

**iffparse**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> iffparse		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		March 28, 2025	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>iffparse</b>	<b>1</b>
1.1	iffparse.doc	1
1.2	HookEntry	1
1.3	iffparse.library/AllocIFF	2
1.4	iffparse.library/AllocLocalItem	2
1.5	iffparse.library/CloseClipboard	3
1.6	iffparse.library/CloseIFF	4
1.7	iffparse.library/CollectionChunk	4
1.8	iffparse.library/CollectionChunks	5
1.9	iffparse.library/CurrentChunk	6
1.10	iffparse.library/EntryHandler	7
1.11	iffparse.library/ExitHandler	8
1.12	iffparse.library/FindCollection	9
1.13	iffparse.library/FindLocalItem	10
1.14	iffparse.library/FindProp	11
1.15	iffparse.library/FindPropContext	11
1.16	iffparse.library/FreeIFF	12
1.17	iffparse.library/FreeLocalItem	13
1.18	iffparse.library/GoodID	13
1.19	iffparse.library/GoodType	14
1.20	iffparse.library/IDtoStr	15
1.21	iffparse.library/InitIFF	15
1.22	iffparse.library/InitIFFasClip	17
1.23	iffparse.library/InitIFFasDOS	17
1.24	iffparse.library/LocalItemData	18
1.25	iffparse.library/OpenClipboard	19
1.26	iffparse.library/OpenIFF	19
1.27	iffparse.library/ParentChunk	20
1.28	iffparse.library/ParseIFF	21
1.29	iffparse.library/PopChunk	22

---

1.30	iffparse.library/PropChunk . . . . .	23
1.31	iffparse.library/PropChunks . . . . .	23
1.32	iffparse.library/PushChunk . . . . .	24
1.33	iffparse.library/ReadChunkBytes . . . . .	25
1.34	iffparse.library/ReadChunkRecords . . . . .	26
1.35	iffparse.library/SetLocalItemPurge . . . . .	26
1.36	iffparse.library/StopChunk . . . . .	27
1.37	iffparse.library/StopChunks . . . . .	28
1.38	iffparse.library/StopOnExit . . . . .	29
1.39	iffparse.library/StoreItemInContext . . . . .	29
1.40	iffparse.library/StoreLocalItem . . . . .	30
1.41	iffparse.library/WriteChunkBytes . . . . .	31
1.42	iffparse.library/WriteChunkRecords . . . . .	32

---

## Chapter 1

# iffparse

### 1.1 iffparse.doc

HookEntry()	FreeIFF()	PropChunk()
AllocIFF()	FreeLocalItem()	PropChunks()
AllocLocalItem()	GoodID()	PushChunk()
CloseClipboard()	GoodType()	ReadChunkBytes()
CloseIFF()	IDtoStr()	ReadChunkRecords()
CollectionChunk()	InitIFF()	SetLocalItemPurge()
CollectionChunks()	InitIFFasClip()	StopChunk()
CurrentChunk()	InitIFFasDOS()	StopChunks()
EntryHandler()	LocalItemData()	StopOnExit()
ExitHandler()	OpenClipboard()	StoreItemInContext()
FindCollection()	OpenIFF()	StoreLocalItem()
FindLocalItem()	ParentChunk()	WriteChunkBytes()
FindProp()	ParseIFF()	WriteChunkRecords()
FindPropContext()	PopChunk()	

### 1.2 HookEntry

#### NAME

HookEntry -- call-back stub vector (LANGUAGE SPECIFIC LINK ROUTINE)

#### SYNOPSIS

This function is never called directly by the client.

#### FUNCTION

HookEntry's purpose is to do language-specific setup and conversion of parameters passed from a library to a client call-back routine. Under Kickstart 2.0, a standard for call-backs has been established. The registers will contain the following items:

A0:	pointer to hook that enabled us to get here.
A2:	pointer to "object."
A1:	pointer to "message packet."

In iffparse, the "object" will vary from routine to routine. The "message packet" is also specific to the operation involved (RTFM!).

THIS ROUTINE IS NOT PART OF IFFPARSE. It, or something similar, is part of the compiler vendor's link library. (If it's not there, cobbling up your own isn't too hard.)

SEE ALSO

EntryHandler(), ExitHandler(), InitIFF(), SetLocalItemPurge(),  
utility/hooks.h (A must-read; LOTS of details in there)

### 1.3 iffparse.library/AllocIFF

NAME

AllocIFF -- Create a new IFFHandle structure.

SYNOPSIS

```
iff = AllocIFF ()  
d0
```

```
struct IFFHandle *iff;
```

FUNCTION

Allocates a new IFFHandle structure and initializes the basic values. This function is the only supported way to create an IFFHandle structure since there are private fields that need to be initialized.

INPUTS

RESULT

iff - pointer to IFFHandle structure or NULL if the allocation failed.

EXAMPLE

NOTES

BUGS

SEE ALSO

FreeIFF()

### 1.4 iffparse.library/AllocLocalItem

NAME

AllocLocalItem -- Create a local context item structure.

