local BYTE *wBuffer   = NULL;
local LONG wNBytes    = 0; /* buffer size in bytes.*/
local LONG wIndex     = 0; /* index of next available byte.*/
local LONG wWaterline = 0; /* Count of # bytes to be written.
                            * Different than wIndex because of GSeek.*/

/*----------- GOpen ----------------------------------------------------*/
LONG GOpen(filename, openmode)   char *filename;  LONG openmode; {
    return( Open(filename, openmode) );
    }

/*----------- GClose ---------------------------------------------------*/
LONG GClose(file)  BPTR file; {
    LONG signal = 0, signal2;
    if (file == wFile)
        signal = GWriteUndeclare(file);
    signal2 = Close(file);      /* Call Close even if trouble with write.*/
    if (signal2 < 0)
        signal = signal2;
    return( signal );
    }

/*----------- GRead ----------------------------------------------------*/
LONG GRead(file, buffer, nBytes)   BPTR file;  BYTE *buffer;  LONG nBytes; {
    LONG signal = 0;
    /* We don't yet read directly from the buffer, so flush it to disk and
     * let the DOS fetch it back. */
    if (file == wFile)
        signal = GWriteFlush(file);
    if (signal >= 0)
        signal = Read(file, buffer, nBytes);
    return( signal );
    }

/* ---------- GWriteFlush ----------------------------------------------*/
```

```
LONG GWriteFlush(file)  BPTR file; {
    LONG gWrite = 0;
    if (wFile != NULL  &&  wBuffer != NULL  &&  wIndex > 0)
        gWrite = Write(wFile, wBuffer, wWaterline);
    wWaterline = wIndex = 0;    /* No matter what, make sure this happens.*/
    return( gWrite );
    }

/* ---------- GWriteDeclare ----------------------------------------*/
LONG GWriteDeclare(file, buffer, nBytes)
    BPTR file;  BYTE *buffer;  LONG nBytes; {
    LONG gWrite = GWriteFlush(wFile);  /* Finish any existing usage.*/
    if ( file==NULL  ||  (file==wFile  &&  buffer==NULL)  ||  nBytes<=3) {
        wFile = NULL;   wBuffer = NULL;     wNBytes = 0; }
    else {
        wFile = file;   wBuffer = buffer;   wNBytes = nBytes; }
    return( gWrite );
    }

/* ---------- GWrite -----------------------------------------------*/
LONG GWrite(file, buffer, nBytes)   BPTR file;  BYTE *buffer;  LONG nBytes; {
    LONG gWrite = 0;

    if (file == wFile  &&  wBuffer != NULL) {
        if (wNBytes >= wIndex + nBytes) {
            /* Append to wBuffer.*/
            movmem(buffer, wBuffer+wIndex, nBytes);
            wIndex += nBytes;
            if (wIndex > wWaterline)
                wWaterline = wIndex;
            nBytes = 0;         /* Indicate data has been swallowed.*/
            }
        else {
            wWaterline = wIndex;     /* We are about to overwrite any
                * data above wIndex, up to at least the buffer end.*/
            gWrite = GWriteFlush(file);  /* Write data out in proper order.*/
            }
        }
    if (nBytes > 0  &&  gWrite >= 0)
        gWrite += Write(file, buffer, nBytes);
    return( gWrite );
    }

/* ---------- GSeek ------------------------------------------------*/
LONG GSeek(file, position, mode)
    BPTR file;   LONG position;   LONG mode; {
    LONG gSeek = -2;
    LONG newWIndex = wIndex + position;

    if (file == wFile  &&  wBuffer != NULL) {
        if (mode == OFFSET_CURRENT  &&
            newWIndex >= 0  &&  newWIndex <= wWaterline) {
            gSeek = wIndex;     /* Okay; return *OLD* position */
            wIndex = newWIndex;
            }
        else {
```



```
            /* We don't even try to optimize the other cases.*/
            gSeek = GWriteFlush(file);
            if (gSeek >= 0)    gSeek = -2;  /* OK so far */
            }
        }
    if (gSeek == -2)
        gSeek = Seek(file, position, mode);
    return( gSeek );
    }

#else /* not GIO_ACTIVE */

void GIODummy() { }     /* to keep the compiler happy */

#endif GIO_ACTIVE
```

```
/*-------------------------------------------------------------------*/
/*  GIOCall.c:  An example of calling the Generic I/O Speed-up.      */
/*                  1/23/86                                          */
/*                                                                   */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                */
/* This software is in the public domain.                            */
/*                                                                   */
/* This version for the Commodore-Amiga computer.                    */
/*                                                                   */
/*-------------------------------------------------------------------*/
    main(...) {
    LONG file;
    int success;
    ...
    success = (0 != (file = GOpen(...)));
    /* A TmpRas is a good buffer to use for a variety of short-term uses.*/
    if (success)
        success = PutObject(file, ob, tmpRas.RasPtr, tmpRas.Size);
    success &= (0 <= GClose(file));
    }

/*----- PutObject writes a DVCS object out as a disk file.-----*/
BOOL PutObject(file, ob, buffer, bufsize)
    LONG file;  struct Object *ob;  BYTE *buffer;  LONG bufsize; {
    int success = TRUE;

    if (bufsize > 2*BODY_BUFSIZE) {
        /* Give buffer to speed-up writing.*/
        GWriteDeclare(file, buffer+BODY_BUFSIZE, bufsize-BODY_BUFSIZE);
        bufsize = BODY_BUFSIZE;  /* Used by PutObject for other purposes.*/
        }
    ...
        /* Use GWrite and GSeek instead of Write and Seek.*/
    success &= (0 <= GWrite(file, address, length));
    ...
    success &= (0 <= GWriteUndeclare(file));
                /* Release the speed-up buffer.*/
                /* This is not necessary if GClose is used to close the file,
                 * but it can't hurt.*/
    return( (BOOL)success );
    }
```



```
/*-------------------------------------------------------------------- */
/* IFFCheck.C  Print out the structure of an IFF-85 file,      1/23/86  */
/* checking for structural errors.                                      */
/*                                                                      */
/* DO NOT USE THIS AS A SKELETAL PROGRAM FOR AN IFF READER!             */
/* See ShowILBM.C for a skeletal example.                               */
/*                                                                      */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                   */
/* This software is in the public domain.                               */
/*                                                                      */
/* This version for the Commodore-Amiga computer.                       */
/*                                                                      */
/*----------------------------------------------------------------------*/
#include "iff/iff.h"


/* ---------- IFFCheck ------------------------------------------------*/
/* [TBD] More extensive checking could be done on the IDs encountered in the
 * file. Check that the reserved IDs "FOR1".."FOR9", "LIS1".."LIS9", and
 * "CAT1".."CAT9" aren't used. Check that reserved IDs aren't used as Form
 * types. Check that all IDs are made of 4 printable characters (trailing
 * spaces ok). */

typedef struct {
    ClientFrame clientFrame;
    int levels;         /* # groups currently nested within.*/
    } Frame;

char MsgOkay[] = { "----- (IFF_OKAY) A good IFF file." };
char MsgEndMark[] = {"----- (END_MARK) How did you get this message??" };
char MsgDone[] = { "----- (IFF_DONE) How did you get this message??" };
char MsgDos[] = { "----- (DOS_ERROR) The DOS gave back an error." };
char MsgNot[] = { "----- (NOT_IFF) not an IFF file." };
char MsgNoFile[] = { "----- (NO_FILE) no such file found." };
char MsgClientError[] = {"----- (CLIENT_ERROR) IFF Checker bug."};
char MsgForm[] = { "----- (BAD_FORM) How did you get this message??" };
char MsgShort[] = { "----- (SHORT_CHUNK) How did you get this message??" };
char MsgBad[] = { "----- (BAD_IFF) a mangled IFF file." };

/* MUST GET THESE IN RIGHT ORDER!!*/
char *IFFPMessages[-(int)LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };

/* FORWARD REFERENCES */
extern IFFP GetList(GroupContext *);
extern IFFP GetForm(GroupContext *);
```

```
extern IFFP GetProp(GroupContext *);
extern IFFP GetCat (GroupContext *);

void IFFCheck(name)  char *name; {
    IFFP iffp;
    BPTR file = Open(name, MODE_OLDFILE);
    Frame frame;

    frame.levels = 0;
    frame.clientFrame.getList = GetList;
    frame.clientFrame.getForm = GetForm;
    frame.clientFrame.getProp = GetProp;
    frame.clientFrame.getCat  = GetCat ;

    printf("----- Checking file '%s' -----\n", name);
    if (file == 0)
        iffp = NO_FILE;
    else
        iffp = ReadIFF(file, (ClientFrame *)&frame);

    Close(file);
    printf("%s\n", IFFPMessages[-iffp]);
    }

main(argc, argv)   int argc;  char **argv; {
    if (argc != 1+1) {
        printf("Usage: 'iffcheck filename'\n");
        exit(0);
        }
    IFFCheck(argv[1]);
    }

/* ---------- Put... ----------------------------------------------------*/

PutLevels(count)   int count; {
    for ( ;  count > 0;  --count) {
        printf(".");
        }
    }

PutID(id)  ID id; {
    printf("%c%c%c%c",
           (char)((id>>24L) & 0x7f),
           (char)((id>>16L) & 0x7f),
           (char)((id>>8)   & 0x7f),
           (char)(id        & 0x7f) );
    }

PutN(n)   int n; {
    printf(" %d ", n);
    }

/* Put something like "...BMHD 14" or "...LIST 14 PLBM". */
PutHdr(context)  GroupContext *context; {
    PutLevels( ((Frame *)context->clientFrame)->levels );
    PutID(context->ckHdr.ckID);
```



```
    PutN(context->ckHdr.ckSize);

    if (context->subtype != NULL_CHUNK)
        PutID(context->subtype);

    printf("\n");
    }

/* ---------- AtLeaf ----------------------------------------------------*/

/* At Leaf chunk.  That is, a chunk which does NOT contain other chunks.
 * Print "ID size".*/
IFFP AtLeaf(context)  GroupContext *context; {

    PutHdr(context);
    /* A typical reader would read the chunk's contents, using the "Frame"
     * for local data, esp. shared property settings (PROP).*/
    /* IFFReadBytes(context, ...buffer, context->ckHdr->ckSize); */
    return(IFF_OKAY);
    }

/* ---------- GetList ---------------------------------------------------*/
/* Handle a LIST chunk.  Print "LIST size subTypeID".
 * Then dive into it.*/
IFFP GetList(parent)  GroupContext *parent; {
    Frame newFrame;

    newFrame = *(Frame *)parent->clientFrame;  /* copy parent's frame*/
    newFrame.levels++;

    PutHdr(parent);

    return( ReadIList(parent, (ClientFrame *)&newFrame) );
    }

/* ---------- GetForm ---------------------------------------------------*/
/* Handle a FORM chunk.  Print "FORM size subTypeID".
 * Then dive into it.*/
IFFP GetForm(parent)   GroupContext *parent; {
    /*CompilerBug register*/ IFFP iffp;
    GroupContext new;
    Frame newFrame;

    newFrame = *(Frame *)parent->clientFrame;  /* copy parent's frame*/
    newFrame.levels++;

    PutHdr(parent);

    iffp = OpenRGroup(parent, &new);
    CheckIFFP();
    new.clientFrame = (ClientFrame *)&newFrame;

    /* FORM reader for Checker. */
    /* LIST, FORM, PROP, CAT already handled by GetF1ChunkHdr. */
    do {if ( (iffp = GetF1ChunkHdr(&new)) > 0 )
            iffp = AtLeaf(&new);
```

```
        } while (iffp >= IFF_OKAY);

    CloseRGroup(&new);
    return(iffp == END_MARK ? IFF_OKAY : iffp);
    }

/* ---------- GetProp ----------------------------------------------------*/
/* Handle a PROP chunk.  Print "PROP size subTypeID".
 * Then dive into it.*/
IFFP GetProp(listContext)  GroupContext *listContext; {
    /*CompilerBug register*/ IFFP iffp;
    GroupContext new;

    PutHdr(listContext);

    iffp = OpenRGroup(listContext, &new);
    CheckIFFP();

    /* PROP reader for Checker. */
    ((Frame *)listContext->clientFrame)->levels++;

    do {if ( (iffp = GetPChunkHdr(&new)) > 0 )
            iffp = AtLeaf(&new);
        } while (iffp >= IFF_OKAY);

    ((Frame *)listContext->clientFrame)->levels--;

    CloseRGroup(&new);
    return(iffp == END_MARK ? IFF_OKAY : iffp);
    }

/* ---------- GetCat -----------------------------------------------------*/
/* Handle a CAT chunk.  Print "CAT size subTypeID".
 * Then dive into it.*/
IFFP GetCat(parent)  GroupContext *parent;  {
    IFFP iffp;

    ((Frame *)parent->clientFrame)->levels++;

    PutHdr(parent);

    iffp = ReadICat(parent);

    ((Frame *)parent->clientFrame)->levels--;
    return(iffp);
    }
```



```
/*-------------------------------------------------------------------------*
 * IFFR.C  Support routines for reading IFF-85 files.          1/23/86
 * (IFF is Interchange Format File.)
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *
 * Uses "gio".  Either link with gio.c, or set the GIO_ACTIVE flag to 0
 * in gio.h.
 *-------------------------------------------------------------------------*/
#include "iff/gio.h"
#include "iff/iff.h"

/* ----- Private subroutine FileLength() ---------------------------------*/
/* Returns the length of the file or else a negative IFFP error code
 * (NO_FILE or DOS_ERROR).  AmigaDOS-specific implementation.
 * SIDE EFFECT: Thanks to AmigaDOS, we have to change the file's position
 * to find its length.
 * Now if Amiga DOS maintained fh_End, we'd just do this:
 *    fileLength = (FileHandle *)BADDR(file)->fh_End; */
LONG FileLength(file)  BPTR file;  {
    LONG fileLength = NO_FILE;

    if (file > 0)  {
        GSeek(file, 0, OFFSET_END);     /* Seek to end of file.*/
        fileLength = GSeek(file, 0, OFFSET_CURRENT);
            /* Returns position BEFORE the seek, which is #bytes in file. */
        if (fileLength < 0)
            fileLength = DOS_ERROR;     /* DOS being absurd.*/
        }

    return(fileLength);
    }

/* ---------- Read -------------------------------------------------------*/

/* ---------- OpenRIFF ---------------------------------------------------*/
IFFP OpenRIFF(file0, new0, clientFrame)
        BPTR file0;  GroupContext *new0;  ClientFrame *clientFrame; {
    register BPTR file = file0;
    register GroupContext *new = new0;
    IFFP iffp = IFF_OKAY;

    new->parent       = NULL;           /* "whole file" has no parent.*/
    new->clientFrame  = clientFrame;
    new->file         = file;
    new->position     = 0;
    new->ckHdr.ckID   = new->subtype    = NULL_CHUNK;
    new->ckHdr.ckSize = new->bytesSoFar = 0;

    /* Set new->bound and go to the file's beginning. */
    new->bound = FileLength(file);
    if (new->bound < 0)
        iffp = new->bound;                  /* File system error! */
```

```
    else if ( new->bound < sizeof(ChunkHeader) )
        iffp = NOT_IFF;                     /* Too small for an IFF file. */
    else
        GSeek(file, 0, OFFSET_BEGINNING);  /* Go to file start. */

    return(iffp);
    }

/* ---------- OpenRGroup ------------------------------------------------*/
IFFP OpenRGroup(parent0, new0)   GroupContext *parent0, *new0; {
    register GroupContext *parent = parent0;
    register GroupContext *new    = new0;
    IFFP iffp = IFF_OKAY;

    new->parent       = parent;
    new->clientFrame  = parent->clientFrame;
    new->file         = parent->file;
    new->position     = parent->position;
    new->bound        = parent->position + ChunkMoreBytes(parent);
    new->ckHdr.ckID   = new->subtype    = NULL_CHUNK;
    new->ckHdr.ckSize = new->bytesSoFar = 0;

    if ( new->bound > parent->bound  ||  IS_ODD(new->bound) )
        iffp = BAD_IFF;
    return(iffp);
    }

/* ---------- CloseRGroup -----------------------------------------------*/
IFFP CloseRGroup(context)   GroupContext *context; {
    register LONG position;

    if (context->parent == NULL) {
        } /* Context for whole file.*/
    else {
        position = context->position;
        context->parent->bytesSoFar += position - context->parent->position;
        context->parent->position = position;
        }
    return(IFF_OKAY);
    }

/* ---------- SkipFwd ---------------------------------------------------*/
/* Skip over bytes in a context. Won't go backwards.*/
/* Updates context->position but not context->bytesSoFar.*/
/* This implementation is AmigaDOS specific.*/
IFFP SkipFwd(context, bytes)   GroupContext *context;  LONG bytes; {
    IFFP iffp = IFF_OKAY;

    if (bytes > 0) {
        if (-1 == GSeek(context->file, bytes, OFFSET_CURRENT))
            iffp = BAD_IFF;     /* Ran out of bytes before chunk complete.*/
        else
            context->position += bytes;
        }
    return(iffp);
    }
```



```
/* ---------- GetChunkHdr -----------------------------------------------*/
ID GetChunkHdr(context0)   GroupContext *context0;  {
    register GroupContext *context = context0;
    register IFFP iffp;
    LONG remaining;

    /* Skip remainder of previous chunk & padding. */
    iffp = SkipFwd(context,
        ChunkMoreBytes(context) + IS_ODD(context->ckHdr.ckSize));
    CheckIFFP();

    /* Set up to read the new header. */
    context->ckHdr.ckID = BAD_IFF;      /* Until we know it's okay, mark it BAD.*/
    context->subtype    = NULL_CHUNK;
    context->bytesSoFar = 0;

    /* Generate a psuedo-chunk if at end-of-context. */
    remaining = context->bound - context->position;
    if (remaining == 0) {
        context->ckHdr.ckSize = 0;
        context->ckHdr.ckID   = END_MARK;
        }

    /* BAD_IFF if not enough bytes in the context for a ChunkHeader.*/
    else if (sizeof(ChunkHeader) > remaining) {
        context->ckHdr.ckSize = remaining;
        }

    /* Read the chunk header (finally). */
    else {
        switch (
            GRead(context->file, (BYTE *)&context->ckHdr, sizeof(ChunkHeader))
            ) {
            case -1: return(context->ckHdr.ckID = DOS_ERROR);
            case 0:  return(context->ckHdr.ckID = BAD_IFF);
            }

        /* Check: Top level chunk must be LIST or FORM or CAT. */
        if (context->parent == NULL)
            switch(context->ckHdr.ckID) {
                case FORM:  case LIST:  case CAT:  break;
                default:    return(context->ckHdr.ckID = NOT_IFF);
                }

        /* Update the context. */
        context->position += sizeof(ChunkHeader);
        remaining         -= sizeof(ChunkHeader);

        /* Non-positive ID values are illegal and used for error codes.*/
        /* We could check for other illegal IDs...*/
        if (context->ckHdr.ckID <= 0)
            context->ckHdr.ckID = BAD_IFF;

        /* Check: ckSize negative or larger than # bytes left in context? */
        else if (context->ckHdr.ckSize < 0  ||
```

```
                context->ckHdr.ckSize > remaining) {
            context->ckHdr.ckSize = remaining;
            context->ckHdr.ckID   = BAD_IFF;
            }

        /* Automatically read the LIST, FORM, PROP, or CAT subtype ID */
        else switch (context->ckHdr.ckID) {
            case LIST:  case FORM:  case PROP:  case CAT:  {
                iffp = IFFReadBytes(context,
                                    (BYTE *)&context->subtype,
                                    sizeof(ID));
                if (iffp != IFF_OKAY)
                    context->ckHdr.ckID = iffp;
                break; }
            }

        }
    return(context->ckHdr.ckID);
    }

/* ---------- IFFReadBytes ------------------------------------------------*/
IFFP IFFReadBytes(context, buffer, nBytes)
    GroupContext *context;   BYTE *buffer;   LONG nBytes; {
    register IFFP iffp = IFF_OKAY;

    if (nBytes < 0)
        iffp = CLIENT_ERROR;
    else if (nBytes > ChunkMoreBytes(context))
        iffp = SHORT_CHUNK;
    else if (nBytes > 0)
        switch ( GRead(context->file, buffer, nBytes) ) {
            case -1: {iffp = DOS_ERROR; break; }
            case 0:  {iffp = BAD_IFF;   break; }
            default: {
                context->position   += nBytes;
                context->bytesSoFar += nBytes;
                }
            }

    return(iffp);
    }

/* ---------- SkipGroup ---------------------------------------------------*/
IFFP SkipGroup(context)  GroupContext *context;  {
    }   /* Nothing to do, thanks to GetChunkHdr */

/* ---------- ReadIFF -----------------------------------------------------*/
IFFP ReadIFF(file, clientFrame)  BPTR file;  ClientFrame *clientFrame;  {
    /*CompilerBug register*/ IFFP iffp;
    GroupContext context;

    iffp = OpenRIFF(file, &context);
    context.clientFrame = clientFrame;

    if (iffp == IFF_OKAY)
        switch (iffp = GetChunkHdr(&context)) {
```



```
            case FORM: { iffp = (*clientFrame->getForm)(&context); break; }
            case LIST: { iffp = (*clientFrame->getList)(&context); break; }
            case CAT : { iffp = (*clientFrame->getCat )(&context); break; }
            /* default: Includes IFF_DONE, BAD_IFF, NOT_IFF... */
            }

    CloseRGroup(&context);

    if (iffp > 0)               /* Make sure we don't return an ID.*/
        iffp = NOT_IFF;         /* GetChunkHdr should've caught this.*/
    return(iffp);
    }

/* ---------- ReadIList ---------------------------------------------------*/
IFFP ReadIList(parent, clientFrame)
    GroupContext *parent;  ClientFrame *clientFrame; {
    GroupContext listContext;
    IFFP iffp;
    BOOL propOk = TRUE;

    iffp = OpenRGroup(parent, &listContext);
    CheckIFFP();

    /* One special case test lets us handle CATs as well as LISTs.*/
    if (parent->ckHdr.ckID == CAT)
        propOk = FALSE;
    else
        listContext.clientFrame = clientFrame;

    do {
        switch (iffp = GetChunkHdr(&listContext)) {
            case PROP: {
                if (propOk)
                    iffp = (*clientFrame->getProp)(&listContext);
                else
                    iffp = BAD_IFF;
                break;
                }
            case FORM: { iffp = (*clientFrame->getForm)(&listContext); break; }
            case LIST: { iffp = (*clientFrame->getList)(&listContext); break; }
            case CAT : { iffp = (*clientFrame->getCat )(&listContext); break; }
            /* default: Includes END_MARK, IFF_DONE, BAD_IFF, NOT_IFF... */
            }
        if (listContext.ckHdr.ckID != PROP)
            propOk = FALSE;     /* No PROPs allowed after this point.*/
        } while (iffp == IFF_OKAY);

    CloseRGroup(&listContext);

    if (iffp > 0)       /* Only chunk types above are allowed in a LIST/CAT.*/
        iffp = BAD_IFF;
    return(iffp == END_MARK ? IFF_OKAY : iffp);
    }

/* ---------- ReadICat ----------------------------------------------------*/
/* By special arrangement with the ReadIList implement'n, this is trivial.*/
```

```
IFFP ReadICat(parent)  GroupContext *parent;  {
    return( ReadIList(parent, NULL) );
    }

/* ---------- GetFChunkHdr ----------------------------------------------*/
ID GetFChunkHdr(context)   GroupContext *context; {
    register ID id;

    id = GetChunkHdr(context);
    if (id == PROP)
        context->ckHdr.ckID = id = BAD_IFF;
    return(id);
    }

/* ---------- GetF1ChunkHdr ---------------------------------------------*/
ID GetF1ChunkHdr(context)   GroupContext *context; {
    register ID id;
    register ClientFrame *clientFrame = context->clientFrame;

    switch (id = GetChunkHdr(context))  {
        case PROP: { id = BAD_IFF; break; }
        case FORM: { id = (*clientFrame->getForm)(context); break; }
        case LIST: { id = (*clientFrame->getList)(context); break; }
        case CAT : { id = (*clientFrame->getCat )(context); break; }
        /* Default: let the caller handle other chunks */
        }
    return(context->ckHdr.ckID = id);
    }

