Libraries

COLLABORATORS

	TITLE : Libraries	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 33 IFFParse Library

The iffparse.library was created to help simplify the job of parsing IFF files. Unlike other IFF libraries, iffparse.library is not form-specific. This means that the job of interpreting the structure and contents of the IFF file is up to you. Previous IFF file parsers were either simple but not general, or general but not simple. IFFParse tries to be both simple and general.

1.2 33 IFFParse Library / The Structure of IFF Files

Many people have a misconception that IFF means image files. This is not the case. IFF (short for Interchange File Format) is a method of portably storing structured information in machine-readable form. The actual information can be anything, but the manner in which it is stored is very specifically detailed. This specification is the IFF standard.

The IFF standard was originally designed in 1985 by Electronic Arts in conjunction with a committee of developers. The full standard along with file descriptions and sample code is published in the Amiga ROM Kernel Reference Manual: Devices (3rd edition).

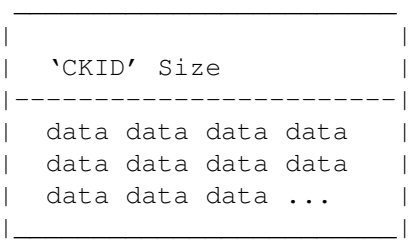
The goal of the IFF standard is to allow customers to move their own data between independently developed software products. The types of data objects to exchange are open-ended and currently include plain and formatted text, raster and structured graphics, fonts, music, sound effects, musical instrument descriptions, and animation. IFF addresses these needs by defining a standard for self-identifying file structures and rules for accessing these files.

Chunks: The Building Blocks of IFF Composite Data Types

1.3 33 / The Structure of IFF Files / Chunks: The Building Blocks of IFF

IFF files contain various types and amounts of data grouped in data chunks, each starting with a four-letter ASCII identifier (the chunk ID) followed by a 32-bit length count (the chunk size). The identifier and length count make it possible for IFF readers to skip over chunks that they don't understand or don't care about, and extract only the information they need. It may be helpful to think of these chunks as the building blocks of an IFF file (Figure 33-1).

Figure 33-1: The Chunk - The Building Block of IFF



The 'CKID' (chunk ID) characters of a real chunk will be a four letter identifier such as 'BODY' or 'CMAP', specifying the type and format of data stored in the chunk. Most chunks contain a simple defined grouping of byte, word, and long variables similar to the contents of a C structure. Others such as sound sample and bitmap image body chunks, contain a stream of compressed data.

Chunk writing is fairly straightforward with the one caveat that all chunks must be word-aligned. Therefore an odd-length chunk must be followed by a pad byte of zero (which is not counted in the size of the chunk). When chunks are nested, the enclosing chunk must state the total size of its composite contents (including any pad bytes).

About Chunk Length.

Every IFF data chunk begins with a 4-character identifier field followed by a 32-bit size field (these 8 bytes are sometimes referred to as the chunk header). The size field is a count of the rest of the data bytes in the chunk, not including any pad byte. Hence, the total space occupied by a chunk is given by its size field (rounded to the nearest even number) + 8 bytes for the chunk header.

1.4 33 / The Structure of IFF Files / Composite Data Types

Standard IFF files generally contain a variable number of chunks within one of the standard IFF composite data types (which may be thought of as chunks which contain other chunks). The IFF specification defines three basic composite data types, 'FORM', 'CAT ', and 'LIST'. A special

composite data type, the ‘PROP’, is found only inside a LIST.

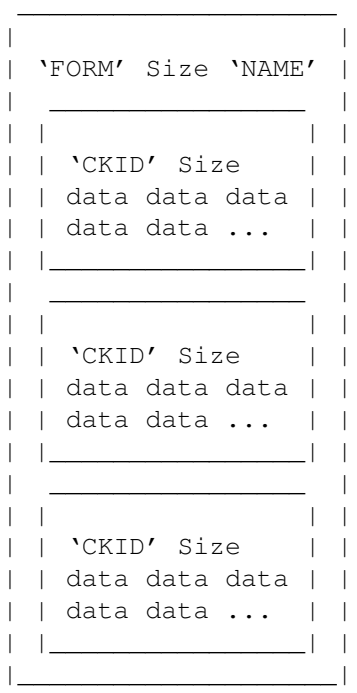
Table 33-1: Usage of Composite IFF Types

FORM	A grouping of chunks which describe one set of data
LIST	A grouping of the same type of FORMs with shared properties in a PROP
PROP	May appear in a LIST to define chunks shared by several FORMs
CAT	A concatenation of related FORMs or LISTs

The special IFF composite data types, like simple chunks, start with a 4-character identifier (such as ‘FORM’), followed by a 32-bit length. But their data begins with a second 4-character identifier that tells you what type of data is in the composite. For example, a FORM ILBM contains chunks describing a bitmap image, and a FORM FTXT contains chunks describing formatted text.

It may help to think of each composite data type as a box containing chunks, the IFF building blocks. Figure 33-2 shows a diagram of a typical composite.

Figure 33-2: The FORM - The Most Common IFF File Type



The example FORM in figure 33-2 is outlined below showing the size of chunks within the FORM. Notice that the size of a composite includes its second 4-character identifier (shown below as ‘NAME’).

```
.FORM 72 NAME  (72 is the size of the FORM starting with the "N" of NAME)
..CKID 22      (then 22 bytes of data, so chunk header + chunk size is 30)
```

```
 ..CKID 10      (then 10 bytes of data, so chunk header + chunk size is 18)
 ..CKID 12      (then 12 bytes of data, so chunk header + chunk size is 20)
```

Note

In the example above, indentation represents the nesting of the chunks within the FORM. Real FORMs and chunks would have different four-character identifiers rather than 'NAME' and 'CKID':

Any odd-length chunks must have a pad byte of zero at the end of the chunk. As shown below, this pad byte is not counted in the size of the chunk but is counted in the size of any composite types (such as FORM) that contain an odd-length chunk. Warning: some IFF readers and writers do not deal with this properly.

```
 .FORM 72 NAME  (72 is the size of the FORM starting with the "N" of NAME)
 ..CKID 21      (then 21 bytes of data, so chunk header + chunk size is 29)
 ..0            (pad byte of zero, size 1)
 ..CKID 10      (then 10 bytes of data, so chunk header + chunk size is 18)
 ..CKID 12      (then 12 bytes of data, so chunk header + chunk size is 20)
```

Most IFF files are of the composite type FORM. Generally, a FORM is a simple grouping of chunks that provide information about some data, and the data itself. Although some standard chunks have common usage within different FORMS, the identity and format of most chunks within a FORM are relative to the definition and specification of the FORM. For example, a CRNG chunk in a FORM ILBM may have a totally different format and contents than a chunk named CRNG in a different type of FORM.

One of the most popular IFF FORMs that has been defined is the ILBM standard. This is how most Amiga bitmap imagery is stored. Since this is the most common IFF file, it is used frequently as an example.

Here is the output of a program called Sift.c that displays a text description of the contents of an IFF file (Sift.c is listed at the end of this chapter). The file shown below is a fairly simple ILBM.

```
 .FORM 11068 ILBM
 ..BMHD 20
 ..CAMG 4
 ..CMAP 12
 ..BODY 10995
```

Computing the Size of a FORM.

The size of the FORM (11068 bytes) is equal to the sum of the sizes stated in each chunk contained within the FORM, plus 8 bytes for the overhead of each chunk header (4 bytes for the 4-character chunk ID, and 4 bytes for the 32-bit chunk size), plus 4 bytes for the FORM's own 4-character ID ('ILBM'), plus 1 byte more for each pad byte that follow any odd-length chunks.

1.5 33 IFFParse Library / Parsing an IFF File

Chunk reading requires a parser to scan each chunk and dispatch the proper access/conversion procedure. For a simple IFF file, such parsing may be relatively easy, consisting mainly reading in the data of desired chunks, and seeking over unwanted chunks (and the pad byte after odd-length chunks). Interpreting nested chunks is more complex, and requires a method for keeping track of the current context, i.e., the data which is still relevant at any particular depth into the nest. The original IFF specifications compare an IFF file to a C program. Such a metaphor can be useful in understanding the scope of of the chunks in an IFF file.

The IFFParse library addresses general IFF parsing requirements by providing a run-time library which can extract the chunks you want from an IFF file, with the ability to pass control to you when it reaches a chunk that requires special processing such as decompression. IFFParse also understands complex nested IFF formats and will keep track of the context for you.