SYNOPSIS

```
item = AllocLocalItem (type, id, ident, usize)  
d0 d1 d2 d3
```

```
struct LocalContextItem *item;  
LONG type, id, ident, usize;
```

## FUNCTION

Allocates and initializes a LocalContextItem structure with "usize" bytes of associated user data. This is the only supported way to create such an item. The user data can be accessed with the LocalItemData function. An item created with this function automatically has its purge vectors set up correctly to dispose of itself and its associated user data area. Any additional cleanup should be done with a user-supplied purge vector.

## INPUTS

type,id - additional longword identification values.  
ident - longword identifier for class of context item.  
usize - number of bytes of user data to allocate for this item.

## RESULT

item - pointer to initialized LocalContextItem or NULL if the allocation failed.

## EXAMPLE

## NOTES

## BUGS

## SEE ALSO

FreeLocalItem(), LocalItemData(), StoreLocalItem(),  
StoreItemInContext(), SetLocalItemPurge()

## 1.5 iffparse.library/CloseClipboard

## NAME

CloseClipboard -- Close and free an open ClipboardHandle.

## SYNOPSIS

```
CloseClipboard (clip)
               a0
```

```
struct ClipboardHandle *clip;
```

## FUNCTION

Closes the clipboard.device and frees the ClipboardHandle structure.

## INPUTS

clip - pointer to ClipboardHandle struct created with  
OpenClipboard.

## RESULT

## EXAMPLE

## NOTES

## BUGS

## SEE ALSO

```
OpenClipboard(), InitIFFasClip()
```

## 1.6 iffparse.library/CloseIFF

### NAME

CloseIFF -- Close an IFF context.

### SYNOPSIS

```
CloseIFF (iff)
        a0
```

```
struct IFFHandle *iff;
```

### FUNCTION

Completes an IFF read or write operation by closing the IFF context established for this IFFHandle struct. The IFFHandle struct itself is left ready for re-use and a new context can be opened with OpenIFF(). This function can be used for cleanup if a read or write fails partway through.

As part of its cleanup operation, CloseIFF() calls the client-supplied stream hook vector. The IFFStreamCmd packet will be set as follows:

```
sc_Command:      IFFCMD_CLEANUP
sc_Buf:          (Not applicable)
sc_NBytes:       (Not applicable)
```

This operation is NOT permitted to fail; any error code returned will be ignored (best to return 0, though). DO NOT write to this structure.

### INPUTS

```
iff      - pointer to IFFHandle struct previously opened with
           OpenIFF().
```

### RESULT

### EXAMPLE

### NOTES

### BUGS

### SEE ALSO

```
OpenIFF(), InitIFF()
```

## 1.7 iffparse.library/CollectionChunk

### NAME

CollectionChunk -- declare a chunk type for collection.

---



## SYNOPSIS

```
error = CollectionChunk (iff, type, id)
      d0                a0    d0    d1
```

```
LONG          error;
struct IFFHandle *iff;
LONG          type;
LONG          id;
```

## FUNCTION

Installs an entry handler for chunks with the given type and id so that the contents of those chunks will be stored as they are encountered. This is like PropChunk() except that more than one chunk of this type can be stored in lists which can be returned by FindCollection(). The storage of these chunks still follows the property chunk scoping rules for IFF files so that at any given point, stored collection chunks will be valid in the current context.

## INPUTS

```
iff      - pointer to IFFHandle struct (does not need to be open).
type     - type code for the chunk to declare (ex. "ILBM").
id       - identifier for the chunk to declare (ex. "CRNG").
```

## RESULT

```
error    - 0 if successful or an IFFERR_#? error code if not
           successful.
```

## EXAMPLE

## NOTES

## BUGS

## SEE ALSO

```
CollectionChunks(), FindCollection(), PropChunk()
```

## 1.8 iffparse.library/CollectionChunks

## NAME

```
CollectionChunks -- Declare many collection chunks at once.
```

## SYNOPSIS

```
error = CollectionChunks (iff, list, n)
      d0                a0    a1    d0
```

```
LONG          error;
struct IFFHandle *iff;
LONG          *list;
LONG          n;
```

## FUNCTION

Declares multiple collection chunks from a list. The list argument is a pointer to an array of long words arranged in pairs. The format for the list is as follows:

TYPE1, ID1, TYPE2, ID2, ..., TYPEn, IDn

The argument *n* is the number of pairs. `CollectionChunks()` just calls `CollectionChunk()` *n* times.

#### INPUTS

`iff` - pointer to `IFFHandle` struct.  
`list` - pointer to array of longword chunk types and identifiers.  
`n` - number of chunks to declare.

#### RESULT

`error` - 0 if successful or an `IFFERR_#?` error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

`CollectionChunk()`

## 1.9 iffparse.library/CurrentChunk

#### NAME

`CurrentChunk` -- Get context node for current chunk.

#### SYNOPSIS

```
top = CurrentChunk (iff)
d0                                a0

struct ContextNode *top;
struct IFFHandle   *iff;
```

#### FUNCTION

Returns top context node for the given `IFFHandle` struct. The top context node corresponds to the chunk most recently pushed on the stack, which is the chunk where the stream is currently positioned. The `ContextNode` structure contains information on the type of chunk currently being parsed (or written), its size and the current position within the chunk.

#### INPUTS

`iff` - pointer to `IFFHandle` struct.

#### RESULT

`top` - pointer to top context node or `NULL` if none.