/* ---------- GetPChunkHdr ----------------------------------------------*/
ID GetPChunkHdr(context)   GroupContext *context; {
    register ID id;

    id = GetChunkHdr(context);
    switch (id) {
        case LIST:  case FORM:  case PROP:  case CAT:  {
            id = context->ckHdr.ckID = BAD_IFF;
            break; }
        }
    return(id);
    }
```



```
/*-------------------------------------------------------------------------*
 * IFFW.C  Support routines for writing IFF-85 files.          1/23/86
 * (IFF is Interchange Format File.)
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *-------------------------------------------------------------------------*/
#include "iff/iff.h"
#include "iff/gio.h"

/* ---------- IFF Writer ------------------------------------------------*/

/* A macro to test if a chunk size is definite, i.e. not szNotYetKnown.*/
#define Known(size)    ( (size) != szNotYetKnown )

/* Yet another weird macro to make the source code simpler...*/
#define IfIffp(expr)  {if (iffp == IFF_OKAY)  iffp = (expr);}

/* ---------- OpenWIFF --------------------------------------------------*/

IFFP OpenWIFF(file, new0, limit)  BPTR file; GroupContext *new0; LONG limit; {
    register GroupContext *new = new0;
    register IFFP iffp = IFF_OKAY;

    new->parent       = NULL;
    new->clientFrame  = NULL;
    new->file         = file;
    new->position     = 0;
    new->bound        = limit;
    new->ckHdr.ckID   = NULL_CHUNK;  /* indicates no current chunk */
    new->ckHdr.ckSize = new->bytesSoFar = 0;

    if (0 > Seek(file, 0, OFFSET_BEGINNING))    /* Go to start of the file.*/
        iffp = DOS_ERROR;
    else if ( Known(limit) && IS_ODD(limit) )
        iffp = CLIENT_ERROR;
    return(iffp);
    }

/* ---------- StartWGroup -----------------------------------------------*/
IFFP StartWGroup(parent, groupType, groupSize, subtype, new)
      GroupContext *parent, *new; ID groupType, subtype; LONG groupSize;  {
    register IFFP iffp;

    iffp = PutCkHdr(parent, groupType, groupSize);
    IfIffp( IFFWriteBytes(parent, (BYTE *)&subtype, sizeof(ID)) );
    IfIffp( OpenWGroup(parent, new) );
    return(iffp);
    }

/* ---------- OpenWGroup ------------------------------------------------*/
IFFP OpenWGroup(parent0, new0)  GroupContext *parent0, *new0; {
    register GroupContext *parent = parent0;
    register GroupContext *new    = new0;
```

```
    register LONG ckEnd;
    register IFFP iffp = IFF_OKAY;

    new->parent       = parent;
    new->clientFrame  = parent->clientFrame;
    new->file         = parent->file;
    new->position     = parent->position;
    new->bound        = parent->bound;
    new->ckHdr.ckID   = NULL_CHUNK;
    new->ckHdr.ckSize = new->bytesSoFar = 0;

    if ( Known(parent->ckHdr.ckSize) ) {
        ckEnd = new->position + ChunkMoreBytes(parent);
        if ( new->bound == szNotYetKnown || new->bound > ckEnd )
            new->bound = ckEnd;
        };

    if ( parent->ckHdr.ckID == NULL_CHUNK || /* not currently writing a chunk*/
         IS_ODD(new->position) ||
         (Known(new->bound) && IS_ODD(new->bound)) )
        iffp = CLIENT_ERROR;
    return(iffp);
    }

/* ---------- CloseWGroup ----------------------------------------------*/
IFFP CloseWGroup(old0)  GroupContext *old0; {
    register GroupContext *old = old0;
    IFFP iffp = IFF_OKAY;

    if ( old->ckHdr.ckID != NULL_CHUNK )  /* didn't close the last chunk */
        iffp = CLIENT_ERROR;
    else if ( old->parent == NULL ) {     /* top level file context */
        if (GWriteFlush(old->file) < 0)  iffp = DOS_ERROR;
        }
    else {                                /* update parent context */
        old->parent->bytesSoFar += old->position - old->parent->position;
        old->parent->position = old->position;
        };
    return(iffp);
    }

/* ---------- EndWGroup ------------------------------------------------*/
IFFP EndWGroup(old)  GroupContext *old;  {
    register GroupContext *parent = old->parent;
    register IFFP iffp;

    iffp = CloseWGroup(old);
    IfIffp( PutCkEnd(parent) );
    return(iffp);
    }

/* ---------- PutCk ----------------------------------------------------*/
IFFP PutCk(context, ckID, ckSize, data)
      GroupContext *context; ID ckID; LONG ckSize; BYTE *data; {
    register IFFP iffp = IFF_OKAY;
```



```
    if ( ckSize == szNotYetKnown )
        iffp = CLIENT_ERROR;
    IfIffp( PutCkHdr(context, ckID, ckSize) );
    IfIffp( IFFWriteBytes(context, data, ckSize) );
    IfIffp( PutCkEnd(context) );
    return(iffp);
    }

/* ---------- PutCkHdr -------------------------------------------------*/
IFFP PutCkHdr(context0, ckID, ckSize)
      GroupContext *context0;  ID ckID;  LONG ckSize; {
    register GroupContext *context = context0;
    LONG minPSize = sizeof(ChunkHeader); /* physical chunk >= minPSize bytes*/

    /* CLIENT_ERROR if we're already inside a chunk or asked to write
     * other than one FORM, LIST, or CAT at the top level of a file */
    /* Also, non-positive ID values are illegal and used for error codes.*/
    /* (We could check for other illegal IDs...)*/
    if ( context->ckHdr.ckID != NULL_CHUNK  ||  ckID <= 0 )
        return(CLIENT_ERROR);
    else if (context->parent == NULL)  {
        switch (ckID)  {
            case FORM:  case LIST:  case CAT:  break;
            default: return(CLIENT_ERROR);
            }
        if (context->position != 0)
            return(CLIENT_ERROR);
        }

    if ( Known(ckSize) ) {
        if ( ckSize < 0 )
            return(CLIENT_ERROR);
        minPSize += ckSize;
        };
    if ( Known(context->bound)  &&
         context->position + minPSize > context->bound )
        return(CLIENT_ERROR);

    context->ckHdr.ckID   = ckID;
    context->ckHdr.ckSize = ckSize;
    context->bytesSoFar   = 0;
    if (0 >
        GWrite(context->file, (BYTE *)&context->ckHdr, sizeof(ChunkHeader))
        )
        return(DOS_ERROR);
    context->position += sizeof(ChunkHeader);
    return(IFF_OKAY);
    }

/* ---------- IFFWriteBytes --------------------------------------------*/
IFFP IFFWriteBytes(context0, data, nBytes)
     GroupContext *context0;  BYTE *data;  LONG nBytes; {
    register GroupContext *context = context0;

    if ( context->ckHdr.ckID == NULL_CHUNK  ||  /* not in a chunk */
         nBytes < 0  ||                         /* negative nBytes */
```

```
        (Known(context->bound)  &&              /* overflow context */
          context->position + nBytes > context->bound)  ||
        (Known(context->ckHdr.ckSize)  &&       /* overflow chunk */
          context->bytesSoFar + nBytes > context->ckHdr.ckSize) )
      return(CLIENT_ERROR);

    if (0 > GWrite(context->file, data, nBytes))
      return(DOS_ERROR);

    context->bytesSoFar += nBytes;
    context->position   += nBytes;
    return(IFF_OKAY);
    }

/* ---------- PutCkEnd ----------------------------------------------------*/
IFFP PutCkEnd(context0)  GroupContext *context0; {
    register GroupContext *context = context0;
    WORD zero = 0;      /* padding source */

    if ( context->ckHdr.ckID == NULL_CHUNK )  /* not in a chunk */
      return(CLIENT_ERROR);

    if ( context->ckHdr.ckSize == szNotYetKnown ) {
      /* go back and set the chunk size to bytesSoFar */
      if ( 0 >
GSeek(context->file, -(context->bytesSoFar + sizeof(LONG)), OFFSET_CURRENT) ||
          0 >
GWrite(context->file, (BYTE *)&context->bytesSoFar, sizeof(LONG))  ||
          0 >
GSeek(context->file, context->bytesSoFar, OFFSET_CURRENT)  )
        return(DOS_ERROR);
      }
    else {  /* make sure the client wrote as many bytes as planned */
      if ( context->ckHdr.ckSize != context->bytesSoFar )
        return(CLIENT_ERROR);
      };

    /* Write a pad byte if needed to bring us up to an even boundary.
     * Since the context end must be even, and since we haven't
     * overwritten the context, if we're on an odd position there must
     * be room for a pad byte. */
    if ( IS_ODD(context->bytesSoFar) ) {
      if ( 0 > GWrite(context->file, (BYTE *)&zero, 1) )
        return(DOS_ERROR);
      context->position += 1;
      };

    context->ckHdr.ckID   = NULL_CHUNK;
    context->ckHdr.ckSize = context->bytesSoFar = 0;
    return(IFF_OKAY);
    }
```



```
/*--------------------------------------------------------------------*/
/* ilbm2raw.c                                                         */
/*                  2/4/86                                            */
/* Reads in ILBM, outputs raw format, which is                        */
/* just the planes of bitmap data followed by the color map           */
/*                                                                    */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                 */
/* This software is in the public domain.                             */
/*                                                                    */
/* This version for the Commodore-Amiga computer.                     */
/*                                                                    */
/*      Callable from CLI only                                        */
/*--------------------------------------------------------------------*/

#include "iff/intuall.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "iff/readpict.h"
#include "iff/remalloc.h"

#undef NULL
#include "lattice/stdio.h"
/*--------------------------------------------------------------------------*/
/*      Iff error messages                                                  */
/*--------------------------------------------------------------------------*/

char MsgOkay[] = { "----- (IFF_OKAY) A good IFF file." };
char MsgEndMark[] = {"----- (END_MARK) How did you get this message??" };
char MsgDone[] = { "----- (IFF_DONE) How did you get this message??" };
char MsgDos[] = { "----- (DOS_ERROR) The DOS gave back an error." };
char MsgNot[] = { "----- (NOT_IFF) not an IFF file." };
char MsgNoFile[] = { "----- (NO_FILE) no such file found." };
char MsgClientError[] = {"----- (CLIENT_ERROR) IFF Checker bug."};
char MsgForm[] = { "----- (BAD_FORM) How did you get this message??" };
char MsgShort[] = { "----- (SHORT_CHUNK) How did you get this message??" };
char MsgBad[] = { "----- (BAD_IFF) a mangled IFF file." };

/* MUST GET THESE IN RIGHT ORDER!!*/
char *IFFPMessages[-LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };

LONG GfxBase;

/*--------------------------------------------------------------------*/
```

```
SaveBitMap(name,bm,cols)
    UBYTE *name;
    struct BitMap *bm;
    SHORT *cols;
    {
    SHORT i;
    LONG nb,plsize;
    LONG file = Open( name, MODE_NEWFILE);
    if( file == 0 )  {
        printf(" couldn't open %s \n",name);
        return (-1);    /* couldnt open a load-file */
        }
    plsize = bm->BytesPerRow*bm->Rows;
    for (i=0; i<bm->Depth; i++) {
        nb =  Write(file, bm->Planes[i], plsize);
        if (nb<plsize) break;
        }
    Write(file, cols, (1<<bm->Depth)*2);        /* save color map */
    Close(file);
    return(0);
    }

struct BitMap bitmap = {0};

char depthString[] = "0";       /* Replaced with desired digit below.*/

ILBMFrame ilbmFrame;    /* Top level "client frame".*/

/** main() ***************************************************************/

UBYTE defSwitch[] = "b";

void main(argc, argv)  int argc;  char **argv;  {
    LONG iffp, file;
    UBYTE fname[40];
    GfxBase = (LONG)OpenLibrary("graphics.library",0);
    if (GfxBase==NULL) exit(0);

    if (argc) {
        /* Invoked via CLI.  Make a lock for current directory. */
        if (argc < 2) {
            printf("Usage from CLI: 'ilbm2raw filename '\n");
            }
        else {

            file = Open(argv[1], MODE_OLDFILE);

            if (file) {
                iffp = ReadPicture(file, &bitmap, &ilbmFrame, ChipAlloc);
                Close(file);
                if (iffp != IFF_DONE) {
                    printf(" Couldn't read file %s \n", argv[1]);
                    printf("%s\n",IFFPMessages[-iffp]);
                    }
                else {
```



```
                    strcpy(fname,argv[1]);

                    if (ilbmFrame.bmHdr.pageWidth > 320) {
                        if (ilbmFrame.bmHdr.pageHeight > 200)
                            strcat(fname, ".hi");
                        else strcat(fname, ".me");
                        }
                    else     strcat(fname, ".lo");

                    depthString[0] = '0' + bitmap.Depth;
                    strcat(fname, depthString);

                    printf(" Creating file %s \n", fname);
                    SaveBitMap(fname, &bitmap, ilbmFrame.colorMap);
                    }
                }
            else printf(" Couldn't open file: %s. \n", argv[1]);

            if (bitmap.Planes[0])  RemFree(bitmap.Planes[0]);

            printf("\n");
            }
        }
    CloseLibrary(GfxBase);
    exit(0);
    }
```

```
/*------------------------------------------------------------------*/
/*                                                                  */
/* ILBMDump.c: reads in ILBM, prints out ascii representation,      */
/*  for including in C files.                                       */
/*                                                                  */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.               */
/* This software is in the public domain.                           */
/*                                                                  */
/* This version for the Commodore-Amiga computer.                   */
/*                                                                  */
/*  Callable from CLI ONLY                                          */
/*  Jan 31, 1986                                                    */
/*------------------------------------------------------------------*/



#include "iff/intuall.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "iff/readpict.h"
#include "iff/remalloc.h"

#undef NULL
#include "lattice/stdio.h"
/*------------------------------------------------------------------*/
/*      Iff error messages                                          */
/*------------------------------------------------------------------*/

char MsgOkay[] = { "----- (IFF_OKAY) A good IFF file." };
char MsgEndMark[] = {"----- (END_MARK) How did you get this message??" };
char MsgDone[] = { "----- (IFF_DONE) How did you get this message??" };
char MsgDos[] = { "----- (DOS_ERROR) The DOS gave back an error." };
char MsgNot[] = { "----- (NOT_IFF) not an IFF file." };
char MsgNoFile[] = { "----- (NO_FILE) no such file found." };
char MsgClientError[] = {"----- (CLIENT_ERROR) IFF Checker bug."};
char MsgForm[] = { "----- (BAD_FORM) How did you get this message??" };
char MsgShort[] = { "----- (SHORT_CHUNK) How did you get this message??" };
char MsgBad[] = { "----- (BAD_IFF) a mangled IFF file." };

/* MUST GET THESE IN RIGHT ORDER!!*/
char *IFFPMessages[-LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };

/* this returns a string containing characters after the
        last '/' or ':' */
```



```
GetSuffix(to, fr) UBYTE *to, *fr; {
    int i;
    UBYTE c,*s = fr;
    for (i=0; ;i++) {
        c = *s++;
        if (c == 0) break;
        if (c == '/') fr = s;
        else if (c == ':') fr = s;
        }
    strcpy(to,fr);
    }

LONG GfxBase;
struct BitMap bitmap = {0};

ILBMFrame ilbmFrame;    /* Top level "client frame".*/

/** main() ***************************************************************/

UBYTE defSwitch[] = "b";

void main(argc, argv)  int argc;  char **argv;  {
    UBYTE *sw;
    FILE *fp;
    LONG iffp,file;
    UBYTE name[40], fname[40];
    GfxBase = (LONG)OpenLibrary("graphics.library",0);
    if (GfxBase==NULL) exit(0);

    if (argc) {
        /* Invoked via CLI.  Make a lock for current directory. */
        if (argc < 2) {
            printf("Usage from CLI: 'ILBMDump filename switch-string'\n");
            printf(" where switch-string = \n");
            printf("  <nothing> : Bob format (default)\n");
            printf("  s         : Sprite format (with header and trailer words)\n");
            printf("  sn        : Sprite format (No header and trailer words)\n");
            printf("  a         : Attached sprite (with header and trailer)\n");
            printf("  an        : Attached sprite (No header and trailer)\n");
            printf(" Add 'c' to switch list to output CR's with LF's   \n");
            }
        else {
            sw = (argc>2)? argv[2]: defSwitch;

            file = Open(argv[1], MODE_OLDFILE);

            if (file) {
                iffp = ReadPicture(file, &bitmap, &ilbmFrame, ChipAlloc);
                Close(file);
                if (iffp != IFF_DONE) {
                    printf(" Couldn't read file %s \n", argv[1]);
                    printf("%s\n",IFFPMessages[-iffp]);
                    }
                else {
                    printf(" Creating file %s.c \n",argv[1]);
                    GetSuffix(name,argv[1]);
```

```
                strcpy(fname,argv[1]);
                strcat(fname,".c");
                fp = fopen(fname,"w");
                BMPrintCRep(&bitmap,fp,name,sw);
                fclose(fp);
                }
            }
        else printf(" Couldn't open file: %s. \n", argv[1]);

        if (bitmap.Planes[0])  RemFree(bitmap.Planes[0]);

        printf("\n");
        }
    }
CloseLibrary(GfxBase);
exit(0);
}
```



```
/*-----------------------------------------------------------------------*
 * ILBMR.C  Support routines for reading ILBM files.          11/27/85
 * (IFF is Interchange Format File.)
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *-----------------------------------------------------------------------*/
#include "iff/packer.h"
#include "iff/ilbm.h"


/* ---------- GetCMAP ----------------------------------------------------*/
/* pNColorRegs is passed in as a pointer to the number of ColorRegisters
 * caller has space to hold.  GetCMAP sets to the number actually read.*/
IFFP GetCMAP(ilbmContext, colorMap, pNColorRegs)
     GroupContext *ilbmContext;  WORD *colorMap;  UBYTE *pNColorRegs;
   {
   register int nColorRegs;
   register IFFP iffp;
   ColorRegister colorReg;

   nColorRegs = ilbmContext->ckHdr.ckSize / sizeofColorRegister;
   if (*pNColorRegs < nColorRegs)   nColorRegs = *pNColorRegs;
   *pNColorRegs = nColorRegs;   /* Set to the number actually there.*/

   for ( ;  nColorRegs > 0;  --nColorRegs)  {
      iffp = IFFReadBytes(ilbmContext, (BYTE *)&colorReg,sizeofColorRegister);
      CheckIFFP();
      *colorMap++ = ( ( colorReg.red   >> 4 ) << 8 ) |
                    ( ( colorReg.green >> 4 ) << 4 ) |
                    ( ( colorReg.blue  >> 4 )      );
      }
   return(IFF_OKAY);
   }

/*---------- GetBODY ----------------------------------------------------*/
/* NOTE: This implementation could be a LOT faster if it used more of the
 * supplied buffer. It would make far fewer calls to IFFReadBytes (and
 * therefore to DOS Read) and to movemem. */
IFFP GetBODY(context, bitmap, mask, bmHdr, buffer, bufsize)
     GroupContext *context;  struct BitMap *bitmap;  BYTE *mask;
     BitMapHeader *bmHdr;  BYTE *buffer;  LONG bufsize;
   {
   register IFFP iffp;
   UBYTE srcPlaneCnt = bmHdr->nPlanes;   /* Haven't counted for mask plane yet*/
   WORD srcRowBytes = RowBytes(bmHdr->w);
   LONG bufRowBytes = MaxPackedSize(srcRowBytes);
   int nRows = bmHdr->h;
   Compression compression = bmHdr->compression;
   register int iPlane, iRow, nEmpty;
   register WORD nFilled;
   BYTE *buf, *nullDest, *nullBuf, **pDest;
   BYTE *planes[MaxSrcPlanes]; /* array of ptrs to planes & mask */
```

```
   if (compression > cmpByteRun1)
      return(CLIENT_ERROR);

   /* Complain if client asked for a conversion GetBODY doesn't handle.*/
   if ( srcRowBytes  !=  bitmap->BytesPerRow  ||
        bufsize < bufRowBytes * 2  ||
        srcPlaneCnt > MaxSrcPlanes )
      return(CLIENT_ERROR);

   if (nRows > bitmap->Rows)
      nRows = bitmap->Rows;

   /* Initialize array "planes" with bitmap ptrs; NULL in empty slots.*/
   for (iPlane = 0; iPlane < bitmap->Depth; iPlane++)
      planes[iPlane] = (BYTE *)bitmap->Planes[iPlane];
   for ( ;  iPlane < MaxSrcPlanes;  iPlane++)
      planes[iPlane] = NULL;

   /* Copy any mask plane ptr into corresponding "planes" slot.*/
   if (bmHdr->masking == mskHasMask) {
      if (mask != NULL)
         planes[srcPlaneCnt] = mask;  /* If there are more srcPlanes than
               * dstPlanes, there will be NULL plane-pointers before this.*/
      else
         planes[srcPlaneCnt] = NULL;  /* In case more dstPlanes than src.*/
      srcPlaneCnt += 1;  /* Include mask plane in count.*/
      }

   /* Setup a sink for dummy destination of rows from unwanted planes.*/
   nullDest = buffer;
   buffer  += srcRowBytes;
   bufsize -= srcRowBytes;

   /* Read the BODY contents into client's bitmap.
    * De-interleave planes and decompress rows.
    * MODIFIES: Last iteration modifies bufsize.*/
   buf = buffer + bufsize;  /* Buffer is currently empty.*/
   for (iRow = nRows; iRow > 0; iRow--)  {
      for (iPlane = 0; iPlane < srcPlaneCnt; iPlane++)  {

         pDest = &planes[iPlane];

         /* Establish a sink for any unwanted plane.*/
         if (*pDest == NULL) {
            nullBuf = nullDest;
            pDest   = &nullBuf;
            }

         /* Read in at least enough bytes to uncompress next row.*/
         nEmpty  = buf - buffer;          /* size of empty part of buffer.*/
         nFilled = bufsize - nEmpty;      /* this part has data.*/
         if (nFilled < bufRowBytes) {
            /* Need to read more.*/

            /* Move the existing data to the front of the buffer.*/
            /* Now covers range buffer[0]..buffer[nFilled-1].*/
```



```
            movmem(buf, buffer, nFilled);  /* Could be moving 0 bytes.*/

            if (nEmpty > ChunkMoreBytes(context)) {
               /* There aren't enough bytes left to fill the buffer.*/
               nEmpty = ChunkMoreBytes(context);
               bufsize = nFilled + nEmpty;  /* heh-heh */
               }

            /* Append new data to the existing data.*/
            iffp = IFFReadBytes(context, &buffer[nFilled], nEmpty);
            CheckIFFP();

            buf     = buffer;
            nFilled = bufsize;
            nEmpty  = 0;
            }

         /* Copy uncompressed row to destination plane.*/
         if (compression == cmpNone) {
            if (nFilled < srcRowBytes)  return(BAD_FORM);
            movmem(buf, *pDest, srcRowBytes);
            buf    += srcRowBytes;
            *pDest += srcRowBytes;
            }
         else
         /* Decompress row to destination plane.*/
            if ( UnPackRow(&buf, pDest, nFilled,  srcRowBytes) )
                  /*  pSource, pDest, srcBytes, dstBytes  */
               return(BAD_FORM);
         }
      }

   return(IFF_OKAY);
   }
```

```
/*-----------------------------------------------------------------------*
 * ILBMW.C  Support routines for writing ILBM files.            1/23/86
 * (IFF is Interchange Format File.)
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *-----------------------------------------------------------------------*/
#include "iff/packer.h"
#include "iff/ilbm.h"

