OS-9/IFF
SUPPORT LIBRARY

REPRODUCTION NOTICE

The software described in this document is intended to be used on a single computer system. OptImage expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of OptImage and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

For additional copies of this software and/or documentation, or if you have questions concerning the above notice, the documentation, and/or software, please contact the OptImage representative in your area.

DISCLAIMER

The information contained herein is believed to be accurate as of the date of publication, however, OptImage will not be liable for any damages, including indirect or consequential, from the use of OptImage-provided software or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

COPYRIGHT

Publication Editor: Walden Miller & Steve McClellan
Publication date: December 10, 1992
Documentation Part Number: None
Revision B

TRADEMARKS

OS-9 is a trademark of Microware Systems Corporation.
OS-9/68000 is a trademark of Microware Systems Corporation.
CD-I is a trademark of N.V. Philips and Sony Corporation

OptImage Interactive Services Company, L.P.
1501 50th Street, Suite 100
West Des Moines, Iowa 50265-5961

Phone numbers:

In the USA:	**1 800 CDI-5484**
	(515) 225-7000
	(515) 225-0252 FAX
In Europe:	**+31 (40) 73 6228**
In Japan:	**(03) 3 665 9740**

Online Information:

Applelink® D6431
America Online®, Keyword: OptImage
OptImage B.B.S. (515) 225-1933

Table of Contents

OS-9/Support Library C Functions

CD-I IFF Specification: Version 0.99 6 January 1989

A Quick Introduction to IFF

"EA IFF 85" Standard For Interchange Format Files: August 1, 1987

Audio Interchange File Format: AIFF

Revision Log

OS-9/IFF Support Library C Functions

Introduction

The OS-9/IFF Support Library provides a relativly simple method for reading and writing correct IFF formated data. Developers are encouraged to use these libraries to help maintain compatibility among different implementations of IFF reader/writer code.

The OS-9/IFF Support Library is provided in source code format to aid tool developers conform to the 0.99 CD-I IFF specification.

This manual is divided into six chapters: The first chapter details the actual OS-9 IFF C functions, while the remaining chapters are the IFF Specification and its appendices. The IFF Specification and its appendices are reprinted with permission from the various authors.

iff_close() Stop Reading/Writing IFF File

SYNOPSIS:

```
#include <iff.h>

int iff_close(context)
IFF_CONTEXT *context;          /* struct allocated by iff_open() */
```

FUNCTION: This function stops the processing of the IFF file specified by **context**. If the file was open for writing on an RBF device, the size in the **CAT** will be updated with the actual size of the data written.

On error, -1 is returned.

iff_open() Read/Write first 12 bytes of IFF File

SYNOPSIS:

```
#include <iff.h>

IFF_CONTEXT *iff_open(path,rwmode,catid)
short   path;           /* previously opened path to IFF File */
short   rwmode;         /* mode to open file:  IFF_READ or IFF_WRITE */
int     catid;          /* type of CAT to be written */
```

FUNCTION:

This function reads or writes the first 12 bytes of information to/from a previously opened IFF file. **Iff_open()** allocates and initializes an **IFF_CONTEXT** structure for this. The structure is allocated by the IFF code and a pointer to it is returned to the caller. The **IFF_CONTEXT** structure is defined in **iff.h** and below:

```
typedef struct {
    short   path;           /* path number for file */
    short   mode;           /* read/write mode for this path */
    short   nextformread;   /* next FORM has already been read */
    int     filepos;        /* current file position */
    short   filetype;       /* file type (RBF or PIPE) */
    int     catid;          /* CAT id */
    int     catpos;         /* file position for size of CAT */
    int     catsize;        /* original size of CAT */
    int     catlen;         /* current length of CAT */
    int     formid;         /* FORM id */
    int     formpos;        /* file position for size of FORM */
    int     formsize;       /* original size of FORM */
    int     formlen;        /* current length of FORM */
    int     datapos;        /* file position for size of data */
    int     datasize;       /* original size of data */
    int     datalen;        /* current length of data */
    int     framepos;       /* file position of SampleFrame */
    int     framefctr;      /* SampleFrame = datasize/framefctr */
} IFF_CONTEXT;
```

rwmode specifies whether **path** will be used for reading or writing. This parameter may have the value **IFF_READ** or **IFF_WRITE**. If the file is open for writing then **catid** is used to specify the type of **CAT** to be written. If the file is open for reading then the **CAT** id in the file must match **catid**. If **catid** is **IFF_CAT_UNKNOWN** then any **CAT** id is allowed. If the file being read does not contain an enclosing **CAT**, then the id must be specified as unknown.

Null is returned on error.

iff_read_aiff() — Read AIFF Header

SYNOPSIS:

```
#include <iff.h>
#include <iff_aiff.h>

int iff_read_aiff(context,aiff)
IFF_CONTEXT *context;       /* struct allocated by iff_open() */
IFF_AIFF    *aiff;          /* aiff chunk */
```

FUNCTION: This function reads the header portion of an **AIFF** type **FORM** from the file specified by **context**. This function returns the size of the data.

If there is no **FORM** active (i.e. **iff_read_what_next()** has not been called) or the current **FORM** is not of type **AIFF**, then -1 is returned along with the error code in **errno**.

iff_read_data() — Read FORM data

SYNOPSIS:

```
#include <iff.h>

int iff_read_data(context,buffer,length)
IFF_CONTEXT *context;       /* struct allocated by iff_open() */
char        *buffer;        /* ptr to data buffer */
int         length;         /* length of data to write */
```

FUNCTION: This function reads data for a specific **FORM** from the file specified by **context**. The header for the **FORM** must have already been read by the appropriate **iff_read** function. The number of bytes read is returned. Zero is returned on end of **FORM**.

On error, -1 is returned.

iff_read_imag() — Read FORM IMAG Header

SYNOPSIS:

```
#include <iff.h>
#include <iff_imag.h>

int iff_read_imag(context,imag)
IFF_CONTEXT *context;        /* struct allocated by iff_open() */
IFF_IMAG    *imag;           /* imag chunk */
```

FUNCTION: This function reads the header portion of an **IMAG** type **FORM** from the file specified by **context**. The variables **imag->plte_data**, **imag->yuv_data**, and **imag->user_-data** must be initialize to zero or point to a sufficiently large buffer. This function returns the size of the data.

If there is no **FORM** active (i.e. **iff_read_what_next()** has not been called) or the current **FORM** is not of type **IMAG**, then -1 is returned along with the error code in **errno**.

iff_read_what_next() — Read Next FORM Type/Size

SYNOPSIS:

```
#include <iff.h>

int iff_read_what_next(context)
IFF_CONTEXT *context;        /* struct allocated by iff_open() */
```

FUNCTION: This function skips to the start of the next **FORM** within the file specified by **context** and reads its type and size. If the current **FORM** size is unknown, an error will be returned. The **FORM** type is returned to the caller. This must be done so that the application will know which **FORM** reader to called to read the actual header data. Zero is return on end of file.

On error, -1 is returned.

iff_seek() — Seek To New File Position

SYNOPSIS:

```
#include <iff.h>

int iff_seek(context,position,place)
IFF_CONTEXT *context;
int         position,
            place;
```

FUNCTION: This function will reposition the file pointer within the file and data chunk pointed to by **context**, to the byte offset given in **position**. The argument **place** specifies the base from which the offset is based: if 0, from the beginning of the data chunk, if 1, from the current position, or if 2, from the end of the data chunk.

The value returned is the resulting position in the file. If there is an error, -1 is returned and the appropriate error code is placed in the global variable **errno**.

If the **FORM** header has not been read with the appropriate **iff_read** function or the data chunk does not have a known size, then an error is returned.

iff_skip_data() — Skip FORM data

SYNOPSIS:

```
#include <iff.h>

int iff_skip_data(context,length)
IFF_CONTEXT *context;       /* struct allocated by iff_open() */
int         length;         /* length of data to skip */
```

FUNCTION: This function will skip the data for the current **FORM** in the file specified by **context**. If the **FORM** header has not been read with the appropriate **iff_read** function, then an error is returned. If the specified **length** is zero, a skip is done to the end of the **FORM** data. The number of bytes actually skipped is returned. Zero is returned if already at end of **FORM**.

On error, -1 is returned.

iff_write_aiff() — Write AIFF Chunk

SYNOPSIS:

```
#include <iff.h>
#include <iff_aiff.h>

int iff_write_aiff(context,aiff,datasize)
IFF_CONTEXT *context;     /* struct allocated by iff_open() */
IFF_AIFF    aiff;         /* AIFF chunk */
int         datasize;     /* size of data to write */
```

FUNCTION: This function will write everything in the specified **AIFF** chunk except the actual audio data to the file specified by **context**. If the size of the data is unknown, then it will be updated when the writing of the data is complete. In this case, the reader will not be able to read the correct amount of data unless it already knows its length.

On error, -1 is returned.

iff_write_data() — Write Form data

SYNOPSIS:

```
#include <iff.h>

int iff_write_data(context,buffer,length)
IFF_CONTEXT *context;     /* struct allocated by iff_open() */
char        *buffer;      /* ptr to data buffer */
int         length;       /* length of data to write */
```

FUNCTION: This function writes the data pointed to by **buffer** for a specific IFF **FORM** from the file specified by **context**. The **FORM** must have already been started by the appropriate **iff_write** function. This function returns the number of bytes actually written.

On error, -1 is returned.

iff_write_end() — Stop Writing FORM data

SYNOPSIS:

```
#include <iff.h>

int iff_write_end(context)
IFF_CONTEXT *context;          /* struct allocated by iff_open() */
```

FUNCTION: This function is used to stop writing of data to a specific **FORM** from the file specified by **context**. If the file was open for writing on an RBF device then the **FORM** and **DATA** sizes will be updated with the actual sizes. The data size is also extended to be an even number of bytes.

On error, -1 is returned.

iff_write_imag() — Write IMAG Chunk

SYNOPSIS:

```
#include <iff.h>
#include <iff_imag.h>

int iff_write_imag(context,imag,datasize)
IFF_CONTEXT *context;          /* struct allocated by iff_open */
IFF_IMAG    imag;              /* IMAG chunk */
int         datasize;          /* size of data to write */
```

FUNCTION: This function will write everything in the specified **IMAG** chunk except the actual video data to the file specified by **context**. If the size of the data is unknown, then it will be updated when the writing of the data is complete. In this case, the reader will not be able to read the correct amount of data unless it already knows its length.

On error, -1 is returned.

CD-I IFF Specification:

Version 0.99
6 January 1989

Preface

This specification is the result of a design session held at the studios of American Interactive Media in Los Angeles, California on May 23-24, 1988. The session was attended by the following people, who share equally in the credit for this document, but are not to be held liable for any misrepresentations of their ideas by the author:

Andy Davidson, AIM
Tom DiDomenico, OptImage
Alain Delpuch, Thomson
Todd Earles, Microware
Dave Feldman, ICOM
Todd MacMillan, Sun Microsystems
Steve Minar, Electronic Arts
George Rebane, Philips/AIM

Changes to the original specification (summarized in Appendix D) were motivated by com-

ments from the participants in the above session, plus those of the following people:

Don Leeper
Drew Topel, AIM
Robert Patton, AIM
Jay Zipnick, ICOM
Jerry Morrison, Electronic Arts
Mitsu Hadeishi, Electronic Arts
Steve McClellan, Microware
Eric Miller, Microware
Dave Clark, Philips

Accompanying this specification is a set of software libraries, in source code format, which tool developers can incorporate in their applications to process CD-I IFF files. Questions about the specification or the libraries should be directed to the author:

Andrew Davidson
American Interactive Media
11050 Santa Monica Boulevard - Suite 300
Los Angeles, California 90025
USA

Telephone: 213-473-4050
Fax: 213-477-4953

Introduction

CD-I IFF is the standard format in which all files used in the CD-I production process are stored. This specification documents that format and describes the kinds of data which may be stored in a CD-I IFF file.