Basic Functions and Structures of IFFParse Library

1.6 33 / / Basic Functions and Structures of IFFParse Library

The structures and flags of the IFFParse library are defined in the include files <libraries/iffparse.h> and <libraries/iffparse.i>. IFF files are manipulated through a structure called an IFFHandle. Only some of the fields in the IFFHandle are publicly documented. The rest are managed internally by IFFParse. This handle is passed to all IFFParse functions, and contains the current parse state and position in the file. An IFFHandle is obtained by calling AllocIFF(), and freed through FreeIFF(). This is the only legal way to obtain and dispose of an IFFHandle.

The public portion of if IFFHandle is defined as follows:

```
    /*
     * Structure associated with an active IFF stream.
     * "iff_Stream" is a value used by the client's read/write/seek
     * functions - it will not be accessed by the library itself and
     * can have any value (could even be a pointer or a BPTR).
     */
    struct IFFHandle {
            ULONG   iff_Stream;
            ULONG   iff_Flags;
            LONG    iff_Depth;      /*  Depth of context stack.  */
            /*  There are private fields hiding here.  */
    };
```

1.7 33 IFFParse Library / Stream Management

A stream is a linear array of bytes that may be accessed sequentially or randomly. DOS files are streams. IFFParse uses Release 2 Hook structures

(defined in <utility/hooks.h>) to implement a general stream management facility. This allows the IFFParse library to read, write, and seek any type of file handle or device by using an application-provided hook function to open, read, write, seek and close the stream.

Built on top of this facility, IFFParse has two internal stream managers: one for unbuffered DOS files (AmigaDOS filehandles), and one for the Clipboard.

Initialization Termination Custom Streams

1.8 33 / Stream Management / Initialization

As shown above, the IFFHandle structure contains the public field iff_Stream. This is a longword that must be initialized to contain something meaningful to the stream manager. IFFParse never looks at this field itself (at least not directly). Your iff_Stream may be an AmigaDOS filehandle, or an IFFParse ClipboardHandle, or a custom stream of any type if you provide your own custom stream handler (see the section on "Custom Stream Handlers" later in this chapter). You must initialize your IFFHandle structure's iff_Stream to your stream, and then initialize the IFFHandle's flags and stream hook.

Three sample initializations are shown here. In the case of the internal DOS stream manager, iff_Stream is an AmigaDOS filehandle as returned by Open():

```
iff->iff_Stream = (ULONG) Open ("filename", MODE_OLDFILE);
if(iff->iff_Stream) InitIFFasDOS (iff);
    /* use internal DOS stream manager */
```

In the case of the internal Clipboard stream manager, iff_Stream is a pointer to an IFFParse ClipboardHandle structure (the OpenClipboard() call used here is a function of iffparse.library, not the clipboard.device):

```
iff->iff_Stream = (ULONG) OpenClipboard (PRIMARY_CLIP);
InitIFFasClip (iff); /* use internal Clipboard stream manager  */
```

When using a custom handle such as an fopen() file handle, or a device other than the clipboard, you must provide your own flags and stream handler:

```
iff->iff_Stream = (ULONG) OpenMyCustomStream("foo");
InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
```

IFFParse "knows" the seek capabilities of DOS and ClipboardHandle streams, so InitIFFasDOS() and InitIFFasClip() set the flags for you.

```
You May Change the Seek Bits in iff_Flags:
------------------------------------------
IFFParse sets IFFF_FSEEK | IFFF_RSEEK for DOS files.  This is not
always true (e.g., pipes).  If you know that a DOS file has different
seek characteristics, your application may correct the seek bits in
iff_Flags after calling InitIFFasDOS().
```

When using InitIFF() to provide a custom handler, you must also provide flags to tell IFFParse the capabilities of your stream. The flags are:

```
    IFFF_FSEEK:  This stream has forward-seek capability only.
    IFFF_RSEEK:  This stream has random-seek capability.
                 IFFF_RSEEK tends to imply IFFF_FSEEK, but it's
                 best to specify both.
```

If neither flag is specified, you're telling IFFParse that it can't seek through the stream.

After your stream is initialized, call OpenIFF():

```
    error = OpenIFF (iff, IFFF_READ);
```

Once you establish a read/write mode (by passing IFFF_READ or IFFF_WRITE), you remain in that mode until you call CloseIFF().

1.9 33 / Stream Management / Termination

Termination is simple. Just call CloseIFF(iff). This may be called at any time, and terminates IFFParse's transaction with the stream. The stream itself is not closed. The IFFHandle may be reused; you may safely call OpenIFF() on it again. You are responsible for closing the streams you opened. A stream used in an IFFHandle must generally remain open until you CloseIFF().

1.10 33 / Stream Management / Custom Streams

A custom stream handler can allow you (and iffparse.library) to use your compiler's own file I/O functions such as fopen(), fread() and fwrite(), rather than the lower-level AmigaDOS equivalents Open(), Read(), and Write(). A custom stream handler could also be used to read or write IFF files from an Exec device or an unusual handler or filesystem.

If you install your own stream handler function, iffparse.library will call your function whenever it needs to read, write, or seek on your file. Your stream handler function will then perform these stream actions for iffparse.library. See the "Custom Stream Handlers" section for more information on custom stream handlers.

1.11 33 IFFParse Library / Parsing

This is both simple and complicated. It's simple in that it's just one call. It's complicated in that you have to seize control of the parser to get your data.

The parser operates automatically, scanning the file, verifying syntax and

layout rules. If left to its default behavior, it will scan through the entire file until it reaches the end, whereupon it will tell you that it got to the end.

The whole scanning procedure is controlled through one call:

```
error = ParseIFF (iff, controlmode);
```

The control modes are IFFPARSE_SCAN, IFFPARSE_STEP and IFFPARSE_RAWSTEP. For now, only the IFFPARSE_SCAN control mode is considered.

1.12 33 / Parsing / Controlling Parsing

ParseIFF(), if left to itself, wouldn't do anything useful. Ideally, it should stop at strategic places so we can look at the chunks. Here's where it can get complicated.

There are many functions provided to help control the parsing process; only the common ones are covered here. Additional functions are described in the Autodocs for iffparse.library (for the complete Autodocs, refer to the Amiga ROM Kernel Reference Manual: Includes and Autodocs also by Addison-Wesley).

1.13 33 / / Controlling Parsing / StopChunk()

You can instruct the parser to stop when it encounters a specific IFF chunk by using the function StopChunk():

```
error = StopChunk (iff, ID_ILBM, ID_BODY);
```

When the parser encounters the requested chunk, parsing will stop, and ParseIFF() will return the value zero. The stream will be positioned ready to read the first data byte in the chunk. You may then call ReadChunkBytes() or ReadChunkRecords() to pull the data out of the chunk.

You may call StopChunk() any number of times for any number of different chunk types. If you wish to identify the chunk on which you've stopped, you may call CurrentChunk() to get a pointer to the current ContextNode, and examine the cn_Type and cn_ID fields.

Using StopChunk() for every chunk, you can parse IFF files in a manner very similar to the way you're probably doing it now, using a state machine. However, this would be a terrible underuse of IFFParse.

1.14 33 / / Controlling Parsing / PropChunk()/FindProp()

In the case of a FORM ILBM, certain chunks are defined as being able to appear in any order. Among these are the BMHD, CMAP, and CAMG. Typically, BMHD appears first, followed by CMAP and CAMG, but you can't make this assumption. The IFF and ILBM standards require you to assume these chunks will appear in any order. So ideally, what you'd like to do is collect them as they arrive, but not do anything with them until you actually need them.

This is where PropChunk() comes in. The syntax for PropChunk() is identical to StopChunk():

```
    error = PropChunk (iff, ID_ILBM, ID_BMHD);
```

When you call ParseIFF(), the parser will look for chunks declared with PropChunk(). When it sees them, the parser will internally copy the contents of the chunk into memory for you before continuing its parsing.

When you're ready to examine the contents of the chunk, you use the function FindProp():

```
    StoredProperty = FindProp (iff, ID_ILBM, ID_BMHD);
```

FindProp() returns a pointer to a struct StoredProperty, which contains the chunk size and data. If the chunk was never encountered, NULL is returned. This permits you to process the property chunks in any order you wish, regardless of how they appeared in the file. This provides much better control of data interpretation and also reduces headaches. The following fragment shows how ILBM BitMapHeader data could be accessed after using ParseIFF() with PropChunk(iff, ID_ILBM, ID_BMHD):

```
    struct StoredProperty *sp;      /* defined in iffparse.h */
    struct BitMapHeader *bmhd;      /* defined in IFF spec   */

    if (sp = FindProp(iff, ID_ILBM, ID_BMHD))
            {
            /* If property is BMHD, sp->sp_Data is ptr to data in BMHD */
            bmhd = (struct BitMapHeader *)sp->sp_Data;
            printf("BMHD: PageWidth      = %ld\n",bmhd->PageWidth);
            }
```

1.15 33 / Parsing / Putting It Together

With just StopChunk(), PropChunk(), and ParseIFF(), you can write a viable ILBM display program. Since IFFParse knows all about IFF structure and scoping, such a display program would have the added ability to parse complex FORMs, LISTs, and CATs and attempt to find imagery buried within.