/*---------- InitBMHdr ------------------------------------------------*/
IFFP InitBMHdr(bmHdr0, bitmap, masking, compression, transparentColor,
          pageWidth, pageHeight)
        BitMapHeader *bmHdr0;  struct BitMap *bitmap;
        WORD masking;           /* Masking */
        WORD compression;       /* Compression */
        WORD transparentColor;  /* UWORD */
        WORD pageWidth, pageHeight;
    {
    register BitMapHeader *bmHdr = bmHdr0;
    register WORD rowBytes = bitmap->BytesPerRow;

    bmHdr->w = rowBytes << 3;
    bmHdr->h = bitmap->Rows;
    bmHdr->x = bmHdr->y = 0;    /* Default position is (0,0).*/
    bmHdr->nPlanes = bitmap->Depth;
    bmHdr->masking = masking;
    bmHdr->compression = compression;
    bmHdr->pad1 = 0;
    bmHdr->transparentColor = transparentColor;
    bmHdr->xAspect = bmHdr->yAspect = 1;
    bmHdr->pageWidth = pageWidth;
    bmHdr->pageHeight = pageHeight;

    if (pageWidth = 320)
        switch (pageHeight) {
            case 200: {bmHdr->xAspect = x320x200Aspect;
                       bmHdr->yAspect = y320x200Aspect; break;}
            case 400: {bmHdr->xAspect = x320x400Aspect;
                       bmHdr->yAspect = y320x400Aspect; break;}
            }
    else if (pageWidth = 640)
        switch (pageHeight) {
            case 200: {bmHdr->xAspect = x640x200Aspect;
                       bmHdr->yAspect = y640x200Aspect; break;}
            case 400: {bmHdr->xAspect = x640x400Aspect;
                       bmHdr->yAspect = y640x400Aspect; break;}
            }

    return( IS_ODD(rowBytes) ? CLIENT_ERROR : IFF_OKAY );
    }

/*---------- PutCMAP --------------------------------------------------*/
IFFP PutCMAP(context, colorMap, depth)
```



```
    GroupContext *context;  WORD *colorMap;  UBYTE depth;
    {
    register LONG nColorRegs;
    IFFP iffp;
    ColorRegister colorReg;

    if (depth > MaxAmDepth)    depth = MaxAmDepth;
    nColorRegs = 1 << depth;

    iffp = PutCkHdr(context, ID_CMAP, nColorRegs * sizeofColorRegister);
    CheckIFFP();

    for ( ;  nColorRegs;  --nColorRegs)  {
        colorReg.red   = ( *colorMap >> 4 ) & 0xf0;
        colorReg.green = ( *colorMap      ) & 0xf0;
        colorReg.blue  = ( *colorMap << 4 ) & 0xf0;
        iffp = IFFWriteBytes(context, (BYTE *)&colorReg, sizeofColorRegister);
        CheckIFFP();
        ++colorMap;
        }

    iffp = PutCkEnd(context);
    return(iffp);
    }

/*---------- PutBODY --------------------------------------------------*/
/* NOTE: This implementation could be a LOT faster if it used more of the
 * supplied buffer. It would make far fewer calls to IFFWriteBytes (and
 * therefore to DOS Write). */
IFFP PutBODY(context, bitmap, mask, bmHdr, buffer, bufsize)
      GroupContext *context;  struct BitMap *bitmap;  BYTE *mask;
      BitMapHeader *bmHdr;  BYTE *buffer;  LONG bufsize;
    {
    IFFP iffp;
    LONG rowBytes = bitmap->BytesPerRow;
    int dstDepth = bmHdr->nPlanes;
    Compression compression = bmHdr->compression;
    int planeCnt;                   /* number of bit planes including mask */
    register int iPlane, iRow;
    register LONG packedRowBytes;
    BYTE *buf;
    BYTE *planes[MaxAmDepth + 1]; /* array of ptrs to planes & mask */

    if ( bufsize < MaxPackedSize(rowBytes)  ||   /* Must buffer a comprsd row*/
         compression > cmpByteRun1  ||           /* bad arg */
         bitmap->Rows != bmHdr->h   ||           /* inconsistent */
         rowBytes != RowBytes(bmHdr->w)  ||      /* inconsistent*/
         bitmap->Depth < dstDepth   ||           /* inconsistent */
         dstDepth > MaxAmDepth )                 /* too many for this routine*/
       return(CLIENT_ERROR);

    planeCnt = dstDepth + (mask == NULL ? 0 : 1);

    /* Copy the ptrs to bit & mask planes into local array "planes" */
    for (iPlane = 0; iPlane < dstDepth; iPlane++)
      planes[iPlane] = (BYTE *)bitmap->Planes[iPlane];
```

```
    if (mask != NULL)
       planes[dstDepth] = mask;

    /* Write out a BODY chunk header */
    iffp = PutCkHdr(context, ID_BODY, szNotYetKnown);
    CheckIFFP();

    /* Write out the BODY contents */
    for (iRow = bmHdr->h; iRow > 0; iRow--)  {
       for (iPlane = 0; iPlane < planeCnt; iPlane++)  {

          /* Write next row.*/
          if (compression == cmpNone) {
             iffp = IFFWriteBytes(context, planes[iPlane], rowBytes);
             planes[iPlane] += rowBytes;
             }

          /* Compress and write next row.*/
          else {
             buf = buffer;
             packedRowBytes = PackRow(&planes[iPlane], &buf, rowBytes);
             iffp = IFFWriteBytes(context, buffer, packedRowBytes);
             }

          CheckIFFP();
          }
       }

    /* Finish the chunk */
    iffp = PutCkEnd(context);
    return(iffp);
    }
```



```
#include "exec/types.h"
movmem(s,d,n)
{
        UBYTE *s,*d;
        int n;
        if(s < d)
        {
                do
                {
                        *d++ = *s++;
                }
                while(n-- > 0);
        }
        else
        {
                s += n;
                d += n;
                do
                {
                        *(--d) = *(--s);
                }
                while(n-- > 0);
        }
}
```

```
/*-----------------------------------------------------------------------*
 * packer.c Convert data to "cmpByteRun1" run compression.     11/15/85
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 *      control bytes:
 *       [0..127]    : followed by n+1 bytes of data.
 *       [-1..-127]  : followed by byte to be repeated (-n)+1 times.
 *       -128        : NOOP.
 *
 * This version for the Commodore-Amiga computer.
 *-----------------------------------------------------------------------*/
#include "iff/packer.h"

#define DUMP    0
#define RUN     1

#define MinRun 3
#define MaxRun 128
#define MaxDat 128

LONG putSize;
#define GetByte()       (*source++)
#define PutByte(c)      { *dest++ = (c);    ++putSize; }

char buf[256];  /* [TBD] should be 128?  on stack?*/

BYTE *PutDump(dest, nn)  BYTE *dest;  int nn; {
        int i;

        PutByte(nn-1);
        for(i = 0;  i < nn;  i++)   PutByte(buf[i]);
        return(dest);
        }

BYTE *PutRun(dest, nn, cc)   BYTE *dest;  int nn, cc; {
        PutByte(-(nn-1));
        PutByte(cc);
        return(dest);
        }

#define OutDump(nn)    dest = PutDump(dest, nn)
#define OutRun(nn,cc)  dest = PutRun(dest, nn, cc)

/*---------- PackRow ----------------------------------------------------*/
/* Given POINTERS TO POINTERS, packs one row, updating the source and
   destination pointers.  RETURNs count of packed bytes.*/
LONG PackRow(pSource, pDest, rowSize)
    BYTE **pSource, **pDest;   LONG rowSize; {
    BYTE *source, *dest;
    char c,lastc = '\0';
    BOOL mode = DUMP;
    short nbuf = 0;             /* number of chars in buffer */
    short rstart = 0;           /* buffer index current run starts */
```



```
    source = *pSource;
    dest = *pDest;
    putSize = 0;
    buf[0] = lastc = c = GetByte();  /* so have valid lastc */
    nbuf = 1;   rowSize--;      /* since one byte eaten.*/


    for (;  rowSize;  --rowSize) {
        buf[nbuf++] = c = GetByte();
        switch (mode) {
                case DUMP:
                        /* If the buffer is full, write the length byte,
                           then the data */
                        if (nbuf>MaxDat) {
                                OutDump(nbuf-1);
                                buf[0] = c;
                                nbuf = 1;   rstart = 0;
                                break;
                                }

                        if (c == lastc) {
                            if (nbuf-rstart >= MinRun) {
                                if (rstart > 0) OutDump(rstart);
                                mode = RUN;
                                }
                            else if (rstart == 0)
                                mode = RUN;     /* no dump in progress,
                                so can't lose by making these 2 a run.*/
                            }
                        else  rstart = nbuf-1;          /* first of run */
                        break;

                case RUN: if ( (c != lastc)|| ( nbuf-rstart > MaxRun)) {
                        /* output run */
                        OutRun(nbuf-1-rstart,lastc);
                        buf[0] = c;
                        nbuf = 1; rstart = 0;
                        mode = DUMP;
                        }
                        break;
                }

        lastc = c;
        }

    switch (mode) {
        case DUMP: OutDump(nbuf); break;
        case RUN: OutRun(nbuf-rstart,lastc); break;
        }
    *pSource = source;
    *pDest = dest;
    return(putSize);
    }
```

```
/** putpict.c ***************************************************/
/* PutPict().  Given a BitMap and a color map in RAM on the     */
/* Amiga, outputs as an ILBM.  See /iff/ilbm.h & /iff/ilbmw.c.  */
/*                    23-Jan-86                                 */
/*                                                              */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.           */
/* This software is in the public domain.                       */
/*                                                              */
/* This version for the Commodore-Amiga computer.               */
/*                                                              */
/****************************************************************/
#include "iff/intuall.h"
#include "iff/gio.h"
#include "iff/ilbm.h"
#include "iff/putpict.h"

#define MaxDepth 5
static IFFP ifferror = 0;

#define CkErr(expression)  {if (ifferror == IFF_OKAY) ifferror = (expression);}

/*****************************************************************************/
/* IffErr                                                                    */
/*                                                                           */
/* Returns the iff error code and resets it to zero                          */
/*                                                                           */
/*****************************************************************************/
IFFP IffErr()
   {
   IFFP i;
   i = ifferror;
   ifferror = 0;
   return(i);
   }

/*****************************************************************************/
/* PutPict()                                                                 */
/*                                                                           */
/* Put a picture into an IFF file                                            */
/* Pass in mask == NULL for no mask.                                         */
/*                                                                           */
/* Buffer should be big enough for one packed scan line                      */
/* Buffer used as temporary storage to speed-up writing.                     */
/* A large buffer, say 8KB, is useful for minimizing Write and Seek calls.   */
/* (See /iff/gio.h & /iff/gio.c).                                            */
/*****************************************************************************/

BOOL PutPict(file, bm, pageW, pageH, colorMap, buffer, bufsize)
      LONG file; struct BitMap *bm;
        WORD pageW,pageH;
        WORD *colorMap;
      BYTE *buffer;  LONG bufsize;
   {
   BitMapHeader bmHdr;
   GroupContext fileContext, formContext;
```



```
    ifferror = InitBMHdr(&bmHdr,
        bm,
        mskNone,
        cmpByteRun1,
        0,
        pageW,
        pageH );

/* use buffered write for speedup, if it is big-enough for both
 * PutBODY's buffer and a gio buffer.*/
#define BODY_BUFSIZE 512
    if (ifferror == IFF_OKAY  &&  bufsize > 2*BODY_BUFSIZE) {
        if (GWriteDeclare(file, buffer+BODY_BUFSIZE, bufsize-BODY_BUFSIZE) < 0)
            ifferror = DOS_ERROR;
        bufsize = BODY_BUFSIZE;
        }

    CkErr(OpenWIFF(file, &fileContext, szNotYetKnown) );
    CkErr(StartWGroup(&fileContext, FORM, szNotYetKnown, ID_ILBM, &formContext) );

    CkErr(PutCk(&formContext, ID_BMHD, sizeof(BitMapHeader), (BYTE *)&bmHdr));

    if (colorMap!=NULL)
        CkErr( PutCMAP(&formContext, colorMap, (UBYTE)bm->Depth) );
    CkErr( PutBODY(&formContext, bm, NULL, &bmHdr, buffer, bufsize) );

    CkErr( EndWGroup(&formContext) );
    CkErr( CloseWGroup(&fileContext) );
    if (GWriteUndeclare(file) < 0  &&  ifferror == IFF_OKAY)
        ifferror = DOS_ERROR;
    return( (BOOL)(ifferror != IFF_OKAY) );
    }
```

```
/** raw2ilbm.c ********************************************************************
/*  Read in a "raw" bitmap (dump of the bitplanes in a screen)  */
/*  Display it, and write it out as an ILBM file.               */
/*  23-Jan-86                                                   */
/*                                                              */
/*  Usage from CLI: 'Raw2ILBM  source dest fmt(low,med,hi)      */
/*     nplanes'                                                 */
/*  Supports the three common Amiga screen formats.             */
/*              'low' is 320x200,                               */
/*              'med' is 640x200,                               */
/*              'hi' is 640x400.                                */
/*               'nplanes' is the number of bitplanes.          */
/*  The default is low-resolution, 5 bitplanes                  */
/*              (32 colors per pixel).                          */
/*                                                              */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.           */
/* This software is in the public domain.                       */
/*                                                              */
/* This version for the Commodore-Amiga computer.               */
/*                                                              */
/********************************************************************/

#include "iff/intuall.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "iff/putpict.h"

#define MIN(a,b)  ((a)<(b)?(a):(b))
#define MAX(a,b)  ((a)>(b)?(a):(b))

/* general usage pointers */
LONG IconBase;  /* Actually, "struct IconBase *" if you've got some ".h" file*/
struct GfxBase *GfxBase;


/* Globals for displaying an image */
struct RastPort rP;
struct RasInfo rasinfo;
struct View v = {0};
struct ViewPort vp = {0};
struct View *oldView = 0;       /* so we can restore it */

/* ---------------------------------- */
DisplayPic(bm, colorMap) struct BitMap *bm; UWORD *colorMap;  {

    oldView = GfxBase->ActiView;        /* so we can restore it */

    InitView(&v);
    InitVPort(&vp);
    v.ViewPort = &vp;
    InitRastPort(&rP);
    rP.BitMap = bm;
    rasinfo.BitMap = bm;

    /* Always show the upper left-hand corner of this picture. */
```



```
    rasinfo.RxOffset = 0;
    rasinfo.RyOffset = 0;

    vp.DWidth = bm->BytesPerRow*8;      /* Physical display WIDTH */
    vp.DHeight = bm->Rows;      /* Display height */

    /* Always display it in upper left corner of screen.*/

    if (vp.DWidth <= 320) vp.Modes = 0;
        else vp.Modes = HIRES;
    if (vp.DHeight > 200) {
        v.Modes |= LACE;
        vp.Modes |= LACE;
        }
    vp.RasInfo = &rasinfo;
    MakeVPort(&v,&vp);
    MrgCop(&v);
    LoadView(&v);       /* show the picture */
    WaitBlit();
    WaitTOF();
    if (colorMap) LoadRGB4(&vp, colorMap,(1 << bm->Depth));
    }

UnDispPict() {
    if (oldView) {
        LoadView(oldView);      /* switch back to old view */
        FreeVPortCopLists(&vp);
        FreeCprList(v.LOFCprList);
        }
    }

PrintS(msg)  char *msg; {   printf(msg);    }

void GoodBye(msg)  char *msg; {   PrintS(msg);   PrintS("\n");   exit(0);   }

struct BitMap bitmap = {0};
SHORT cmap[32];

AllocBitMap(bm) struct BitMap *bm; {
    int i;
    LONG psz = bm->BytesPerRow*bm->Rows;
    UBYTE *p = (UBYTE *)AllocMem(bm->Depth*psz, MEMF_CHIP|MEMF_PUBLIC);
    for (i=0; i<bm->Depth; i++)  {
        bm->Planes[i] = p;
        p += psz;
        }
    }

FreeBitMap(bm) struct BitMap *bm;  {
    if (bitmap.Planes[0])  {
        FreeMem(bitmap.Planes[0],
                bitmap.BytesPerRow * bitmap.Rows * bitmap.Depth);
        }
    }

BOOL LoadBitMap(file,bm,cols)
```

```
    LONG file;
    struct BitMap *bm;
    SHORT *cols;
    {
    SHORT i;
    LONG nb,plsize;
    plsize = bm->BytesPerRow*bm->Rows;
    for (i=0; i<bm->Depth; i++) {
        nb =  Read(file, bm->Planes[i], plsize);
        if (nb<plsize) BltClear(bm->Planes[i],plsize,1);
        }
    if (cols) {
        nb = Read(file, cols, (1<<bm->Depth)*2);        /* load color map */
        return( (BOOL) (nb == (1<<bm->Depth)*2) );
        }
    return((BOOL) FALSE);
    }


/** main() ************************************************************************/

UBYTE defSwitch[] = "b";

#define BUFSIZE  16000

static SHORT maxDepth[3] = {5,4,4};

void main(argc, argv)  int argc;  char **argv;  {
    SHORT fmt,depth,pwidth,pheight;
    UBYTE *buffer;
    BOOL hadCmap;
    LONG file;
    if( !(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0)) )
        GoodBye("No graphics.library");
    if( !(IconBase = OpenLibrary("icon.library",0)) )
        GoodBye("No icon.library");
    if (argc) {
        if (argc < 3) {
            printf(
"Usage from CLI: 'Raw2ILBM  source dest fmt(low,med,hi) nplanes'\n");
            goto bailout;
            }
        fmt = 0;
        depth = 5;
        if (argc>3)
            switch(*argv[3]) {
                case 'l': fmt = 0; break;
                case 'm': fmt = 1; break;
                case 'h': fmt = 2; break;
                }
        if (argc>4) depth = *argv[4]-'0';
        depth = MAX(1, MIN(maxDepth[fmt],depth));
        pwidth = fmt? 640: 320;
        pheight = (fmt>1)? 400: 200;
        InitBitMap(&bitmap, depth, pwidth, pheight);
        AllocBitMap(&bitmap);
```



```
        file = Open(argv[1], MODE_OLDFILE);

        if (file)  {
            DisplayPic(&bitmap,NULL);
            hadCmap = LoadBitMap(file,&bitmap, cmap);
            if (hadCmap) LoadRGB4(&vp, cmap, 1<<bitmap.Depth);
            Close(file);
            file = Open(argv[2], MODE_NEWFILE);
            buffer = (UBYTE *)AllocMem(BUFSIZE, MEMF_CHIP|MEMF_PUBLIC);
            PutPict(file, &bitmap, pwidth, pheight,
                hadCmap? cmap: NULL, buffer, BUFSIZE);
            Close(file);
            FreeMem(buffer,BUFSIZE);
            }
        else printf(" Couldn't open file '%s' \n",argv[2]);
        }

    UnDispPict();
    FreeBitMap(&bitmap);

    bailout:
        CloseLibrary(GfxBase);
        CloseLibrary(IconBase);
        exit(0);
    }
```

```
/** ReadPict.c ************************************************************
 *
 * Read an ILBM raster image file.                              23-Jan-86.
 *
 * By Jerry Morrison, Steve Shaw, and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * USE THIS AS AN EXAMPLE PROGRAM FOR AN IFF READER.
 *
 * The IFF reader portion is essentially a recursive-descent parser.
 ************************************************************************/

#define LOCAL   static

#include "iff/intuall.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "iff/readpict.h"

/* This example's max number of planes in a bitmap. Could use MaxAmDepth. */
#define EXDepth 5
#define maxColorReg (1<<EXDepth)
#define MIN(a,b)  ((a)<(b)?(a):(b))

#define SafeFreeMem(p,q)  {if(p)FreeMem(p,q);}

/* Define the size of a temporary buffer used in unscrambling the ILBM rows.*/
#define bufSz 512

/*------------ ILBM reader ------------------------------------------------*/
/* ILBMFrame is our "client frame" for reading FORMs ILBM in an IFF file.
 * We allocate one of these on the stack for every LIST or FORM encountered
 * in the file and use it to hold BMHD & CMAP properties. We also allocate
 * an initial one for the whole file.
 * We allocate a new GroupContext (and initialize it by OpenRIFF or
 * OpenRGroup) for every group (FORM, CAT, LIST, or PROP) encountered. It's
 * just a context for reading (nested) chunks.
 *
 * If we were to scan the entire example file outlined below:
 *    reading          proc(s)                 new                 new
 *
 * --whole file--   ReadPicture+ReadIFF   GroupContext        ILBMFrame
 * CAT              ReadICat                GroupContext
 *   LIST           GetLiILBM+ReadIList       GroupContext        ILBMFrame
 *     PROP ILBM    GetPrILBM                   GroupContext
 *       CMAP       GetCMAP
 *       BMHD       GetBMHD
 *     FORM ILBM    GetFoILBM                   GroupContext        ILBMFrame
 *       BODY       GetBODY
 *     FORM ILBM    GetFoILBM                   GroupContext        ILBMFrame
 *       BODY       GetBODY
 *   FORM ILBM      GetFoILBM                 GroupContext        ILBMFrame
 */

/* NOTE: For a small version of this program, set Fancy to 0.
```



```
 * That'll compile a program that reads a single FORM ILBM in a file, which
 * is what DeluxePaint produces. It'll skip all LISTs and PROPs in the input
 * file. It will, however, look inside a CAT for a FORM ILBM.
 * That's suitable for 90% of the uses.
 *
 * For a fancier version that handles LISTs and PROPs, set Fancy to 1.
 * That'll compile a program that dives into a LIST, if present, to read
 * the first FORM ILBM. E.g. a DeluxePrint library of images is a LIST of
 * FORMs ILBM.
 *
 * For an even fancier version, set Fancy to 2. That'll compile a program
 * that dives into non-ILBM FORMs, if present, looking for a nested FORM ILBM.
 * E.g. a DeluxeVideo C.S. animated object file is a FORM ANBM containing a
 * FORM ILBM for each image frame. */
#define Fancy  0

/* Global access to client-provided pointers.*/
LOCAL Allocator *gAllocator = NULL;
LOCAL struct BitMap *gBM = NULL;        /* client's bitmap.*/
LOCAL ILBMFrame *giFrame = NULL;        /* "client frame".*/

/** GetFoILBM() ***********************************************************
 *
 * Called via ReadPicture to han+le every FORM encountered in an IFF file.
 * Reads FORMs ILBM and skips all others.
 * Inside a FORM ILBM, it stops once it reads a BODY. It complains if it
 * finds no BODY or if it has no BMHD to decode the BODY.
 *
 * Once we find a BODY chunk, we'll allocate the BitMap and read the image.
 *
 ************************************************************************/
LOCAL BYTE bodyBuffer[bufSz];
IFFP GetFoILBM(parent)  GroupContext *parent;  {
   /*compilerBug register*/ IFFP iffp;
   GroupContext formContext;
   ILBMFrame ilbmFrame;         /* only used for non-clientFrame fields.*/
   register int i;
   LONG plsize; /* Plane size in bytes. */
   int nPlanes; /* number of planes in our display image */

    /* Handle a non-ILBM FORM. */
    if (parent->subtype != ID_ILBM) {
#if Fancy >= 2
        /* Open a non-ILBM FORM and recursively scan it for ILBMs.*/
        iffp = OpenRGroup(parent, &formContext);
        CheckIFFP();
        do {
            iffp = GetF1ChunkHdr(&formContext);
            } while (iffp >= IFF_OKAY);
        if (iffp == END_MARK)
            iffp = IFF_OKAY;    /* then continue scanning the file */
        CloseRGroup(&formContext);
        return(iffp);
#else
        return(IFF_OKAY); /* Just skip this FORM and keep scanning the file.*/
#endif
```