#### EXAMPLE

#### NOTES

#### BUGS

---

SEE ALSO

PushChunk(), PopChunk(), ParseIFF(), ParentChunk()

## 1.10 iffparse.library/EntryHandler

NAME

EntryHandler -- Add an entry handler to the IFFHandle context.

SYNOPSIS

```
error = EntryHandler (iff, type, id, position, hook, object)
      d0              a0      d0      d1      d2      a1      a2
```

```
LONG          error;
struct IFFHandle *iff;
LONG          type, id, position;
struct Hook   *hook;
APTR          object;
```

FUNCTION

Installs an entry handler vector for a specific type of chunk into the context for the given IFFHandle struct. Type and id are the longword identifiers for the chunk to handle. The hook is a client-supplied standard 2.0 Hook structure, properly initialized. Position tells where to put the handler in the context. The handler will be called whenever the parser enters a chunk of the given type, so the IFF stream will be positioned to read the first data byte in the chunk. The handler will execute in the same context as whoever called ParseIFF(). The handler will be called (through the hook) with the following arguments:

```
A0:      the Hook pointer you passed.
A2:      the 'object' pointer you passed.
A1:      pointer to a LONG containing the value
         IFFCMD_ENTRY.
```

The error code your call-back routine returns will affect the parser in three different ways:

Return value	Result
-----	-----
0:	Normal success; ParseIFF() will continue through the file.
IFF_RETURN2CLIENT:	ParseIFF() will stop and return the value 0. (StopChunk() is internally implemented using this return value.)
Any other value:	ParseIFF() will stop and return the value you supplied. This is how errors should be returned.

INPUTS

```
iff      - pointer to IFFHandle struct.
type     - type code for chunk to handle (ex. "ILBM").
id       - ID code for chunk to handle (ex. "CMAP").
position - Local context item position. One of the IFFSLI_#? codes.
hook     - pointer to Hook structure.
```

object - a client-defined pointer which is passed in A2 during call-back.

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

Returning the values IFFERR\_EOF or IFFERR\_EOC from the call-back routine *\*may\** confuse the parser.

There is no way to explicitly remove a handler once installed. However, by installing a do-nothing handler using IFFSLI\_TOP, previous handlers will be overridden until the context expires.

#### SEE ALSO

ExitHandler(), StoreLocalItem(), StoreItemInContext(),  
utility/hooks.h

## 1.11 iffparse.library/ExitHandler

#### NAME

ExitHandler -- Add an exit handler to the IFFHandle context.

#### SYNOPSIS

```
error = ExitHandler (iff, type, id, position, hook, object)
d0              a0      d0    d1      d2          a1      a2
```

```
LONG           error;
struct IFFHandle *iff;
LONG           type, id, position;
struct Hook     *hook;
APTR           object;
```

#### FUNCTION

Installs an exit handler vector for a specific type of chunk into the context for the given IFFHandle struct. Type and id are the longword identifiers for the chunk to handle. The hook is a client-supplied standard 2.0 Hook structure, properly initialized. Position tells where to put the handler in the context. The handler will be called just before the parser exits the given chunk in the "pause" parse state. The IFF stream may not be positioned predictably within the chunk. The handler will execute in the same context as whoever called ParseIFF(). The handler will be called (through the hook) with the following arguments:

```
A0:      the Hook pointer you passed.
A2:      the 'object' pointer you passed.
A1:      pointer to a LONG containing the value
          IFFCMD_EXIT.
```

The error code your call-back routine returns will affect the parser in three different ways:

Return value	Result
-----	-----
0:	Normal success; ParseIFF() will continue through the file.
IFF_RETURN2CLIENT:	ParseIFF() will stop and return the value 0. (StopChunk() is internally implemented using this return value.)
Any other value:	ParseIFF() will stop and return the value you supplied. This is how errors should be returned.

#### INPUTS

iff - pointer to IFFHandle struct.  
 type - type code for chunk to handle (ex. "ILBM").  
 id - identifier code for chunk to handle (ex. "CMAP").  
 position - local context item position. One of the IFFSLI\_#? codes.  
 hook - pointer to Hook structure.  
 object - a client-defined pointer which is passed in A2 during call-back.

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

Returning the values IFFERR\_EOF or IFFERR\_EOC from the call-back routine *may* confuse the parser.

There is no way to explicitly remove a handler once installed. However, by installing a do-nothing handler using IFFSLI\_TOP, previous handlers will be overridden until the context expires.

#### SEE ALSO

EntryHandler(), StoreLocalItem(), StoreItemInContext(),  
 utility/hooks.h

## 1.12 iffparse.library/FindCollection

#### NAME

FindCollection -- Get a pointer to the current list of collection items.

#### SYNOPSIS

```
ci = FindCollection (iff, type, id)
d0                  a0      d0      d1

struct CollectionItem *ci;
struct IFFHandle      *iff;
```



matched.

EXAMPLE

NOTES

BUGS

It really should have some sort of wildcarding capability.

SEE ALSO

StoreLocalItem()

## 1.14 iffparse.library/FindProp

NAME

FindProp -- Search for a stored property chunk.