Such an ILBM reader might appear as follows:

```
iff = AllocIFF();
iff->iff_Stream = Open ("shuttle dog", MODE_OLDFILE);
```

```
InitIFFasDOS (iff);
OpenIFF (iff, IFFF_READ);

PropChunk (iff, ID_ILBM, ID_BMHD);
PropChunk (iff, ID_ILBM, ID_CMAP);
PropChunk (iff, ID_ILBM, ID_CAMG);
StopChunk (iff, ID_ILBM, ID_BODY);
ParseIFF (iff, IFFPARSE_SCAN);


if (bmhdprop = FindProp (iff, ID_ILBM, ID_BMHD))
    configurescreen (bmhdprop);
else
    bye ("No BMHD, no picture.");


if (cmapprop = FindProp (iff, ID_ILBM, ID_CMAP))
    setcolors (cmapprop);
else
    usedefaultcolors ();


if (camgprop = FindProp (iff, ID_ILBM, ID_CAMG))
    setdisplaymodes (camgprop);


decodebody (iff);
showpicture ();
CloseIFF (iff);
Close (iff->iff_Stream);
FreeIFF (iff);


    Open the Library.
    -----------------
    Application programs must always open iffparse.library before using
    the functions outlined above.

    Only Example Programs Skip Error Checking.
    ------------------------------------------
    Error checking is not used in the example above for the sake of
    clarity.  A real application should always check for errors.
```

1.16 33 / Parsing / Other Chunk Management Functions

```
Several other functions are available for controlling the parser.

 CollectionChunk() and FindCollection()
 StopOnExit()
 EntryHandler()
 ExitHandler()
```

1.17 33 / / Management Functions / CollectionChunk() and FindCollection()

PropChunk() keeps only one copy of the declared chunk (the one currently in scope). CollectionChunk() collects and keeps all instances of a specified chunk. This is useful for chunks such as the ILBM CRNG chunk, which can appear multiple times in a FORM, and which don't override previous instances of themselves. CollectionChunk() is called identically to PropChunk():

```
    error = CollectionChunk (iff, type, id);
```

When you wish to find the collected chunks currently in scope, you use the function FindCollection():

```
    ci = FindCollection (iff, type, id);
```

You will be returned a pointer to a CollectionItem, which is part of a singly-linked list of all copies of the specified chunk collected so far that are currently in scope.

```
    struct CollectionItem {
            struct CollectionItem   *ci_Next;
            LONG                    ci_Size;
            UBYTE                   *ci_Data;
    };
```

The size of this copy of the chunk is in the CollectionItem's ci_Size field. The ci_Data field points to the chunk data itself. The ci_Next field points to the next CollectionItem in the list. The last element in the list has ci_Next set to NULL.

The most recently-encountered instance of the chunk will be first in the list, followed by earlier chunks. Some might consider this ordering backwards.

If NULL is returned, the specified chunk was never encountered.

1.18 33 / / Other Chunk Management Functions / StopOnExit()

Whereas StopChunk() will stop the parser just as it enters the declared chunk, StopOnExit() will stop just before it leaves the chunk. This is useful for finding the end of FORMs, which would indicate that you've collected all possible data in this FORM and may now act on it.

```
    /* Ask ParseIFF() to stop with IFFERR_EOC when leaving a FORM ILBM */
    StopOnExit(iff,ID_ILBM, ID_FORM);
```

1.19 33 / / Other Chunk Management Functions / EntryHandler()

This is used to install your own custom chunk entry handler. See the "Custom Chunk Handlers" section below for more information. StopChunk(), PropChunk(), and CollectionChunk() are internally built on top of this.

1.20 33 / / Other Chunk Management Functions / ExitHandler()

This is used to install your own custom chunk exit handler. See the "Custom Chunk Handlers" section below for more information. StopOnExit() is internally built on top of this.

1.21 33 / Parsing / Reading Chunk Data

To read data from a chunk, use the functions ReadChunkBytes() and ReadChunkRecords(). Both calls truncate attempts to read past the end of a chunk. For odd-length chunks, the parser will skip over the pad bytes for you. Remember that for chunks which have been gathered using PropChunk() (or CollectionChunk()), you may directly reference the data by using FindProp() (or FindCollection()) to get a pointer to the data. ReadChunkBytes() is commonly used when loading and decompressing bitmap and sound sample data or sequentially reading in data chunks such as FTXT CHRS text chunks. See the code listing ClipFTXT.c for an example usage of ReadChunkBytes().

1.22 33 / Parsing / Other Parsing Modes

In addition to the mode IFFPARSE_SCAN, there are the modes IFFPARSE_STEP and IFFPARSE_RAWSTEP.

IFFPARSE_RAWSTEP IFFPARSE_STEP

1.23 33 / / Other Parsing Modes / IFFPARSE_RAWSTEP

This mode causes the parser to progress through the stream step by step, rather than in the automated fashion provided by IFFPARSE_SCAN. In this mode, ParseIFF() will return upon every entry to and departure from a context.

When the parser enters a context, ParseIFF() will return zero. CurrentChunk() will report the type and ID of the chunk just entered, and the stream will be positioned to read the first byte in the chunk. When entering a FORM, LIST, CAT or PROP chunk, the longword containing the type (e.g., ILBM, FTXT, etc.) is read by the parser. In this case, the stream will be positioned to read the byte immediately following the type.)

When the parser leaves a context, ParseIFF() will return the value IFFERR_EOC. This is not strictly an error, but an indication that you are about to leave the current context. CurrentChunk() will report the type and ID of the chunk you are about to leave. The stream is not positioned predictably within the chunk.

The parser does not call any installed chunk handlers when using this mode (e.g., property chunks declared with PropChunk() will not be collected).

See the example program, Sift.c, for a demonstration of IFFPARSE_RAWSTEP.

1.24 33 / / Other Parsing Modes / IFFPARSE_STEP

This mode is identical to IFFPARSE_RAWSTEP, except that, before control returns to your code, the chunk handler (if any) for the chunk is invoked.

1.25 33 IFFParse Library / Writing IFF Files

IFFParse provides facilities for writing IFF files. Again, IFFParse makes no assumptions about the data you're writing, and concerns itself with verifying the syntax of your output.

1.26 33 / Writing IFF Files / Creating Chunks In a File

Because the IFF specification has nesting and scoping rules, you can nest chunks inside one another. One instance is the BMHD chunk, which is commonly nested inside a FORM chunk. Thus, it is necessary for you to inform IFFParse when you are starting and ending chunks.

1.27 33 / / Creating Chunks In a File / PushChunk()

To tell IFFParse you are about to begin writing a new chunk, you use the function PushChunk():

```
    error = PushChunk (iff, ID_ILBM, ID_BMHD,chunksize);
```

The chunk ID and size are written to the stream. IFFParse will enforce the chunk size you specified; attempts to write past the end of the chunk will be truncated. If, as a chunk size argument, you pass IFFSIZE_UNKNOWN, the chunk will be expanded in size as you write data to it.

1.28 33 / / Creating Chunks In a File / PopChunk()

When you are through writing data to a chunk, you complete the write by calling PopChunk():

```
    error = PopChunk (iff);
```

If you wrote fewer bytes than you declared with PushChunk(), PopChunk() will return an error. If you specified IFFSIZE_UNKNOWN, PopChunk() will seek backward on the stream and write the final size. If you specified a chunk size that was odd, PopChunk() will write the pad byte automatically.

PushChunk() and PopChunk() nest; every call to PushChunk() must have a corresponding call to PopChunk().

1.29 33 / Writing IFF Files / Writing Chunk Data

Writing the IFF chunk data is done with either the WriteChunkBytes() or
WriteChunkRecords() functions.

```
    error = WriteChunkBytes (iff, buf, size);
    error = WriteChunkRecords (iff, buf, recsize, numrec);
```

If you specified a valid chunk size when you called PushChunk(),
WriteChunkBytes() and WriteChunkRecords() will truncate attempts to write
past the end of the chunk.