```
      }

  ilbmFrame = *(ILBMFrame *)parent->clientFrame;
  iffp = OpenRGroup(parent, &formContext);
  CheckIFFP();

  do switch (iffp = GetFChunkHdr(&formContext)) {
     case ID_BMHD: {
       ilbmFrame.foundBMHD = TRUE;
       iffp = GetBMHD(&formContext, &ilbmFrame.bmHdr);
       break; }
     case ID_CMAP: {
       ilbmFrame.nColorRegs = maxColorReg;  /* we have room for this many */
       iffp = GetCMAP(
          &formContext, (WORD *)&ilbmFrame.colorMap[0], &ilbmFrame.nColorRegs);
                     /* was &ilbmFrame.colorMap, (fixed) robp. */
       break; }
     case ID_BODY: {
        if (!ilbmFrame.foundBMHD)  return(BAD_FORM);   /* No BMHD chunk! */

        nPlanes = MIN(ilbmFrame.bmHdr.nPlanes, EXDepth);
        InitBitMap(
           gBM,
           nPlanes,
           ilbmFrame.bmHdr.w,
           ilbmFrame.bmHdr.h);
        plsize = RowBytes(ilbmFrame.bmHdr.w) * ilbmFrame.bmHdr.h;
        /* Allocate all planes contiguously.  Not really necessary,
         * but it avoids writing code to back-out if only enough memory
         * for some of the planes.
         * WARNING: Don't change this without changing the code that
         * Frees these planes.
         */
        if (gBM->Planes[0] =
              (PLANEPTR) (*gAllocator)(nPlanes * plsize))
           {
           for (i = 1; i < nPlanes; i++)
               gBM->Planes[i] = (PLANEPTR) gBM->Planes[0] + plsize*i;
           iffp = GetBODY(
               &formContext,
               gBM,
               NULL,
               &ilbmFrame.bmHdr,
               bodyBuffer,
               bufSz);
           if (iffp == IFF_OKAY) iffp = IFF_DONE;      /* Eureka */
           *giFrame = ilbmFrame;  /* Copy fields to client's frame.*/
           }
        else
           iffp = CLIENT_ERROR;        /* not enough RAM for the bitmap */
        break; }
    case END_MARK: { iffp = BAD_FORM; break; } /* No BODY chunk! */
    } while (iffp >= IFF_OKAY);  /* loop if valid ID of ignored chunk or a
                      * subroutine returned IFF_OKAY (no errors).*/

if (iffp != IFF_DONE)  return(iffp);
```



```
    /* If we get this far, there were no errors. */
    CloseRGroup(&formContext);
    return(iffp);
    }

/** Notes on extending GetFoILBM ***************************************************
 *
 * To read more kinds of chunks, just add clauses to the switch statement.
 * To read more kinds of property chunks (GRAB, CAMG, etc.) add clauses to
 * the switch statement in GetPrILBM, too.
 *
 * To read a FORM type that contains a variable number of data chunks--e.g.
 * a FORM FTXT with any number of CHRS chunks--replace the ID_BODY case with
 * an ID_CHRS case that doesn't set iffp = IFF_DONE, and make the END_MARK
 * case do whatever cleanup you need.
 *
 *********************************************************************************/

/** GetPrILBM() ********************************************************************
 *
 * Called via ReadPicture to handle every PROP encountered in an IFF file.
 * Reads PROPs ILBM and skips all others.
 *
 *********************************************************************************/
#if Fancy
IFFP GetPrILBM(parent)  GroupContext *parent;  {
   /*compilerBug register*/ IFFP iffp;
   GroupContext propContext;
   ILBMFrame *ilbmFrame = (ILBMFrame *)parent->clientFrame;

   if (parent->subtype != ID_ILBM)
      return(IFF_OKAY); /* just continue scaning the file */

   iffp = OpenRGroup(parent, &propContext);
   CheckIFFP();

   do switch (iffp = GetPChunkHdr(&propContext)) {
      case ID_BMHD: {
        ilbmFrame->foundBMHD = TRUE;
        iffp = GetBMHD(&propContext, &ilbmFrame->bmHdr);
        break; }
      case ID_CMAP: {
        ilbmFrame->nColorRegs = maxColorReg; /* we have room for this many */
        iffp = GetCMAP(
          &propContext, (WORD *)&ilbmFrame->colorMap, &ilbmFrame->nColorRegs);
        break; }
      } while (iffp >= IFF_OKAY);  /* loop if valid ID of ignored chunk or a
                        * subroutine returned IFF_OKAY (no errors).*/

   CloseRGroup(&propContext);
   return(iffp == END_MARK ? IFF_OKAY : iffp);
   }
#endif

/** GetLiILBM() ********************************************************************
```

```
 *
 * Called via ReadPicture to handle every LIST encountered in an IFF file.
 *
 ******************************************************************************/
#if Fancy
IFFP GetLiILBM(parent)  GroupContext *parent;  {
    ILBMFrame newFrame; /* allocate a new Frame */

    newFrame = *(ILBMFrame *)parent->clientFrame;  /* copy parent frame */

    return( ReadIList(parent, (ClientFrame *)&newFrame) );
    }
#endif

/** ReadPicture() ************************************************************/
IFFP ReadPicture(file, bm, iFrame, allocator)
   LONG file;
   struct BitMap *bm;
   ILBMFrame *iFrame;   /* Top level "client frame".*/

        /* **** ERROR IN SOURCE CODE, WAS jFrame, now iFrame */
        /* fixed */

   Allocator *allocator;
   {
   IFFP iffp = IFF_OKAY;

#if Fancy
   iFrame->clientFrame.getList = GetLiILBM;
   iFrame->clientFrame.getProp = GetPrILBM;
#else
   iFrame->clientFrame.getList = SkipGroup;
   iFrame->clientFrame.getProp = SkipGroup;
#endif
   iFrame->clientFrame.getForm = GetFoILBM;
   iFrame->clientFrame.getCat  = ReadICat ;

   /* Initialize the top-level client frame's property settings to the
    * program-wide defaults. This example just records that we haven't read
    * any BMHD property or CMAP color registers yet. For the color map, that
    * means the default is to leave the machine's color registers alone.
    * If you want to read a property like GRAB, init it here to (0, 0). */
   iFrame->foundBMHD  = FALSE;
   iFrame->nColorRegs = 0;

   gAllocator = allocator;
   gBM = bm;
   giFrame = iFrame;
  /* Store a pointer to the client's frame in a global variable so that
   * GetFoILBM can update client's frame when done.  Why do we have so
   * many frames & frame pointers floating around causing confusion?
   * Because IFF supports PROPs which apply to all FORMs in a LIST,
   * unless a given FORM overrides some property.
   * When you write code to read several FORMs,
   * it is ssential to maintain a frame at each level of the syntax
   * so that the properties for the LIST don't get overwritten by any
```



```
   * properties specified by individual FORMs.
   * We decided it was best to put that complexity into this one-FORM example,
   * so that those who need it later will have a useful starting place.
   */

   iffp = ReadIFF(file, (ClientFrame *)iFrame);
   return(iffp);
   }
```

```
/** RemAlloc.c ************************************************************/
/*  ChipAlloc(), ExtAlloc(), RemAlloc(), RemFree().                       */
/*  ALLOCators which REMember the size allocated, for simpler freeing.    */
/*                                                                        */
/* Date      Who Changes                                                  */
/* --------- --- ----------------------------------------------------------*/
/* 16-Jan-86 sss Created from DPaint/DAlloc.c                             */
/* 23-Jan-86 jhm Include Compiler.h, check for size > 0 in RemAlloc.      */
/* 25-Jan-86 sss Added ChipNoClearAlloc,ExtNoClearAlloc                   */
/*                                                                        */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.                     */
/* This software is in the public domain.                                 */
/*                                                                        */
/* This version for the Commodore-Amiga computer.                         */
/*                                                                        */
/************************************************************************/
#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#include "exec/nodes.h"
#include "exec/memory.h"
#include "iff/remalloc.h"

/** RemAlloc *************************************************************/
UBYTE *RemAlloc(size,flags) LONG size, flags; {
    register LONG *p = NULL;    /* (LONG *) for the sake of p++, below */
    register LONG asize = size+4;
    if (size > 0)
        p = (LONG *)AllocMem(asize,flags);
    if (p != NULL)
        *p++ = asize;   /* post-bump p to point at clients area*/
    return((UBYTE *)p);
    }

/** ChipAlloc ************************************************************/
UBYTE *ChipAlloc(size) LONG size; {
    return(RemAlloc(size, MEMF_CLEAR|MEMF_PUBLIC|MEMF_CHIP));
    }

/** ChipNoClearAlloc *****************************************************/
UBYTE *ChipNoClearAlloc(size) LONG size; {
    return(RemAlloc(size, MEMF_PUBLIC|MEMF_CHIP));
    }

/** ExtAlloc *************************************************************/
UBYTE *ExtAlloc(size) LONG size; {
    return(RemAlloc(size, MEMF_CLEAR|MEMF_PUBLIC));
    }

/** ExtNoClearAlloc ******************************************************/
UBYTE *ExtNoClearAlloc(size) LONG size; {
    return(RemAlloc(size, MEMF_PUBLIC));
    }

/** RemFree **************************************************************/
```



```
UBYTE *RemFree(p) UBYTE *p; {
    if (p != NULL) {
        p -= 4;
        FreeMem(p, *((LONG *)p));
        }
    return(NULL);
    }
```

```
/** ShowILBM.c ***********************************************************
 *
 * Read an ILBM raster image file and display it.       24-Jan-86.
 *
 * By Jerry Morrison, Steve Shaw, and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * USE THIS AS AN EXAMPLE PROGRAM FOR AN IFF READER.
 *
 * The IFF reader portion is essentially a recursive-descent parser.
 * The display portion is specific to the Commodore Amiga computer.
 *
 * NOTE: This program displays an image, pauses, then exits.
 *
 * Usage from CLI:
 *   showilbm picture1 [picture2] ...
 *
 * Usage from WorkBench:
 * Click on ShowILBM, hold down shift key, click on each picture to show,
 * Double-click on final picture to complete the selection, release the
 * shift key.
 *
 ************************************************************************/

/* If you are constructing a Makefile, here are the names of the files
 * that you'll need to compile and link with to use showilbm:

        showilbm.c
        readpict.c
        remalloc.c
        ilbmr.c
        iffr.c
        unpacker.c
        gio.c

        and you'll have to get movmem() from lc.lib

 * robp.
 * ********************************************************************** */


#include "iff/intuall.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "workbench/workbench.h"
#include "workbench/startup.h"
#include "iff/readpict.h"
#include "iff/remalloc.h"

#define LOCAL static

#define MIN(a,b)  ((a)<(b)?(a):(b))
#define MAX(a,b)  ((a)>(b)?(a):(b))

/* general usage pointers */
```



```
struct GfxBase *GfxBase;
LONG IconBase;  /* Actually, "struct IconBase *" if you've got some ".h" file*/

/* For displaying an image */
LOCAL struct RastPort rP;
LOCAL struct BitMap bitmap0;
LOCAL struct RasInfo rasinfo;
LOCAL struct View v = {0};
LOCAL struct ViewPort vp = {0};

LOCAL ILBMFrame iFrame;

/* Define the size of a temporary buffer used in unscrambling the ILBM rows.*/
#define bufSz 512

/* Message strings for IFFP codes. */
LOCAL char MsgOkay[]        = {
      "(IFF_OKAY) Didn't find a FORM ILBM in the file." };
LOCAL char MsgEndMark[]     = { "(END_MARK) How did you get this message?" };
LOCAL char MsgDone[]        = { "(IFF_DONE) All done."};
LOCAL char MsgDos[]         = { "(DOS_ERROR) The DOS returned an error." };
LOCAL char MsgNot[]         = { "(NOT_IFF) Not an IFF file." };
LOCAL char MsgNoFile[]      = { "(NO_FILE) No such file found." };
LOCAL char MsgClientError[] = {
      "(CLIENT_ERROR) ShowILBM bug or insufficient RAM."};
LOCAL char MsgForm[]        = { "(BAD_FORM) A malformed FORM ILBM." };
LOCAL char MsgShort[]       = { "(SHORT_CHUNK) A malformed FORM ILBM." };
LOCAL char MsgBad[]         = { "(BAD_IFF) A mangled IFF file." };

/* THESE MUST APPEAR IN RIGHT ORDER!! */
LOCAL char *IFFPMessages[-(int)LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };

/** DisplayPic() *********************************************************
 *
 * Interface to Amiga graphics ROM routines.
 *
 ************************************************************************/
DisplayPic(bm, ptilbmFrame)
    struct BitMap *bm;  ILBMFrame *ptilbmFrame;  {
    int i;
    struct View *oldView = GfxBase->ActiView;   /* so we can restore it */

    InitView(&v);
    InitVPort(&vp);
    v.ViewPort = &vp;
```

```
    InitRastPort(&rP);
    rP.BitMap = bm;
    rasinfo.BitMap = bm;

    /* Always show the upper left-hand corner of this picture. */
    rasinfo.RxOffset = 0;
    rasinfo.RyOffset = 0;

    vp.DWidth = MAX(ptilbmFrame->bmHdr.w, 4*8);
    vp.DHeight = ptilbmFrame->bmHdr.h;

#if 0
    /* Specify where on screen to put the ViewPort. */
    vp.DxOffset = ptilbmFrame->bmHdr.x;
    vp.DyOffset = ptilbmFrame->bmHdr.y;
#else
    /* Always display it in upper left corner of screen.*/
#endif

    if (ptilbmFrame->bmHdr.pageWidth <= 320)
        vp.Modes = 0;
    else vp.Modes = HIRES;
    if (ptilbmFrame->bmHdr.pageHeight > 200) {
        v.Modes |= LACE;
        vp.Modes |= LACE;
        }
    vp.RasInfo = &rasinfo;
    MakeVPort(&v,&vp);
    MrgCop(&v);
    LoadView(&v);       /* show the picture */
    WaitBlit();
    WaitTOF();
    LoadRGB4(&vp, ptilbmFrame->colorMap, ptilbmFrame->nColorRegs);

    for (i = 0; i < 5*60; ++i)  WaitTOF();      /* Delay 5 seconds. */

    LoadView(oldView);  /* switch back to old view */
    }

/** stuff for main0() ***********************************************************/
LOCAL struct WBStartup *wbStartup = 0;  /* 0 unless started from WorkBench.*/

PrintS(msg)  char *msg; {
    if (!wbStartup)  printf(msg);
    }

void GoodBye(msg)  char *msg; {
    PrintS(msg);   PrintS("\n");
    exit(0);
    }

/** OpenArg() ******************************************************************
 *  Given a "workbench argument" (a file reference) and an I/O mode.
 *  It opens the file.
 ******************************************************************************/
LONG OpenArg(wa, openmode)  struct WBArg *wa;   int openmode; {
```



```
    LONG olddir;
    LONG file;
    if (wa->wa_Lock)   olddir = CurrentDir(wa->wa_Lock);
    file = Open(wa->wa_Name, openmode);
    if (wa->wa_Lock)   CurrentDir(olddir);
    return(file);
    }

/** main0() ********************************************************************/
void main0(wa)  struct WBArg *wa;  {
    LONG file;
    IFFP iffp = NO_FILE;

    /* load and display the picture */
    file = OpenArg(wa, MODE_OLDFILE);
    if (file)
        iffp = ReadPicture(file, &bitmap0, &iFrame, ChipAlloc);
        /* Allocates BitMap using ChipAlloc().*/
    Close(file);
    if (iffp == IFF_DONE)
        DisplayPic(&bitmap0, &iFrame);

    PrintS(" ");   PrintS(IFFPMessages[-iffp]);   PrintS("\n");

    /* cleanup */
    if (bitmap0.Planes[0])  {
        RemFree(bitmap0.Planes[0]);
                /* ASSUMES allocated all planes via a single ChipAlloc call.*/
        FreeVPortCopLists(&vp);
        FreeCprList(v.LOFCprList);
        }
    }

/** main() *********************************************************************/

void main(argc, argv)  int argc;  char **argv;  {
    struct WBArg wbArg, *wbArgs;
    LONG olddir;
/*sss    struct Process *myProcess; */

    if( !(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0)) )
        GoodBye("No graphics.library");
    if( !(IconBase = OpenLibrary("icon.library",0)) )
        GoodBye("No icon.library");
    if (!argc) {
        /* Invoked via workbench */
        wbStartup = (struct WBStartup *)argv;
        wbArgs = wbStartup->sm_ArgList;
        argc = wbStartup->sm_NumArgs;
        while (argc >= 2) {
            olddir = CurrentDir(wbArgs[1].wa_Lock);
            main0(&wbArgs[1]);
            argc--;   wbArgs = &wbArgs[1];
            }
#if 0
        /* [TBD] We want to get an error msg to the Workbench user... */
```

```
        if (argc < 2) {
            PrintS ("Usage from workbench:\n");
            PrintS (" Click mouse on Show-ILBM, Then hold 'SHIFT' key\n");
            GoodBye(" while double-click on file to display.");
            }
#endif
        }
    else {
        /* Invoked via CLI.  Make a lock for current directory.
         * Eventually, scan name, separate out directory reference?*/
        if (argc < 2)
            GoodBye("Usage from CLI: 'Show-ILBM filename'");
/*sss   myProcess = (struct Process *)FindTask(0); */
        wbArg.wa_Lock = 0; /*sss myProcess->pr_CurrentDir; */
        while (argc >= 2) {
            wbArg.wa_Name = argv[1];
            PrintS("Showing file ");   PrintS(wbArg.wa_Name);   PrintS(" ...");
            main0(&wbArg);
            PrintS("\n");
            argc--;   argv = &argv[1];
            }
        }
    CloseLibrary(GfxBase);
    CloseLibrary(IconBase);
    exit(0);
    }
```



```
/*---------------------------------------------------------------------*
 * unpacker.c Convert data from "cmpByteRun1" run compression. 11/15/85
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 *      control bytes:
 *       [0..127]    : followed by n+1 bytes of data.
 *       [-1..-127]  : followed by byte to be repeated (-n)+1 times.
 *       -128        : NOOP.
 *
 * This version for the Commodore-Amiga computer.
 *---------------------------------------------------------------------*/
#include "iff/packer.h"


/*---------- UnPackRow ------------------------------------------------*/

#define UGetByte()      (*source++)
#define UPutByte(c)     (*dest++ = (c))

/* Given POINTERS to POINTER variables, unpacks one row, updating the source
 * and destination pointers until it produces dstBytes bytes. */
BOOL UnPackRow(pSource, pDest, srcBytes0, dstBytes0)
        BYTE **pSource, **pDest;  WORD srcBytes0, dstBytes0; {
    register BYTE *source = *pSource;
    register BYTE *dest   = *pDest;
    register WORD n;
    register BYTE c;
    register WORD srcBytes = srcBytes0, dstBytes = dstBytes0;
    BOOL error = TRUE;  /* assume error until we make it through the loop */
    WORD minus128 = -128;  /* get the compiler to generate a CMP.W */

    while( dstBytes > 0 )  {
        if ( (srcBytes -= 1) < 0 )  goto ErrorExit;
        n = UGetByte();

        if (n >= 0) {
            n += 1;
            if ( (srcBytes -= n) < 0 )  goto ErrorExit;
            if ( (dstBytes -= n) < 0 )  goto ErrorExit;
            do {  UPutByte(UGetByte());  } while (--n > 0);
            }

        else if (n != minus128) {
            n = -n + 1;
            if ( (srcBytes -= 1) < 0 )  goto ErrorExit;
            if ( (dstBytes -= n) < 0 )  goto ErrorExit;
            c = UGetByte();
            do {  UPutByte(c);  } while (--n > 0);
            }
        }
    error = FALSE;      /* success! */

ErrorExit:
    *pSource = source;  *pDest = dest;
```

```
    return(error);
    }
```

```
FROM lstartup.o,iffcheck.o,iffr.o,gio.o
LIBRARY lc.lib,amiga.lib
TO iffcheck

FROM lstartup.o,iffcheck.o,iffr.o
LIBRARY lc.lib,amiga.lib
TO iffcheck

FROM lstartup.o, ilbm2raw.o, readpict.o, ilbmr.o, unpacker.o, iffr.o*
remalloc.o
LIBRARY lc.lib, amiga.lib
TO ilbm2raw

FROM lstartup.o, ilbmdump.o, readpict.o, ilbmr.o, unpacker.o, iffr.o*
remalloc.o, bmprintc.o
LIBRARY lc.lib, amiga.lib
TO ilbmdump

FROM lstartup.o, raw2ilbm.o, putpict.o, ilbmw.o, packer.o, iffw.o, gio.o
LIBRARY lc.lib, amiga.lib
TO raw2ilbm

FROM lstartup.o, raw2ilbm.o, putpict.o, ilbmw.o, packer.o, iffw.o
LIBRARY lc.lib, amiga.lib
TO raw2ilbm

FROM lstartup.o,showilbm.o,readpict.o,ilbmr.o,unpacker.o,iffr.o,remalloc.o*
gio.o
LIBRARY lc.lib,amiga.lib
TO showilbm

FROM lstartup.o,showilbm.o,readpict.o,ilbmr.o,unpacker.o,iffr.o,remalloc.o
TO showilbm
LIBRARY lc.lib,amiga.lib
```

ADDENDA

# NEW RESERVED FORM/CHUNK NAMES

1. ACBM (contiguous bitmap form - see ACBM in docs/addenda)

2. WORD (a document form - see WORD in docs/addenda)

3. HEAD (an idea processor form - see HEAD in docs/addenda)

4. ANIM (a cel animation form - see ANIM in docs/addenda)

The following FORM and Chunk names are reserved for FORMS which are under construction by independent developers. When any of these forms are finalized, a complete spec will be posted on BIX in amiga.dev/iff, and probably to usenet (a mainframe network).

1. SPLT (another proposed animation FORM)
   SHDR (proposed chunks)
   CELL
   COLR

2. RGB4 (FORM for 4 bit R G B pixel information)
   COMP (chunk containing compression table for the FORM)

   The RGB4 FORM contains a BMHD which will specify 2 as its Compression. BMHD compression value 2 has been reserved for this algorithm which is a modified Huffman encoding.

Carolyn Scheppner CBM 07/16/87

## IFF FORM / CHUNK DESCRIPTION

Form/Chunk ID: FORM ACBM (Amiga Contiguous BitMap)
Chunk ABIT (Amiga BITplanes)

Date Submitted: 05/29/86
Submitted by: Carolyn Scheppner CBM

### FORM

FORM ID: ACBM (Amiga Contiguous BitMap)

FORM Description:

FORM ACBM has the same format as FORM ILBM except the normal BODY chunk (InterLeaved BitMap) is replaced by an ABIT chunk (Amiga BITplanes).

FORM Purpose:

To enable faster loading/saving of screens, especially from Basic, while retaining the flexibility and portability of IFF format files.

### CHUNKS

Chunk ID: ABIT (Amiga BITplanes)

Chunk Description:

The ABIT chunk contains contiguous bitplane data. The chunk contains sequential data for bitplane 0 through bitplane n.

Chunk Purpose:

To enable loading/storing of bitmaps with one DOS Read/Write per bitplane. Significant speed increases are realized when loading/saving screens from Basic.

### SUPPORTING SOFTWARE

(Public Domain, available soon via Fish PD disk, various networks)

LoadILBM-SaveACBM (AmigaBasic)
Loads and displays an IFF ILBM pic file (Graphicraft, DPaint, Images). Optionally saves the screen in ACBM format.

LoadACBM (AmigaBasic)
Loads and display an ACBM format pic file.



SaveILBM (AmigaBasic)
Saves a demo screen as an ILBM pic file which can be loaded into Graphicraft, DPaint, Images.

New FORM: ANIM (for CEL Animations)

The ANIM Short Delta mode is used in Videoscape-3D (Aegis).

Additional modes may be added to ANIM in the future. Any additions to the ANIM spec will be posted on BIX in amiga.dev/iff. An ANIM player has been posted on BIX in the listings area (ANIM.arc).

# A N I M
## An IFF Format For CEL Animations

prepared by:
Sparta Inc.
23041 de la Carlota
Laguna Hills, Calif 92653
714) 768-8161
contact: Gary Bonham

also by:
Aegis Development Co.
2115 Pico Blvd.
Santa Monica, Calif 90405
213) 392-9972

## 1.0 Introduction

The ANIM IFF format was developed at Sparta originally for the production of animated video sequences on the Amiga computer. The intent was to be able to store, and play back, sequences of frames and to minimize both the storage space on disk (through compression) and playback time (through efficient de-compression algorithms). It was desired to maintain maximum compatibility with existing IFF formats and to be able to display the initial frame as a normal still IFF picture.

The basic ANIM format described here has been in use for over a year in-house at Sparta with the XOR mode. The delta mode is a recent, and very effective, addition/improvement.

### 1.1 IFF File Format Overview

The general philosophy of ANIMs is to present the initial frame as a normal, run-length-encoded, IFF picture. Subsequent frames are then described by listing only their differences from a previous frame. Normally, the "previous" frame is two frames back as that is the frame remaining in the hidden screen buffer when double-buffering is used. To better understand this, suppose one has two screens, called A and B, and the ability to instantly switch the display from one to the other. The normal playback mode is to load the initial frame into A and duplicate it into B. Then frame A is displayed on the screen. Then the differences for frame 2 are used to alter screen B and it is displayed. Then the differences for frame 3 are used to alter screen A and it is displayed, and so on. Note that frame 2 is stored as differences from frame 1, but all other frames are stored as differences from two frames



back.

ANIM is an IFF FORM and its basic format is as follows (this assumes the reader has a basic understanding of IFF format files):