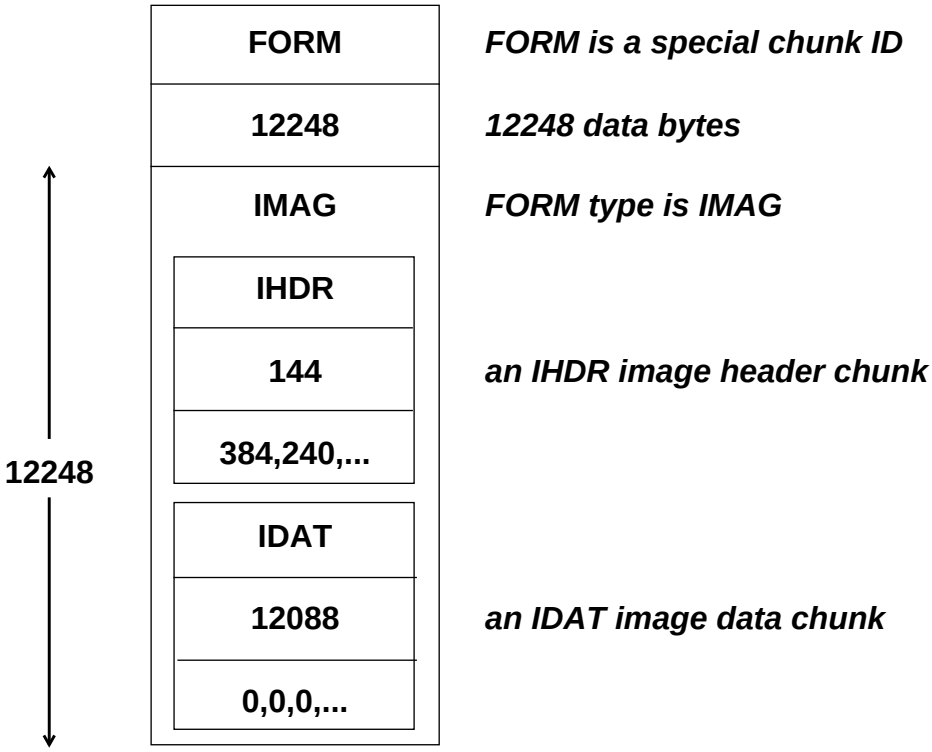
The underlying framework for this standard is that of IFF - Interchange File Format. This existing standard, developed by Electronic Arts, is documented completely in ***EA IFF 85 - Standard for Interchange Files*** and explained succinctly in ***A Quick Introduction to IFF***. It is recommended that the reader have an understanding of IFF before reading this specification.

IFF is a framework upon which interchange formats can be built for files used in a variety of application areas. The CD-I community has adopted this framework for its use, and has developed a standard set of file formats for use by CD-I tool developers. The goal of this standard is to allow data to be exchanged conveniently among different software tools and different computer systems. It is not intended to provide a data base management system for the storage of CD-I data.

IFF for CD-I

A CD-I IFF file may contain a single image, a single sound, or any combination of data types which are defined in this specification and allowed by the rules of IFF. See ***A Quick Introduction to IFF*** for a description of IFF files in general.

An example of a typical CD-I IFF file (one with a single image contained in it) is shown below:



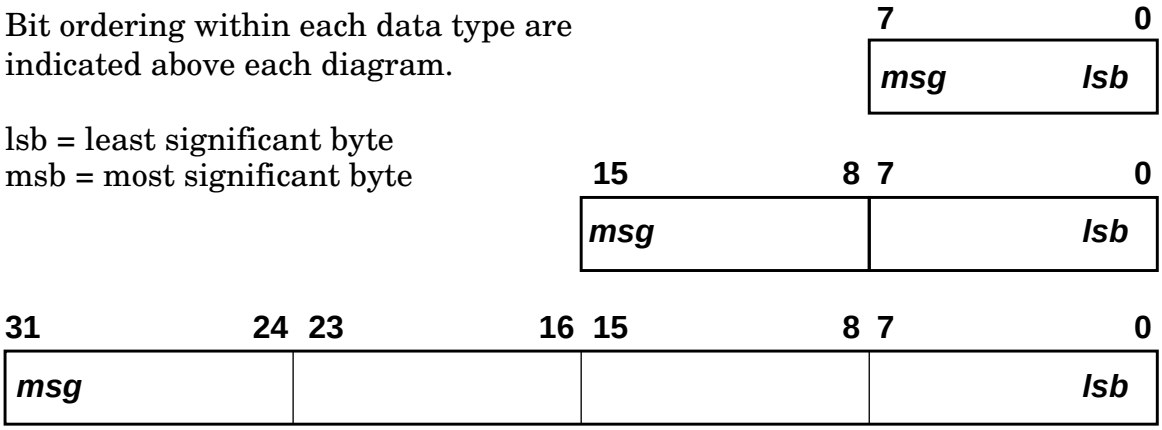
Data Definitions

The following definitions apply to all CD-I IFF data. Note that the convention for these definitions is as defined for the Motorola 68000 family processors.

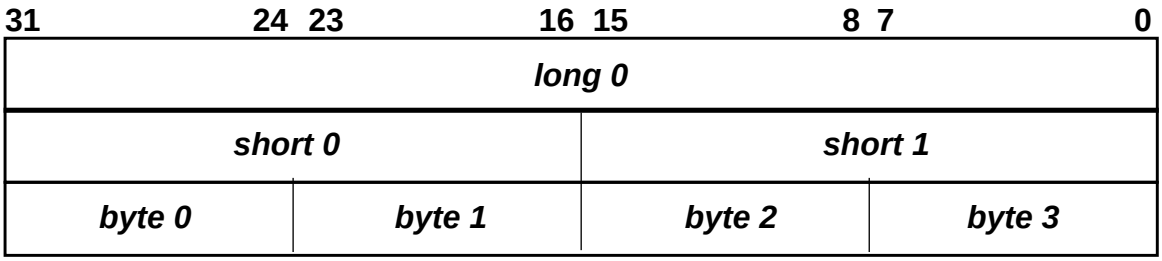
Data Types

Bit ordering within each data type are indicated above each diagram.

lsb = least significant byte
msb = most significant byte



Data Ordering



Common Declarations

The following data types are common to all of the chunk definitions listed below:

```
typedef char                IFF_8;      /* 8-bit signed int */
typedef short               IFF_16;     /* 16-bit signed int */
typedef long                IFF_32;     /* 32-bit signed int */

typedef unsigned char       IFF_U_8;    /* 8-bit unsigned int */
typedef unsigned short      IFF_U_16;   /* 16-bit unsigned int */
typedef unsigned long       IFF_U_32;   /* 32-bit unsigned int */

typedef struct
{                                       /* color value tuple */
    IFF_U_8    r,g,b;
} IFF_COLOR;

/*  NOTE: the total size of this struct is 24 bits, which is not an integral number of
    16-bit words.  Many C compilers will automatically insert a pad byte into struc-
    tures to force word alignment.  It is the responsibility of the application program
    when reading and writing these structures to insure that no pad bytes are
    present in the file.  Each color value on disk must occupy exactly 24 bits.
*/

typedef struct
{                                       /* YUV value tuple */
    IFF_U_8    y,u,v;
} IFF_YUV;

/*  NOTE: the total size of this struct is 24 bits, which is not an integral number of
    16-bit words. Many C compilers will automatically insert a pad byte into struc-
    tures to force word (16-bit) alignment.  It is the responsibility of the application
    program when reading and writing these structures to insure that no pad bytes
    are present in the file. Each YUV value on disk must occupy exactly 24 bits.
*/
```

FORM & Chunk Definitions

IMAGs (Image) FORM

All CD-I image formats (RGB, CLUT, DYUV, and RL) are stored in a single **FORM** called **IMAG**. The **IMAG FORM** has no fields or data of its own; it simply signals the beginning of the collection of chunks to follow.

The first chunk in every **IMAG FORM** must be a standard header, of type **IHDR**. Following that are a number of optional chunks, depending upon the type of image being represented. The last chunk is, with one exception noted below, always of type **IDAT**, and contains the actual image data. The component chunks of an **IMAG FORM** are as follows:

IMAG	CD-I image
IHDR	standard header
IPAR	optional image parameters
PLTE	optional palette
YUVS	optional DYUV start values
USER	optional user-defined data
IDAT	actual image data

Although CLUT images will likely have palettes associated with them (and contained in the same **IMAG FORM**), it is also possible to have a file which contains just a palette. In that case, there is no **IDAT** chunk in the **IMAG FORM**, and the last chunk is a **PLTE** instead.

IHDR (Image Header) Chunk

The image header contains fields whose contents define the structure and organization of the image stored in the remaining chunks in the **FORM**. The **IHDR** chunk, which is required and must be the first chunk in the **IMAG FORM**, is defined as follows:

```
/* image header */
typedef struct
{
    IFF_U_16   ihdr_width;          /* logical width of image (number of significant
                                       pixels in each scan line) */

    IFF_U_16   ihdr_line_size;      /* physical width of image (number of bytes in
                                       each scan line, including any data required at
                                       the end of each scan line for padding [see de-
                                       scription of each model's IDAT chunk for pad-
                                       ding rules]) This field is not used when ih-
                                       dr_model = IFF_MDL_RL7 or IFF_MDL_RL3.
                                       When ihdr_model = IFF_MDL_RGB555, this
                                       field defines the size of one scan line of the up-
                                       per or  lower portion of the pixel data, but not
                                       the size of them both together. */

    IFF_U_16   ihdr_height;         /* logical height of image (number of scan lines)
                                       */

    IFF_U_16   ihdr_model;          /* image model (coding method) */

    IFF_U_16   ihdr_depth;          /* physical size of pixel (number of bits per pixel
                                       used in storing image data) When
                                       ihdr_model = IFF_MDL_RL7 or IFF_MDL_-
                                       RL3, this value only represents the size of a
                                       single pixel; the size of a run of pixels is inde-
                                       terminate */

    IFF_U_8    ihdr_dyuv_kind;      /* if ihdr_model = IFF_MDL_DYUV, indicates
                                       whether there is one DYUV start value for all
                                       scan lines (in ihdr_dyuv_start), or whether
                                       each scan line has its own start value in the
                                       YUVS chunk which follows */

    IFF_YUV    ihdr_dyuv_start;     /* start values for DYUV image if
                                       ihdr_model = IFF_MDL_DYUV and
                                       ihdr_dyuv_kind = IFF_DYUV_ONE */

} IFF_IHDR;
```

Note that the total size of the image data is NOT stored in the image header. This value, which is necessary for processing the data and includes all padding and is independent of the model in effect, is found in the (standard IFF) byte count field of the **IDAT** chunk.

```
/* Definitions for values used in the field ihdr_model */

#define IFF_MDL_RGB888      1    /* red, green, blue - 8 bits per color */
#define IFF_MDL_RGB555      2    /* Green Book absolute RGB */
#define IFF_MDL_DYUV        3    /* Green Book Delta YUV */
#define IFF_MDL_CLUT8       4    /* Green Book 8 bit CLUT */
#define IFF_MDL_CLUT7       5    /* Green Book 7 bit CLUT */
#define IFF_MDL_CLUT4       6    /* Green Book 4 bit CLUT */
#define IFF_MDL_CLUT3       7    /* Green Book 3 bit CLUT */
#define IFF_MDL_RL7         8    /* Green Book runlength coded 7 bit CLUT */
#define IFF_MDL_RL3         9    /* Green Book runlength coded 3 bit CLUT */
#define IFF_MDL_PLTE        10   /* color palette only */

/* Definitions for values used in the field ihdr_dyuv_kind */

#define IFF_DYUV_ONE        0    /* one start value for all scan lines */
#define IFF_DYUV_EACH       1    /* start value for each scan line */
```

For the following values of the field **ihdr_model**:

IFF_MDL_RGB888
IFF_MDL_RGB555
IFF_MDL_DYUV
IFF_MDL_CLUT8
IFF_MDL_CLUT7
IFF_MDL_CLUT4
IFF_MDL_CLUT3

the following relationships apply:

image size (in IDATS chunk) = **ihdr_height * ihdr_line_size**
ihdr_line_size = (ihdr_width * ihdr_depth)/8 + required padding

See the description of each model's **IDAT** chunk for information on required padding on each scan line.