Code to write an ILBM file might take the following form:

```
    iff = AllocIFF ();
    iff->iff_Stream = Open ("foo", MODE_NEWFILE);
    InitIFFasDOS (iff);
    OpenIFF (iff, IFFF_WRITE);

    PushChunk (iff, ID_ILBM, ID_FORM, IFFSIZE_UNKNOWN);

    PushChunk (iff, ID_ILBM, ID_BMHD, sizeof (struct BitMapHeader));
    WriteChunkBytes (iff, &bmhd, sizeof (bmhd));
    PopChunk (iff);

    PushChunk (iff, ID_ILBM, ID_CMAP, cmapsize);
    WriteChunkBytes (iff, cmapdata, cmapsize);
    PopChunk (iff);

    PushChunk (iff, ID_ILBM, ID_BODY, IFFSIZE_UNKNOWN);
    packwritebody (iff);
    PopChunk (iff);

    PopChunk (iff);

    CloseIFF (iff);
    Close (iff->iff_Stream);
    FreeIFF (iff);
```

Again, error checking is not present for clarity. See the example code
ClipFTXT.c which writes a simple FTXT clip to the clipboard.

1.30 33 / Writing IFF Files / A Note On Seekability

As you can see from the above examples, IFFParse works best with a stream
that can seek randomly. However, it is not possible to seek on some
streams (e.g., pipes).

IFFParse will read and write streams with limited or no seek capability.
In the case of reading, only forward-seek capability is desirable.
Failing this, IFFParse will fake forward seeks with a number of short
reads.

In the case of writing, if the stream lacks random seek capability,

```
IFFParse will buffer the entire contents of the file until you do the
final PopChunk(), or when you CloseIFF() the handle.  At that time, the
entire stream will be written in one go.  This buffering happens whether
or not you specify all the chunk sizes to PushChunk().

    About the Internal Buffering.
    -----------------------------
    The current implementation of this internal buffering could be
    more efficient.  Be aware that Commodore reserves the right to alter
    this behavior of the parser, to improve performance or reduce memory
    requirements.  We mention this behavior on the off chance it is
    important to you.
```

1.31 33 IFFParse Library / Context Functions

```
Internally, IFFParse maintains IFF nesting and scoping context via a
context stack.  The PushChunk() and PopChunk() functions get their names
from this basic idea of the iffparse.library.  Direct access to this stack
is not allowed.  However, many functions are provided to assist in
examining and manipulating the context stack.

    About the Context Stack.
    ------------------------
    It is probably easier to think of a stack of blocks on a table in
    front of you when reading this discussion.

As the nesting level increases (as would happen when parsing a nested LIST
or FORM), the depth of the context stack increases; new elements are added
to the top.  When these contexts expire, the ContextNodes are deleted and
the stack shrinks.

 Context Nodes
 The Default Context
 Context-Specific Data: LocalContextItems
 Storing LCIs
 Some Interesting Internal Details
```

1.32 33 / Context Functions / Context Nodes

```
The current context is said to be the top element on the stack.
Contextual information is stored in a structure called a ContextNode:

    struct ContextNode {
            struct MinNode  cn_Node;
            LONG            cn_ID;
            LONG            cn_Type;
            LONG            cn_Size; /*  Size of this chunk             */
            LONG            cn_Scan; /*  # of bytes read/written so far */
            /*  There are private fields hiding here.  */
            };

 CurrentChunk()    ParentChunk()
```

1.33 33 / / Context Nodes / CurrentChunk()

You can obtain a pointer to the current ContextNode through the function CurrentChunk():

```
    currentnode = CurrentChunk (iff);
```

The ContextNode tells you the type, ID, and size of the currently active chunk. If there is no currently active context, NULL is returned.

1.34 33 / / Context Nodes / ParentChunk()

To find the parent of a context, you call ParentChunk() on the relevant ContextNode:

```
    parentnode = ParentChunk(currentnode);
```

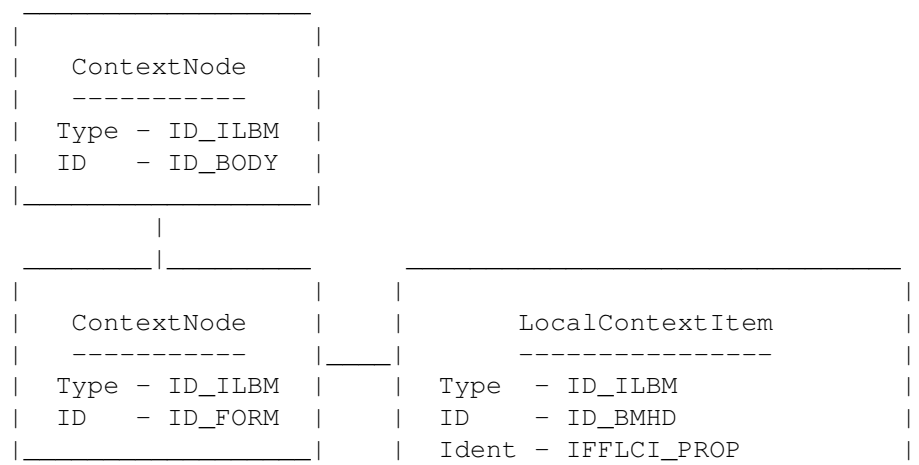
If there is no parent context, NULL is returned.

1.35 33 / Context Functions / The Default Context

When you first obtain an IFFHandle through AllocIFF(), a hidden default context node is created. You cannot get direct access to this node through CurrentChunk() or ParentChunk(). However, using StoreLocalItem(), you can store information in this context.

1.36 33 / Context Functions / Context-Specific Data: LocalContextItems

ContextNodes can contain application data specific to that context. These data objects are called LocalContextItems. LocalContextItems (LCIs) are a grey-box structure which contain a type, ID and identification field. LCIs are used to store context-sensitive data. The format of this data is application-defined. A ContextNode can contain as many LCIs as you want.



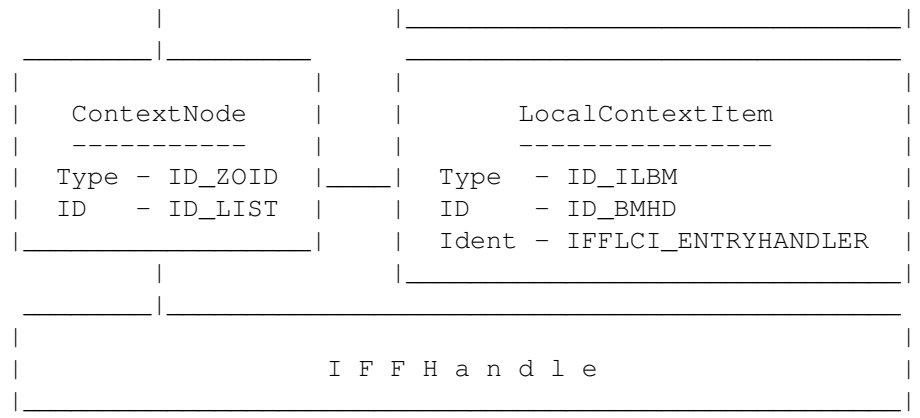


Figure 33-3: IFFParse Context Stack

Figure 33-3 shows the relationship between the IFFHandle, the ContextNodes, and the LCIs. The IFFHandle is the root of the tree, so to speak. ContextNodes "grow" out of the IFFHandle. In turn, LCIs "grow" out of the ContextNodes. What grows out of an LCI is client-defined.

AllocLocalItem() LocalItemData()

1.37 33 / / Context-Specific Data: LocalContextItems / AllocLocalItem()

To create an LCI, you use the function AllocLocalItem():

```
lci = AllocLocalItem (type, id, ident, datasize);
```

If successful, you will be returned a pointer to an LCI having the specified type, ID, and identification values; and with datasize bytes of buffer space for your application to use.

1.38 33 / / Context-Specific Data: LocalContextItems / LocalItemData()

To get a pointer to an LCIs data buffer, you use LocalItemData():

```
buf = LocalItemData (lci);
```

You may read and write the buffer to your heart's content; it is yours. You should not, however, write beyond the end of the buffer. The size of the buffer is what you asked for when you called AllocLocalItem().

1.39 33 / Context Functions / Storing LCIs

Once you've created and initialized an LCI, you'll want to attach it to a ContextNode. Though a ContextNode can have many LCIs, a given LCI can be linked to only one ContextNode. Once linked, an LCI cannot be removed

from a ContextNode (this may change in the future). Storing an LCI in a ContextNode is done with the functions StoreLocalItem() and StoreItemInContext().