```
FORM ANIM
. FORM ILBM          first frame
. . BMHD                 normal type IFF data
. . CMAP
. . BODY
. FORM ILBM          frame 2
. . ANHD                 animation header chunk
. . DLTA                 delta mode data
. FORM ILBM          frame 3
. . ANHD
. . DLTA
    ...
```

The initial FORM ILBM can contain all the normal ILBM chunks, such as CRNG, etc. The BODY will normally be a standard run-length-encoded data chunk (but may be any other legal compression mode as indicated by the BMHD).

The subsequent FORMs ILBM contain an ANHD, instead of a BMHD, which duplicates some of BMHD and has additional parameters pertaining to the animation frame. The DLTA chunk contains the data for the two available delta compression modes. If the older XOR compression mode is used, then a BODY chunk will be here. In addition, other chunks may be placed in each of these as deemed necessary (and as code is placed in player programs to utilize them). A good example would be CMAP chunks to alter the color palette. A basic assumption in ANIMs is that the size of the bitmap, and the display mode (e.g. HAM) will not change through the animation.

### 1.2 Recording ANIMs

To record an ANIM will require three bitmaps - one for creation of the next frame, and two more for a "history" of the previous two frames for performing the compression calculations (e.g. the delta mode calculations).

There are three frame-to-frame compression methods currently defined:

#### 1.2.1 XOR mode

This mode is the original and is included here for compatibility with some programs which still can output this mode. In general, the delta modes are far superior. The creation of XOR mode is quite simple. One simply performs an exclusive-or (XOR) between all corresponding bytes of the new frame and two frames back. This results in a new bitmap with 0 bits wherever the two frames were identical, and 1 bits where they are different. Then this new bitmap is saved using run-length-encoding. A major

obstacle of this mode is in the time consumed in performing the XOR upon reconstructing the image.

### 1.2.2 Long Delta mode

This mode stores the actual new frame long-words which are different, along with the offset in the bitmap. The exact format is shown and discussed in section 2 below. Each plane is handled separately, with no data being saved if no changes take place in a given plane. Strings of 2 or more long-words in a row which change can be run together so offsets do not have to be saved for each one.

Constructing this data chunk usually consists of having a buffer to hold the data, and calculating the data as one compares the new frame, long-word by long-word, with two frames back.

### 1.2.3 Short Delta mode

This mode is identical to the Long Delta mode except that short-words are saved instead of long-words. In most instances, this mode results in a smaller DLTA chunk. The Long Delta mode is mainly of interest in improving the playback speed when used on a 32-bit 68020 Turbo Amiga.

## 1.3 Playing ANIMs

Playback of ANIMs will usually require two buffers, as mentioned above, and double-buffering between them. The frame data from the ANIM file is used to modify the hidden frame to the next frame to be shown. When using the XOR mode, the usual run-length-decoding routine can be easily modified to do the exclusive-or operation required. Note that runs of zero bytes, which will be very common, can be ignored, as an exclusive or of any byte value to a byte of zero will not alter the original byte value.

# 2.0 Chunk Formats

## 2.1 ANHD Chunk

The ANHD chunk consists of the following data structure:

```
UBYTE operation (=0 set directly,
                 =1 XOR mode,
                 =2 Long Delta mode,
                 =3 Short Delta mode)
UBYTE mask      (XOR mode only - plane mask where each
                 bit is set =1 if there is data and =0
                 if not.)
UWORD w,h       (XOR mode only - width and height of the
                 area represented by the BODY to eliminate
                 unnecessary un-changed data)
WORD  x,y       (XOR mode only - position of rectangular
                 area representd by the BODY)
ULONG abstime   (currently unused - timing for a frame
                 relative to the time the first frame
```



```
                 was displayed - in jiffies (1/60 sec))
ULONG reltime   (timing for frame relative to time
                 previous frame was displayed - in
                 jiffies (1/60 sec))
UBYTE interleave (unused so far - indicates how may frames
                 back this data is to modify. =0 defaults
                 to indicate two frames back (for double
                 buffering). =n indicates n frames back.
                 The main intent here is to allow values
                 of =1 for special applications where
                 frame data would modify the immediately
                 previous frame)
UBYTE pad[21]   (this is a pad for future use for future
                 compression modes. Some of these - maybe
                 16 - are intentionally provided for use
                 by Jim Kent for operation codes for each
                 plane and other ancillary data he
                 requested)
```

## 2.2 DLTA Chunk

This chunk is a basic data chunk used to hold the delta compression data. The minimum size of this chunk is 32 bytes as the first 8 long-words are byte pointers into the chunk for the data for each of up to 8 bitplanes. The pointer for the plane data starting immediately following these 8 pointers will have a value of 32 as the data starts in the 33-rd byte of the chunk (indes value of 32 due to zero-base indexing).

The data for a given plane consists of groups of data words. In Long Delta mode, these groups consist of both short and long words - short words for offsets and numbers, and long words for the actual data. In Short Delta mode, the groups are identical except data words are also shorts so all data is short words. Each group consists of a starting word which is an offset. If the offset is positive then it indicates the increment in long or short words (whichever is appropriate) through the bitplane. In other words, if you were reconstructing the plane, you would start a pointer (to shorts or longs depending on the mode) to point to the first word of the bitplane. Then the offset would be added to it and the following data word would be placed at that position. Then the next offset would be added to the pointer and the following data word would be placed at that position. And so on... The data terminates with an offset equal to 0xFFFF.

A second interpretation is given if the offset is negative. In that case, the absolute value is the offset+2. Then the following short-word indicates the number of data words that follow. Following that is the indicated number of contiguous data words (longs or shorts depending on mode) which are to be placed in contiguous locations in the bitplane.

If there are no changed words in a given plane, then the pointer in the first 32 bytes of the chunk is =0.

## IFF FORM / CHUNK DESCRIPTION

Form/Chunk ID: Chunk DPPV (DPaint II ILBM perspective chunk)

Date Submitted: 12/86 (?)
Submitted by: Dan Silva

Chunk Description:

The DPPV chunk describes the perspective state of a DPaint ILBM.

Chunk Purpose:

For reinstatement of the perspective state.

Chunk Spec:

```
/* The chunk identifier DPPV */
#define ID_DPPV    MakeID('D','P','P','V')

typedef LONG LongFrac;
typedef struct ( LongFrac x,y,z; )  LFPoint;
typedef LongFrac  APoint[3];

typedef union {
   LFPoint l;
   APoint  a;
   } UPoint;

/* values taken by variable rotType */
#define ROT_EULER  0
#define ROT_INCR   1

/* Disk record describing Perspective state */

typedef struct {
   WORD     rotType;           /* rotation type */
   WORD     iA, iB, iC;        /* rotation angles (in degrees) */
   LongFrac Depth;             /* perspective depth */
   WORD     uCenter, vCenter;  /* coords of center perspective,
                                * relative to backing bitmap,
                                * in Virtual coords
                                */
   WORD     fixCoord;          /* which coordinate is fixed */
   WORD     angleStep;         /* large angle stepping amount */
   UPoint   grid;              /* gridding spacing in X,Y,Z */
   UPoint   gridReset;         /* where the grid goes on Reset */
   UPoint   gridBrCenter;      /* Brush center when gris was last on,
                                * as reference point
                                */
   UPoint   permBrCenter;      /* Brush center the last time the mouse
                                * button was clicked, a rotation performed,
                                * or motion along "fixed" axis
                                */
```



```
   LongFrac rot[3][3];         /* rotation matrix */
   } PerspState;
```

## SUPPORTING SOFTWARE

DPaint II by Dan Silva for Electronic Arts

TITLE: HEAD (FORM used by Flow - New Horizons Software, Inc.)

## IFF FORM / CHUNK DESCRIPTION

Form/Chunk ID: FORM HEAD, Chunks NEST, TEXT, FSCC

Date Submitted: 03/87
Submitted by: James Bayless - New Horizons Software, Inc.

## FORM

FORM ID: HEAD

FORM Description:

FORM HEAD is the file storage format of the Flow idea processor by New Horizons Software, Inc. Currently only the TEXT and NEST chunks are used. There are plans to incorporate FSCC and some additional chunks for headers and footers.

## CHUNKS

CHUNK ID: NEST

This chunk consists of only of a word (two byte) value that gives the new current nesting level of the outline. The initial nesting level (outermost level) is zero. It is necessary to include a NEST chunk only when the nesting level changes. Valid changes to the nesting level are either to decrease the current value by any amount (with a minimum of 0) or to increase it by one (and not more than one).

CHUNK ID: TEXT

This chunk is the actual text of a heading. Each heading has a TEXT chunk (even if empty). The text is not NULL terminated - the chunk size gives the length of the heading text.

CHUNK ID: FSCC

This chunk gives the Font/Style/Color changes in the heading from the most recent TEXT chunk. It should occur immediately after the TEXT chunk it modifies. The format is identical to the FSCC chunk for the IFF form type 'WORD' (for compatibility), except that only the 'Location' and 'Style' values are used (i.e., there can be currently only be style changes in an outline heading). The structure definition is:

```
typedef struct {
   UWORD   Location;   /* Char location of change */
   UBYTE   FontNum;    /* Ignored */
```



```
   UBYTE   Style;      /* Amiga style bits */
   UBYTE   MiscStyle;  /* Ignored */
   UBYTE   Color;      /* Ignored */
   UWORD   pad;        /* Ignored */
} FSCChange;
```

The actual chunk consists of an array of these structures, one entry for each Style change in the heading text.

TITLE:  WORD  (word processing FORM used by ProWrite)

IFF FORM / CHUNK DESCRIPTION

Form/Chunk IDs:
   FORM   WORD
   Chunks FONT,COLR,DOC,HEAD,FOOT,PCTS,PARA,TABS,PAGE,TEXT,FSCC,PINF

Date Submitted: 03/87
Submitted by:  James Bayless - New Horizons Software, Inc.

FORM

FORM ID:  WORD

FORM Purpose:  Document storage (supports color, fonts, pictures)

FORM Description:

This include file describes FORM WORD and its Chunks

```
/*
 *      IFF Form WORD structures and defines
 *      Copyright (c) 1987 New Horizons Software, Inc.
 *
 *      Permission is hereby granted to use this file in any and all
 *      applications.  Modifying the structures or defines included
 *      in this file is not permitted without written consent of
 *      New Horizons Software, Inc.
 */

#include ":IFF/ILBM.h"        /* Makes use of ILBM defines */

#define ID_WORD      MakeID('W','O','R','D')      /* Form type */

#define ID_FONT      MakeID('F','O','N','T')      /* Chunks */
#define ID_COLR      MakeID('C','O','L','R')
#define ID_DOC       MakeID('D','O','C',' ')
#define ID_HEAD      MakeID('H','E','A','D')
#define ID_FOOT      MakeID('F','O','O','T')
#define ID_PCTS      MakeID('P','C','T','S')
#define ID_PARA      MakeID('P','A','R','A')
#define ID_TABS      MakeID('T','A','B','S')
#define ID_PAGE      MakeID('P','A','G','E')
#define ID_TEXT      MakeID('T','E','X','T')
#define ID_FSCC      MakeID('F','S','C','C')
#define ID_PINF      MakeID('P','I','N','F')

/*
 *   Special text characters for page number, date, and time
 *   Note:  ProWrite currently supports only PAGENUM_CHAR, and only in
 *      headers and footers
 */
```



```
#define PAGENUM_CHAR    0x80
#define DATE_CHAR       0x81
#define TIME_CHAR       0x82

/*
 *   Chunk structures follow
 */

/*
 *   FONT - Font name/number table
 *   There are one of these for each font/size combination
 *   These chunks should appear at the top of the file (before document data)
 */

typedef struct {
   UBYTE   Num;          /* 0 .. 255 */
   UWORD   Size;
/* UBYTE   Name[];       */   /* NULL terminated, without ".font" */
} FontID;

/*
 *   COLR - Color translation table
 *   Translates from color numbers used in file to ISO color numbers
 *   Should be at top of file (before document data)
 *   Note:  Currently ProWrite only checks these values to be its current map,
 *      it does no translation as it does for FONT chunks
 */

typedef struct {
   UBYTE   ISOColors[8];
} ISOColors;

/*
 *   DOC - Begin document section
 *   All text and paragraph formatting following this chunk and up to a
 *   HEAD, FOOT, or PICT chunk belong to the document section
 */

#define PAGESTYLE_1   0       /* 1, 2, 3 */
#define PAGESTYLE_I   1       /* I, II, III */
#define PAGESTYLE_i   2       /* i, ii, iii */
#define PAGESTYLE_A   3       /* A, B, C */
#define PAGESTYLE_a   4       /* a, b, c */

typedef struct {
   UWORD   StartPage;       /* Starting page number */
   UBYTE   PageNumStyle;    /* From defines above */
   UBYTE   pad1;
   LONG    pad2;
} DocHdr;

/*
 *   HEAD/FOOT - Begin header/footer section
 *   All text and paragraph formatting following this chunk and up to a
 *   DOC, HEAD, FOOT, or PICT chunk belong to this header/footer
```

```
 *   Note:  This format supports multiple headers and footers, but currently
 *      ProWrite only allows a single header and footer per document
 */

#define PAGES_NONE    0
#define PAGES_LEFT    1
#define PAGES_RIGHT   2
#define PAGES_BOTH    3

typedef struct {
   UBYTE   PageType;        /* From defines above */
   UBYTE   FirstPage;       /* 0 = Not on first page */
   LONG    pad;
} HeadHdr;

/*
 *   PCTS - Begin picture section
 *   Note:  ProWrite currently requires NPlanes to be three (3)
 */

typedef struct {
   UBYTE   NPlanes;       /* Number of planes used in picture bitmaps */
   UBYTE   pad;
} PictHdr;

/*
 *   PARA - New paragraph format
 *   This chunk should be inserted first when a new section is started (DOC,
 *      HEAD, or FOOT), and again whenever the paragraph format changes
 */

#define SPACE_SINGLE    0
#define SPACE_DOUBLE    0x10

#define JUSTIFY_LEFT    0
#define JUSTIFY_CENTER  1
#define JUSTIFY_RIGHT   2
#define JUSTIFY_FULL    3

#define MISCSTYLE_NONE   0
#define MISCSTYLE_SUPER  1      /* Superscript */
#define MISCSTYLE_SUB    2      /* Subscript */

typedef struct {
   UWORD   LeftIndent;     /* In decipoints (720 dpi) */
   UWORD   LeftMargin;
   UWORD   RightMargin;
   UBYTE   Spacing;        /* From defines above */
   UBYTE   Justify;        /* From defines above */
   UBYTE   FontNum;        /* FontNum, Style, etc. for first char in para*/
   UBYTE   Style;          /* Standard Amiga style bits */
   UBYTE   MiscStyle;      /* From defines above */
   UBYTE   Color;          /* Internal number, use COLR to translate */
   LONG    pad;
} ParaFormat;
```



```
/*
 *   TABS - New tab stop types/locations
 *   Use an array of values in each chunk
 *   Like the PARA chunk, this should be inserted whenever the tab settings
 *      for a paragraph change
 *   Note:  ProWrite currently does not support TAB_CENTER
 */

#define TAB_LEFT     0
#define TAB_CENTER   1
#define TAB_RIGHT    2
#define TAB_DECIMAL  3

typedef struct {
   UWORD   Position;       /* In decipoints */
   UBYTE   Type;
   UBYTE   pad;
} TabStop;

/*
 *   PAGE - Page break
 *   Just a marker -- this chunk has no data
 */

/*
 *   TEXT - Paragraph text (one block per paragraph)
 *   Block is actual text, no need for separate structure
 *   If the paragraph is empty, this is an empty chunk -- there MUST be
 *   a TEXT block for every paragraph
 *   Note:  The only ctrl characters ProWrite can currently handle in TEXT
 *   chunks are Tab and PAGENUM_CHAR, ie no Return's, etc.
 */

/*
 *   FSCC - Font/Style/Color changes in previous TEXT block
 *   Use an array of values in each chunk
 *   Only include this chunk if the previous TEXT block did not have
 *      the same Font/Style/Color for all its characters
 */

typedef struct {
   UWORD   Location;       /* Character location in TEXT chunk of change */
   UBYTE   FontNum;
   UBYTE   Style;
   UBYTE   MiscStyle;
   UBYTE   Color;
   UWORD   pad;
} FSCChange;

/*
 *   PINF - Picture info
 *   This chunk must only be in a PCTS section
 *   Must be followed by ILBM BODY chunk
 *   Pictures are treated independently of the document text (like a
 *      page-layout system), this chunk includes information about what
 *      page and location on the page the picture is at
```

```
 *   Note:  ProWrite currently only supports mskTransparentColor and
 *      mskHasMask masking
 */

typedef struct {
   UWORD          Width, Height;    /* In pixels */
   UWORD          Page;          /* Which page picture is on (0..max) */
   UWORD          XPos, YPos;       /* Location on page in decipoints */
   Masking        Masking;       /* Like ILBM format */
   Compression    Compression;    /* Like ILBM format */
   UBYTE          TransparentColor;    /* Like ILBM format */
   UBYTE          pad;
} PictInfo;

/* end */
```

Electronic Arts has defined two more SEvents in SMUS:

| SID Value | Next Data Byte |
|---|---|
| #define SID_Clef 135 | 0=treble, 1=bass, 2=alto, 3=tenor |
| #define SID_Tempo 136 | beats per minute (0-255) |

```
/*------------------------------------------------------------------------*
 * SMUS.H  Definitions for Simple MUSical score.   2/12/86
 *
 * By Jerry Morrison and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *------------------------------------------------------------------------*/
#ifndef SMUS_H
#define SMUS_H

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#include "iff/iff.h"

#define ID_SMUS       MakeID('S', 'M', 'U', 'S')
#define ID_SHDR       MakeID('S', 'H', 'D', 'R')
#define ID_NAME       MakeID('N', 'A', 'M', 'E')
#define ID_Copyright  MakeID('(', 'c', ')', ' ')
#define ID_AUTH       MakeID('A', 'U', 'T', 'H')
#define ID_ANNO       MakeID('A', 'N', 'N', 'O')

#define ID_INS1       MakeID('I', 'N', 'S', '1')
#define ID_TRAK       MakeID('T', 'R', 'A', 'K')

/* ---------- SScoreHeader ------------------------------------------------*/
typedef struct {
    UWORD tempo;        /* tempo, 128ths quarter note/minute */
    UBYTE volume;       /* playback volume 0 through 127 */
    UBYTE ctTrack;      /* count of tracks in the score */
    } SScoreHeader;

/* ---------- NAME --------------------------------------------------------*/
/* NAME chunk contains a CHAR[], the musical score's name. */

/* ---------- Copyright (c) -----------------------------------------------*/
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

/* ---------- AUTH --------------------------------------------------------*/
/* AUTH chunk contains a CHAR[], the name of the score's author. */

/* ---------- ANNO --------------------------------------------------------*/
/* ANNO chunk contains a CHAR[], the author's text annotations. */

/* ---------- INS1 --------------------------------------------------------*/
/* Constants for the RefInstrument's "type" field. */
#define INS1_Name  0    /* just use the name; ignore data1, data2 */
#define INS1_MIDI  1    /* <data1, data2> = MIDI <channel, preset> */

typedef struct {
    UBYTE iRegister;    /* set this instrument register number */
    UBYTE type;         /* instrument reference type (see above) */
    UBYTE data1, data2; /* depends on the "type" field */
    char name[60];      /* instrument name */
```



```
    } RefInstrument;

/* ---------- TRAK --------------------------------------------------------*/
/* TRAK chunk contains an SEvent[]. */

/* SEvent: Simple musical event. */
typedef struct {
    UBYTE sID;          /* SEvent type code */
    UBYTE data;         /* sID-dependent data */
    } SEvent;

/* SEvent type codes "sID". */
#define SID_FirstNote     0
#define SID_LastNote    127     /* sIDs in the range SID_FirstNote through
                                 * SID_LastNote (sign bit = 0) are notes. The
                                 * sID is the MIDI tone number (pitch). */
#define SID_Rest        128     /* a rest; same data format as a note. */

#define SID_Instrument  129     /* set instrument number for this track. */
#define SID_TimeSig     130     /* set time signature for this track. */
#define SID_KeySig      131     /* set key signature for this track. */
#define SID_Dynamic     132     /* set volume for this track. */
#define SID_MIDI_Chnl   133     /* set MIDI channel number (sequencers) */
#define SID_MIDI_Preset 134     /* set MIDI preset number (sequencers) */
#define SID_Clef        135     /* inline clef change.
                                 * 0=Treble, 1=Bass, 2=Alto, 3=Tenor. */
#define SID_Tempo       136     /* Inline tempo change in beats per minute.*/

/* SID values 144 through 159: reserved for Instant Music SEvents. */

/* The remaining sID values up through 254: reserved for future
 * standardization. */
#define SID_Mark        255     /* SID reserved for an end-mark in RAM. */

/* ---------- SEvent FirstNote..LastNote or Rest --------------------------*/
typedef struct {
    unsigned tone     :8,       /* MIDI tone number 0 to 127; 128 = rest */
             chord    :1,       /* 1 = a chorded note */
             tieOut   :1,       /* 1 = tied to the next note or chord */
             nTuplet  :2,       /* 0 = none, 1 = triplet, 2 = quintuplet,
                                 * 3 = septuplet */
             dot      :1,       /* dotted note; multiply duration by 3/2 */
             division :3;       /* basic note duration is 2**-division:
                                 * 0 = whole note, 1 = half note, 2 = quarter
                                 * note, ... 7 = 128th note */
    } SNote;

/* Warning: An SNote is supposed to be a 16-bit entity.
 * Some C compilers will not pack bit fields into anything smaller
 * than an int. So avoid the actual use of this type unless you are certain
 * that the compiler packs it into a 16-bit word.
 */

/* You may get better object code by masking, ORing, and shifting using the
 * following definitions rather than the bit-packed fields, above. */
#define noteChord  (1<<7)       /* note is chorded to next note */
```

```
#define noteTieOut (1<<6)       /* note/chord is tied to next note/chord */

#define noteNShift 4                    /* shift count for nTuplet field */
#define noteN3     (1<<noteNShift)      /* note is a triplet */
#define noteN5     (2<<noteNShift)      /* note is a quintuplet */
#define noteN7     (3<<noteNShift)      /* note is a septuplet */
#define noteNMask  noteN7               /* bit mask for the nTuplet field */

#define noteDot    (1<<3)               /* note is dotted */

#define noteDShift 0                    /* shift count for division field */
#define noteD1     (0<<noteDShift)      /* whole note division */
#define noteD2     (1<<noteDShift)      /* half note division */
#define noteD4     (2<<noteDShift)      /* quarter note division */
#define noteD8     (3<<noteDShift)      /* eighth note division */
#define noteD16    (4<<noteDShift)      /* sixteenth note division */
#define noteD32    (5<<noteDShift)      /* thirty-secondth note division */
#define noteD64    (6<<noteDShift)      /* sixty-fourth note division */
#define noteD128   (7<<noteDShift)      /* 1/128 note division */
#define noteDMask  noteD128             /* bit mask for the division field */

#define noteDurMask 0x3F                /* bit mask for all duration fields
                                         * division, nTuplet, dot */