SYNOPSIS

```
sp = FindProp (iff, type, id)
d0          a0      d0      d1
```

```
struct StoredProperty      *sp;
struct IFFHandle           *iff;
LONG                       type, id;
```

FUNCTION

Searches for the stored property which is valid in the given context. Property chunks are automatically stored by ParseIFF() when pre-declared by PropChunk() or PropChunks(). The StoredProperty struct, if found, contains a pointer to a data buffer containing the contents of the stored property.

INPUTS

iff        - pointer to IFFHandle struct.  
type      - type code for chunk to search for (ex. "ILBM").  
id        - identifier code for chunk to search for (ex. "CMAP").

RESULT

sp        - pointer to stored property, if found, or NULL if none found.

EXAMPLE

NOTES

BUGS

SEE ALSO

PropChunk(), PropChunks()

## 1.15 iffparse.library/FindPropContext

---

## NAME

FindPropContext -- Get the property context for the current state.

## SYNOPSIS

```
cn = FindPropContext (iff)
d0                                a0
```

```
struct ContextNode *cn;
struct IFFHandle   *iff;
```

## FUNCTION

Locates the context node which would be the scoping chunk for properties in the current parsing state. (Huh?) This is used for locating the proper scoping context for property chunks i.e. the scope from which a property would apply. This is usually the FORM or LIST with the highest precedence in the context stack.

If you don't understand this, read the IFF spec a couple more times.

## INPUTS

iff        - pointer to IFFHandle struct.

## RESULT

cn        - ContextNode of property scoping chunk.

## EXAMPLE

## NOTES

## BUGS

## SEE ALSO

CurrentChunk(), ParentChunk(), StoreItemInContext()

## 1.16 iffparse.library/FreeIFF

## NAME

FreeIFF -- Deallocate an IFFHandle struct.

## SYNOPSIS

```
FreeIFF (iff)
a0
```

```
struct IFFHandle *iff;
```

## FUNCTION

Deallocates all resources associated with this IFFHandle struct. The struct MUST have already been closed with CloseIFF().

## INPUTS

iff        - pointer to IFFHandle struct to free.

## RESULT



EXAMPLE

NOTES

BUGS

SEE ALSO

AllocIFF(), CloseIFF()

## 1.17 iffparse.library/FreeLocalItem

NAME

FreeLocalItem -- Deallocate a local context item structure.

SYNOPSIS

```
FreeLocalItem (lci)
               a0
```

```
struct LocalContextItem *lci;
```

FUNCTION

Frees the memory for the local context item and any associated user memory as allocated with AllocLocalItem. User purge vectors should call this function after they have freed any other resources associated with this item.

Note that FreeLocalItem() does NOT call the custom purge vector set up through SetLocalItemPurge(); all it does is free the local context item. (This implies that your custom purge vector would want to call this to ultimately free the LocalContextItem.) (This description still seems muddy; how to clear it up?)

INPUTS

lci            - pointer to LocalContextItem created with AllocLocalItem.

RESULT

EXAMPLE

NOTES

BUGS

SEE ALSO

AllocLocalItem()

## 1.18 iffparse.library/GoodID

NAME

GoodID -- Test if an identifier follows the IFF 85 specification.

SYNOPSIS

---

```
isok = GoodID (id)
      d0          d0
```

```
LONG isok, id;
```

#### FUNCTION

Tests the given longword identifier to see if it meets all the EA IFF 85 specifications for a chunk ID. If so, it returns non-zero, otherwise 0.

#### INPUTS

id            - potential 32 bit identifier.

#### RESULT

isok        - non-zero if this is a valid ID, 0 otherwise.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

GoodType()

## 1.19 iffparse.library/GoodType

#### NAME

GoodType -- Test if a type follows the IFF 85 specification.

#### SYNOPSIS

```
isok = GoodType (type)
      d0          d0
```

```
LONG isok, type;
```

#### FUNCTION

Tests the given longword type identifier to see if it meets all the EA IFF 85 specifications for a FORM type (requirements for a FORM type are more stringent than those for a simple chunk ID). If it complies, GoodType() returns non-zero, otherwise 0.

#### INPUTS

type        - potential 32 bit format type identifier.

#### RESULT

isok        - non-zero if this is a valid type id, 0 otherwise.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

---

```
GoodID()
```

## 1.20 iffparse.library/IDtoStr

### NAME

IDtoStr -- Convert a longword identifier to a null-terminated string.

### SYNOPSIS

```
str = IDtoStr (id, buf)
d0          d0  a0
```

```
STRPTR  str;
LONG    id;
STRPTR  buf;
```

### FUNCTION

Writes the ASCII equivalent of the given longword ID into buf as a null-terminated string.

### INPUTS

```
id      - longword ID.
buf     - character buffer to accept string (at least 5 chars).
```

### RESULT

```
str     - the value of 'buf'.
```

### EXAMPLE

### NOTES

### BUGS

### SEE ALSO

## 1.21 iffparse.library/InitIFF

### NAME

InitIFF -- Initialize an IFFHandle struct as a user stream.

### SYNOPSIS

```
InitIFF (iff, flags, streamhook)
a0      d0      a1
```

```
struct IFFHandle *iff;
LONG             flags;
struct Hook      *streamhook;
```

### FUNCTION

Initializes an IFFHandle as a general user-defined stream by allowing the user to declare a hook that the library will call to accomplish the low-level reading, writing, and seeking of the stream. Flags are the stream I/O flags for the specified stream; typically a

combination of the IFFF\_?SEEK flags.