StoreLocalItem() StoreItemInContext() FindLocalItem()

1.40 33 / / Storing LCIs / StoreLocalItem()

The StoreLocalItem() function is called as follows:

```
error = StoreLocalItem (iff, lci, position);
```

The position argument determines where the LCI is stored. The possible values are IFFSLI_ROOT, IFFSLI_TOP, and IFFSLI_PROP.

IFFSLI_ROOT causes StoreLocalItem() to store your LCI in the default ContextNode.

IFFSLI_TOP gets your LCI stored in the top (current) ContextNode.

```
The LCI Ends When the Current Context Ends.
-------------------------------------------
When the current context expires, your LCI will be deleted by
the parser.
```

IFFSLI_PROP causes your LCI to be stored in the topmost context from which a property would apply. This is usually the topmost FORM or LIST chunk. For example, suppose you had a deeply nested ILBM FORM, and you wanted to store the BMHD property in its correct context such that, when the current FORM context expired, the BMHD property would be deleted. IFFSLI_PROP will cause StoreLocalItem() to locate the proper context for such scoping, and store the LCI there. See the section on "Finding the Prop Context" for additional information on the scope of properties.

1.41 33 / / Storing LCIs / StoreItemInContext()

StoreItemInContext() is used when you already have a pointer to the ContextNode to which you want to attach your LCI. It is called like so:

```
StoreItemInContext (iff, lci, contextnode);
```

StoreItemInContext() links your LCI into the specified ContextNode. Then it searches the ContextNode to see if there is another LCI with the same type, ID, and identification values. If so, the old one is deleted.

1.42 33 / / Storing LCIs / FindLocalItem()

After you've stored your LCI in a ContextNode, you will no doubt want to be able to find it again later. You do this with the function FindLocalItem(), which is called as follows:

```
    lci = FindLocalItem (iff, type, id, ident);
```

FindLocalItem() attempts to locate an LCI having the specified type, ID, and identification values. The search proceeds as follows (refer to Figure 33-3 to understand this better).

FindLocalItem() starts at the top (current) ContextNode and searches all LCIs in that context. If no matching LCIs are found, it proceeds down the context stack to the next ContextNode and searches all its LCIs. The process repeats until it finds the desired LCI (whereupon it returns a pointer to it), or reaches the end without finding anything (where it returns NULL).

```
    Context Stack Position.
    -----------------------
    LCIs higher in the stack will "override" lower LCIs with the
    same type, ID, and identification field.  This is how property
    scoping is handled.  As ContextNodes are popped off the context
    stack, all its LCIs are deleted as well.  See the section on
    "Freeing LCIs" below for additional information on deleting LCIs.
```

1.43 33 / Context Functions / Some Interesting Internal Details

```
    WARNING:
    --------
    This section details some internal implementation details of
    iffparse.library which may help you to understand it better.  Use of
    the following information to do "clever" things in an application is
    forbidden and unsupportable.  Don't even think about it.
```

It turns out that StoredProperties, CollectionItems, and entry and exit handlers are all implemented using LCIs. For example, when you call FindProp(), you are actually calling a front-end to FindLocalItem(). The mysterious identification value (which has heretofore never been discussed) is a value which permits you to differentiate between LCIs having the same type and ID.

For instance, suppose you called PropChunk(), asking it to store an ILBM BMHD. PropChunk() will install an entry handler in the form of an LCI, having type equal to 'ILBM', ID equal to 'BMHD', and an identification value of IFFLCI_ENTRYHANDLER.

When an ILBM BMHD is encountered, the entry handler is called, and it creates and stores another LCI having type equal to 'ILBM', ID equal to 'BMHD' and an identification value of IFFLCI_PROP.

Thus, when you call FindProp(), it merely calls FindLocalItem() with your type and ID, and supplies IFFLCI_PROP for the identification value.

Therefore, handlers, StoredProperties, CollectionItems and your own custom LCIs can never be confused with each other, since they all have unique identification values. Since they are all handled (and searched for) in the same way, they all "override" each other in a consistent way. Just as StoredProperties higher in the context stack will be found and returned

before identical ones in lower contexts, so will chunk handlers be found and invoked before ones lower on the context stack (recall FindLocalItem()'s search procedure).

This means you can temporarily override a chunk handler by installing an identical handler in a higher context. The handler will persist until the context in which it is stored expires, after which, the original one regains scope.

1.44 33 IFFParse Library / Error Handling

If at any time during reading or writing you encounter an error, the IFFHandle is left in an undefined state. Upon detection of an error, you should perform an abort sequence and CloseIFF() the IFFHandle. Once CloseIFF() has been called, the IFFHandle is restored to normalcy and may be reused.

1.45 33 IFFParse Library / Advanced Topics

This section discusses customizing of IFFParse data handling. For applications with special needs, IFFParse supports both custom stream handlers and custom chunk handlers.

1.46 33 / Advanced Topics / Custom Stream Handlers

As discussed earlier, IFFParse contains built-in stream handlers for AmigaDOS file handles as returned by Open(), and for the clipboard.device. If you are using AmigaDOS filehandles or the clipboard.device, you need not supply a custom stream handler.

If you wish to use your compiler's own file I/O functions (such as fread()) or need to read or write to an unusual handler or Exec device, you must provide a custom stream handler for your IFFHandle. Your custom stream handler will be called to perform all reads, writes, and seeks on your custom stream. The allows you to use compiler file I/O functions, Exec device commands, or any other method to perform the requested stream operations.

If you are implementing your own custom stream handler, you will need to know the mechanics of hook call-backs, and how to interpret the parameters. An IFFParse custom stream handler is simply a function in your code that follows Release 2 hook function conventions (Hook functions are also known as callback functions. See the "Utility Library" chapter for more details).

1.47 33 / / Custom Stream Handlers / Installing a Custom Stream Handler

To initialize your IFFHandle to point to your custom stream and stream handler, you must open your stream and place a pointer to the stream in your IFFHandle's iff_Stream field. This pointer could be the return from fopen(), or an Exec device I/O request, or any other type of pointer which will provide your custom stream handler with a way to manage your stream. For example:

```
    iff->iff_Stream = (ULONG) fopen("foo");
```

Next, you must install your custom handler for this stream, and also tell IFFParse the seek capabilities of your stream. To install your custom stream handler, you must first set up a Hook structure (<utility/hooks.h>) to point to your stream handling function. Then use InitIFF() to install your stream handler and also tell IFFParse the seek capabilities of your stream. There is "some assembly required".

Release 2 hook function calling conventions specify a register interface. For an IFFParse custom stream hook, at entry, A0 contains a pointer to the hook, A2 is a pointer to your IFFHandle, and A1 is a pointer to a command packet, which tells you what to do. A6 contains a pointer to IFFParseBase. You may trash A0, A1, D0, and D1. All other registers must be preserved. You return your error code in D0. A return of 0 indicates success. A non-zero return indicates an error.

If your compiler supports registered function parameters, you may use a registerized C function entry as the h_Entry hook entry point:

```
/* mystreamhandler - SAS C custom stream handler with */
/* registerized arguments                             */
static LONG __saveds __asm
mystreamhandler (
    register __a0 struct Hook               *hook,
    register __a2 struct IFFHandle          *iff,
    register __a1 struct IFFStreamCmd       *actionpkt
    )
{
/*
 * Code to handle the stream commands - see end this section
 * for a complete example custom stream handler function.
 *
 */
}

/* Initialization of Hook structure for registerized  */
/* handler function                                   */
struct Hook mystreamhook {
    { NULL },
    (ULONG (*)()) mystreamhandler,
                  /* h_Entry, registerized function entry */
    NULL,
    NULL };

/* Open custom stream and InitIFF to custom stream handler */
```

```
if ( iff->iff_Stream = (ULONG) myopen("foo") )
    {
    InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
    }
```

Alternately, you could initialize your Hook structure's h_Entry to point to a standard assembler stub which would push the register arguments on the stack and then call a standard args C function. In this case, you must store the address of your C function in the h_SubEntry field of the Hook structure so the assembler stub can find and call your C entry point. A sample assembler hook entry stub follows. This would be assembled as hookentry.o and linked with your C code.

```
* HookEntry.asm for SAS C

        INCLUDE "exec/types.i"
        INCLUDE "utility/hooks.i"

        XDEF    _HookEntry

        section code

 _HookEntry:
        move.l  a6,-(sp)
        move.l  a1,-(sp)                ; push message packet pointer
        move.l  a2,-(sp)                ; push object pointer
        move.l  a0,-(sp)                ; push hook pointer
        move.l  h_SubEntry(a0),a0       ; fetch C entry point ...
        jsr     (a0)                    ; ... and call it
        lea     12(sp),sp               ; fix stack
        move.l  (sp)+,a6
        rts

        end
```