IPAR (Image Parameters) Chunk

Image parameters are fields whose contents provide information associated with the image, but not required in order to display it.

The **IPAR** chunk, which is optional, is defined as follows:

```
/* image parameters */
typedef struct
{
    IFF_16      ipar_x_off,
                ipar_y_off;         /* offset of origin in source image
                                       [0 ≤ ipar_x_off ≤ ipar_x_page]
                                       [0 ≤ ipar_y_off ≤ ipar_y_page] */
    IFF_U_16    ipar_x_ratio,
                ipar_y_ratio;       /* aspect ratio of pixels in source image;
                                       ratio is ipar_y_ratio/ipar_x_ratio.
                                       E.g. 1.333:1 is ipar_y_ratio=4,
                                       ipar_x_ratio=3 */
    IFF_U_16    ipar_x_page,
                ipar_y_page;        /* size of source image */
    IFF_U_16    ipar_x_grab,
                ipar_y_grab;        /* location of hot spot within image */
    IFF_COLOR ipar_trans;           /* transparent color */
    IFF_COLOR ipar_mask;            /* mask color */
} IFF_IPAR;
```

PLTE (Palette) Chunk

A palette is a set of color values from which the colors in a CLUT image are assigned. It is a zero-based indexed table of color values.

The **PLTE** chunk, which is optional, is defined as follows:

```
/* palette */
typedef struct
{
    IFF_U_16     plte_offset;          /* offset (entry number) from start of
                                          CLUT where this palette is to be
                                          loaded */
    IFF_U_16     plte_count;           /* number of entries in this palette */
    IFF_COLOR plte_data[plte_count];   /* color values for palette */
} IFF_PLTE;

/* NOTE that one entry in a palette consists of a single color, which includes red,
   green, and blue values. See the description about storage allocation of this type in
   the description of IFF_COLOR in the Common Declarations section above.
*/
```

YUVS (YUV Start Values) Chunk

YUV start values are a set of DYUV values which define the color of the first pixel on each scan line in an image.

If the value of the field **ihdr_model** in the **IHDR** chunk is equal to **IFF_MDL_DYUV**, then the **YUVS** chunk must be present if the value of the field **ihdr_dyuv_kind** in the **IHDR** chunk is equal to **IFF_DYUV_EACH** and should not be present if the value of that field is equal to **IFF_DYUV_ONE**. The **YUVS** chunk is defined as follows:

```
/* YUV start values */
typedef struct
{
    IFF_YUV yuvs_value[ihdr_height];        /*YUV value for each scan line - the
                                              number of these values is equal to
                                              the value of the field ihdr_height in
                                              the IHDR chunk */
} IFF_YUVS;
```

USER (User-Defined Data) Chunk

The **USER** chunk is provided merely as a convenient way of storing comments, copyright notices, or other text information. It should not be used for storing processing parameters or other data.

The format of the **USER** chunk, which is optional, is defined by the application which processes it and is not specified here.

IDAT (Image Data) Chunk

The **IDAT** chunk contains the actual data representing the image stored in the file. The format of this data depends upon the model in effect for the image, as defined in the **ihdr_model** field of the **IHDR** chunk. The cases are described on the following pages.

In all cases, the order in which pixels appear is as follows:

P(0,0), P(0,1), P(0,2), ... P(0,ihdr_width),
P(1,0), P(1,1), P(1,2), ... P(1,ihdr_width),
...
P(ihdr_height,0), P(ihdr_height,1), P(ihdr_height,2), ...
P(ihdr_height,ihdr_width)

where **ihdr_height** and **ihdr_width** are the values in those fields in the **IHDR** chunk, and the notation **P(m,n)** means the **n**th pixel on the **m**th scan line.

IFF_MDL_RGB888

The RGB888 color model is a general purpose, machine-independent RGB image representation. Many images are stored in their original digital format using this representation. It is not related to the CD-I internal RGB representation. That color model is described below, in the **IFF_MDL_RGB555** section.

For RGB888 images, the data is sinply the color (RGB values) for each pixel in the image. The format of that data is as follows:

```
/* IFF_MDL_RGB888 image data */

typedef struc
{
    IFF_COLOR idat_rgb888[n];        /* This is simply a stream of color values for
                                        each pixel in the image. No padding bytes
                                        are inserted in the data. The value of n =
                                        ihdr_width * ihdr_height, where ihdr_width
                                        and ihdr_height are the values found in
                                        those fields in the IHDR chunk. */
} IFF_IDAT_RGB888;
```

IFF_MDL_RGB555

For RGB555 images, the data is an absolute representation of the color of each pixel as described in the CD-I Green Book. In this model, one 16 bit word is used to represent each pixel's color, but the word is split into two halves, and each half (byte) is stored separately. The format of the data is as follows:

```
/* IFF_MDL_RGB555 image data */
typedef struct
{
    IFF_U_8  idat_rgb555_upper [n];     /* Upper bytes of each pixel */
    IFF_U_8  idat_rgb555_lower [n];     /* Lower bytes of each pixel */
                                        /* n = ihdr_line_size *
                                           ihdr_height, where
                                           ihdr_line_size and ihdr_height are the
                                           values found in those fields in the IHDR
                                           chunk. The first pixel of each scan line
                                           must begin in a longword boundary. If
                                           necessary, pad bytes must be present
                                           after the last pixel in each scan line to
                                           achieve this. The value of ihdr_line_size
                                           includes any pad bytes, while the value
                                           of ihdr_width does not. Note that for
                                           this color model, the value of
                                           ihdr_line_size does not represent the
                                           scan line's entire pixel data, but only
                                           that of the upper or lower portion. */
} IFF_IDAT_RGB555;
```

IFF_MDL_DYUV

For DYUV images, the data is a sequence of two byte pairs as described in the CD-I Green Book. (Each pair of bytes represents two adjacent pixels on a scan line.) The format of the data is as follows:

```
/* IFF_MDL_DYUV image data */
typedef struct
{
    IFF_U_8  idat_dyuv [n];        /* This is simply a stream of bytes which contains the DYUV data.
                                      n = ihdr_line_size * ihdr_height where ihdr_line_size and ihdr_height are the values
                                      found in those fields in the IHDR chunk. The first pixel of each scan line must begin on a
                                      longword boundary. If necessary, pad bytes must be present after the last pixel in each
                                      scan line to achieve this. The value of ihdr_line_size includes any pad bytes, while the
                                      value of
                                      ihdr_width does not. */
} IFF_IDAT_DYUV;
```

IFF_MDL_CLUT8
IFF_MDL_CLUT7
IFF_MDL_CLUT4
IFF_MDL_CLUT3

For CLUT images, the image data are sinply indices into a palette for each pixel in the image. The size of the index depends upon the particular CLUT model being represented, and is defined in the CD-I Green Book. The format of the data is as follows:

```
/* IFF_MDL_CLUT image data */
typedef struct
{
    IFF_U_8  idat_clut[n];          /* This is a stream of bytes which contains the
                                       CLUT index data.
                                       n = ihdr_line_size * ihdr_height, where ih-
                                       dr_line_size and ihdr_height are the values
                                       found in those fields in the IHDR chunk. The
                                       first pixel of each scan line must begin on a
                                       longword boundary. If necessary, pad bytes
                                       must be present after the last pixel in each
                                       scan line to achieve this. The value of ih-
                                       dr_line_size includes any pad bytes, while the
                                       value of
                                       ihdr_width does not. */
} IFF_IDAT_CLUT;
```

IFF_MDL_RL7
IFF_MDL_RL3

For RL images, the data is a sequence of bytes which represents run-length coding of CLUT image data. This encoding scheme is sensitive to image content, and there is no predictable relationship between the dimensions of the original image and the size of its run-length coded version. The exact encoding scheme is described in the CD-I Green Book. The format of the data in this model is as follows:

```
/* IFF_MDL_RL image data */
typedef struct
{
    IFF_U_8  idat_rl [n];          /*  This is simply a stream of bytes which contains the RL data.  The total number of bytes depends upon the picture contents and is found in the IFF byte count field of this chunk.
                                   */
} IFF_IDAT_RL;
```

IFF_MDL_PLTE

For palettes, the **IDAT** chunk does not exist, but a **PLTE** chunk is present instead. The format of that chunk has been documented above.

AIFF (Audio) FORM

The CD-I audio formats (PCM and ADPCM) are stored in a single **FORM** called **AIFF**. The format of this **FORM** is exactly as specified in the chapter ***Audio Interchange File Format: AIFF, A Standard for Sampled Sound Files, Version 1.2***.

The **AIFF FORM**, as specified in that chapter, is used to represent PCM audio data. It is extended to support CD-I audio data with one additional chunk definition. The existing **AIFF FORM** stores the actual sampled PCM data in an **SSND** chunk. If, instead of an **SSND** chunk, an **APCM** chunk is encountered in the file, then the data contained in it is assumed to be ADPCM data, rather than PCM data. The physical representation of this part of the ADPCM data is identical to that of PCM data; the contents are identified only by the id of the enclosing chunk.

end of chapter 2

NOTES

CD-I IFF Specification:

A Quick Introduction To IFF

Jerry Morrison, Electronic Arts

10-17-88

IFF is the Amiga-standard ***Interchange File Format***. It's designed to work across all CPUs.

Why IFF?

Did you ever have this happen to your picture file?

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

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

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

What Is IFF?

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

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

IFF is also a design idea.

programs should use interchange formats for their everyday storage

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

What's The Trick?

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

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

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

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

What's An IFF File Look Like?