The stream vector is called with the following arguments:

A0: pointer to streamhook.  
 A2: pointer to IFFHandle struct.  
 A1: pointer to IFFStreamCmd struct.

The IFFStreamCmd packet appears as follows:

sc\_Command: Contains an IFFCMD\_#? value  
 sc\_Buf: Pointer to memory buffer  
 sc\_NBytes: Number of bytes involved in operation

The values taken on by sc\_Command, and their meaning, are as follows:

#### IFFCMD\_INIT:

Prepare your stream for reading. This is used for certain streams that can't be read immediately upon opening, and need further preparation. (The clipboard.device is an example of such a stream.) This operation is allowed to fail; any error code will be returned directly to the client. sc\_Buf and sc\_NBytes have no meaning here.

#### IFFCMD\_CLEANUP:

Terminate the transaction with the associated stream. This is used with streams that can't simply be closed. (Again, the clipboard is an example of such a stream.) This operation is not permitted to fail; any error returned will be ignored (best to return 0, though). sc\_Buf and sc\_NBytes have no meaning here.

#### IFFCMD\_READ:

Read from the stream. You are to read sc\_NBytes from the stream and place them in the buffer pointed to by sc\_Buf. Any (non-zero) error returned will be remapped by the parser into IFFERR\_READ.

#### IFFCMD\_WRITE:

Write to the stream. You are to write sc\_NBytes to the stream from the buffer pointed to by sc\_Buf. Any (non-zero) error returned will be remapped by the parser into IFFERR\_WRITE.

#### IFFCMD\_SEEK:

Seek on the stream. You are to perform a seek on the stream relative to the current position. sc\_NBytes is signed; negative values mean seek backward, positive values mean seek forward. sc\_Buf has no meaning here. Any (non-zero) error returned will be remapped by the parser into IFFERR\_SEEK.

All errors are returned in D0. A return of 0 indicates success. UNDER NO CIRCUMSTANCES are you permitted to write to the IFFStreamCmd structure.

#### INPUTS

iff - pointer to IFFHandle structure to initialize.  
 flags - stream I/O flags for the IFFHandle.  
 hook - pointer to Hook structure.

#### RESULT

EXAMPLE

NOTES

BUGS

SEE ALSO

utility/hooks.h

## 1.22 iffparse.library/InitIFFasClip

NAME

InitIFFasClip -- Initialize an IFFHandle as a clipboard stream.

SYNOPSIS

```
InitIFFasClip (iff)
               a0
```

```
struct IFFHandle *iff;
```

FUNCTION

Initializes the given IFFHandle to be a clipboard stream. The function initializes the stream processing vectors to operate on streams of the ClipboardHandle type. The iff\_Stream field will still need to be initialized to point to a ClipboardHandle as returned from OpenClipboard().

INPUTS

iff            - pointer to IFFHandle struct.

RESULT

EXAMPLE

NOTES

BUGS

SEE ALSO

OpenClipboard()

## 1.23 iffparse.library/InitIFFasDOS

NAME

InitIFFasDOS -- Initialize an IFFHandle as a DOS stream.

SYNOPSIS

```
InitIFFasDOS (iff)
               a0
```

```
struct IFFHandle *iff;
```

---

**FUNCTION**

The function initializes the given IFFHandle to operate on DOS streams. The iff\_Stream field will need to be initialized as a BPTR returned from the DOS function Open().

**INPUTS**

iff        - pointer to IFFHandle struct.

**RESULT****EXAMPLE****NOTES****BUGS****SEE ALSO**

## 1.24 iffparse.library/LocalItemData

**NAME**

LocalItemData -- Get pointer to user data for local context item.

**SYNOPSIS**

```
data = LocalItemData (lci)
      d0              a0
```

```
UBYTE                    *data;
struct LocalContextItem   *lci;
```

**FUNCTION**

Returns pointer to the user data associated with the given local context item. The size of the data area depends on the "usize" argument used when allocating this item. If the pointer to the item given (lci) is NULL, the function also returns NULL.

**INPUTS**

lci        - pointer to local context item or NULL.

**RESULT**

data       - pointer to user data area or NULL if lci is NULL.

**EXAMPLE****NOTES****BUGS**

Currently, there is no way to determine the size of the user data area; you have to 'know'.

**SEE ALSO**

AllocLocalItem(), FreeLocalItem()

---

## 1.25 iffparse.library/OpenClipboard

### NAME

OpenClipboard -- Create a handle on a clipboard unit.

### SYNOPSIS

```
ch = OpenClipboard (unit)
d0                                d0
```

```
struct ClipboardHandle    *ch;
LONG                      unit;
```

### FUNCTION

Opens the clipboard.device and opens a stream for the specified unit (usually PRIMARY\_CLIP). This handle structure will be used as the clipboard stream for IFFHandles initialized as clipboard streams by InitIFFasClip().

### INPUTS

unit      - clipboard unit number (usually PRIMARY\_CLIP).

### RESULT

ch        - pointer to ClipboardHandle structure or NULL if unsuccessful.