When using an assembler HookEntry stub, your C program's custom stream handler interface would be initialized as follows:

```
extern LONG HookEntry();        /* The assembler entry */

/* mystreamhandler - a standard args C function custom stream handler*/
static LONG mystreamhandler ( struct Hook *hook,
                              struct IFFHandle *iff,
                              struct IFFStreamCmd *actionpkt )
{
/* Code to handle the stream commands - see end of this section
 * for a complete example custom stream handler function.
 */
}

/* Initialization of Hook for asm HookEntry and std args C function */
struct Hook mystreamhook {
    { NULL },
```

```
    (ULONG (*)()) HookEntry,        /* h_Entry, assembler stub entry */
    (ULONG (*)()) mystreamhandler,
                            /* h_SubEntry, std args function entry */
    NULL };

/* Open custom stream and InitIFF to custom stream handler */
if ( iff->iff_Stream = (ULONG) myopen("foo") )
        {
        InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
        }
```

1.48 33 / / Custom Stream Handlers / Inside a Custom Stream Handler

When the library calls your stream handler, you'll be passed a pointer to the Hook structure (hook in the example used here), a pointer to the IFFHandle (iff), and a pointer to an IFFStreamCmd structure (actionpkt). Your stream pointer may be found in iff->iff_Stream where you previously stored it. The IFFStreamCmd (actionpkt) will describe the action that IFFParse needs you to perform on your stream:

```
/* Custom stream handler is passed struct IFFStreamCmd *actionpkt */
struct IFFStreamCmd {
    LONG    sc_Command;     /*  Operation to be performed (IFFCMD_) */
    APTR    sc_Buf;         /*  Pointer to data buffer              */
    LONG    sc_NBytes;      /*  Number of bytes to be affected      */
    };

/* Possible call-back command values.  (Using 0 as the value for
 * IFFCMD_INIT was, in retrospect, probably a bad idea.)
 */
#define IFFCMD_INIT     0   /*  Prepare the stream for a session    */
#define IFFCMD_CLEANUP  1   /*  Terminate stream session            */
#define IFFCMD_READ     2   /*  Read bytes from stream              */
#define IFFCMD_WRITE    3   /*  Write bytes to stream               */
#define IFFCMD_SEEK     4   /*  Seek on stream                      */
#define IFFCMD_ENTRY    5   /*  You just entered a new context      */
#define IFFCMD_EXIT     6   /*  You're about to leave a context     */
#define IFFCMD_PURGELCI 7   /*  Purge a LocalContextItem            */
```

Your custom stream handler should perform the requested action on your custom stream, and then return 0 for success or an IFFParse error if an error occurred. The following code demonstrates a sample stream handler for a stream which was opened with a compiler's fopen() buffered file I/O function:

```
static LONG mystreamhandler ( struct Hook *hook,
                              struct IFFHandle *iff,
                              struct IFFStreamCmd *actionpkt )
{
    register FILE   *stream;
    register LONG   nbytes, error;
    register UBYTE  *buf;
```

```
stream  = (FILE *) iff->iff_Stream;    /* get stream pointer */
nbytes  = actionpkt->sc_NBytes;        /* length for command */
buf     = (UBYTE *) actionpkt->sc_Buf; /* buffer for the command */

/* Now perform the action specified by the actionpkt->sc_Command */

switch (actionpkt->sc_Command) {
case IFFCMD_READ:
    /*
     * IFFCMD_READ means read sc_NBytes from the stream and place
     * it in the memory pointed to by sc_Buf.  Be aware that
     * sc_NBytes may be larger than can be contained in an int.
     * This is important if you plan on recompiling this for
     * 16-bit ints, since fread() takes int arguments.
     *
     * Any error code returned will be remapped by IFFParse into
     * IFFERR_READ.
     */
    error = (fread (buf, 1, nbytes, stream) != nbytes);
    break;

case IFFCMD_WRITE:
    /*
     * IFFCMD_WRITE is analogous to IFFCMD_READ.
     *
     * Any error code returned will be remapped by IFFParse into
     * IFFERR_WRITE.
     */
    error = (fwrite (buf, 1, nbytes, stream) != nbytes);
    break;

case IFFCMD_SEEK:
    /*
     * IFFCMD_SEEK asks that you performs a seek relative to the
     * current position.  sc_NBytes is a signed number,
     * indicating seek direction (positive for forward, negative
     * for backward).  sc_Buf has no meaning here.
     *
     * Any error code returned will be remapped by IFFParse into
     * IFFERR_SEEK.
     */
    error = (fseek (stream, nbytes, 1) == -1);
    break;

case IFFCMD_INIT:
    /*
     * IFFCMD_INIT means to prepare your stream for reading.
     * This is used for certain streams that can't be read
     * immediately upon opening, and need further preparation.
     * This operation is allowed to fail;  the error code placed
     * in D0 will be returned directly to the client.
     *
     * An example of such a stream is the clipboard.  The
     * clipboard.device requires you to set the io_ClipID and
     * io_Offset to zero before starting a read.  You would
```

```
         * perform such a sequence here.  (Stdio files need no such
         * preparation, so we simply return zero for success.)
         */
    case IFFCMD_CLEANUP:
        /*
         * IFFCMD_CLEANUP means to terminate the transaction with
         * the associated stream.  This is used for streams that
         * can't simply be closed.  This operation is not allowed to
         * fail;  any error returned will be ignored.
         *
         * An example of such a stream is (surprise!) the clipboard.
         * It requires you to explicitly end reads by CMD_READing
         * past the end of a clip, and end writes by sending a
         * CMD_UPDATE.  You would perform such a sequence here.
         * (Again, stdio needs no such sequence.)
         */
        error = 0;
        break;
    }
    return (error);
}
```

1.49 33 / Advanced Topics / Custom Chunk Handlers

Like custom stream handlers, custom chunk handlers are implemented using Release 2 Hook structures. See the previous section for details on how a handler function may be interfaced using a Hook structure.

There are two types of chunk handlers: entry handlers and exit handlers. Entry handlers are invoked just after the parser enters the chunk; the stream will be positioned to read the first byte in the chunk. (If the chunk is a FORM, LIST, CAT, or PROP, the longword type will be read by the parser; the stream will be positioned to read the byte immediately following the type.) Exit handlers are invoked just before the parser leaves the chunk; the stream is not positioned predictably within the chunk.

1.50 33 / / Custom Chunk Handlers / Installing a Custom Chunk Handler

To install an entry handler, you call EntryHandler() in the following manner:

```
    error = EntryHandler (iff, type, id, position, hookptr, object);
```

An exit handler is installed by saying:

```
    error = ExitHandler (iff, type, id, position, hookptr, object);
```

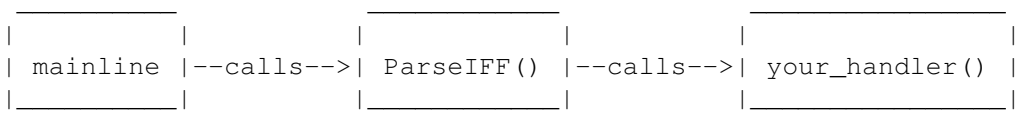
In both cases, a handler is installed for chunks having the specified type and id. The position argument specifies in what context to install the handler, and is identical to the position argument used by StoreLocalItem(). The hookptr argument given above is a pointer to your Hook structure.

1.51 33 / / Custom Chunk Handlers / Inside a Custom Chunk Handler

The mechanics of receiving parameters through the Hook are covered in the "Custom Stream Handlers" section, and are not duplicated here. Refer to the EntryHandler() and ExitHandler() Autodocs for the contents of the registers upon entry.

Once inside your handler, you can call nearly all functions in the iffparse.library, however, you should avoid calling ParseIFF() from within chunk handlers.

Your handler runs in the same environment as whoever called ParseIFF(). The propagation sequence is:



Thus, your handler runs on your mainline's stack, and can call any OS functions the mainline code can. (Your handler will have to set the global base pointer if your code uses base-relative addressing.)