/* Field access: */
#define IsChord(snote)   (((UWORD)snote) & noteChord)
#define IsTied(snote)    (((UWORD)snote) & noteTieOut)
#define NTuplet(snote)   ((((UWORD)snote) & noteNMask) >> noteNShift)
#define IsDot(snote)     (((UWORD)snote) & noteDot)
#define Division(snote)  ((((UWORD)snote) & noteDMask) >> noteDShift)

/* ---------- TimeSig SEvent -------------------------------------------*/
typedef struct {
    unsigned type     :8,       /* = SID_TimeSig */
             timeNSig :5,       /* time signature "numerator" timeNSig + 1 */
             timeDSig :3;       /* time signature "denominator" is
                                 * 2**timeDSig: 0 = whole note, 1 = half
                                 * note, 2 = quarter note, ...
                                 * 7 = 128th note */
    } STimeSig;

#define timeNMask  0xF8         /* bit mask for timeNSig field */
#define timeNShift 3            /* shift count for timeNSig field */

#define timeDMask  0x07         /*  bit mask for timeDSig field */

/* Field access: */
#define TimeNSig(sTime)   ((((UWORD)sTime) & timeNMask) >> timeNShift)
#define TimeDSig(sTime)    (((UWORD)sTime) & timeDMask)

/* ---------- KeySig SEvent --------------------------------------------*/
/* "data" value 0 = Cmaj; 1 through 7 = G,D,A,E,B,F#,C#;
 * 8 through 14 = F,Bb,Eb,Ab,Db,Gb,Cb.                                  */

/* ---------- Dynamic SEvent -------------------------------------------*/
```



```
/* "data" value is a MIDI key velocity 0..127. */


/* ---------- SMUS Reader Support Routines -----------------------------*/

/* Just call this to read a SHDR chunk. */
#define GetSHDR(context, ssHdr)  \
    IFFReadBytes(context, (BYTE *)ssHdr, sizeof(SScoreHeader))

/* ---------- SMUS Writer Support Routines -----------------------------*/

/* Just call this to write a SHDR chunk. */
#define PutSHDR(context, ssHdr)  \
    PutCk(context, ID_SHDR, sizeof(SScoreHeader), (BYTE *)ssHdr)

#endif
```

```
/*****************************************************************************
 * 8SVX.H  Definitions for 8-bit sampled voice (VOX).   2/10/86
 *
 * By Jerry Morrison and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *****************************************************************************/
#ifndef EIGHTSVX_H
#define EIGHTSVX_H

#ifndef COMPILER_H
#include "iff/compiler.h"
#endif

#include "iff/iff.h"

#define ID_8SVX      MakeID('8', 'S', 'V', 'X')
#define ID_VHDR      MakeID('V', 'H', 'D', 'R')
#define ID_NAME      MakeID('N', 'A', 'M', 'E')
#define ID_Copyright MakeID('(', 'c', ')', ' ')

#define ID_AUTH      MakeID('A', 'U', 'T', 'H')
#define ID_ANNO      MakeID('A', 'N', 'N', 'O')

#define ID_BODY      MakeID('B', 'O', 'D', 'Y')

#define ID_ATAK      MakeID('A', 'T', 'A', 'K')
#define ID_RLSE      MakeID('R', 'L', 'S', 'E')

/* ---------- Voice8Header ------------------------------------------------*/
typedef LONG Fixed;     /* A fixed-point value, 16 bits to the left of
                         * the point and 16 to the right. A Fixed is a
                         * number of 2**16ths, i.e. 65536ths. */
#define Unity 0x10000L  /* Unity = Fixed 1.0 = maximum volume */

/* sCompression: Choice of compression algorithm applied to the samples. */
#define sCmpNone       0        /* not compressed */
#define sCmpFibDelta   1        /* Fibonacci-delta encoding (Appendix C) */
                                /* Could be more kinds in the future. */
typedef struct {
    ULONG oneShotHiSamples,     /* # samples in the high octave 1-shot part */
          repeatHiSamples,      /* # samples in the high octave repeat part */
          samplesPerHiCycle;    /* # samples/cycle in high octave, else 0 */
    UWORD samplesPerSec;        /* data sampling rate */
    UBYTE ctOctave,             /* # of octaves of waveforms */
          sCompression;         /* data compression technique used */
    Fixed volume;               /* playback nominal volume from 0 to Unity
                                 * (full volume). Map this value into
                                 * the output hardware's dynamic range.
                                 */
    } Voice8Header;

/* ---------- NAME --------------------------------------------------------*/
/* NAME chunk contains a CHAR[], the voice's name. */
```



```
/* ---------- Copyright ---------------------------------------------------*/
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

/* ---------- AUTH --------------------------------------------------------*/
/* AUTH chunk contains a CHAR[], the author's name. */

/* ---------- ANNO --------------------------------------------------------*/
/* ANNO chunk contains a CHAR[], the author's text annotations. */

/* ---------- Envelope ATAK & RLSE ----------------------------------------*/
typedef struct {
    UWORD duration;     /* segment duration in milliseconds, > 0 */
    Fixed dest;         /* destination volume factor */
    } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope. */

/* The envelope defines a function of time returning Fixed values.
 * It's used to scale the nominal volume specified in the Voice8Header.
 */

/* ---------- BODY --------------------------------------------------------*/
/* BODY chunk contains a BYTE[], array of audio data samples. */
/* (8-bit signed numbers, -128 through 127.) */


/* ---------- 8SVX Reader Support Routines --------------------------------*/

/* Just call this macro to read a VHDR chunk. */
#define GetVHDR(context, vHdr)  \
    IFFReadBytes(context, (BYTE *)vHdr, sizeof(Voice8Header))


/* ---------- 8SVX Writer Support Routines --------------------------------*/

/* Just call this macro to write a VHDR chunk. */
#define PutVHDR(context, vHdr)  \
    PutCk(context, ID_VHDR, sizeof(Voice8Header), (BYTE *)vHdr)

#endif
```

Example linkage for Read8svx.c.  Note that iffr.o is the object of
one of the standard IFF reader modules.  For this link, all modules
must be compiled with the -v flag on LC2.

```
FROM     LIB:AStartup.obj, Read8svx.o, dUnpack.o, iffr.o
LIBRARY  LIB:Amiga.lib
TO       Read8svx
```

```
/** Read8SVX.c ************************************************************
 *
 * Read a sound sample from an IFF file.  21Jan85
 *
 * By Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 ************************************************************************/

#include "exec/types.h"
#include "exec/exec.h"
#include "libraries/dos.h"
#include "iff/8svx.h"

/* Message strings for IFFP codes. */
char MsgOkay[]        = { "(IFF_OKAY) No FORM 8SVX in the file." };
char MsgEndMark[]     = { "(END_MARK) How did you get this message?" };
char MsgDone[]        = { "(IFF_DONE) All done."};
char MsgDos[]         = { "(DOS_ERROR) The DOS returned an error." };
char MsgNot[]         = { "(NOT_IFF) Not an IFF file." };
char MsgNoFile[]      = { "(NO_FILE) No such file found." };
char MsgClientError[] = { "(CLIENT_ERROR) Read8SVX bug or insufficient RAM."};
char MsgForm[]        = { "(BAD_FORM) A malformed FORM 8SVX." };
char MsgShort[]       = { "(SHORT_CHUNK) A malformed FORM 8SVX." };
char MsgBad[]         = { "(BAD_IFF) A mangled IFF file." };

/* THESE MUST APPEAR IN RIGHT ORDER!! */
char *IFFPMessages[-LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };

typedef struct {
   ClientFrame clientFrame;
   UBYTE foundVHDR;
   UBYTE pad1;
   Voice8Header sampHdr;
   } SVXFrame;


/* NOTE: For a simple version of this program, set Fancy to 0.
 * That'll compile a program that skips all LISTs and PROPs in the input
 * file. It will look in CATs for FORMs 8SVX. That's suitable for most uses.
 *
 * For a fancy version that handles LISTs and PROPs, set Fancy to 1. */

#define Fancy  1
```



```
BYTE *buf;
int szBuf;

/** DoSomethingWithSample() ***********************************************
 *
 * Interface to Amiga sound driver.
 *
 ************************************************************************/
DoSomethingWithSample(sampHdr)  Voice8Header *sampHdr;  {
    BYTE *t;
    printf("\noneShotHiSamples=%ld", sampHdr->oneShotHiSamples);
    printf("\nrepeatHiSamples=%ld", sampHdr->repeatHiSamples);
    printf("\nsamplesPerHiCycle=%ld", sampHdr->samplesPerHiCycle);
    printf("\nsamplesPerSec=%ld", sampHdr->samplesPerSec);
    printf("\nctOctave=%ld", sampHdr->ctOctave);
    printf("\nsCompression=%ld", sampHdr->sCompression);
    printf("\nvolume=0x%lx", sampHdr->volume);
    /* Decompress, if needed. */
    if (sampHdr->sCompression) {
        t = (BYTE *)AllocMem(szBuf<<1, MEMF_CHIP);
        DUnpack(buf, szBuf, t);
        FreeMem(buf, szBuf);
        buf = t;
        szBuf <<= 1;
        };
    printf("\nData = %3ld %3ld %3ld %3ld %3ld %3ld %3ld %3ld",
           buf[0],buf[1],buf[2],buf[3],buf[4],buf[5],buf[6],buf[7]);
    printf("\n       %3ld %3ld %3ld %3ld %3ld %3ld %3ld %3ld ...\n",
           buf[8+0],buf[8+1],buf[8+2],buf[8+3],buf[8+4],buf[8+5],
           buf[8+6],buf[8+ 7]);
    }

/** ReadBODY() ************************************************************
 *
 * Read a BODY into RAM.
 *
 ************************************************************************/
IFFP ReadBODY(context)  GroupContext *context;  {
    IFFP iffp;

    szBuf  = ChunkMoreBytes(context);
    buf  = (BYTE *)AllocMem(szBuf, MEMF_CHIP);
    if (buf == NULL)
        iffp = CLIENT_ERROR;
    else
        iffp = IFFReadBytes(context, (BYTE *)buf, szBuf);
    CheckIFFP();
    }

/** GetFo8SVX() ***********************************************************
 *
 * Called via ReadSample to handle every FORM encountered in an IFF file.
 * Reads FORMs 8SVX and skips all others.
 * Inside a FORM 8SVX, it reads BODY. It complains if it
 * doesn't find an VHDR before the BODY.
```

```
 *
 * [TBD] We could read and print out any NAME and "(c) " chunks.
 *
 ****************************************************************************/
IFFP GetFo8SVX(parent)  GroupContext *parent;  {
   /*compilerBug register*/ IFFP iffp;
   GroupContext formContext;
   SVXFrame smusFrame;          /* only used for non-clientFrame fields.*/

   if (parent->subtype != ID_8SVX)
      return(IFF_OKAY); /* just continue scaning the file */

   smusFrame = *(SVXFrame *)parent->clientFrame;
   iffp = OpenRGroup(parent, &formContext);
   CheckIFFP();

   do switch (iffp = GetFChunkHdr(&formContext)) {
      case ID_VHDR: {
        smusFrame.foundVHDR = TRUE;
        iffp = GetVHDR(&formContext, &smusFrame.sampHdr);
        break; }
      case ID_BODY: {
        if (!smusFrame.foundVHDR)
            iffp = BAD_FORM;            /* Need an VHDR chunk first! */
        else iffp = ReadBODY(&formContext);
        break; }
      case END_MARK: {
        if (!smusFrame.foundVHDR)
            iffp = BAD_FORM;
        else
            iffp = IFF_DONE;
        break; }
      } while (iffp >= IFF_OKAY);  /* loop if valid ID of ignored chunk or a
                              * subroutine returned IFF_OKAY (no errors).*/

    if (iffp != IFF_DONE)  return(iffp);

    /* If we get this far, there were no errors. */
    CloseRGroup(&formContext);
    DoSomethingWithSample(&smusFrame.sampHdr);
    FreeMem(buf, szBuf);
    return(iffp);
    }

/** Notes on extending GetFo8SVX ***************************************
 *
 * To read more kinds of chunks, just add clauses to the switch statement.
 * To read more kinds of property chunks (like NAME) add clauses to
 * the switch statement in GetPr8SVX, too.
 *
 ****************************************************************************/

/** GetPr8SVX() ***************************************************************
 *
 * Called via ReadSample to handle every PROP encountered in an IFF file.
 * Reads PROPs 8SVX and skips all others.
```



```
 *
 ****************************************************************************/
#if Fancy
IFFP GetPr8SVX(parent)  GroupContext *parent;  {
   /*compilerBug register*/ IFFP iffp;
   GroupContext propContext;
   SVXFrame *svxFrame = (SVXFrame *)parent->clientFrame; /* Subclass */

   if (parent->subtype != ID_8SVX)
      return(IFF_OKAY); /* just continue scaning the file */

   iffp = OpenRGroup(parent, &propContext);
   CheckIFFP();

   do switch (iffp = GetPChunkHdr(&propContext)) {
      case ID_VHDR: {
        svxFrame->foundVHDR = TRUE;
        iffp = GetVHDR(&propContext, &svxFrame->sampHdr);
        break; }
      } while (iffp >= IFF_OKAY);  /* loop if valid ID of ignored chunk or a
                              * subroutine returned IFF_OKAY (no errors).*/

   CloseRGroup(&propContext);
   return(iffp == END_MARK ? IFF_OKAY : iffp);
   }
#endif

/** GetLi8SVX() ***************************************************************
 *
 * Called via ReadSample to handle every LIST encountered in an IFF file.
 *
 ****************************************************************************/
#if Fancy
IFFP GetLi8SVX(parent)  GroupContext *parent;  {
    SVXFrame newFrame;  /* allocate a new Frame */

    newFrame = *(SVXFrame *)parent->clientFrame;  /* copy parent frame */

    return( ReadIList(parent, (ClientFrame *)&newFrame) );
    }
#endif

/** ReadSample() **************************************************************
 *
 * Read IFF 8SVX, given a file handle open for reading.
 *
 ****************************************************************************/
IFFP ReadSample(file)  LONG file;  {
   SVXFrame sFrame;     /* Top level "client frame".*/
   IFFP iffp = IFF_OKAY;

#if Fancy
   sFrame.clientFrame.getList = GetLi8SVX;
   sFrame.clientFrame.getProp = GetPr8SVX;
#else
   sFrame.clientFrame.getList = SkipGroup;
```

```
   sFrame.clientFrame.getProp = SkipGroup;
#endif
   sFrame.clientFrame.getForm = GetFo8SVX;
   sFrame.clientFrame.getCat  = ReadICat ;

   /* Initialize the top-level client frame's property settings to the
    * program-wide defaults. This example just records that we haven't read
    * any VHDR properties yet.
    * If you want to read another property, init it's fields in sFrame. */
   sFrame.foundVHDR = FALSE;
   sFrame.pad1      = 0;

   iffp = ReadIFF(file, (ClientFrame *)&sFrame);

   return(iffp);
   }

/** main0() ***************************************************************/
void main0(filename)  char *filename;  {
    LONG file;
    IFFP iffp = NO_FILE;
    file = Open(filename, MODE_OLDFILE);
    if (file)
        iffp = ReadSample(file);
    Close(file);
    printf(" %s\n", IFFPMessages[-iffp]);
    }

/** main() ****************************************************************/
void main(argc, argv)  int argc;  char **argv;  {
    printf("Reading file '%s' ...", argv[1]);
    if (argc < 2)
        printf("\nfilename required\n");
    else
        main0(argv[1]);
    }
```

```
/* DUnpack.c --- Fibonacci Delta decompression by Steve Hayes */

#include <exec/types.h>

/* Fibonacci delta encoding for sound data */
BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Unpack Fibonacci-delta encoded data from n byte source
 * buffer into 2*n byte dest buffer, given initial data
 * value x.  It returns the lats data value x so you can
 * call it several times to incrementally decompress the data.
 */

BYTE D1Unpack(source,n,dest,x)
BYTE source[], dest[];
LONG n;
BYTE x;
   {
   BYTE d;
   LONG i, lim;

   lim = n << 1;
   for (i=0; i < lim; ++i)
      {
      /* Decode a data nibble, high nibble then low nibble */
      d = source[i >> 1];     /* get a pair of nibbles */
      if (i & 1)              /* select low or high nibble */
         d &= 0xf;            /* mask to get the low nibble */
      else
         d >>= 4;             /* shift to get the high nibble */
      x += codeToDelta[d];    /* add in the decoded delta */
      dest[i] = x;            /* store a 1 byte sample */
      }
   return(x);
   }

/* Unpack Fibonacci-delta encoded data from n byte
 * source buffer into 2*(n-2) byte dest buffer.
 * Source buffer has a pad byte, an 8-bit initial
 * value, followed by n-2 bytes comprising 2*(n-2)
 * 4-bit encoded samples.
 */

void DUnpack(source, n, dest)
BYTE source[], dest[];
LONG n;
   {
   D1Unpack(source+2, n-2, dest, source[1]);
   }
```

```
/*
 *  ScreenSave.c --  Carolyn Scheppner  CBM  10/86
 *                   Saves front screen as ILBM file
 *                   Saves a CAMG chunk for HAM, etc.
 *                   Creates icon for ILBM file
 *
 *     Uses IFF rtns by J.Morrison and S.Shaw of Electronic Arts
 *
 *       (all C code including IFF modules compiled with -v on LC2)
 * Linkage information:
 * FROM     AStartup.obj, ScreenSave.o, iffw.o, ilbmw.o, packer.o
 * TO       ScreenSave
 * LIBRARY  Amiga.lib, LC.lib
 *
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <graphics/gfxbase.h>
#include <graphics/rastport.h>
#include <graphics/gfx.h>
#include <graphics/view.h>

#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <workbench/workbench.h>
#include <workbench/startup.h>

#include <iff/ilbm.h>

/* From AStartup */
extern  LONG  stdin, stdout, stderr;

/* CAMG Stuff */
typedef struct {
   ULONG ViewModes;
   } CamgChunk;

#define PutCAMG(context, camg)  \
    PutCk(context, ID_CAMG, sizeof(CamgChunk),(BYTE *)camg)

#define bufSize 512

/* Other Stuff */
struct IntuitionBase *IntuitionBase;
struct GfxBase       *GfxBase;
ULONG  IconBase;

struct Screen   *frontScreen;

struct ViewPort *picViewPort;
struct BitMap   *picBitMap;
WORD            *picColorTable;
ULONG            picViewModes;
```



```
BOOL fromWB, newStdio;

#define INBUFSZ 40
char sbuf[INBUFSZ];
char nbuf[INBUFSZ];

char conSpec[] = "CON:0/40/639/160/ ScreenSave ";

/* Definitions for ILBM Icon */
USHORT  ILBMimagedata[] = {
 0xFFFF, 0xFFFC,
 0xC000, 0x000C,
 0xC000, 0x000C,
 0xC1E7, 0x9E0C,
 0xC1F8, 0x7E0C,
 0xC078, 0x780C,
 0xC187, 0x860C,
 0xC078, 0x780C,
 0xC1F8, 0x7E0C,
 0xC1E7, 0x9E0C,
 0xC000, 0x000C,
 0xC000, 0x000C,
 0xFFFF, 0xFFFC,
 0x0000, 0x0000,
 0x0000, 0x0000,
/**/
 0xFFFF, 0xFFFC,
 0xFFFF, 0xFFFC,
 0xF800, 0x007C,
 0xF9E0, 0x1E7C,
 0xF980, 0x067C,
 0xF807, 0x807C,
 0xF81F, 0xE07C,
 0xF807, 0x807C,
 0xF980, 0x067C,
 0xF9E0, 0x1E7C,
 0xF800, 0x007C,
 0xFFFF, 0xFFFC,
 0xFFFF, 0xFFFC,
 0x0000, 0x0000,
 0x0000, 0x0000,
/**/
 };

struct Image ILBMimage = {
   0,0,                         /* Leftedge, Topedge */
   30,15,                       /* Width Height */
   2,                           /* Depth */
   &ILBMimagedata[0],           /* Data for image */
   3,0                          /* PlanePick, PlaneOnOff */
   };

struct DiskObject ILBMobject = {
   WB_DISKMAGIC,
   WB_DISKVERSION,
```

```
    /* Gadget Structure */
    NULL,                       /* Ptr to next gadget */
    0,0,                        /* Leftedge, Topedge */
    30,15,                      /* Width, Height */
    GADGHBOX|GADGIMAGE,         /* Flags */
    RELVERIFY|GADGIMMEDIATE,    /* Activation */
    BOOLGADGET,                 /* Type */
    (APTR)&ILBMimage,           /* Render */
    NULL,                       /* Select Render */
    NULL,                       /* Text */
    NULL,NULL,NULL,NULL,        /* Exclude, Special, ID, UserData */

    4,                          /* WBObject type */
    ":Display",                 /* Default tool */
    NULL,                       /* Tool Types */
    NO_ICON_POSITION,           /* Current X */
    NO_ICON_POSITION,           /* Current Y */
    NULL,NULL,NULL,             /* Drawer, ToolWindow, Stack */
    };


main(argc, argv)
int argc;
char **argv;
    {
    LONG            file;
    IFFP            iffp = NO_FILE;
    char            *filename;
    int l;

    newStdio = FALSE;
    fromWB = (argc==0) ? TRUE : FALSE;

    if((fromWB) && (!(newStdio = openStdio(&conSpec[0]))))
        {
        return(0);
        }

    if ((IntuitionBase =
        (struct IntuitionBase *)OpenLibrary("intuition.library",0))==NULL)
            cleanexit("Can't open intuition.library\n");

    if ((GfxBase =
        (struct GfxBase *)OpenLibrary("graphics.library",0))==NULL)
        cleanexit("Can't open graphics.library\n");

    if ((IconBase = OpenLibrary("icon.library",0))==NULL )
        cleanexit("Can't open icon.library\n");

    printf("ScreenSave --- C. Scheppner  CBM  10/86\n");
    printf("    Saves the front screen as an IFF ILBM file\n");
    printf("    A CAMG chunk is saved (for HAM pics, etc.)\n\n");

    if(argc>1)                  /* Passed filename via command line  */
        {
        filename = argv[1];
```



```
        }
    else
        {
        printf("Enter filename for save: ");
        l = gets(&nbuf[0]);

        if(l==0)                /* No filename - Exit */
            {
            cleanexit("\nScreen not saved, filename required\n");
            }
        else
            {
            filename = &nbuf[0];
            }
        }

    if (!(file = Open(filename, MODE_NEWFILE)))
        cleanexit("Can't open output file\n");

    Write(file,"x",1);  /* 1.1 so Seek to beginning works ? */

    printf("Click here and press <RETURN> when ready: ");
    gets(&sbuf[0]);
    printf("Front screen will be saved in 10 seconds\n");
    Delay(500);

    Forbid();
    frontScreen  = IntuitionBase->FirstScreen;
    Permit();

    picViewPort = &( frontScreen->ViewPort );
    picBitMap = (struct BitMap*)picViewPort->RasInfo->BitMap;
    picColorTable = (WORD *)picViewPort->ColorMap->ColorTable;
    picViewModes = (ULONG)picViewPort->Modes;

    printf("\nSaving...\n");

    iffp = PutPicture(file, picBitMap, picColorTable, picViewModes);
    Close(file);

    if (iffp == IFF_OKAY)
        {
        printf("Screen saved\n");
        if(!(PutDiskObject(filename,&ILBMobject)))
            {
            cleanexit("Error saving icon\n");
            }
        printf("Icon saved\n");
        }
    cleanexit("Done\n");
    }


cleanexit(s)
    char  *s;
    {
```

```
    if(*s) printf(s);
    if ((fromWB)&&(*s))     /* Wait so user can read messages */
        {
        printf("\nPRESS RETURN TO EXIT\n");
        gets(&sbuf[0]);
        }
    cleanup();
    exit();
    }

cleanup()
    {
    if (newStdio)   closeStdio();
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    if (IconBase) CloseLibrary(IconBase);
    }


openStdio(conspec)
char *conspec;
    {
    LONG wfile;
    struct Process *proc;
    struct FileHandle *handle;

    if (!(wfile = Open(conspec,MODE_NEWFILE)))  return(0);
    stdin  = wfile;
    stdout = wfile;
    stderr = wfile;
    handle = (struct FileHandle *)(wfile << 2);
    proc = (struct Process *)FindTask(NULL);
    proc->pr_ConsoleTask = (APTR)(handle->fh_Type);
    proc->pr_CIS = (BPTR)stdin;
    proc->pr_COS = (BPTR)stdout;
    return(1);
    }

closeStdio()
    {
    struct Process *proc;
    struct FileHandle *handle;

    if (stdin > 0)  Close(stdin);
    stdin  = -1;
    stdout = -1;
    stderr = -1;
    handle = (struct FileHandle *)(stdin << 2);
    proc = (struct Process *)FindTask(NULL);
    proc->pr_ConsoleTask = NULL;
    proc->pr_CIS = NULL;
    proc->pr_COS = NULL;
    }


gets(s)
```



```
char *s;
    {
    int l = 0, max = INBUFSZ - 1;

    while (((*s = getchar()) !='\n' )&&(l < max)) s++, l++;
    *s = NULL;
    return(l);
    }