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

char typeID[4]	**'CMAP'**	**in an ILBM file, CMAP means "color map"**
unsigned long dataSize	**48**	**48 data bytes**
char data[]	**0, 0, 0, 255, 255, 255, ...**	**16 3-byte color values: black, white, ...**

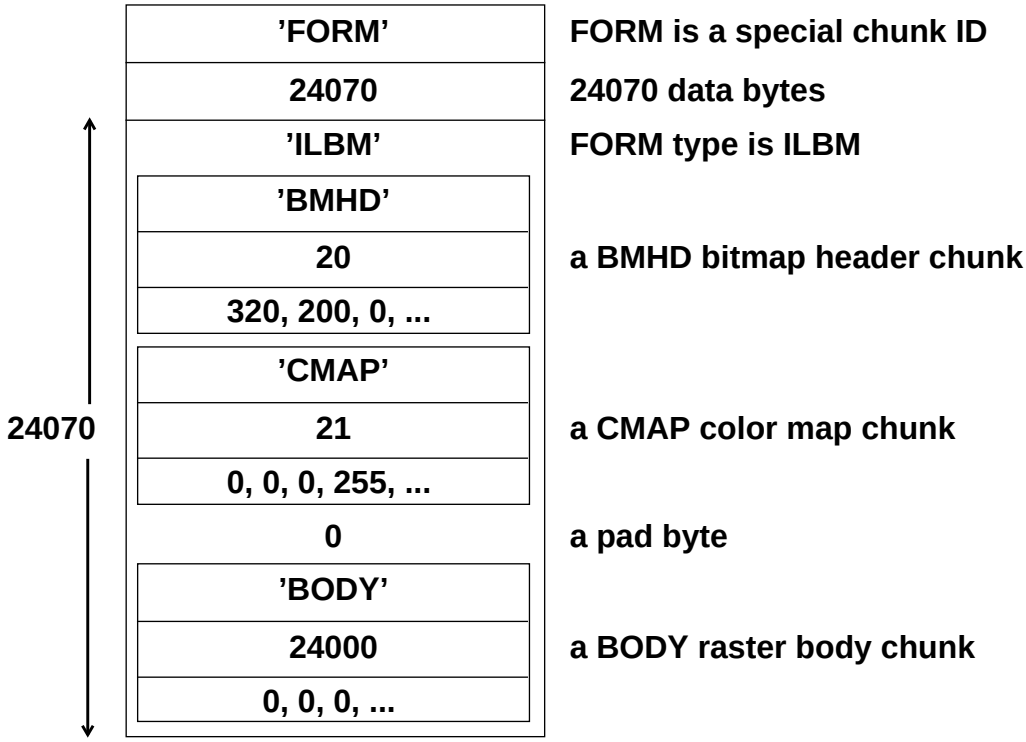
A chunk is made of a 4-character type identifier, a data byte count, and the data bytes. It's basically a Macintosh "resource" with a 32-bit size.

Fine points:

- Every 16- and 32-bit number is stored in 68000 byte order--highest byte first. An intel CPU must reverse the 2- or 4-byte sequence of each number. This applies to chunk dataSize fields and to numbers inside chunk data. It does not affect character strings and byte data because you can't reverse a 1-byte sequence. But it does affect the 32-bit math used in IFF's MakeID macro.
- Every 16-and 32-bit number is stored on an even address so a 68000 can read IFF data in RAM.
- Every odd-length chunk must be followed by a 0 pad byte. This pad byte is not counted in dataSize.
- An ID is made of 4 ASCII characters in the range " " (space, hex 20) through "~" (tilde, hex 7E).
- IDs are compared using a quick 32-bit equality test. Case matters.

A chunk usually holds a C struct (Pascal record) or any array. E.g. an 'ILBM' picture has a 'BMHD' bitmap header (struct) chunk and a 'BODY' raster body (array) chunk.

To construct an IFF file, just put a file type ID (like 'ILBM') and chunks one after another (with any pad bytes) into a wrapper chunk called a 'FORM' (Think "FILE").



A FORM always contains one 4-character FORM type ID (a file type, in this case ‘ILBM’) followed by any number of data chunks. In this example, the FORM type is ‘ILBM’, which stands for "InterLeaved BitMap". ILBM is an IFF standard for bitplane raster pictures. The bitplanes are interleaved so you can process a scan line at a time. This example has 3 chunks. Note the pad byte after the odd length chunk.

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

How To Read An IFF File?

Given the C subroutine GetChunkHeader

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

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

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

This loop processes each chunk by dispatching to a routine that reads the specific type of chunk data. We don’t assume a particular order of chunks. This is a simple parser.

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

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

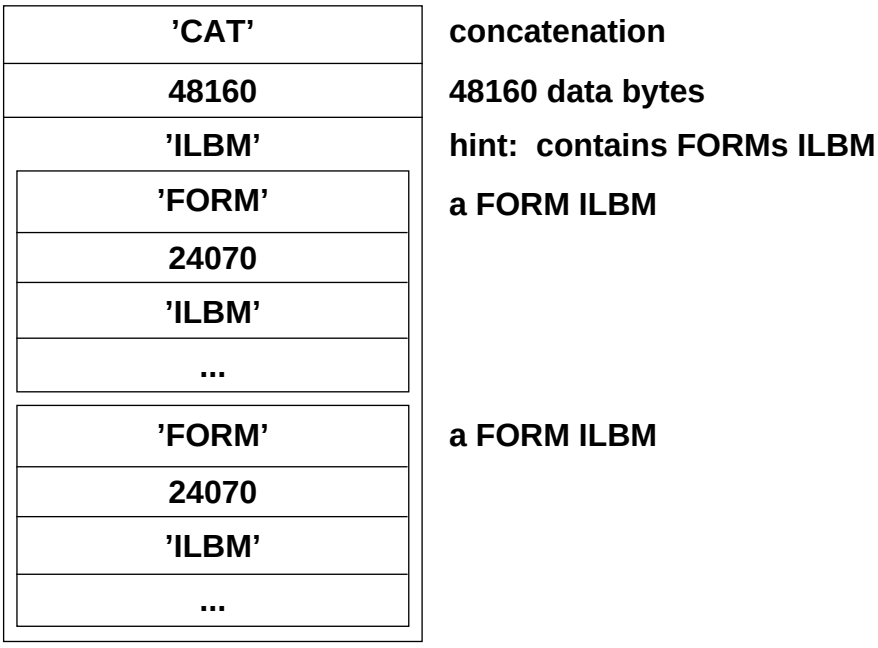
File Extensibility

These techniques make IFF files extensible and forward/backward compatible:

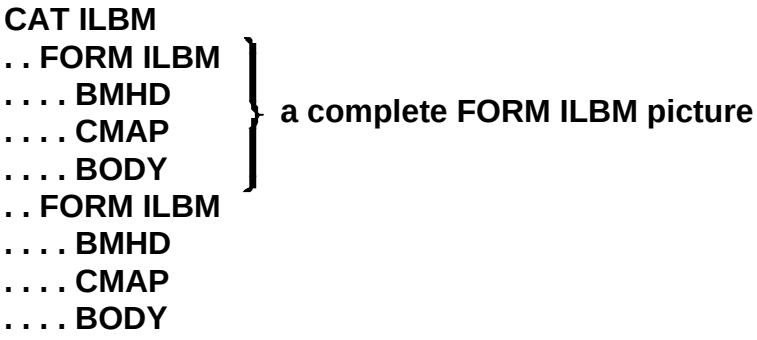
- Design chunk contents for compatibility across environments and for longevity.
- The standards team for a FORM type can extend one if its chunks that contains a study by appending new, optional struct fields.
- Anyone can define new FORM types as well as new chunk types within a FORM type.
- A chunk can be superseded by a new chunk type, e.g. to store more bits per RGB color register. New programs can output the old chunk (for backward compatibility) along with the new chunk.
- If you must change data in an incompatible way, change the chunk ID or the FORM type ID.

Advanced Topics: CAT, LIST, And PROP (not all that important)

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



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



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

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

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

Hints On ILBM Files

Avoid some pitfalls when reading ILBM files.

- Don’t ignore the BitMapHeader.masking field. A bitmap with a mask (such as a partially-transparent DPaint brush or a DPaint picture with a stencil) will read as garbage if you don’t de-interleave the mark.

- Don’t assume all images are compressed. Narrow images aren’t usually run-compressed since the would actually make them longer.

- Don’t assume a particular image size. You may encounter overscan pictures and PAL pictures.

There's a better way to read a BODY than the example IFF code. The GetBODY routine should call a GetScanline routine once per scan line, which calls a GetRow routine for each bitplane in the file. This in turn calls a GetUnpackedBytes routine, which calls a GetBytes routine as needed and unpacks the result. (If the picture is uncompressed, GetRow calls GetBytes directly.) Since the unpacker knows how many packed bytes to read, this avoids juggling buffers for a memory-to-memory UnpackBytes routine.

Caution: If you make many AmigaDOS calls to read or write a few bytes at a time, performance will be mud! AmigaDOS has a very high overhead per call, even with RAM disk. So use buffered read/write routines.

Different hardware display devices have different color resolutions:

Device	R:G:B Bits	maxColor
Mac SE	1	1
IBM EGA	2:2:2	3
Atari ST	3:3:3	7
Amiga	4:4:4	15
CD-I	5:5:5	31
IBM VGA	6:6:6	63
Mac II	8:8:8	255

ILBM CMAP colors are 8:8:8 bits of R:G:B. To read them into hardware colors, just take the high-order bits. E.g. to convert ILBM's 8-bit Red R8 to Amiga's 4-bit red R4, right-shift R4:=R8>>4.

To write hardware colors to ILBM colors, the ILBM specification says just set the high bits (R8:=R4<<4). But you can transmit higher contrast to foreign display devices by scaling the data [0..maxColor] to the full range [0..255]. In other words, R8:=(Rn x 255) ÷ maxColor. (Example: EGA color 1:2:3 scales to 85:170:255.) This makes a big difference where maxColor is less than 15. In the extreme case, Mac SE white (1) should be converted to ILBM white (255), not to IBLM gray (128).

CGA and EGA subtleties

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

IBM CGA colors are 4 bits stored in a byte as xxxxIRGB. (EGA colors in 200 scan line modes are the same as CGA colors, but stored in memory as xxx1xRGB.) That's 3 high-order bits (one for each of R,G, and B) plus one low-order "Intensity" bit for all 3 components R,G, and B. Exception: IBM monitors show IRGB=0110 as brown, which is really the EGA color R:G:B=2:1:0, not dark yellow 2:2:0.

NOTES

"EA IFF 85" Standard For Interchange Format Files

August 1, 1987

1. Introduction

Standards are Good for Software Developers

As home computer hardware evolves to better and better media machines, the demand increase for higher quality, more detailed data. Data development gets more expensive, requires more expenses and better tools, and has to be shared across projects. Thank 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 contectual 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 **I**nterchange **F**ile **F**ormat. 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 is 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 (TM) 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 (copyright) 1981 by Software Arts, Inc. DIF (TM) is the format for speadsheet data interchange developed by Software Arts., Inc. DIF (TM) is a trademark of Software Arts, Inc.

Electronic Arts (TM) is a trademark of Electronic Arts.

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

Inside Macintosh (copyright) 1982, 1983, 1985 Apple Computer, Inc., a programmer's reference manual. Apple "r" is a trademark of Apple Computer, Inc. Macintosh (Trademark) 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 (copyright) 1984, 1982, 1980, 1979 by Motorola, Inc.

PostScript Language Manual (copyright) 1984 Adobe Systems Incorporated. PostScript (trademark) is a trademark of Adobe Systems, Inc. Times and Helvetica (trademark) are trademarks of Allied Corporation.

InterScript: A proposal for a Standard for the Inerchange of Editable Documents (copyright) 1984 Xerox Corporation.

Introduction to InterScript (copyright) 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 is 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 a 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 (conversions).

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 of 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 trade-off 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 structure 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 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 InterScirpt-compatible editor is supposed to preserve the objects it doesn't understand and even maintain nested cross-references. So a simple word processor would let you edit the text of a fancy document without discarding the equations or disrupting the equation numbers.

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

3. Primitive Data Types

Atomic components such as integers and characters that are interpretable directly by the CPU are specified in one format for all processors. We chose a format that's most convenient for the Motorola MC68000 process (M68000 16/32-Bit Microprocessor Programmer's Reference Manual).