When leaving the handler, you must follow the standard register preservation conventions (D0/D1/A0/A1 may be trashed, all others must be preserved). D0 contains your return code, which will affect the parser in a number of ways:

If you return zero (a normal, uneventful return), ParseIFF() will continue normally. If you return the value IFFERR_RETURN2CLIENT, ParseIFF() will stop and return the value zero to the mainline code.

If you return any other value, ParseIFF() will stop and return that value to the mainline code. This is how you should return error conditions to the client code.

1.52 33 / / Custom Chunk Handlers / The Object Parameter

The object parameter supplied to EntryHandler() and ExitHandler() is a pointer which will be passed to your handler when invoked. This pointer can be anything; the parser doesn't use it directly. As an example, you might pass the pointer to the IFFHandle. This way, when your handler is called, you'll be able to easily perform operations on the IFFHandle within your handler. Such code might appear as follows:

```
    error = EntryHandler (iff, ID_ILBM, ID_BMHD, IFFSLI_ROOT, hook, iff);
```

And your handler would be declared as follows:

```
    /* Registerized handler hook h_Entry using */
    /* SAS C registerized parameters           */
    LONG __asm MyHandler (register __a0 struct Hook      *hook,
                          register __a1 LONG             *cmd,
                          register __a2 struct IFFHandle *iff)

    /* Standard args handler hook h_SubEntry using */
    /* HookEntry.asm as Hook h_Entry               */
    LONG MyHandler ( struct Hook      *hook,
                     LONG             *cmd,
                     struct IFFHandle *iff )
```

From within your handler, you could then call CurrentChunk(), ReadChunkBytes(), or nearly any other operation on the IFFHandle. Please refer to the EntryHandler() and ExitHandler() Autodocs for additional information on the use of chunk handlers.

1.53 33 / Advanced Topics / Finding The PROP Context

Earlier it was mentioned that supplying a position value of IFFSLI_PROP to StoreLocalItem() would store it in the topmost property scope. FindPropContext() is the routine that finds that topmost context.

Property chunks (such as the BMHD, CMAP, and others) have dominion over the FORM that contains them; they are said to be "in scope" and their definition persists until the FORM's context ends. Thus, a property chunk has a scoping level equal to the FORM that contains it; when the FORM ends, the property dies with it.

Consider a more complicated example. Suppose you have a LIST with a PROP in it. PROPs are the global variables of LISTs; thus a property chunk declared in a PROP will persist until the LIST's context ends.

This is what FindPropContext() is looking for; a context level in which a property chunk may be installed.

FindPropContext() starts at the parent of the current context (second from the top of the context stack) and starts searching downward, looking for the first FORM or LIST context it finds. If it finds one, it returns a pointer to that ContextNode. If it can't find a suitable context level, it returns NULL.

FindPropContext() is called as follows:

```
    struct ContextNode *cn;

    cn = FindPropContext (iff);
```

1.54 33 / Advanced Topics / Freeing LCIs

Ordinarily, the parser will automatically delete LCIs you have allocated and installed. However, you may have a case where simply FreeMem()ing your LCI is not enough; you may need to free some ancillary memory, or decrement a counter, or send a signal, or something. This is where SetLocalItemPurge() comes in. It is called as follows:

```
SetLocalItemPurge (lci, hookptr);
```

When the parser is ready to delete your LCI, your purge handler code will be called through the Hook you supplied. You can then perform all your necessary operations. One of these operations should be to free the LCI itself. This is done with FreeLocalItem():

```
FreeLocalItem (lci);
```

This deallocates the memory used to store the LCI and the client buffer allocated with it. FreeLocalItem() is only called as part of a custom purge handler.

As with custom chunk handlers, your purge handler executes in the same environment as the mainline code that called ParseIFF(). It is recommended that you keep purge handlers short and to the point; super clever stuff should be reserved for custom chunk handlers, or for the client's mainline code. Custom purge handlers must always work; failures will be ignored.

1.55 33 IFFParse Library / IFF FORM Specifications

The specifications for Amiga IFF formats are maintained by Commodore Applications and Technical Support (CATS) and updated periodically. The latest specifications are published in the Amiga ROM Kernel Reference Manual: Devices (3rd edition) and also available in electronic form directly from CATS. Between updates of the IFF Manual, selected new FORMs and changes to existing FORMs are documented in Amiga Mail a technical newsletter for Amiga developers published by Commodore's CATS group.

Some of the most commonly used IFF FORMs are the four that were originally specified in the EA IFF-85 standard:

ILBM	Bitmap images and palettes
FTXT	Simple formatted text
SMUS	Simple musical scores
8SVX	8-bit sound samples

Of these four, ILBM is the most commonly encountered FORM, and FTXT is becoming increasingly important since the Release 2 conclip command passes clipped console text through the clipboard as FTXT. All data clipped to the clipboard must be in an IFF format.

This section will provide a brief summary of the ILBM and FTXT FORMs and
their most used common chunks. Please consult the EA-IFF specifications
for additional information.

 FORM ILBM   FORM FTXT

1.56 33 / IFF FORM Specifications / FORM ILBM

The IFF file format for graphic images on the Amiga is called FORM ILBM
(InterLeaved BitMap). It follows a standard parsable IFF format.

 ILBM.BMHD BitMapHeader Chunk
 Sample Hex Dump of an ILBM
 Interpreting ILBMs
 ILBM BODY Compression
 Interpreting the Scan Line Data
 Other ILBM Notes

1.57 33 / / FORM ILBM / ILBM.BMHD BitMapHeader Chunk

The most important chunk in a FORM ILBM is the BMHD (BitMapHeader) chunk
which describes the size and compression of the image:

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

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

/* The BitMapHeader structure expressed as a C structure */
typedef struct {
    UWORD w, h;                 /* raster width & height in pixels      */
    WORD  x, y;                 /* pixel position for this image        */
    UBYTE nPlanes;              /* # bitplanes (without mask, if any)   */
    Masking     masking;        /* One of the values above.  Usually 0  */
    Compression compression;    /* One of the values above.  Usually 1  */
    UBYTE reserved1;            /* reserved; ignore on read, write as 0 */
    UWORD transparentColor;     /* transparent color number. Usually 0  */
    UBYTE xAspect, yAspect;     /* pixel aspect, a ratio width : height */
    WORD  pageWidth, pageHeight;    /* source "page" size in pixels */
} BitMapHeader;
```

1.58 33 / / FORM ILBM / Sample Hex Dump of an ILBM

```
    WARNING:
    --------
    This hex dump is shown only to help the reader understand how IFF
    chunks appear in a file.  You cannot ever depend on any particular
    ILBM chunk being at any particular offset into the file.  IFF files
    are composed, in their simplest form, of chunks within a FORM.  Each
    chunk starts with a 4-letter chunkID, followed by a 32-bit length of
    the rest of the chunk.  You must parse IFF files, skipping past
    unneeded or unknown chunks by seeking their length (+1 if odd length)
    to the next 4-letter chunk ID.  In a real ILBM file, you are likely
    to encounter additional optional chunks.  See the IFF Specification
    listed in the Amiga ROM Kernel Reference Manual: Devices for
    additional information on such chunks.


0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD
0010: 00000014 01400190 00000000 06000100    .....@..........
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....
0030: 00000804 434D4150 00000030 001000E0    ....CMAP...0....
0040: E0E00000 20000050 30303050 50500030    .... ..P000PPP.0
0050: 90805040 70707010 60E02060 E06080D0    ..P@ppp.`. `.`..
0060: A0A0A0A0 90E0C0C0 C0D0A0E0 424F4459    ............BODY
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...  etc.

Interpretation:

      'F O R M' length  'I L B M"B M H D' <- start of BitMapHeader chunk
0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD

                                             Planes Mask
       length  WideHigh XorgYorg PlMkCoRe <- Compression Reserved
0010: 00000014 01400190 00000000 06000100    .....@..........

                                             start of C-AMiGa
      TranAspt PagwPagh 'C A M G' length  <- View modes chunk
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....

dir include:

      ViewMode 'C M A P' length   R g b R <- ViewMode 800=HAM | 4=LACE
0030: 00000804 434D4150 00000030 001000E0    ....CMAP...0....

       g b R g  b R g b  R g b R  g b R g <- Rgb's are for reg0 thru regN
0040: E0E00000 20000050 30303050 50500030    .... ..P000PPP.0

       b R g b  R g b R  g b R g  b R g b
0050: 90805040 70707010 60E02060 E06080D0    ..P@ppp.`. `.`..