/* String Functions */

strlen(s)
char *s;
    {
    int i = 0;
    while(*s++) i++;
    return(i);
    }

strcpy(to,from)
char *to, *from;
    {
    do
        {
        *to++ = *from;
        }
    while(*from++);
    }


/** PutPicture() ********************************************************
 *
 * Put a picture into an IFF file.
 * This procedure calls PutAnILBM, passing in an <x, y> location of <0, 0>,
 * a NULL mask, and a locally-allocated buffer. It also assumes you want to
 * write out all the bitplanes in the BitMap.
 *
 ************************************************************************/
Point2D nullPoint = {0, 0};

IFFP PutPicture(file, bitmap, colorMap, viewmodes)
      LONG file;  struct BitMap *bitmap;
      WORD *colorMap;  ULONG viewmodes;
    {
    BYTE buffer[bufSize];
    return( PutAnILBM(file, bitmap, NULL,
            colorMap, bitmap->Depth, viewmodes,
            &nullPoint, buffer, bufSize) );
    }


/** PutAnILBM() *********************************************************
 *
 * Write an entire BitMap as a FORM ILBM in an IFF file.
 * This version works for any display mode (C. Scheppner).
```

```
 *
 * Normal return result is IFF_OKAY.
 *
 * The utility program IFFCheck would print the following outline of the
 * resulting file:
 *
 *   FORM ILBM
 *     BMHD
 *     CAMG
 *     CMAP
 *     BODY        (compressed)
 *
 ******************************************************************************/
#define CkErr(expression)  {if (ifferr == IFF_OKAY) ifferr = (expression);}

IFFP PutAnILBM(file, bitmap, mask, colorMap, depth,
                                viewmodes, xy, buffer, bufsize)
     LONG file;
     struct BitMap *bitmap;
     BYTE *mask;  WORD *colorMap; UBYTE depth;
     ULONG viewmodes;
     Point2D *xy; BYTE *buffer;  LONG bufsize;
  {
  BitMapHeader bmHdr;
  CamgChunk    camgChunk;
  GroupContext fileContext, formContext;
  IFFP ifferr;
  WORD pageWidth, pageHeight;

  pageWidth  = (bitmap->BytesPerRow) << 3;
  pageHeight = bitmap->Rows;

  ifferr = InitBMHdr(&bmHdr, bitmap, mskNone,
                     cmpByteRun1, 0, pageWidth, pageHeight);
  /* You could write an uncompressed image by passing cmpNone instead
   * of cmpByteRun1 to InitBMHdr. */
  bmHdr.nPlanes = depth;   /* This must be  <= bitmap->Depth */
  if (mask != NULL) bmHdr.masking = mskHasMask;
  bmHdr.x = xy->x;   bmHdr.y = xy->y;

  camgChunk.ViewModes = viewmodes;

  CkErr( OpenWIFF(file, &fileContext, szNotYetKnown) );
  CkErr(StartWGroup(&fileContext, FORM, szNotYetKnown, ID_ILBM, &formContext));

  CkErr( PutBMHD(&formContext, &bmHdr) );
  CkErr( PutCAMG(&formContext, &camgChunk) );
  CkErr( PutCMAP(&formContext, colorMap, depth) );
  CkErr( PutBODY(&formContext, bitmap, mask, &bmHdr, buffer, bufsize) );

  CkErr( EndWGroup(&formContext) );
  CkErr( CloseWGroup(&fileContext) );
  return( ifferr );
  }
```

```
/*
 * cycvb.c --- Dan Silva's DPaint color cycling interrupt code
 *
 *    Use this as an example for interrupt driven color cycling
 *    If compiled with Lattice, use -v flag on LC2
 *    For an example of subtask cycling, see Display.c
 */

#include <exec/types.h>
#include <exec/interrupts.h>
#include <graphics/view.h>
#include <iff/compiler.h>

#define MAXNCYCS 4
#define NO  FALSE
#define YES TRUE
#define LOCAL static

typedef struct {
    SHORT count;
    SHORT rate;
    SHORT flags;
    UBYTE low, high;  /* bounds of range */
    } Range;

/* Range flags values */
#define RNG_ACTIVE  1
#define RNG_REVERSE 2
#define RNG_NORATE 36  /* if rate == NORATE, don't cycle */

/* cycling frame rates */
#define OnePerTick   16384
#define OnePerSec    OnePerTick/60

extern Range  cycles[];
extern BOOL   cycling[];
extern WORD   cycols[];
extern struct ViewPort *vport;
extern SHORT  nColors;


MyVBlank()  {
   int i,j;
   LOCAL  Range *cyc;
   LOCAL  WORD  temp;
   LOCAL  BOOL  anyChange;

#ifdef IS_AZTEC
#asm
       movem.l  a2-a7/d2-d7,-(sp)
       move.l   a1,a4
#endasm
#endif

   if (cycling)  {
      anyChange = NO;
```



```
      for (i=0; i<MAXNCYCS; i++)  {
         cyc = &cycles[i];
         if ( (cyc->low == cyc->high) ||
              ((cyc->flags&RNG_ACTIVE) == 0) ||
              (cyc->rate == RNG_NORATE) )
                 continue;

         cyc->count += cyc->rate;
         if (cyc->count >= OnePerTick)  {
            anyChange = YES;
            cyc->count -= OnePerTick;

            if (cyc->flags&RNG_REVERSE)  {
               temp = cycols[cyc->low];
               for (j=cyc->low; j < cyc->high; j++)
                  cycols[j] = cycols[j+1];
               cycols[cyc->low] = temp;
               }
            else  {
               temp = cycols[cyc->high];
               for (j=cyc->high; j > cyc->low; j--)
                  cycols[j] = cycols[j-1];
               cycols[cyc->low] = temp;
               }
            }
         }
      if (anyChange) LoadRGB4(vport,cycols,nColors);
      }

#ifdef IS_AZTEC
      ;   /* this is necessary */
#asm
      movem.l  (sp)+,a2-a7/d2-d7
#endasm
#endif

   return(0);  /* interrupt routines have to do this */
   }


/*
 *  Code to install/remove cycling interrupt handler
 */

LOCAL char myname[] = "MyVB";  /* Name of interrupt handler */
LOCAL struct Interrupt intServ;

typedef void (*VoidFunc)();

StartVBlank()  {
#ifdef IS_AZTEC
   intServ.is_Data = GETAZTEC();  /* returns contents of register a4 */
#else
   intServ.is_Data = NULL;
#endif
   intServ.is_Code = (VoidFunc)&MyVBlank;
```

```
    intServ.is_Node.ln_Succ = NULL;
    intServ.is_Node.ln_Pred = NULL;
    intServ.is_Node.ln_Type = NT_INTERRUPT;
    intServ.is_Node.ln_Pri  = 0;
    intServ.is_Node.ln_Name = myname;
    AddIntServer(5,&intServ);
    }

StopVBlank() { RemIntServer(5,&intServ); }

/**/
```

```
Mar 12 14:59 1987  Display/Display.with Page 1


FROM      LIB:AStartup.obj,Display.o,myreadpict.o,iffmsgs.o*
          iffr.o,ilbmr.o,unpacker.o
TO        Display
LIBRARY   LIB:Amiga.lib, LIB:LC.lib
```

```
/*
 * Display.c
 *   Read an ILBM file and display as a screen/window until closed.
 *   Simulated close gadget in upper left corner of window.
 *   Clicking below title bar area toggles screen bar for dragging.
 *   Handles normal and HAM ILBM's
 *   Does automatic color cycling (see note below)
 *   Accepts optional 2nd CLI arg for display time in seconds
 *
 *  By Carolyn Scheppner   CBM  03/15/86
 *
 *  Modified 09/02/86 - Only global frame is iFrame
 *                      Use message->MouseX and Y
 *                      Wait() for IDCMP
 *  Modified 10/15/86 - For HAM
 *                      Name changed from SeeILBM to ViewILBM
 *  Modified 11/01/86 - Revised for linkage with myreadpict.c
 *  Modified 11/18/86 - For Astartup ... Amiga.lib, LC.lib linkage
 *  Modified 12/12/86 - Added color cycling at request of Mimetics
 *  Modified 01/06/87 - Tab toggles cycling
 *  Modified 03/03/87 - Recognizes RNG_NORATE (36) as non-active DP CRNG
 *                      Changed name to Display
 *  Modified 03/13/87 - Accepts display time in seconds as 2nd CLI arg
 *
 *     This viewer automatically cycles any ILBM that contains cycling
 *     chunks (CCRT or CRNG) which are marked as active and do not
 *     have a CRNG cycle rate of 36. (To DPaint, rate 36 = don't cycle)
 *     Note that by default, DPaint saves its pics with CRNG (cycling)
 *     chunks flagged as active and with a rate not equal to 36.
 *     This can cause this viewer to cycle DPaint pics which were
 *     not meant to be cycled.  To de-activate the CRNG chunks in
 *     a DPaint pic, either resave the pic after setting the cycle
 *     speed for each range to the lowest position, or use the
 *     "uncycle" program. (usage:  uncycle DPaintPicName)
 *
 *
 * Based on ShowILBM.c, readpict.c    1/86
 *  By Jerry Morrison, Steve Shaw, and Steve Hayes, Electronic Arts.
 *  This software is in the public domain.
 *
 * >>NOTE<<: This example must be linked with additional IFF rtn files.
 *           See linkage information below.
 *
 * The display portion is specific to the Commodore Amiga computer.
 *
 * Linkage Information:
 *  (NOTE: All modules including iff stuff compiled with -v on LC2)
 *
 * FROM     AStartup.obj,Display.o,myreadpict.o,iffr.o,ilbmr.o,unpacker.o
 * TO       Display
 * LIBRARY  Amiga.lib, LC.lib
 *
 */

#include <exec/types.h>
#include <exec/memory.h>
```



```
#include <exec/tasks.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <workbench/startup.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>

#include "iff/ilbm.h"
#include "myreadpict.h"


/* For wbStdio rtns */
extern LONG stdin, stdout, stderr;  /* in Astartup.obj */
char conSpec[] = "CON:0/40/640/140/";
BOOL wbHasStdio = NULL;


/* general usage pointers */
struct GfxBase       *GfxBase;
struct IntuitionBase *IntuitionBase;
struct IntuiMessage  *message;

/* Globals for displaying an image */
struct Screen   *screen1;
struct Window   *window1;
struct RastPort *rport1;
struct ViewPort *vport1;

struct BitMap   tBitMap;      /* Temp BitMap struct for small pics  */

/* For WorkBench startup */
extern struct WBStartup *WBenchMsg;
BOOL   fromWB;
struct FileLock *startLock, *newLock;

/* Other globals */
int   i, error;
BYTE  c;
BOOL  TBtoggle, Done;
ULONG signals, wSig, class, code, pBytes;
SHORT mouseX, mouseY;

char u1[]  = "\n>>> Display <<<   v1  C. Scheppner  CBM  03/87\n";
char u2a[] = "\nUsage: Display ilbmfile [time]\n";
char u2b[] = "\nUsage: Click this icon, SHIFT and DoubleClick on pic\n";
char u3[]  = "Click toggles bar, Tab toggles cycling, Close upper left\n";

/* Structures for new Screen, new Window */

struct   TextAttr       TextFont = {
   "topaz.font",                    /* Font Name   */
   TOPAZ_EIGHTY,                    /* Font Height */
   FS_NORMAL,                       /* Style       */
   FPF_ROMFONT,                     /* Preferences */
   };
```

```
struct   NewScreen      ns = {
   0, 0,                                    /* LeftEdge and TopEdge   */
   0, 0,                                    /* Width and Height       */
   0,                                       /* Depth                  */
   1, 0,                                    /* DetailPen and BlockPen */
   NULL,                                    /* Special display modes  */
   CUSTOMSCREEN,                            /* Screen Type            */
   &TextFont,                               /* Use my font            */
   " <- Close here after clicking below",                  /* Title */
   NULL,                                    /* No gadgets yet         */
   NULL,                                    /* Ptr to CustomBitmap    */
   };

struct   NewWindow      nw = {
   0, 0,                                    /* LeftEdge and TopEdge */
   0, 0,                                    /* Width and Height */
   -1, -1,                                  /* DetailPen and BlockPen */
   MOUSEBUTTONS|VANILLAKEY,                 /* IDCMP Flags */
   ACTIVATE
   |BACKDROP
   |BORDERLESS,                             /* Flags */
   NULL, NULL,                              /* Gadget and Image pointers */
   NULL,                                    /* Title string */
   NULL,                                    /* Put Screen ptr here */
   NULL,                                    /* SuperBitMap pointer */
   0, 0,                                    /* MinWidth and MinHeight */
   0, 0,                                    /* MaxWidth and MaxHeight */
   CUSTOMSCREEN,                            /* Type of window */
   };

USHORT  allBlack[maxColorReg] = {0};

/* For alloc to define new pointer */
#define PDATASZ 12
UWORD   *pdata;

#ifndef MIN
#define MIN(a,b)  ((a)<(b)?(a):(b))
#endif  MIN

extern char *IFFPMessages[];
ILBMFrame   iFrame;       /* my global frame */


/* Cycle Task stuff */
#define CYCLETIME  16384L
#define REVERSE    0x02
#define ACTIVE     0x01

extern VOID cycleTask();
char *cyTaskName = "v2cyTask";
struct Task *cyTask;

/* Data shared with cycleTask */
CrngChunk       *cyCrngs;
struct ViewPort *cyVport;
```



```
int    cyRegs, cyCnt;
USHORT cyMap[maxColorReg];
LONG   cyClocks[maxCycles];
LONG   cyRates[maxCycles];
BOOL   CycleOn, PrepareToDie;

/* For optional time delay */
struct Task *mainTask;
BOOL   TimerOn;
LONG   tSigNum = -1;
ULONG  tSig, dTimer;


/*****************************************************************/
main(argc, argv)
int argc;
char **argv;
   {
   LONG            file;
   IFFP            iffp = NO_FILE;
   struct WBArg    *arg;
   char            *filename;

   fromWB = (argc==0) ? TRUE : FALSE;
   TimerOn = FALSE;

   if((argc>1)&&(*argv[1] != '?'))  /* Passed filename via command line  */
      {
      filename = argv[1];
      if(argc==3)
         {
         TimerOn = TRUE;
         dTimer = 60 * atoi(argv[2]);
         }
      }
   else if ((argc==0)&&(WBenchMsg->sm_NumArgs > 1))
      {                          /* Passed filename via  WorkBench */
      arg = WBenchMsg->sm_ArgList;
      arg++;
      filename   = (char *)arg->wa_Name;
      newLock    = (struct FileLock *)arg->wa_Lock;
      startLock  = (struct FileLock *)CurrentDir(newLock);
      }
   else            /* From WB or CLI, no filename or ? */
      {
      usage();
      cleanexit(" "); /* Space forces my wait for keypress on WB exit */
      }

   if(!(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0)))
      cleanexit("Can't open graphics");

   if(!(IntuitionBase=
         (struct IntuitionBase *)OpenLibrary("intuition.library",0)))
      cleanexit("Can't open intuition");
```

```
   if(!(file = Open(filename, MODE_OLDFILE)))
      cleanexit("Picture file not found");

   iffp = myReadPicture(file,&iFrame);
   Close(file);

   if (!(iffp == IFF_DONE))
      cleanexit("Not an IFF ILBM");


   error = DisplayPic(&iFrame);
   if(error)  cleanexit("Can't open screen or window");

   if(pdata = (UWORD *)AllocMem(PDATASZ,MEMF_CHIP|MEMF_CLEAR))
      {
      pdata[2] = 0x8000;  /* 1 pixel */
      SetPointer(window1,pdata,1,16,0,0);
      }

   /* Set up cycle/timer task */

   mainTask = (struct Task *)FindTask(NULL);
   if((tSigNum = AllocSignal(-1)) == -1)  cleanexit("Can't alloc timerSig");
   tSig = 1 << tSigNum;

   initCycle(&iFrame,vport1);
   cyTask = (struct Task *)CreateTask(cyTaskName,0,cycleTask,4000);
   if(!cyTask)  cleanexit("Can't create cycling task");
   CycleOn = TRUE;

   TBtoggle   = FALSE;      /* Title bar toggle */
   Done       = FALSE;      /* Close flag       */

   wSig = 1<<window1->UserPort->mp_SigBit;
   tSig = 1<<tSigNum;

   while (!Done)
      {
      signals = Wait(wSig|tSig);
      if (signals & wSig)  chkmsg();
      if (signals & tSig)  Done = TRUE;
      }
   cleanup();
   }


initCycle(ptFrame,vp)
ILBMFrame *ptFrame;
struct ViewPort *vp;
   {
   int k;

   CycleOn  = FALSE;
   PrepareToDie = FALSE;
   cyCrngs  = ptFrame->crngChunks;
   cyVport  = vp;
```



```
   cyRegs   = ptFrame->nColorRegs;
   cyCnt    = ptFrame->cycleCnt;

   for(k=0; k<cyRegs; k++)
      {
      cyMap[k] = ptFrame->colorMap[k];
      }

   /* Init Rates and Clocks */
   for(k=0; k<cyCnt; k++)
      {
      /* In DPaint CRNG, rate = RNG_NORATE (36) means don't cycle */
      if(cyCrngs[k].rate == RNG_NORATE)
         {
         cyCrngs[k].rate = 0;
         cyCrngs[k].active &= ~ACTIVE;
         }

      if((cyCrngs[k].active & ACTIVE)&&(cyCrngs[k].rate))
         {
         cyRates[k] = cyCrngs[k].rate;
         }
      else
         {
         cyRates[k] = 0;  /* Means don't cycle to my cycleTask */
         }
      cyClocks[k] = 0;
      }
   }


VOID cycleTask()
   {
   int    k, i, j;
   UBYTE  low, high;
   USHORT cyTmp;
   BOOL   Cycled;

   while(!PrepareToDie)
      {
      WaitTOF();
      if(CycleOn)
         {
         Cycled = FALSE;
         for(k=0; k<cyCnt; k++)
            {
            if(cyRates[k])  /* cyRate 0 = inactive */
               {
               cyClocks[k] += cyRates[k];
               if(cyClocks[k] >= CYCLETIME)
                  {
                  Cycled = TRUE;
                  cyClocks[k] -= CYCLETIME;
                  low = cyCrngs[k].low;
                  high= cyCrngs[k].high;
                  if(cyCrngs[k].active & REVERSE)  /* Reverse cycle */
```

```
                    {
                    cyTmp = cyMap[low];
                    for(i=low,j=low+1; i < high; i++,j++)
                        {
                        cyMap[i] = cyMap[j];
                        }
                    cyMap[high] = cyTmp;
                    }
                else     /* Forward cycle */
                    {
                    cyTmp = cyMap[high];
                    for(i=high,j=high-1; i > low; i--,j--)
                        {
                        cyMap[i] = cyMap[j];
                        }
                    cyMap[low] = cyTmp;
                    }
                }
            }
        }
        if(Cycled)
            {
            LoadRGB4(cyVport,cyMap,cyRegs);
            }
        }
    if(TimerOn)
        {
        if(--dTimer == 0)  Signal(mainTask,tSig);
        }
    }
PrepareToDie = FALSE;
while(TRUE);  /* Busy wait to die */
}

chkmsg()
{
while(message=(struct IntuiMessage *)GetMsg(window1->UserPort))
    {
    class = message->Class;
    code  = message->Code;
    mouseX = message->MouseX;
    mouseY = message->MouseY;

    ReplyMsg(message);
    switch(class)
        {
        case MOUSEBUTTONS:
            if ((code == SELECTDOWN)&&
                (mouseX < 10)&&(mouseY<10))
                {
                Done = TRUE;
                }
            else if ((code == SELECTDOWN)&&
                    ((mouseY>10)||(mouseX>10))&&
                        (TBtoggle==FALSE))
```



```
                {
                TBtoggle = TRUE;
                ShowTitle(screen1,TRUE);
                ClearPointer(window1);
                }
            else if ((code == SELECTDOWN)&&
                    (mouseY>10)&&(TBtoggle==TRUE))
                {
                TBtoggle = FALSE;
                ShowTitle(screen1,FALSE);
                SetPointer(window1,pdata,1,16,0,0);
                }
            break;
        case VANILLAKEY:
            if(code==0x09)  /* Tab toggles Cycle */
                {
                if(CycleOn)
                    {
                    CycleOn = FALSE;
                    WaitTOF();          /* Make sure cyTask saw FALSE */
                    WaitBOVP(vport1);
                    LoadRGB4(vport1,iFrame.colorMap,maxColorReg);
                    }
                else
                    {
                    initCycle(&iFrame,vport1);
                    CycleOn = TRUE;
                    }
                }
            break;
        default:
            break;
        }
    }
}

usage()
{
if ((fromWB)&&(! wbHasStdio))  wbHasStdio = openStdio(conSpec);

if ((!fromWB)||(wbHasStdio))
    {
    Write(stdout,u1,strlen(u1));
    if(!fromWB)  Write(stdout,u2a,strlen(u2a));
    else Write(stdout,u2b,strlen(u2b));
    Write(stdout,u3,strlen(u3));
    }
}

cleanexit(s)
char  *s;
{
if (*s)
    {
    if ((fromWB)&&(! wbHasStdio))  wbHasStdio = openStdio(conSpec);
```

```
      if ((!fromWB)||(wbHasStdio))
         {
         Write(stdout,s,strlen(s));
         Write(stdout,"\n",1);
         }
      if (wbHasStdio)
         {
         Write(stdout,"\nPRESS RETURN TO EXIT\n",22);
         while (getchar() != '\n');
         }
      }
   cleanup();
   if (wbHasStdio) closeStdio();
   exit();
   }


cleanup()
   {
   if(cyTask)
      {
      CycleOn = FALSE;
      PrepareToDie = TRUE;
      while(PrepareToDie);
      DeleteTask(cyTask);
      }

   /* Free timer signal */
   if (tSigNum > -1)  FreeSignal(tSigNum);