N.B.: Part 3 dictates the format for "primitve' data types where-and only where-used in the overall file structure and standard kinds of chunks (Cf. Chucks). 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 filed 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, your 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 stan-

dard 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   */
```

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 not that the FORM type FTXT (formatted text) defines the meaning of these characters within FTXT forms. In particular, character values hex 7F through 9F are control codes while characters hex A0 through hex FF are extended graphic characters like A, 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 restricted 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.

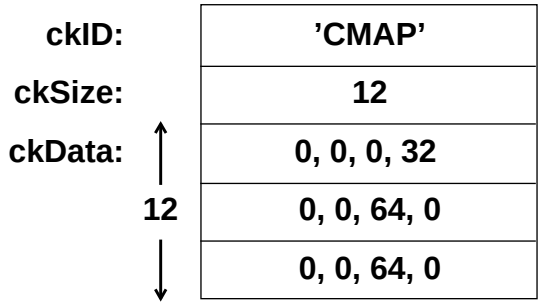
Sometimes it's necessary to make data format changes that aren't backward compatible. Since IDs are used to denote data formats in IFF, new IDs are chosen to denote revised formats. Since programs won't read chunks whose IDs they don't recognize (see Chunks, below), the new IDs keep old programs from 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 ckIDs 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. it 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 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 is 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 propNs are not prop1. The setting prop1 = x has no effect.

The following universal chunk IDs are reserved inside any FORM: "LIST", "FORM", "PROP", "CAT", " ", "LIST", "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 interLeavedBitMap 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 InterLeavedBitMap.

PICS

The data chunk inside a "PICS" data section had 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 InterleavedBitmap 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 overridden for the contents of its local chunks-but not for the chunk structure itself-if your documentation spells out the deviations. If machine-specific type variants are needed, e.g. to store vast numbers of integers in reverse bit order, then outline the conversion algorithm and indicate the variant inside each file, perhaps via different FORM types. Needless to say, variations should be minimized.

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

Separate the central data (like musical notes) from 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 manageable short-let's day <(this is underscored) 256 bytes.

When designing a data object, try to strike a good trade-off 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 created 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, as 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 is 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 the 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 simple 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 tect_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 FORMs 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 FORMs 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 data 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 documents, one page at a time.

Programs should enforce IFF's syntactic rules when reading and writing files. This ensure 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!

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

Single Purpose Files

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

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

on FORM type IDs.

Programmers and users may pick an "intended use" type ad 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 subytpe 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 programers: 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 jut 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 remember 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 "LIST" 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 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 Appendix 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 conditional, 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.

FORMtype 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.

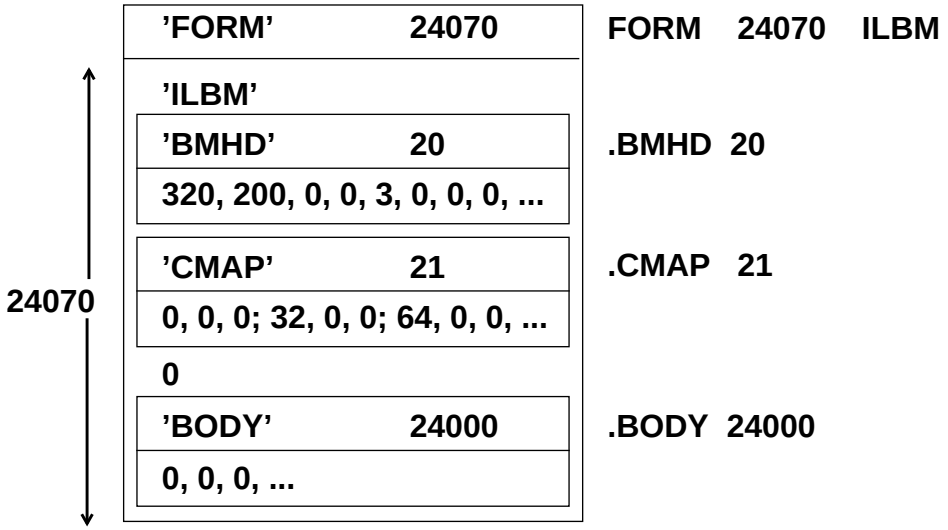
Support Software

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

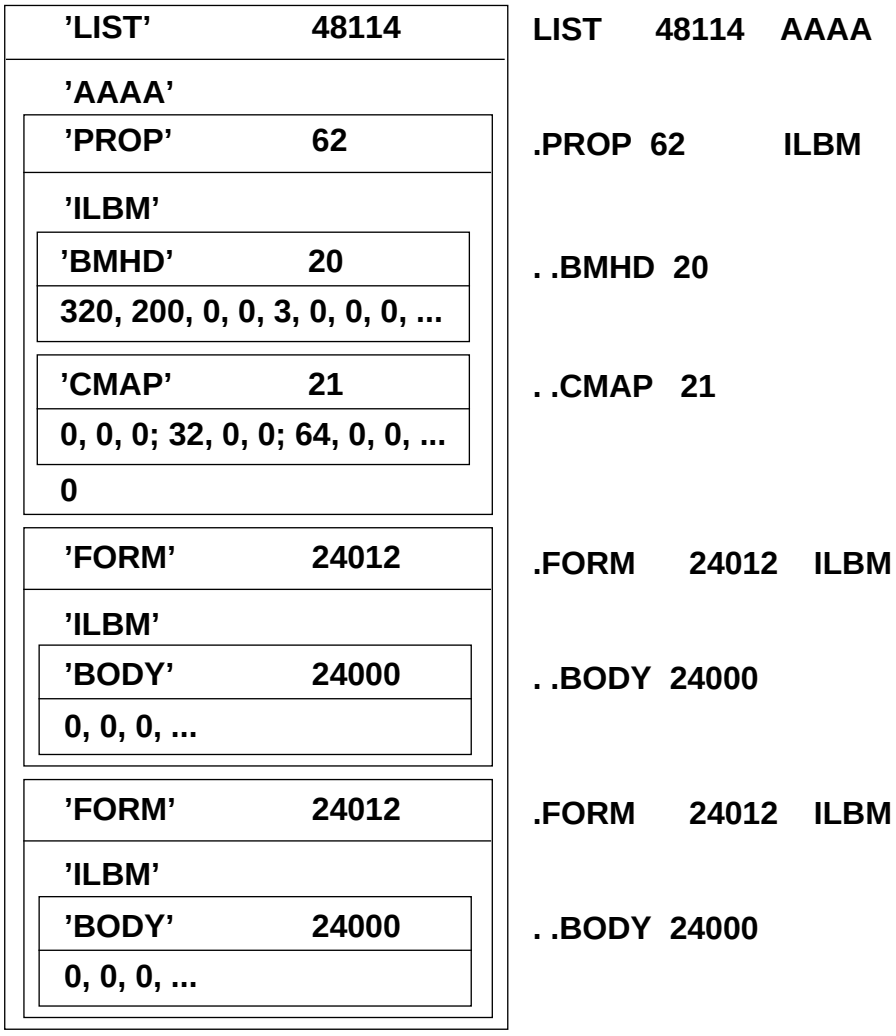
IFF.H, IFFR.C, IFFW.C	IFF reader and writer package. These modules handle many of the details of reliably reading and writing IFF files.
IFFCheck.C	This handy utility program scans an IFF file, checks that the contents are well formed, and prints an outline of the chunks.
Packer.H, Packer.C, UnPacker.C	Run encoder and decoder used for ILBM files.
ILBM.H, ILMBR.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.



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 a la IFFCheck.



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

end of chapter 4

NOTES

AUDIO INTERCHANGE FILE FORMAT: "AIFF"

A Standard For Sample Sound Files Version 1.2 Apple Computer, Inc.

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing for the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths.

Audio IFF conforms to the "EA IFF 85" Standard for Interchange Format Files developed by Electronic Arts.

Audio IFF is primarily an interchange format, although application designers should find it flexible enough to use as a data storage format as well. If an application does choose to use a document. This will facilitate the sharing of sound data between applications.

Audio IFF is the result of several meetings held with music developers over a period of ten months in 1987-88. Apple Computer greatly appreciates the comments and cooperation provided by all developers who helped define this standard.

Another "EA IFF 85" sound storage format is "8SVX" IFF 8-bit Sampled Voice, by Electronic Arts. "8SVX", which handles 8-bit monaural samples, is intended for use with a larger variety of computers, sampled sound instruments, sound software applications, and high fidelity recording devices.

Data Types

A C-like language will be used to describe data structures in this document. The data types used are listed below:

char: 8 bits, signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).

unsigned char: 8 bits, unsigned. Contains any number from zero to 255 (inclusive).

short: 16 bits, signed. Contains any number from -32,768(inclusive).

unsigned short: 16 bits, unsigned. Contains any number from zero to 65,535 (inclusive).

long: 32 bits, signed. Contains any number from -2,147,483,648 to 2,147,483,647(inclusive).

unsigned long: 32 bits, unsigned. Contains any number from zero to 4,294,967,295 (inclusive).

extended: 80 bits IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type Extended).

pstring: Pascal-style string, a one byte count followed by text bytes. The total number of bytes in this data type should be even. A pad byte can be added at the end of the text to accomplish this. This pad byte is not reflected in the count.

ID: 32 bits, the concatenation of four printable ASCII character in the range (SP, 0x20) through '~' (0x7E). Spaces (0x20) cannot precede printing characters; trailing spaces are allowed. Control characters are forbidden.

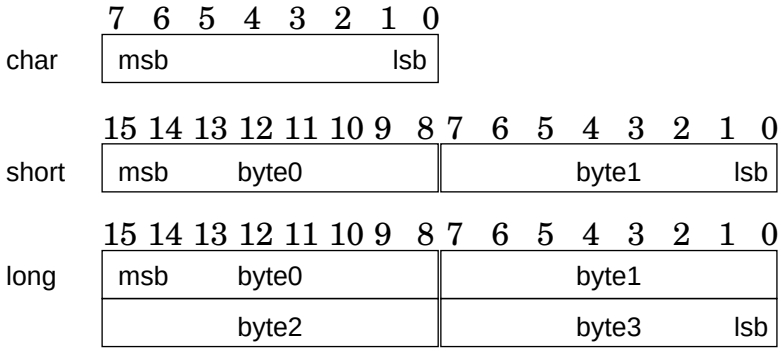
OSType: 32 bits. A concatenation of four characters, as define in Inside Macintosh, vol II.

Constants

Decimal values are referred to as a string of digits, for example 123,0,100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g. 0x0A12,0x1,0x64.

Data Organization

All data is stored in Motorola 68000 format. Data is organized as follows:



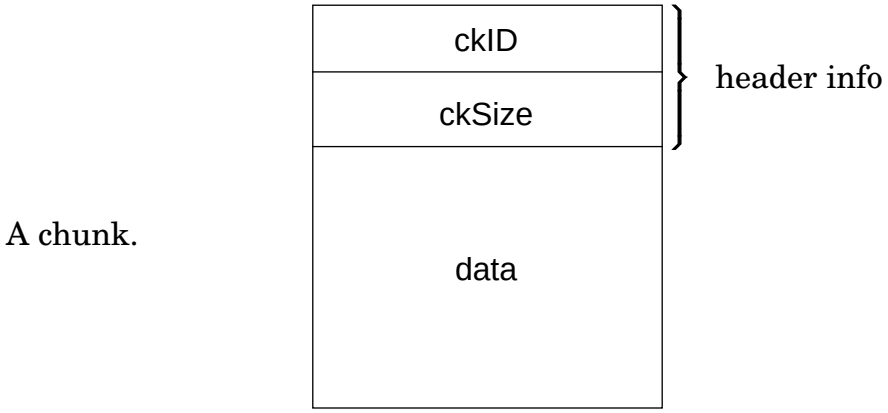
Referring to Audio IFF

The official name for this standard is Audio Interchange File Format. If an application program needs to present the name of this format to a user, such as in a "Save as..." dialog box, the name can be abbreviated to Audio IFF.

File Structure

The ***"EAIFF85" Standard for Interchange Format Files*** defines an overall structure for storing data in files. Audio IFF conforms to the "EAIFF85" standard. This document will describe those portions of "EAIFF85" that are germane to Audio IFF. For a more complete discussion of "EAIFF85", please refer to the document ***"EAIFF85" Standard for Interchange Format Files***.

An "EAIFF85" file is made up of a number of ***chunks*** of data. Chunks are the building blocks of "EAIFF85" files. A chunk consists of some header information followed by data:



A chunk can be represented using our C-like language in the following manner:

```
typedef struct {

   ID          ckID;              /* chunk ID          */
   long        ckSize;            /* chunk Size        */

   char        ckData[];          /* data           */

} Chunk;
```

ckID describes the format of the *data* portion a chunk. A program can determine how to interpret the chunk data by examining **ckID**.

ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**.

ckData contains the data stored in the chunk. The format of this data is determined by **ckID**. If the data is an odd number of bytes in length, a zero pad byte must be added at the end. The pad byte is not included in **ckSize**.

Note that an array with no size specification (e.g. **char ckData[];**) indicates a variable-sized array in our C-like language. This differs from standard C.

An Audio IFF file is a collection of a number of different types of chunks. There is a ***Common Chunk*** which contains important parameters describing the sampled sound, such as it's length and sample rate. There is a ***Sound Data Chunk*** that contains the actual audio samples. There are several other optional chunks that define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in later sections of this document.

The chunks in a Audio IFF file are grouped together in a container chunk. "EAIFF85" defines a number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

```
typedef struct {

    ID          ckID;
    long        ckSize;

    ID          formType;
    char        chunks [];

} Chunk;
```

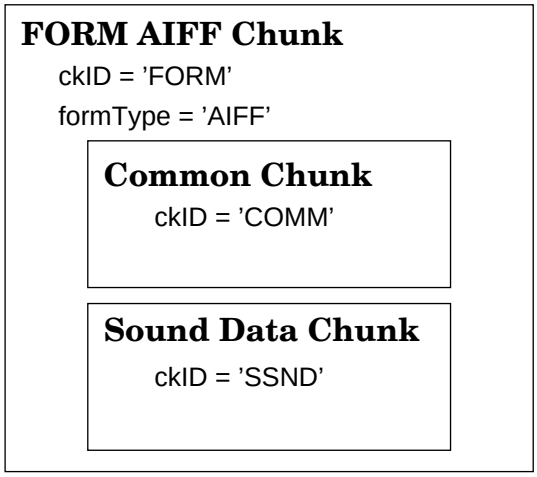
ckID is always 'FORM'. This indicates that this is a FORM chunk.

ckSize contains the size of data portion of the 'FORM; chunk. Note that the data portion has been broken into two parts, **formType** and **chunks[]**.

formType describes what's in the 'FORM' chunk. For Audio IFF files, **formType** is always 'AIFF'. This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of **formType** 'AIFF' is called a ***FORM AIFF***.

chunks are the chunks contained within the FORM. These chunks are called ***local chunks***. A FORM AIFF along with its local chunks make up an Audio IFF file.

Here is an example of a simple Audio IFF file. It consists of a file containing single FORM AIFF which contains two local chunks, a Common Chunk and a Sound Data Chunk.



There are no restrictions on the ordering of local chunks within a FORM AIFF.

On an Apple //, the FORM AIFF is stored in a PRO DOS file. The file type is 0xCB and the aux type is 0x0000.

On a Macintosh, the FORM AIFF is stored in the data fork of an Audio IFF file. The Macintosh file type of an Audio IFF file is 'AIFF'. This is the same as the **formType** of the FORM AIFF.

Macintosh applications should not store any information in Audio IFF file's resource fork, as this information may not be preserved by all applications. Applications can use the ***Application Specific Chunk***, defined later in this document, to store extra information specific to their application.

On an operating system that uses file extensions, such as MS-DOS or UNIX, it is recommended that Audio IFF file names have a ".AIF" extension.

A more detailed example of an AudioIFF file can be found in Appendix A. Please refer to this example as often as necessary while reading the remainder of this document.

Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections. The **ckIDs** for each chunk are also defined.

There are two types of chunks, those that are required and those that are optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has greater than zero length. All other chunks are optional. All applications that use FORM AIFF must be able to read the required chunks, and can choose to selectively ignore the optional chunks. A program the copies a FORM AIFF should copy all of the chunks in the FORM AIFF.

To insure that this standard remains usable by all developers, only Apple Computer, Inc. should define new chunk types for FORM AIFF. If you have suggestions for new chunk types, Apple is happy to listen! Please refer to Appendix B for instructions to how to send comments to Apple.

Common Chunk

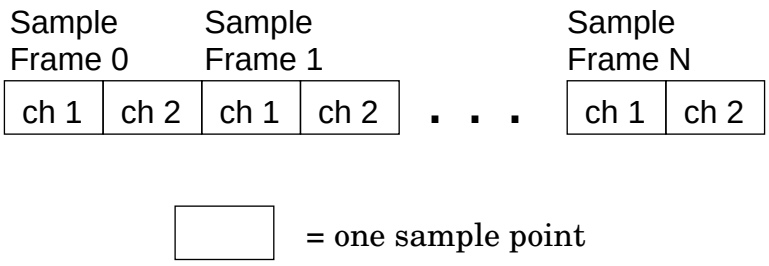
The Common Chunk describes fundamental parameters of the sampled sound.

```
#define CommonID        'COMM'      /* ckID for Common Chunk    */

typedef struct {

    ID                  ckID;
    long                ckSize;

    short               numChannels;
    unsigned long       numSampleFrames;
    short               sampleSize;
    extended            sampleRate;

} CommonChunk;
```

ckID is always 'COMM'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**. For the Common Chunk, **ckSize** is always 18.

numChannels contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, and 4 means four channel sound, etc. Any number of audio channels may be represented.

The actual sound samples are stored in another chunk, the Sound Data Chunk, which will be described shortly. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame. This is illustrated below for the stereo case.



For monophonic sound, a sample frame is a single sample point.

For multichannel sounds, the following conventions should be observed:

	1	2	3	4	5	6
stereo	left	right				
3 channel	left	right	center			
quad	front left	front right	rear left	rear right		
4 channel	left	center	right	surround		
6 channel	left	left center	center	right	right center	surround

numSampleFrames contains the number of sample frames in the ***Sound Data Chunk***. Note that **numSampleFrames** is the number of sample frames, not the number of bytes nor the number of sample points in the ***Sound Data Chunk***. The total number of sample points in the file is **numSampleFrames** times **numChannels**.

sampleSize is the number of bits in each sample point. It can be any number from 1 to 32. The format of a sample point will be described in the next section, the ***Sound Data Chunk***.

sampleRate is the sample rate at which the sound is to be played back, in ***sample frames*** per second.

One and only one Common Chunk is required in every FORM AIFF.

Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

```
#define SoundDataID     'SSND'      /* ckID for Sound Data Chunk */

typedef struct {

    ID                  ckID;
    long                ckSize;

    unsigned long       offset;
    unsigned long       blockSize;
    unsigned char       soundData[];

} SoundDataChunk;
```

ckID is always 'SSND'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize.**

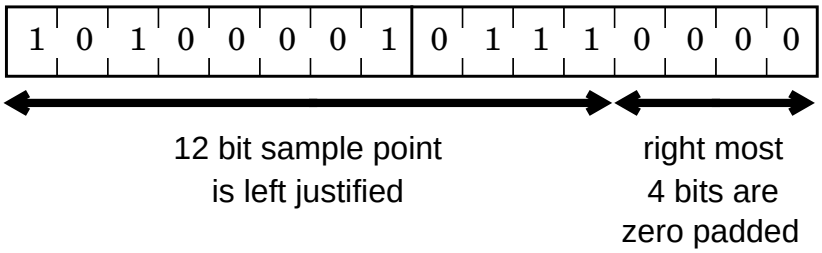
offset determines where the first sample frame in the **soundData** starts. **offset** is in bytes. Most applications won't use **offset** and should set it to zero. Use for a non-zero **offset** is explained in the ***Block-Aligning Sound Data*** section below.

blockSize is used in conjunction with offset for block-aligning sound data. It contains the size in bytes of the blocks that sound data is aligned to. As with **offset**, most applications won't use **blockSize** and should set it to zero. More information on **blockSize** is in the ***Block-aligning Sound Data*** section below.

soundData contains the sample frames that make up the sound. The number of sample frames in the **soundData** is determined by the **numSampleFrames** parameter in the ***Common Chunk***.

Sample Points

Each sample point in a sample frame is a linear, 2's complement value. The sample points are from 1 to 32 bits wide, as determined by the **sampleSize** parameter in the ***Common Chunk***. Sample points are stored in an integral number of contiguous bytes. One to 8 bit wide sample points are stored in one byte, 9 to 16 bit wide sample points are stored in two bytes, 17 to 24 bit wide sample points are stored in 3 bytes, and 25 to 32 bit wide samples are stored in 4 bytes. When the width of a sample point is less than a multiple of 8 bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated below. A 12 bit sample point binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

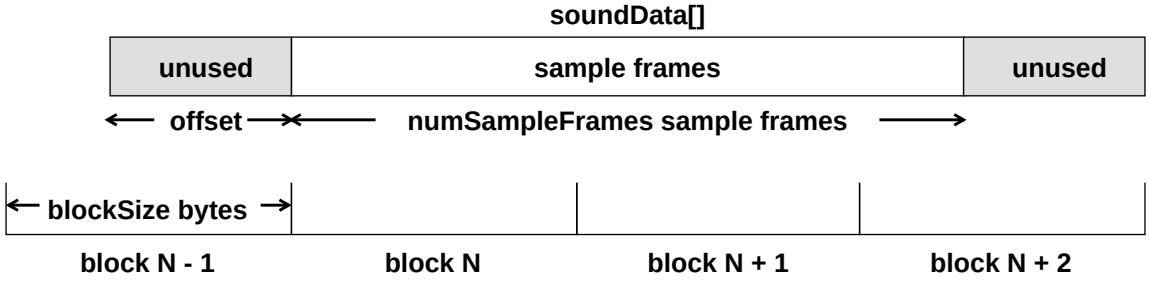


Sample Frames

Sample frames are stored contiguously in order of increasing time. The sample points within a sample frame are packed together, there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

Block-Aligning Sound Data

There may be some applications that, to insure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This can be accomplished with the **offset** and **blockSize** parameters, as shown below.



Block-aligned sound data

In the above figure, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first **offset** bytes of the **soundData**. Note too that the **soundData** array can extend beyond valid sample frames, allowing the **soundData** array to end on a block boundary.

blockSize specifies the size in bytes of the block that is to be aligned to. A **blockSize** of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set **blockSize** and offset to zero when writing Audio IFF files. Applications that write block-aligned sound data should set **blockSize** to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application doesn't preserve alignment, it should set **blockSize** and **offset** to zero. If an application needs to realign sound data to a different sized block, it should update **blockSize** and **offset** accordingly.

The Sound Data Chunk is required unless the **numSampleFames** field in the ***Common Chunk*** is zero. A maximum of one Sound Data Chunk can appear in a FORM AIFF.

Marker Chunk

The Marker Chunk contains markers that point to positions in the sound data. Markers can be used for whatever purposes an application desires. The ***Instrument Chunk***, defined later in this document, uses markers to mark loop beginning and end points, for example.

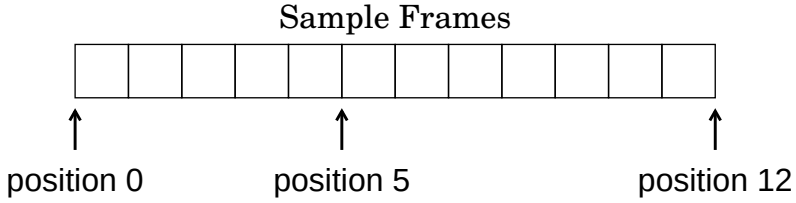
Markers

A marker has the following format.