### EXAMPLE

### NOTES

### BUGS

### SEE ALSO

InitIFFasClip(), CloseClipboard()

## 1.26 iffparse.library/OpenIFF

### NAME

OpenIFF -- Prepare an IFFHandle to read or write a new IFF stream.

### SYNOPSIS

```
error = OpenIFF (iff, rwmode)
d0          a0      d0
```

```
LONG          error;
struct IFFHandle *iff;
LONG          rwmode;
```

### FUNCTION

Initializes an IFFHandle struct for a new read or write. The direction of the I/O is given by the value of rwmode, which can be either IFFF\_READ or IFFF\_WRITE.

As part of its initialization procedure, OpenIFF() calls the client-supplied stream hook vector. The IFFStreamCmd packet will contain

the following:

```
sc_Command:    IFFCMD_INIT
sc_Buf:        (Not applicable)
sc_NBytes:     (Not applicable)
```

This operation is permitted to fail. DO NOT write to this structure.

#### INPUTS

```
iff          - pointer to IFFHandle struct.
rwmode       - IFFF_READ or IFFF_WRITE
```

#### RESULT

```
error        - contains an error code or 0 if successful.
```

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

```
CloseIFF(), InitIFF()
```

## 1.27 iffparse.library/ParentChunk

#### NAME

ParentChunk -- Get the nesting context node for the given chunk.

#### SYNOPSIS

```
parent = ParentChunk (cn)
          d0          a0
```

```
struct ContextNode *parent, *cn;
```

#### FUNCTION

Returns a context node for the chunk containing the chunk for the given context node. This function effectively moves down the context stack into previously pushed contexts. For example, to get a ContextNode pointer for the enclosing FORM chunk while reading a data chunk, use: ParentChunk (CurrentChunk (iff)) to find this pointer. The ContextNode structure contains information on the type of chunk and its size.

#### INPUTS

```
cn          - pointer to a context node.
```

#### RESULT

```
parent      - pointer to the enclosing context node or NULL if none.
```

#### EXAMPLE

#### NOTES

#### BUGS

---



SEE ALSO  
CurrentChunk()

## 1.28 iffparse.library/ParseIFF

### NAME

ParseIFF -- Parse an IFF file from an IFFHandle struct stream.

### SYNOPSIS

```
error = ParseIFF (iff, control)
d0          a0      d0
```

```
LONG          error;
struct IFFHandle *iff;
LONG          control;
```

### FUNCTION

This is the biggie.

Traverses a file opened for read by pushing chunks onto the context stack and popping them off directed by the generic syntax of IFF files. As it pushes each new chunk, it searches the context stack for handlers to apply to chunks of that type. If it finds an entry handler it will invoke it just after entering the chunk. If it finds an exit handler it will invoke it just before leaving the chunk. Standard handlers include entry handlers for pre-declared property chunks and collection chunks and entry and exit handlers for stop chunks - that is, chunks which will cause the ParseIFF() function to return control to the client. Client programs can also provide their own custom handlers.

The control flag can have three values:

#### IFFPARSE\_SCAN:

In this normal mode, ParseIFF() will only return control to the caller when either:

- 1) an error is encountered,
- 2) a stop chunk is encountered, or a user handler returns the special IFF\_RETURN2CLIENT code, or
- 3) the end of the logical file is reached, in which case IFFERR\_EOF is returned.

ParseIFF() will continue pushing and popping chunks until one of these conditions occurs. If ParseIFF() is called again after returning, it will continue to parse the file where it left off.

#### IFFPARSE\_STEP and \_RAWSTEP:

In these two modes, ParseIFF() will return control to the caller after every step in the parse, specifically, after each push of a context node and just before each pop. If returning just before a pop, ParseIFF() will return IFFERR\_EOC, which is not an error, per se, but is just an indication that the most recent context is ending. In STEP

---

mode, ParseIFF() will invoke the handlers for chunks, if any, before returning. In RAWSTEP mode, ParseIFF() will not invoke any handlers and will return right away. In both cases the function can be called multiple times to step through the parsing of the IFF file.

#### INPUTS

iff - pointer to IFFHandle struct.  
control - control code (IFFPARSE\_SCAN, \_STEP or \_RAWSTEP).

#### RESULT

error - 0 or IFFERR\_#? value or return value from user handler.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

PushChunk(), PopChunk(), EntryHandler(), ExitHandler(),  
PropChunk[s](), CollectionChunk[s](), StopChunk(), StopOnExit()

## 1.29 iffparse.library/PopChunk

#### NAME

PopChunk -- Pop top context node off context stack.

#### SYNOPSIS

```
error = PopChunk (iff)
d0          a0

LONG          error;
struct IFFHandle *iff;
```

#### FUNCTION

Pops top context chunk and frees all associated local context items. The function is normally called only for writing files and signals the end of a chunk.

#### INPUTS

iff - pointer to IFFHandle struct.

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

PushChunk()

## 1.30 iffparse.library/PropChunk

### NAME

PropChunk -- Specify a property chunk to store.

### SYNOPSIS

```
error = PropChunk (iff, type, id)
          d0          a0      d0   d1
```

```
LONG          error;
struct IFFHandle *iff;
LONG          type;
LONG          id;
```

### FUNCTION

Installs an entry handler for chunks with the given type and ID so that the contents of those chunks will be stored as they are encountered. The storage of these chunks follows the property chunk scoping rules for IFF files so that at any given point, a stored property chunk returned by FindProp() will be the valid property for the current context.