   /* Note - tBitMap planes were deallocated in DisplayPic() */
   if (window1)
      {
      while(message=(struct IntuiMessage *)GetMsg(window1->UserPort))
         {
         ReplyMsg(message);
         }
      CloseWindow(window1);
      }
   if (screen1) CloseScreen(screen1);
   if (pdata)   FreeMem(pdata,PDATASZ);
   if (IntuitionBase) CloseLibrary(IntuitionBase);
   if (GfxBase)       CloseLibrary(GfxBase);
   if (newLock != startLock)  CurrentDir(startLock);
   }


strlen(s)
char *s;
   {
   int i = 0;
   while(*s++) i++;
   return(i);
   }
```



```
/** getBitMap() ***************************************************************
 *
 * Open screen or temp bitmap.
 *   Returns ptr destBitMap  or  0 = error
 *
 ******************************************************************************/
struct BitMap *getBitMap(ptilbmFrame)
   ILBMFrame *ptilbmFrame;
   {
   int     i, nPlanes, plsize;
   SHORT  sWidth, sHeight, dWidth, dHeight;
   struct BitMap *destBitMap;

   sWidth  = ptilbmFrame->bmHdr.w;
   sHeight = ptilbmFrame->bmHdr.h;
   dWidth  = ptilbmFrame->bmHdr.pageWidth;
   dHeight = ptilbmFrame->bmHdr.pageHeight;
   nPlanes = MIN(ptilbmFrame->bmHdr.nPlanes, EXDepth);

   ns.Width  = dWidth;
   ns.Height = dHeight;
   ns.Depth  = nPlanes;

   if (ptilbmFrame->foundCAMG)
      {
      ns.ViewModes = ptilbmFrame->camgChunk.ViewModes;
      }
   else
      {
      if (ptilbmFrame->bmHdr.pageWidth <= 320)
         ns.ViewModes = 0;
      else
         ns.ViewModes = HIRES;

      if (ptilbmFrame->bmHdr.pageHeight > 256)
         ns.ViewModes |= LACE;
      }

   if ((screen1 = (struct Screen *)OpenScreen(&ns))==NULL)     return(0);

   vport1 = &screen1->ViewPort;
   LoadRGB4(vport1, &allBlack[0], ptilbmFrame->nColorRegs);

   if((ptilbmFrame->camgChunk.ViewModes)&(HAM))  setHam(screen1,FALSE);

   nw.Width  = dWidth;
   nw.Height = dHeight;
   nw.Screen = screen1;

   if ((window1 = (struct Window *)OpenWindow(&nw))==NULL)
      {
      CloseScreen(screen1);
      screen1 = NULL;
      return(0);
```

```
    }

  ShowTitle(screen1, FALSE);

  if ((sWidth == dWidth) && (sHeight == dHeight))
     {
     destBitMap = (struct BitMap *)screen1->RastPort.BitMap;
     }
  else
     {
     InitBitMap( &tBitMap,
                 nPlanes,
                 sWidth,
                 sHeight);

     plsize = RowBytes(ptilbmFrame->bmHdr.w) * ptilbmFrame->bmHdr.h;
     if (tBitMap.Planes[0] =
      (PLANEPTR)AllocMem(nPlanes * plsize, MEMF_CHIP))
        {
        for (i = 1; i < nPlanes; i++)
           tBitMap.Planes[i] = (PLANEPTR)tBitMap.Planes[0] + plsize*i;
        destBitMap = &tBitMap;
        }
     else
        {
        CloseWindow(window1);
        window1 = NULL;
        CloseScreen(screen1);
        screen1 = NULL;
        return(0);  /* can't allocate temp BitMap */
        }
     }
  return(destBitMap);          /* destBitMap allocated */
  }


/** DisplayPic() ************************************************************
 *
 * Display loaded bitmap.  If tBitMap, first transfer to screen.
 *
 ****************************************************************************/
DisplayPic(ptilbmFrame)
   ILBMFrame *ptilbmFrame;
   {
   int    i, row, byte, nrows, nbytes;
   struct BitMap  *tbp, *sbp; /* temp and screen BitMap ptrs */
   UBYTE  *tpp, *spp;         /* temp and screen plane ptrs  */

   if (tBitMap.Planes[0])     /* transfer from tBitMap if nec. */
      {
      tbp = &tBitMap;
      sbp = screen1->RastPort.BitMap;
      nrows  = MIN(tbp->Rows, sbp->Rows);
      nbytes = MIN(tbp->BytesPerRow, sbp->BytesPerRow);

      for (i = 0; i < sbp->Depth; i++)
```



```
         {
         tpp = (UBYTE *)tbp->Planes[i];
         spp = (UBYTE *)sbp->Planes[i];
         for (row = 0; row < nrows; row++)
            {
            tpp = tbp->Planes[i] + (row * tbp->BytesPerRow);
            spp = sbp->Planes[i] + (row * sbp->BytesPerRow);
            for (byte = 0; byte < nbytes; byte++)
               {
               *spp++ = *tpp++;
               }
            }
         }
      /*  Can now deallocate the temp BitMap  */
      FreeMem(tBitMap.Planes[0],
               tBitMap.BytesPerRow * tBitMap.Rows * tBitMap.Depth);
      }

   vport1 = &screen1->ViewPort;
   LoadRGB4(vport1, ptilbmFrame->colorMap, ptilbmFrame->nColorRegs);
   if((ptilbmFrame->camgChunk.ViewModes)&(HAM))  setHam(screen1,TRUE);

   return(0);
   }


/* setHam --- For toggling HAM so HAM pic invisible while loading */
setHam(scr,toggle)
struct Screen *scr;
BOOL   toggle;
   {
   struct ViewPort *vp;
   struct View     *v;

   vp = &(scr->ViewPort);
   v = (struct View *)ViewAddress();
   Forbid();
   if(toggle)
      {
      v->Modes  |= HAM;
      vp->Modes |= HAM;
      }
   else
      {
      v->Modes  &= ~HAM;
      vp->Modes &= ~HAM;
      }
   MakeScreen(scr);
   RethinkDisplay();
   Permit();
   }


/* wbStdio.c --- Open an Amiga stdio window under workbench
 *               For use with AStartup.obj
 */
```

```
openStdio(conspec)
char *conspec;
   {
   LONG wfile;
   struct Process *proc;
   struct FileHandle *handle;

   if (wbHasStdio)  return(1);

   if (!(wfile = Open(conspec,MODE_NEWFILE)))  return(0);
   stdin  = wfile;
   stdout = wfile;
   stderr = wfile;
   handle = (struct FileHandle *)(wfile << 2);
   proc = (struct Process *)FindTask(NULL);

   proc->pr_ConsoleTask = (APTR)(handle->fh_Type);
   proc->pr_CIS = (BPTR)stdin;
   proc->pr_COS = (BPTR)stdout;
   return(1);
   }

closeStdio()
   {
   struct Process *proc;
   struct FileHandle *handle;

   if (! wbHasStdio) return(0);

   if (stdin > 0)  Close(stdin);
   stdin  = -1;
   stdout = -1;
   stderr = -1;
   handle = (struct FileHandle *)(stdin << 2);
   proc = (struct Process *)FindTask(NULL);
   proc->pr_ConsoleTask = NULL;
   proc->pr_CIS = NULL;
   proc->pr_COS = NULL;
   wbHasStdio = NULL;
   }
```

```
/* iffmsgs.c --- The IFF error msgs indexed by iffp
 * Use:  extern char *IFFPMessages[];  in application to access
 */

#ifndef IFF_H
#include "iff/iff.h"
#endif

/* Message strings for IFFP codes. */
char MsgOkay[]        = {"(IFF_OKAY) No FORM of correct type in file." };
char MsgEndMark[]     = {"(END_MARK) How did you get this message?" };
char MsgDone[]        = {"(IFF_DONE) All done."};
char MsgDos[]         = {"(DOS_ERROR) The DOS returned an error." };
char MsgNot[]         = {"(NOT_IFF) Not an IFF file." };
char MsgNoFile[]      = {"(NO_FILE) No such file found." };
char MsgClientError[] = {"(CLIENT_ERROR) Probably insufficient RAM."};
char MsgForm[]        = {"(BAD_FORM) File contains a malformed FORM." };
char MsgShort[]       = {"(SHORT_CHUNK) File contains a short Chunk." };
char MsgBad[]         = {"(BAD_IFF) A mangled IFF file." };

/* THESE MUST APPEAR IN RIGHT ORDER!! */
char *IFFPMessages[-LAST_ERROR+1] = {
    /*IFF_OKAY*/  MsgOkay,
    /*END_MARK*/  MsgEndMark,
    /*IFF_DONE*/  MsgDone,
    /*DOS_ERROR*/ MsgDos,
    /*NOT_IFF*/   MsgNot,
    /*NO_FILE*/   MsgNoFile,
    /*CLIENT_ERROR*/ MsgClientError,
    /*BAD_FORM*/  MsgForm,
    /*SHORT_CHUNK*/  MsgShort,
    /*BAD_IFF*/   MsgBad
    };
```

```
/* myreadpict.h */

#ifndef MYREADPICT_H
#define MYREADPICT_H

#ifndef ILBM_H
#include <iff/ilbm.h>
#endif

#ifndef GRAPHICS_GFX_H
#include <graphics/gfx.h>
#endif


#define EXDepth 6        /* Maximum depth (6=HAM) */
#define maxColorReg 32
#define maxCycles   8
#define RNG_NORATE  36   /* Dpaint uses this rate to mean non-active */

typedef struct {
   ULONG ViewModes;
   } CamgChunk;

typedef struct {
   WORD  pad1;   /* future exp - store 0 here */
   WORD  rate;   /* 60/sec=16384, 30/sec=8192, 1/sec=16384/60=273 */
   WORD  active; /* lo bit 0=no cycle, 1=yes; next bit 1=rvs */
   UBYTE low;    /* range lower */
   UBYTE high;   /* range upper */
   } CrngChunk;

typedef struct {
   WORD  direction;  /* 0=don't cycle, 1=forward, -1=backwards */
   UBYTE start;      /* range lower */
   UBYTE end;        /* range upper */
   LONG  seconds;    /* seconds between cycling */
   LONG  microseconds; /* msecs between cycling */
   WORD  pad;        /* future exp - store 0 here */
   } CcrtChunk;


#define GetCAMG(context, camg)  \
    IFFReadBytes(context, (BYTE *)camg, sizeof(CamgChunk))

#define ID_CRNG  MakeID('C','R','N','G')
#define GetCRNG(context, crng)  \
    IFFReadBytes(context, (BYTE *)crng, sizeof(CrngChunk))

#define ID_CCRT  MakeID('C','C','R','T')
#define GetCCRT(context, ccrt)  \
    IFFReadBytes(context, (BYTE *)ccrt, sizeof(CcrtChunk))


typedef struct {
   ClientFrame clientFrame;
   UBYTE foundBMHD;
```



```
   UBYTE nColorRegs;
   BitMapHeader bmHdr;
   Color4 colorMap[maxColorReg];
   /* If you want to read any other property chunks, e.g. GRAB or CAMG, add
    * fields to this record to store them. */
   UBYTE foundCAMG;
   CamgChunk camgChunk;
   UBYTE cycleCnt;
   CrngChunk crngChunks[maxCycles]; /* I'll convert CCRT to this */
   } ILBMFrame;

typedef UBYTE *UBytePtr;

#ifdef FDwAT
extern IFFP myReadPicture(LONG, ILBMFrame *);
extern struct BitMap *getBitMap(ILBMFrame *);
#else
extern IFFP myReadPicture();
extern struct BitMap *getBitMap();
#endif

#endif MYREADPICT_H
```

```
/** myReadPict.c ***************************************************************
 *
 * Read an ILBM raster image file.                23-Jan-86.
 *
 * Modified version of ReadPict.c
 *   by Jerry Morrison, Steve Shaw, and Steve Hayes, Electronic Arts.
 *   This software is in the public domain.
 *
 * Modified by C. Scheppner  11/86
 *   Handles CAMG chunks for HAM, etc.
 *   Calls user defined routine getBitMap(ilbmFramePtr) when it
 *      reaches the BODY.
 *   getBitMap() can open a screen of the correct size using
 *      information this rtn places in the ilbmFrame, and returns
 *      a pointer to a BitMap structure.  The BitMap structure
 *      tells myReadPicture where it should load the bit planes.
 *
 * Modified by C. Scheppner  12/86
 *   Loads in CCRT or CRNG chunks (converts CCRT to CRNG)
 *
 ******************************************************************************/

#define LOCAL   static

#include "intuition/intuition.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "iff/ilbm.h"
#include "myreadpict.h"    /* cs */

/* Define size of a temporary buffer used in unscrambling the ILBM rows.*/
#define bufSz 512

/*------------ ILBM reader ------------------------------------------------*/
/* ILBMFrame is our "client frame" for reading FORMs ILBM in an IFF file.
 * We allocate one of these on the stack for every LIST or FORM encountered
 * in the file and use it to hold BMHD & CMAP properties. We also allocate
 * an initial one for the whole file.
 * We allocate a new GroupContext (and initialize it by OpenRIFF or
 * OpenRGroup) for every group (FORM, CAT, LIST, or PROP) encountered. It's
 * just a context for reading (nested) chunks.
 *
 * If we were to scan the entire example file outlined below:
 *    reading          proc(s)                 new                 new
 *
 * --whole file--   myReadPicture+ReadIFF GroupContext        ILBMFrame
 * CAT              ReadICat                GroupContext
 *   LIST           GetLiILBM+ReadIList       GroupContext        ILBMFrame
 *     PROP ILBM    GetPrILBM                   GroupContext
 *       CMAP       GetCMAP
 *       BMHD       GetBMHD
 *     FORM ILBM    GetFoILBM                   GroupContext        ILBMFrame
 *       BODY       GetBODY
 *     FORM ILBM    GetFoILBM                   GroupContext        ILBMFrame
 *       BODY       GetBODY
 *   FORM ILBM      GetFoILBM                 GroupContext        ILBMFrame
```



```
 */

/* NOTE: For a small version of this program, set Fancy to 0.
 * That'll compile a program that reads a single FORM ILBM in a file, which
 * is what DeluxePaint produces. It'll skip all LISTs and PROPs in the input
 * file. It will, however, look inside a CAT for a FORM ILBM.
 * That's suitable for 90% of the uses.
 *
 * For a fancier version that handles LISTs and PROPs, set Fancy to 1.
 * That'll compile a program that dives into a LIST, if present, to read
 * the first FORM ILBM. E.g. a DeluxePrint library of images is a LIST of
 * FORMs ILBM.
 *
 * For an even fancier version, set Fancy to 2. That'll compile a program
 * that dives into non-ILBM FORMs, if present, looking for a nested FORM ILBM.
 * E.g. a DeluxeVideo C.S. animated object file is a FORM ANBM containing a
 * FORM ILBM for each image frame. */
#define Fancy  0

/* Global access to client-provided pointers.*/
LOCAL ILBMFrame *giFrame = NULL;   /* "client frame".*/


IFFP handleCAMG(context,frame)
GroupContext *context;
ILBMFrame    *frame;
   {
   IFFP iffp = IFF_OKAY;

   frame->foundCAMG = TRUE;
   iffp = GetCAMG(context, &frame->camgChunk);
   return(iffp);
   }

IFFP handleCRNG(context,frame)
GroupContext *context;
ILBMFrame    *frame;
   {
   IFFP iffp = IFF_OKAY;

   if(frame->cycleCnt < maxCycles)
      {
      iffp = GetCRNG(context,&(frame->crngChunks[frame->cycleCnt]));
      frame->cycleCnt++;
      }
   return(iffp);
   }


IFFP handleCCRT(context,frame)
GroupContext *context;
ILBMFrame    *frame;
   {
   CcrtChunk  ccrtTmp;
   CrngChunk  *ptCrng;
```

```
    IFFP iffp = IFF_OKAY;

    if(frame->cycleCnt < maxCycles)
        {
        iffp = GetCCRT(context, &ccrtTmp);
        ptCrng = &(frame->crngChunks[frame->cycleCnt]);
        if(ccrtTmp.direction)  ccrtTmp.direction = -ccrtTmp.direction;
        ptCrng->active = ccrtTmp.direction & 0x03;
        ptCrng->low = ccrtTmp.start;
        ptCrng->high = ccrtTmp.end;

        /* Convert  CCRT secs/msecs to CRNG timimg
         * 0x4000 = max CRNG rate  (cycle every 1 60th sec)
         * This must be divided by # 60th's between cycles
         * seconds to 60th's is easy
         * msecs to 60th's requires division by 16667
         * this is int math so I add 8334 (half 16667) first for rounding
         */
        ptCrng->rate = 0x4000 /
            ((ccrtTmp.seconds * 60)+((ccrtTmp.microseconds+8334)/16667));
        frame->cycleCnt++;
        }
    return(iffp);
    }


/** GetFoILBM() ***********************************************************
 *
 * Called via myReadPictureto handle every FORM encountered in an IFF file.
 * Reads FORMs ILBM and skips all others.
 * Inside a FORM ILBM, it stops once it reads a BODY. It complains if it
 * finds no BODY or if it has no BMHD to decode the BODY.
 *
 * Once we find a BODY chunk, we'll call user rtn getBitMap() to
 *    allocate the bitmap and planes (or screen) and then read
 *    the BODY into the planes.
 *
 ************************************************************************/
LOCAL BYTE bodyBuffer[bufSz];
IFFP GetFoILBM(parent)  GroupContext *parent;
    {
    /*compilerBug register*/ IFFP iffp;
    GroupContext formContext;
    ILBMFrame ilbmFrame;        /* only used for non-clientFrame fields.*/
    struct BitMap *destBitMap;   /* cs */

    /* Handle a non-ILBM FORM. */
    if (parent->subtype != ID_ILBM)
        {
#if Fancy >= 2
        /* Open a non-ILBM FORM and recursively scan it for ILBMs.*/
        iffp = OpenRGroup(parent, &formContext);
        CheckIFFP();
        do {
            iffp = GetF1ChunkHdr(&formContext);
            } while (iffp >= IFF_OKAY);
```



```
        if (iffp == END_MARK)
            {
            iffp = IFF_OKAY;   /* then continue scanning the file */
            }
        CloseRGroup(&formContext);
        return(iffp);
#else
        return(IFF_OKAY); /* Just skip this FORM and keep scanning the file.*/
#endif
        }

    ilbmFrame = *(ILBMFrame *)parent->clientFrame;
    iffp = OpenRGroup(parent, &formContext);
    CheckIFFP();

    do switch (iffp = GetFChunkHdr(&formContext)) {
        case ID_BMHD: {
            ilbmFrame.foundBMHD = TRUE;
            iffp = GetBMHD(&formContext, &ilbmFrame.bmHdr);
            break; }
        case ID_CAMG: {        /* cs */
            iffp = handleCAMG(&formContext, &ilbmFrame);
            break; }
        case ID_CRNG: {        /* cs */
            iffp = handleCRNG(&formContext, &ilbmFrame);
            break; }
        case ID_CCRT: {        /* cs */
            iffp = handleCCRT(&formContext, &ilbmFrame);
            break; }
        case ID_CMAP: {
            ilbmFrame.nColorRegs = maxColorReg; /* room for this many */
            iffp = GetCMAP(&formContext, (WORD *)ilbmFrame.colorMap,
                           &ilbmFrame.nColorRegs);
            break; }
        case ID_BODY: {        /* cs */
            if (!ilbmFrame.foundBMHD)
                {
                iffp = BAD_FORM;   /* No BMHD chunk! */
                }
            else
                {
                if(destBitMap=(struct BitMap *)getBitMap(&ilbmFrame))
                    {
                    iffp = GetBODY( &formContext,
                                    destBitMap,
                                    NULL,
                                    &ilbmFrame.bmHdr,
                                    bodyBuffer,
                                    bufSz);
                    if (iffp == IFF_OKAY) iffp = IFF_DONE;   /* Eureka */
                    *giFrame = ilbmFrame; /* copy fields to client frame */
                    }
                else
                    {
                    iffp = CLIENT_ERROR;   /* not enough RAM for the bitmap */
                    }
```

```
            }
         break; }

      case END_MARK: {
         iffp = BAD_FORM;
         break; }

      } while (iffp >= IFF_OKAY);
      /* loop if valid ID of ignored chunk or a
       * subroutine returned IFF_OKAY (no errors).*/

   if (iffp != IFF_DONE)  return(iffp);

   CloseRGroup(&formContext);
   return(iffp);
   }

/** Notes on extending GetFoILBM ***********************************************
 *
 * To read more kinds of chunks, just add clauses to the switch statement.
 * To read more kinds of property chunks (GRAB, CAMG, etc.) add clauses to
 * the switch statement in GetPrILBM, too.
 *
 * To read a FORM type that contains a variable number of data chunks--e.g.
 * a FORM FTXT with any number of CHRS chunks--replace the ID_BODY case with
 * an ID_CHRS case that doesn't set iffp = IFF_DONE, and make the END_MARK
 * case do whatever cleanup you need.
 *
 ******************************************************************************/


/** GetPrILBM() ****************************************************************
 *
 * Called via myReadPicture to handle every PROP encountered in an IFF file.
 * Reads PROPs ILBM and skips all others.
 *
 ******************************************************************************/
#if Fancy
IFFP GetPrILBM(parent)  GroupContext *parent;  {
   /*compilerBug register*/ IFFP iffp;
   GroupContext propContext;
   ILBMFrame *ilbmFrame = (ILBMFrame *)parent->clientFrame;

   if (parent->subtype != ID_ILBM)
      return(IFF_OKAY);   /* just continue scaning the file */

   iffp = OpenRGroup(parent, &propContext);
   CheckIFFP();

   do switch (iffp = GetPChunkHdr(&propContext)) {
      case ID_BMHD: {
         ilbmFrame->foundBMHD = TRUE;
         iffp = GetBMHD(&propContext, &ilbmFrame->bmHdr);
         break; }
      case ID_CAMG: {       /* cs */
         iffp = handleCAMG(&propContext, ilbmFrame);
```



```
         break; }
      case ID_CRNG: {       /* cs */
         iffp = handleCRNG(&propContext, ilbmFrame);
         break; }
      case ID_CCRT: {       /* cs */
         iffp = handleCCRT(&propContext, ilbmFrame);
         break; }
      case ID_CMAP: {
         ilbmFrame->nColorRegs = maxColorReg; /* room for this many */
         iffp = GetCMAP(&propContext,
                        (WORD *)&ilbmFrame->colorMap,
                          &ilbmFrame->nColorRegs);
         break; }
      } while (iffp >= IFF_OKAY);
        /* loop if valid ID of ignored chunk or a
         * subroutine returned IFF_OKAY (no errors).*/

   CloseRGroup(&propContext);
   return(iffp == END_MARK ? IFF_OKAY : iffp);
   }
#endif

/** GetLiILBM() ****************************************************************
 *
 * Called via myReadPicture to handle every LIST encountered in an IFF file.
 *
 ******************************************************************************/
#if Fancy
IFFP GetLiILBM(parent)  GroupContext *parent;  {
    ILBMFrame newFrame;   /* allocate a new Frame */

    newFrame = *(ILBMFrame *)parent->clientFrame;  /* copy parent frame */

    return( ReadIList(parent, (ClientFrame *)&newFrame) );
    }
#endif

/** myReadPicture() ************************************************************/
IFFP myReadPicture(file,iFrame)
   LONG file;
   ILBMFrame *iFrame;   /* Top level "client frame".*/
   {
   IFFP iffp = IFF_OKAY;

#if Fancy
   iFrame->clientFrame.getList = GetLiILBM;
   iFrame->clientFrame.getProp = GetPrILBM;
#else
   iFrame->clientFrame.getList = SkipGroup;
   iFrame->clientFrame.getProp = SkipGroup;
#endif
   iFrame->clientFrame.getForm = GetFoILBM;
   iFrame->clientFrame.getCat  = ReadICat ;

   /* Initialize the top-level client frame's property settings to the
    * program-wide defaults. This example just records that we haven't read
```

```
    * any BMHD property or CMAP color registers yet. For the color map, that
    * means the default is to leave the machine's color registers alone.
    * If you want to read a property like GRAB, init it here to (0, 0). */

   iFrame->foundBMHD  = FALSE;
   iFrame->nColorRegs = 0;
   iFrame->foundCAMG  = FALSE;    /* cs */
   iFrame->cycleCnt   = 0;        /* cs */
   giFrame = iFrame;

  /* Store a pointer to the client's frame in a global variable so that
   * GetFoILBM can update client's frame when done.  Why do we have so
   * many frames & frame pointers floating around causing confusion?
   * Because IFF supports PROPs which apply to all FORMs in a LIST,
   * unless a given FORM overrides some property.
   * When you write code to read several FORMs,
   * it is ssential to maintain a frame at each level of the syntax
   * so that the properties for the LIST don't get overwritten by any
   * properties specified by individual FORMs.
   * We decided it was best to put that complexity into this one-FORM example,
   * so that those who need it later will have a useful starting place.
   */

   iffp = ReadIFF(file, (ClientFrame *)iFrame);
   return(iffp);
   }
```

AMIGA™

EA IFF 85
STANDARD FOR INTERCHANGE
FORMAT FILES
08/01/87

commodore COMPUTERS

Computer Systems Division
1200 Wilson Drive
West Chester, PA 19380