```
typedef short           MarkerId;

typedef struct  {

    MarkerId            id;
    unsigned long       position;
    pstring             markerName;

} Marker;
```

id is a number that uniquely identifies the marker within a FORM AIFF. The **id** can be any positive non-zero integer, as long as no other marker within the same FORM AIFF has the same **id**.

The marker's position in the sound data is determined by **position**. Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at **position** zero, while a marker that falls between the first and second sample frame in the sound data is a **position** 1. Note that the units for **position** are sample frames, not bytes nor sample points.



markerName is a Pascal-style text string containing the name of the mark.

Note: Some "EAIFF85" files store strings as C-strings (text bytes followed by a null terminating character) instead of Pascal-style strings. Audio IFF uses **pstrings** because they are more efficiently skipped over when scanning through chunks. Using **pstrings**, a program can skip over a string by adding the string count to the address of the first character. C strings require that each character in the string be examined for the null terminator.

Marker Chunk Format

The format for the data within a Marker Chunk is shown below.

```
#define MarkerID        'MARK'      /* ckID for Marker Chunk */

typedef struct {

    ID                  ckID;
    long                ckSize;

    unsigned short      numMarkers;
    Marker              Markers[];

} MarkerChunk;
```

ckID is always 'MARK'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**.

numMarkers is the number of markers in the Marker Chunk.

numMarkers, if non-zero, it is followed by the markers themselves. Because all fields in a marker are an even number of bytes in length, the length of any marker will always be even. Thus, markers are packed together with no unused bytes between them. The markers need not be ordered in any particular manner.

The Marker Chunk is optional. No more than one Marker Chunk can appear in a FORM AIFF.

Instrument Chunk

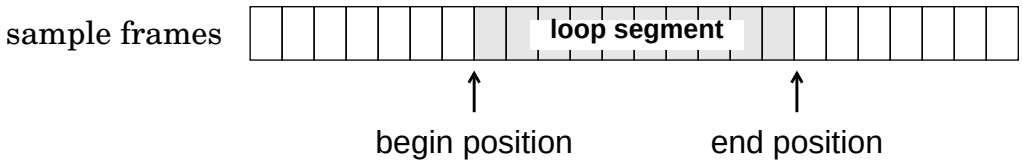
The Instrument Chunk defines basic parameters that an instrument, such as a sampler, could use to play back the sound data.

Looping

Sound data can be looped, allowing a portion of the sound to be repeated in order to lengthen the sound. The structure below describes a loop:

```
typedef struct {

    short           playMode;
    MarkerId        beginLoop;
    MarkerId        endLoop;

} Loop;
```

A loop is marked with two points, a begin position and an end position. There are two ways to play a loop, forward looping and forward/backward looping. In the case of forward looping, playback begins at the beginning of the sound, continues past the begin position and continues to the end position, at which point playback restarts again at the begin position. The segment between the begin and end position, called the ***loop segment***, is played over and over again, until interrupted by something, such as the release of a key on a sampling instrument, for example.



With forward/backward looping, the loop segment is first played from the begin position to the end position, and then played backwards from the end position back to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

playMode specifies which type of looping is to be performed.

```
#define  NoLooping            0
#define  ForwardLooping       1
#define  ForwardBackwardLooping  2
```

If **NoLooping** is specified, then the loop points are ignored during playback.

beginLoop is a marker id that marks the begin position of the loop segment.

endLoop marks the end position of a loop. The begin position must be less than the end position. If this is not the case, then the loop segment has zero or negative length and no looping takes place.

Instrument Chunk Format

The format of the data within an Instrument Chunk is described below.

```
#define InstrumentID        'INST'      /* ckID for Instrument Chunk */

typedef struct {

    ID                      ckID;
    long                    ckSize;

    char                    baseNote;
    char                    detune;
    char                    lowNote;
    char                    highNote;
    char                    lowVelocity;
    char                    highVelocity;
    short                   gain;
    Loop                    sustainLoop;
    Loop                    releaseLoop;

} Instrumentchunk;
```

ckID is always 'INST'. **ckSize** is the size of the data portion of the chunk, in bytes. For the InstrumentChunk, **ckSize** is always 20.

baseNote is the note at which the instrument plays back the sound data without pitch modification. Units are MIDI (MIDI is an acronym for Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.

detune determines how much the instrument should alter the pitch of the sound when it is played back. Units are in ***cents*** (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised.

lowNote and **highNote** specify the suggested range on a keyboard for a playback of the sound data. The sound data should be played if the instrument is requested to play a note between the low and high notes, inclusive. The base note does not have to be within this range. Units for **lowNote** and **highNote** are MIDI note values.

lowVelocity and **highVelocity** specify the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is between low and high velocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

gain is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0 db means no change, 6 db means double the value of each sample point, while -6db means halve the value of each sample point.

sustainLoop specifies a loop that is to be played when an instrument is sustaining a sound.

releaseLoop specifies a loop that is to be played when an instrument is in the release phase of playing back a sound. The release phase usually occurs after a key on an instrument is released.

The InstrumentChunk is optional. No more than one InstrumentChunk can appear in a FORM AIFF.

MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data (please refer to ***Musical Instrument Digital Interface Specification 1.0***, available from the International MIDI Association, for more details on MIDI).

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in this block as well. As more instruments come on the market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the ***Instrument Chunk***. For example, a new sampling instrument may have more than the two loops defined in the ***Instrument Chunk***. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

```
#define MIDDIDataID       'MIDI'      /* ckID for MIDI Data Chunk */

typedef struct {

   ID                     ckID;
   long                   ckSize;

   unsigned char          MIDIdata[];

} MIDIDataChunk;
```

ckID is always 'MIDI'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**.

MIDIData contains a stream of MIDI data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a FORM AIFF. If MIDI System Exclusive messages for several instruments are to be stored in a FORM AIFF, it is better to use one MIDI Data Chunk per instrument than one big MIDI Data Chunk for all of the instruments.

Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

```
#define AudioRecordingID  'AESD'          /* ckID for Audio Recording Chunk */

typedef struct  {

    ID                  ckID;
    long                ckSize;
    unsigned char       AESChannelStatusData [24];

}  AudioRecordingChunk;
```

ckID is always 'AESD'. **ckSize** is the size of the data portion of the chunk, in bytes. For the Audio Recording Chunk, **ckSize** is always 24.

The 24 bytes of **AESChannelStatusData** are specified in the ***AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data***, section 7.1, Channel Status Data. That document describes a format for real-time digital transmission of digital audio between audio devices. This information is duplicated in the Audio Recording Chunk for convenience. Of general interest would be bits 2,3, and 4 of byte0, which describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a FORM AIFF.

Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by manufacturers of applications. For example, an application that edits sound might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, and the like.

```
#define ApplicationSpecificID 'APPL'        /* ckID for Application */
                                              /* Specific Chunk      */

typedef struct {

    ID                  ckID;
    long                ckSize;

    OSType              applicationSignature;
    char                data[];

} ApplicationSpecificChunk;
```

ckID is always 'APPL'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**.

applicationSignature identifies a particular application. For Macintosh applications, this will be the application's four character signature.

data is the data specific to the application.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EAIFF85" has an ***Annotation Chunk*** that can be used for comments, but the Comments Chunk has two features not found in the "EAIFF85" chunk. They are: 1) a timestamp for the comment; and 2) a link to a marker.

Comment

A comment consists of a time stamp, marker id, and a text count followed by text.

```
typedef struct {

    unsigned long       timeStamp;
    MarkerID            marker;
    unsigned short      count;
    char                text;

} Comment;
```

timeStamp indicates when the comment was created. Units are the number of seconds since January 1, 1904. (This time convention is the one used by the Macintosh. For procedures that manipulate the time stamp, see The Operating System Utilities chapter in ***Inside Macintosh, vol II***).

A comment can be linked to a marker. This allows applications to store long descriptions of markers as a comment. If the comment is referring to a marker, then **marker** is the ID of that marker. Otherwise, **marker** is zero, indicating that this comment is not linked to a marker.

count is the length of the text that makes up the comment. This is a 16 bit quantity, allowing much longer comments than would be available with a **pstring**.

text contains the comment itself. This text must be padded with a byte at the end to insure that it is an even number of bytes in length. This pad byte, if present is not included in **count**.

Comments Chunk Format

```
#define CommentID       'COMT'    /* ckID for Comments Chunk.  */

typedef struct  {

    ID                  ckID;
    long                ckSize;

    unsigned short      numComments;
    Comment             comments;

} CommentsChunk;
```

ckID is always 'COMT'. **ckSize** is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by **ckID** and **ckSize**.

numComments contains the number of comments in the Comments Chunk. This is followed by the comments themselves. Comments are always an even number of bytes in length, so there is no padding between comments in the Comments Chunk.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

Text Chunks - Name, Author, Copyright, Annotation

These four chunks are included in the definition of every "EAAIFF85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

```
#define nameID          'NAME'    /* ckID for Name Chunk.      */
#define AuthorID        'AUTH'    /* ckID for Author Chunk.    */
#define CopyrightID     '(c) '    /* ckID for Copyright Chunk. */
#define AnnotationID    'ANNO'    /* ckID for Annotation Chunk. */

typedef  struct {

   ID                   ckID;
   long                 ckSize;

   char                 text [];