       R g b R  g b R g  b R g b 'B O D Y'
0060: A0A0A0A0 90E0C0C0 C0D0A000 424F4459    ............BODY

                                             Compacted
       length   start of body data        <- (Compression=1 above)
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...
```

```
0080: FFBFF800 0F7FF7FC FF04F85A 77AD5DFE    ..........Zw.].  etc.

Simple CAMG ViewModes:  HIRES=0x8000  LACE=0x4  HAM=0x800  HALFBRITE=0x80

( Release 2 ILBMs may contain a LONGWORD ViewPort ModeID in CAMG )
```

1.59 33 / / FORM ILBM / Interpreting ILBMs

ILBM is a fairly simple IFF FORM. All you really need to deal with to extract the image are the following chunks:

BMHD
: Information about the size, depth, compaction method (see interpreted hex dump above).

CAMG
: Optional Amiga ViewModes chunk. Most HAM and HALFBRITE ILBMs should have this chunk. If no CAMG chunk is present, and image is 6 planes deep, assume HAM and you'll probably be right. Some Amiga ViewModes flags are HIRES=0x8000, LACE=0x4, HAM=0x800, HALFBRITE=0x80. 2.0 ILBM writers should write a full 32-bit mode ID in the CAMG. See the IFF Manual for more information on writing and interpreting 32-bit CAMG values.

CMAP
: RGB values for color registers 0 to N. Previously, 4-bit RGB components each left justified in a byte. These should now be stored as a full 8-bit RGB values by duplicating 4-bit values in the high and low nibble of the byte.

BODY
: The pixel data, stored in an interleaved fashion as follows (each line individually compacted if BMHD Compression = 1):

```
plane 0 scan line 0
plane 1 scan line 0
plane 2 scan line 0
...
plane n scan line 0
plane 0 scan line 1
plane 1 scan line 1
etc.
```

Optional Chunks
: Also watch for AUTH Author chunks and (c) copyright chunks and preserve any copyright information if you rewrite the ILBM.

1.60 33 / / FORM ILBM / ILBM BODY Compression

The BODY contains pixel data for the image. Width, Height, and depth (Planes) is specified in the BMHD.

If the BMHD Compression byte is 0, then the scan line data is not compressed. If Compression=1, then each scan line is individually compressed as follows:

```
    while (not produced the desired number of bytes)

            /* get a byte, call it N */

            if (N >= 0 && N <= 127)
                    /* copy the next N+1 bytes literally */

            if (N >= -127 && N <= -1)
                    /* repeat the next byte N+1 times */

            if (N == -128)
                    /* skip it, presumably it's padding */
```

1.61 33 / / FORM ILBM / Interpreting the Scan Line Data

If the ILBM is not HAM or HALFBRITE, then after parsing and uncompacting if necessary, you will have N planes of pixel data. Color register used for each pixel is specified by looking at each pixel thru the planes. For instance, if you have 5 planes, and the bit for a particular pixel is set in planes 0 and 3:

```
        PLANE     4 3 2 1 0
        PIXEL     0 1 0 0 1
```

then that pixel uses color register binary 01001 = 9.

The RGB value for each color register is stored in the CMAP chunk of the ILBM, starting with register 0, with each register's RGB value stored as one byte of R, one byte G, and one byte of B, with each component left justified in the byte. (ie. Amiga R, G, and B components are each stored in the high nibble of a byte)

But, if the picture is HAM or HALFBRITE, it is interpreted differently. Hopefully, if the picture is HAM or HALFBRITE, the package that saved it properly saved a CAMG chunk (look at a hex dump of your file with ASCII interpretation - you will see the chunks - they all start with a 4-ASCII-char chunk ID). If the picture is 6 planes deep and has no CAMG chunk, it is probably HAM. If you see a CAMG chunk, the 'CAMG' is followed by the 32-bit chunk length, and then the 32-bit Amiga view mode flags.

HAM pictures will have the 0x800 bit set in CAMG chunk ViewModes. HALBRITE pictures will have the 0x80 bit set. See the graphics library chapters or the Amiga Hardware Reference Manual for more information on HAM and HALFBRITE modes.

1.62 33 / / FORM ILBM / Other ILBM Notes

Amiga ILBMs images must be stored as an even number of bytes in width. However, the ILBM BMHD field w (width) should describe the actual image width, not the rounded up width as stored.

ILBMs created with Electronic Arts IBM or Amiga Deluxe Paint II packages are compatible (though you may have to use a '.lbm' filename extension on an IBM). The ILBM graphic files may be transferred between the machines (or between the Amiga and IBM sides your Amiga if you have a CBM Bridgeboard card installed) and loaded into either package.

1.63 33 / IFF FORM Specifications / FORM FTXT

The FTXT (Formatted TeXT) form is the standard format for sharing text on the Amiga. The Release 2 console device clip and paste functions, in conjunction with the startup-sequence conclip command, pass clipped console text through the clipboard as FTXT.

By supporting reading and writing of clipboard device FTXT, your application will allow users to cut and paste text between your application, other applications, and Amiga Shell windows.

The FTXT form is very simple. Generally, for clip and paste operations, you will only be concerned with the CHRS (character string) chunks of an FTXT. There may be one or more CHRS chunks, each of which contains a non-NULL-terminated string of ASCII characters whose length is equal to the length (StoredProperty.sp_Size) of the chunk.

Be aware that if you CollectionChunk() the CHRS chunks of an FTXT, the list accumulated by IFFParse will be in reverse order from the chunk order in the file. See the ClipFTXT.c example at the end of this chapter for a simple example of reading (and optionally writing) FTXT to and from the clipboard. See also the "Clipboard Device" and "Console Device" chapters of the Amiga ROM Kernel Reference Manual: Devices for more information on Release 2 console clip and paste.

1.64 33 IFFParse Library / IFFParse Examples

Two examples follow: ClipFTXT.c, and Sift.c. These are simple examples that demonstrate the use of the IFFParse library. More complex examples for displaying and saving ILBMs may be found in the IFF Appendix of the Amiga ROM Kernel Reference Manual: Devices.

Clipboard FTXT Example IFF Scanner Example

1.65 33 IFFParse Library / Function Reference

The following are brief descriptions of the IFFParse functions discussed in this chapter. IFFParse library functions are avialable in Release 2 of the Amiga OS and are backward compatible with older versions of the

system. Further information about these and other IFFParse functions can be found in the 3rd edition of the Amiga ROM Kernel Reference Manual: Includes and Autodocs, also from Addison-Wesley.

Table 33-2: IFFParse Library Functions

Function	Description
AllocIFF()	Creates an IFFHandle structure.
FreeIFF()	Frees the IFFHandle structure created with AllocIFF().
OpenIFF()	Initialize an IFFHandle structure to read or write an IFF stream.
CloseIFF()	Closes an IFF context.
InitIFF()	Initialize an IFFHandle as a user-defined stream.
InitIFFasDOS()	Initialize an IFFHandle as an AmigaDOS stream.
InitIFFasClip()	Initialize an IFFHandle as a clipboard stream.
OpenClipboard()	Create a handle on a clipboard unit for InitIFFasClip().
ParseIFF()	Parse an IFF file from an IFFHandle stream.
ReadChunkBytes()	Read bytes from current chunk into a buffer.
ReadChunkRecords()	Read record elements from the current chunk into a buffer.
StopChunk()	Declare a chunk that should cause ParseIFF() to return.
CurrentChunk()	Get the context node for the current chunk.
PropChunk()	Specify a property chunk to store.
FindProp()	Search for a stored property in a given context.
CollectionChunk()	Declare a chunk type for collection.
FindCollection()	Get a pointer to the current list of collection items.
StopOnExit()	Declare a stop condition for exiting a chunk.
EntryHandler()	Add an entry handler to the IFFHandle context.
ExitHandler()	Add an exit handler to the IFFHandle context.
PushChunk()	Push a given context node onto the top of the context stack.
PopChunk()	Pop the top context node off of the context stack.
CurrentChunk()	Get the top context node for the current chunk.
ParentChunk()	Get the nesting context node for a given chunk.
AllocLocalItem()	Create a LocalContextItem (LCI) structure.
LocalItemData()	Returns a pointer to the user data of a LocalContextItem (LCI).
StoreLocalItem()	Insert a LocalContextItem (LCI).
StoreItemInContext()	Store a LocalContextItem in a given context node.
FindPropContext()	Find the property context for the current state.
FindLocalItem()	Return a LocalContextItem from the context

	stack.
FreeLocalItem()	Free a LocalContextItem (LCI) created with AllocLocalItem().
SetLocalItemPurge()	Set purge vector for a local context item.