### INPUTS

iff        - pointer to IFFHandle struct (does not need to be open).  
type      - type code for the chunk to declare (ex. "ILBM").  
id        - identifier for the chunk to declare (ex. "CMAP").

### RESULT

error     - 0 if successful or an IFFERR\_#? error code if not successful.

### EXAMPLE

### NOTES

### BUGS

### SEE ALSO

PropChunks(), FindProp(), CollectionChunk()

## 1.31 iffparse.library/PropChunks

### NAME

PropChunks -- Declare many property chunks at once.

### SYNOPSIS

```
error = PropChunks (iff, list, n)
          d0          a0      a1   d0
```

```
LONG          error;
```

---

```

struct IFFHandle *iff;
LONG             *list;
LONG             n;

```

#### FUNCTION

Declares multiple property chunks from a list. The list argument is a pointer to an array of long words arranged in pairs, and has the following format:

```

TYPE1, ID1, TYPE2, ID2, ..., TYPEn, IDn

```

The argument n is the number of pairs. PropChunks() just calls PropChunk() n times.

#### INPUTS

```

iff      - pointer to IFFHandle struct.
list     - pointer to array of longword chunk types and identifiers.
n        - number of chunks to declare.

```

#### RESULT

```

error    - 0 if successful or an IFFERR_#? error code if not
           successful.

```

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

PropChunk()

## 1.32 iffparse.library/PushChunk

#### NAME

PushChunk -- Push a new context node on the context stack.

#### SYNOPSIS

```

error = PushChunk (iff, type, id, size)
d0          a0      d0      d1      d2

LONG        error;
struct IFFHandle *iff;
LONG        type, id, size;

```

#### FUNCTION

Pushes a new context node on the context stack by reading it from the stream if this is a read file, or by creating it from the passed parameters if this is a write file. Normally this function is only called in write mode, where the type and id codes specify the new chunk to create. If this is a leaf chunk, i.e. a local chunk inside a FORM or PROP chunk, then the type argument is ignored. If the size is specified then the chunk writing functions will enforce this size. If the size is given as IFFSIZE\_UNKNOWN, the chunk will expand to accommodate whatever is written into it.

## INPUTS

iff        - pointer to IFFHandle struct.  
 type      - chunk type specifier (ex. ILBM) (ignored for read mode or leaf chunks).  
 id        - chunk id specifier (ex. CMAP) (ignored for read mode).  
 size      - size of the chunk to create or IFFSIZE\_UNKNOWN (ignored for read mode).

## RESULT

error     - 0 if successful or an IFFERR\_#? error code if not successful.

## EXAMPLE

## NOTES

## BUGS

## SEE ALSO

PopChunk(), WriteChunkRecords(), WriteChunkBytes()

## 1.33 iffparse.library/ReadChunkBytes

## NAME

ReadChunkBytes -- Read bytes from the current chunk into a buffer.

## SYNOPSIS

```

actual = ReadChunkBytes (iff, buf, size)
    d0                      a0  a1  d0

```

```

LONG          actual;
struct IFFHandle *iff;
UBYTE         *buf;
LONG          size;

```

## FUNCTION

Reads the IFFHandle stream into the buffer for the specified number of bytes. Reads are limited to the size of the current chunk and attempts to read past the end of the chunk will truncate. Function returns positive number of bytes read or a negative error code.

## INPUTS

iff        - pointer to IFFHandle struct.  
 buf        - pointer to buffer area to receive data.  
 size      - number of bytes to read.

## RESULT

actual     - (positive) number of bytes read if successful or a (negative) IFFERR\_#? error code if not successful.

## EXAMPLE

## NOTES

BUGS

SEE ALSO

`ReadChunkRecords()`, `ParseIFF()`, `WriteChunkBytes()`

## 1.34 iffparse.library/ReadChunkRecords

NAME

`ReadChunkRecords` -- Read record elements from the current chunk into a buffer.

SYNOPSIS

```
actual = ReadChunkRecords (iff, buf, rectx, numrec)
      d0                  a0  a1      d0      d1
```

```
LONG          actual;
struct IFFHandle *iff;
UBYTE         *buf;
LONG          rectx, numrec;
```

FUNCTION

Reads records from the current chunk into buffer. Truncates attempts to read past end of chunk (only whole records are read; remaining bytes that are not of a whole record size are left unread and available for `ReadChunkBytes()`).

INPUTS

```
iff      - pointer to IFFHandle struct.
buf      - pointer to buffer area to receive data.
rectx    - size of data records to read.
numrec   - number of data records to read.
```

RESULT

```
actual   - (positive) number of whole records read if successful or a
          (negative) IFFERR_#? error code if not successful.
```

EXAMPLE

NOTES

BUGS

SEE ALSO

`ReadChunkBytes()`, `ParseIFF()`, `WriteChunkRecords()`

## 1.35 iffparse.library/SetLocalItemPurge

NAME

`SetLocalItemPurge` -- Set purge vector for a local context item.