} TextChunk;
```

ckID is either '**NAME**', '**AUTH**', '**(c)** ', or '**ANNO**', depending on whether the chunk as a Name Chunk, Author Chunk, Copyright Chunk, or Annotation Chunk, respectively. For the Copyright Chunk, the 'c' is lowercase and there is a space (0x20) after the close parenthesis.

ckSize is the size of the data portion of the chunk, in this case the text.

text contains pure ASCII characters. It is not a **pstring** nor a C string. The number of characters in text is determined by **ckSize**. The contents of **text** depend on the chunk, as described below:

Name Chunk

text contains the name of the sampled sound. The Name Chunk is optional. No more than one Name chunk may exist within a FORM AIFF.

Author Chunk

text contains one or more author names. An author in this case is the creator of a sampled sound. The Author Chunk is optional. No more than one Author Chunk may exist within a FORM AIFF.

Copyright Chunk

The Copyright Chunk contains a copyright notice for the sound. Text contains a date followed by the copyright owner. The chunk ID '**(c)** ' serves as the copyright characters '©'. For example, a Copyright Chunk containing the text "1988 Apple Computer, Inc." means "© 1988 Apple Computer, Inc."

The Copyright Chunk is optional. No more than one Copyright Chunk may exist within a

FORM AIFF.

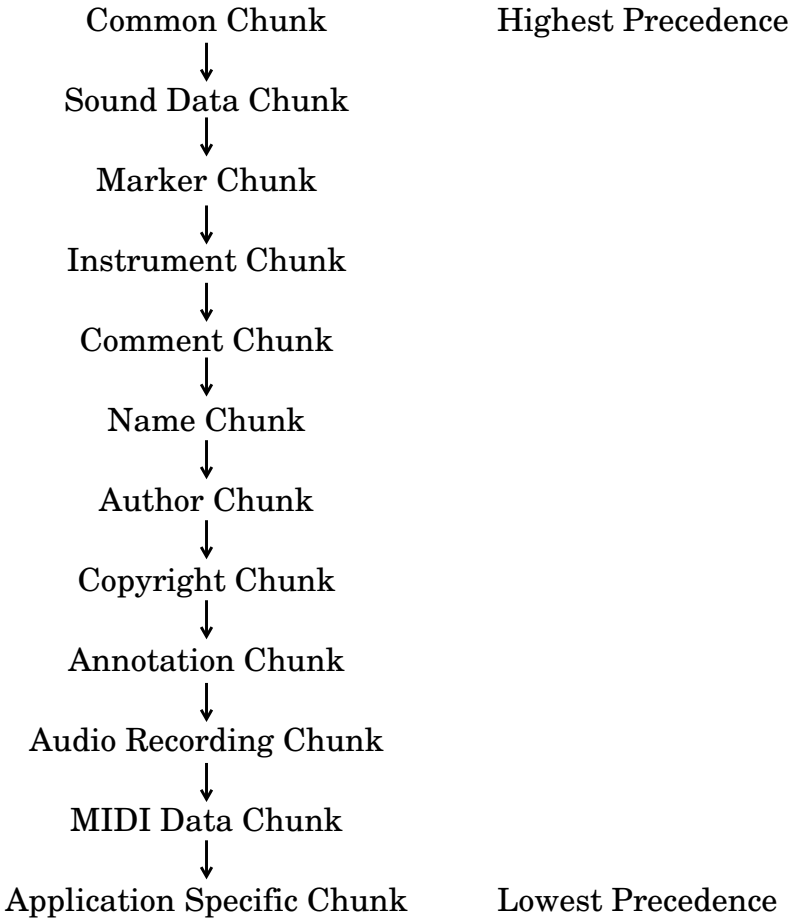
Annotation Chunk

text contains a comment. Use of this chunk is discouraged within FORM AIFF. The more powerful ***Comments Chunk*** should be used instead. The Annotation Chunk is optional. Many Annotation Chunks may exist within a FORM AIFF.

Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the ***Instrument Chunk*** defines loop points and MIDI system exclusive data in the ***MIDI Data Chunk*** may also define loop points. What happens if these loop points are different? How is an application supposed to loop the sound?

Such conflicts are resolved by defining a precedence for chunks:

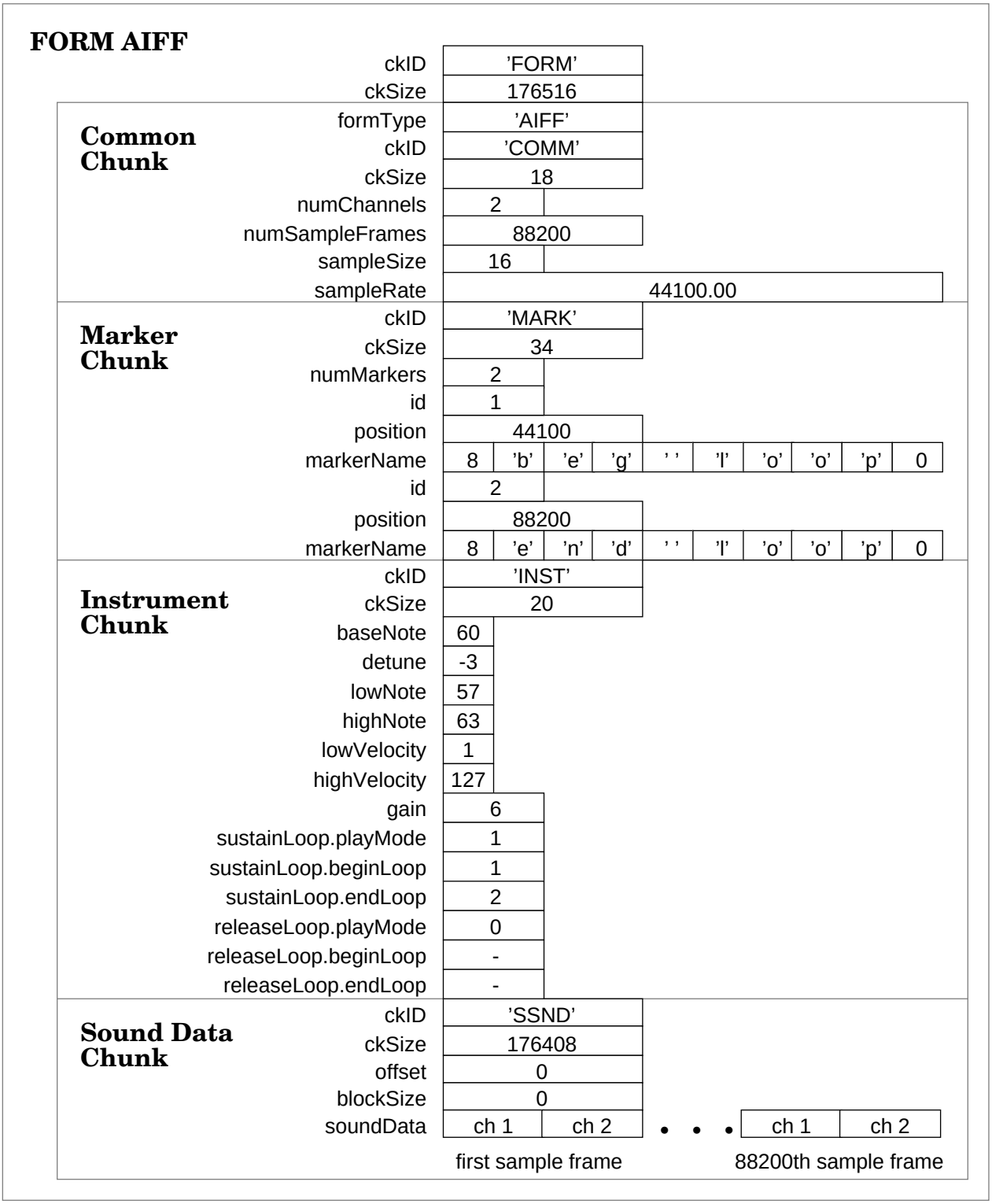


The ***Common Chunk*** has the highest precedence, while the ***Application Specific Chunk*** has the lowest. Information in the ***Common Chunk*** always takes precedence over conflicting information in any other chunk. The ***Application Specific Chunk*** always loses in conflicts with other chunks. By looking at the chunk hierarchy, for example, one sees that the loop points in the ***Instrument Chunk*** take precedence over conflicting loop points found in the ***MIDI Data Chunk***.

It is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly.

Appendix A - An Example

Illustrated below is an example of a FORM AIFF. An Audio IFF file is simply a file containing a single FORM AIFF. On a Macintosh, the FORM AIFF is stored in the data fork of a file and the file type is 'AIFF'.



Appendix B - Sending comments to Apple Computer, Inc.

If you have suggestions for new chunks to be added to the Audio Interchange File Format, please describe the chunk in as much detail as possible, and give an example of its use. Suggestions for new FORMs, ways to group FORM AIFF's into a bank, and new local chunks are welcome. When sending in suggestions, be sure to mention that your comment refers to the ***Audio Interchange File Format: "AIFF"*** document.

Send comments to :

Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, MS: 27-T
Cupertino, CA 95014 USA

References

AES Recommended Practice for Digital audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data, Audio Engineering Society, 60 East 42nd Street, New York, New York 10165

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

"EA IFF 85" Standard for Interchange Format Files, Electronic Arts.

"8SVX" IFF 8-Bit Sampled Voice. Electronic Arts.

Inside Macintosh, Volume II. Apple Computer, Inc., Addison Wesley Publishing Company, Inc., 1986.

Apple ® Numerics Manual, Addison Wesley Publishing Company, Inc., 1986

end of chapter 5

Revision Log

THE FOLLOWING CHANGES WERE MADE BETWEEN THE 0.90 VERSION AND THE DRAFT 0.99 VERSION OF THE SPECIFICATION:

- Added Table of Contents, Preface, Introduction, explanation of CD-I IFF file structure.
- Added Appendices [***A Quick Introduction to IFF***, ***EA IFF 85 - Standard for Interchange Format Files***, ***Audio Interchange File Format: "AIFF"***, ***Revision Log***].
- Changed **int** type in Data Definition section to long in order to eliminate ambiguities among different compilers.
- Modified bit packing scheme to have bits within a pixel right justified instead of left justified. Don't care bits are now where they belong for CD-I in the msbs, not lsbs.
- Changed definition of IFF_COLOR (for CLUT entries) from 16 bits per color [48 bits per pixel] to 8 bits per color [24 bits per pixel]. The specification can be extended later if there is a need for 16 bit colors.
- Made cosmetic and stylistic changes to C code fragments.
- Changed names of all structs to simplify and standardize naming conventions.
- Corrected language in referring to IFF components - "sub-chunk" was an undefined term and FORMs were omitted.
- Added mandatory FORM in each file for specifying the version number of the CD-I IFF specification used to generate that file.
- Added field in IHDR for color mask in addition to transparency. [This moves to IPAR -

see next item.]

- Moved many field from IHDR chunk to new optional IPAR chunk to simplify definition of required parameters for an image. This will allow the use of optional parameters to evolve without affecting the basic definition of a CD-I image.

- Added field in IHDR to indicate whether a DYUV image has one start value (in the header) for all scan lines or whether each scan line has its own start value (in the YUVS chunk). [Formerly, this distinction was made based upon a specific value of the field in the header.]

- Added capability to represent RGB555 directly in Green Book format. This was done since the RGB555 format for CD-I is structurally different from the generic RGBnnn format defined in this specification.

- Corrected definition of the field **ih_rl_len** in IHDR.

- Removed references to palette regions since this is not implemented in the current version.

- Corrected C language description of variable length arrays for disk storage.

- Added warning about pad bytes generated by compiler treatment of data structures. This specification, and library implementations should not depend upon specific compiler practices for alignment of allocated storage. The reader and writer software is responsible for guaranteeing that no pad bytes are inserted or expected on files.

- Added not to USER FORM stating that it should be used only for text comments, not storing processing parameters.

- Eliminated requirement that the first pixel of each scan line in CLUT data be longword aligned. Also removed mention of this alignment restriction in RGBnnn data.

- Filled in definition of DYUV and RL data formats.

- Removed reference to emphasis in audio section.

- Removed reference to IFF_MDL_QHY since it is not supported in this version.

- Adopted version 1.2 of AIFF specification from Apple Computer (instead of prior 1.1 version).

THE FOLLOWING CHANGES WERE MADE BETWEEN THE DRAFT 0.99 VERSION AND THE RELEASED 0.99 VERSION OF THE SPECIFICATION:

- Added picture of CD-I IFF file format to "IFF for CD-I" section

- Removed general capability for RGBnnn (along with the associated complex data packing scheme) and restricted it to just RGB888. While a contraction of capabilities for exchange of data, this will allow much simpler implementations of reading and writing code, and eliminate the need to support formats which are not generally used in the CD-I production process.
- Removed the VERS FORM, as it was semantically incorrect. Future revisions to FORM and chunk definitions will be handled in the IFF standard manner - making new chunk ids.
- Removed **ihdr_compress** field from IHDR chunk, since it was not supported and of questionable utility.
- Tried to improve presentation of pixel ordering in IDAT description.
- Improved description of DYUV, CLUT, & RL data formats.
- Modified structure of RGB555 data to separate upper and lower bytes of pixel data into 2 portions.
- Removed USER FORM, as it was semantically incorrect.
- Added USER chunk for use within IMAG FORM for comments.
- Substantially revised name, order, and semantics of many fields in the IHDR chunk to support the padding of scan line data. This is necessary to provide compatibility with UCM functions which require longword alignment of the first pixel in a scan line. All of this forced a fairly major revision in the definition and usage of the header fields.
- Renamed and renumbered the values for the **ihdr_model** field to allow precise specification of the image coding in effect, rather than just the family of image coding. E.g., added CLUT8, CLUT7, etc. ad values rather than just CLUTn.
- Added explanations of padding rules to each of the IDAT model descriptions.

end of chapter 6

NOTES