SYNOPSIS

```
SetLocalItemPurge (item, purgehook)
```

a0            a1

```
struct LocalContextItem     *item;
struct Hook                *purgehook;
```

#### FUNCTION

Sets a local context item to use a client-supplied cleanup (purge) vector for disposal when its context is popped. The purge vector will be called when the ContextNode containing this local item is popped off the context stack and is about to be deleted itself. If the purge vector has not been set, the parser will use FreeLocalItem to delete the item, but if this function is used to set the purge vector, the supplied vector will be called with the following arguments:

```
A0:        pointer to purgehook.
A2:        pointer to LocalContextItem to be freed.
A1:        pointer to a LONG containing the value
           IFFCMD_PURGELCI.
```

The user purge vector is then responsible for calling FreeLocalItem() as part of its own cleanup. Although the purge vector can return a value, it will be ignored -- purge vectors must always work (best to return 0, though).

#### INPUTS

```
item        - pointer to local context item.
purgehook - pointer to a Hook structure.
```

#### RESULT

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

AllocLocalItem(), FreeLocalItem(), utility/hooks.h

## 1.36 iffparse.library/StopChunk

#### NAME

StopChunk -- Declare a chunk which should cause ParseIFF to return.

#### SYNOPSIS

```
error = StopChunk (iff, type, id)
                  d0            a0        d0        d1
```

```
LONG                error;
struct IFFHandle *iff;
LONG                type;
LONG                id;
```

#### FUNCTION

---

Installs an entry handler for the specified chunk which will cause the ParseIFF() function to return control to the caller when this chunk is encountered. This is only of value when ParseIFF() is called with the IFFPARSE\_SCAN control code.

#### INPUTS

iff - pointer to IFFHandle struct (need not be open).  
 type - type code for chunk to declare (ex. "ILBM").  
 id - identifier for chunk to declare (ex. "BODY").

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

StopChunks(), ParseIFF()

## 1.37 iffparse.library/StopChunks

#### NAME

StopChunks -- Declare many stop chunks at once.

#### SYNOPSIS

```
error = StopChunks (iff, list, n)
    d0              a0    a1  d0
```

```
LONG          error;
struct IFFHandle *iff;
LONG          *list;
LONG          n;
```

#### FUNCTION

(is to StopChunk() as PropChunks() is to PropChunk().)

#### INPUTS

iff - pointer to IFFHandle struct.  
 list - pointer to array of longword chunk types and identifiers.  
 n - number of chunks to declare.

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS



SEE ALSO  
 StopChunk()

## 1.38 iffparse.library/StopOnExit

### NAME

StopOnExit -- Declare a stop condition for exiting a chunk.

### SYNOPSIS

```
error = StopOnExit (iff, type, id)
                d0      a0      d0      d1
```

```
LONG          error;
struct IFFHandle *iff;
LONG          type;
LONG          id;
```

### FUNCTION

Installs an exit handler for the specified chunk which will cause the ParseIFF() function to return control to the caller when this chunk is exhausted. ParseIFF() will return IFFERR\_EOC when the declared chunk is about to be popped. This is only of value when ParseIFF() is called with the IFFPARSE\_SCAN control code.

### INPUTS

```
iff      - pointer to IFFHandle struct (need not be open).
type     - type code for chunk to declare (ex. "ILBM").
id       - identifier for chunk to declare (ex. "BODY").
```

### RESULT

```
error    - 0 if successful or an IFFERR_#? error code if not
           successful.
```

### EXAMPLE

### NOTES

### BUGS

### SEE ALSO

ParseIFF()

## 1.39 iffparse.library/StoreItemInContext

### NAME

StoreItemInContext -- Store local context item in given context node.

### SYNOPSIS

```
StoreItemInContext (iff, item, cn)
                  a0      a1      a2
```

```
struct IFFHandle *iff;
```

```
struct LocalContextItem    *item;
struct ContextNode    *cn;
```

#### FUNCTION

Adds the LocalContextItem to the list of items for the given context node. If an LCI with the same Type, ID, and Ident is already present in the ContextNode, it will be purged and replaced with the new one. This is a raw form of StoreLocalItem.

#### INPUTS

```
iff      - pointer to IFFHandle struct for this context.
item     - pointer to a LocalContextItem to be stored.
cn       - pointer to context node in which to store item.
```

#### RESULT

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

StoreLocalItem()

## 1.40 iffparse.library/StoreLocalItem

#### NAME

StoreLocalItem -- Insert a local context item into the context stack.

#### SYNOPSIS

```
error = StoreLocalItem (iff, item, position)
d0              a0      a1      d0
```

```
LONG              error;
struct IFFHandle   *iff;
struct LocalContextItem    *item;
LONG              position;
```

#### FUNCTION

Adds the local context item to the list of items for one of the context nodes on the context stack and purges any other item in the same context with the same ident, type and id. The position argument determines where in the stack to add the item:

##### IFFSLI\_ROOT:

Add item to list at root (default) stack position.

##### IFFSLI\_TOP:

Add item to the top (current) context node.

##### IFFSLI\_PROP:

Add element in top property context. Top property context is either the top FORM chunk, or the top LIST chunk, whichever is closer to the top of the stack.

Items added to the root context, or added to the top context before

the IFFHandle has been opened or after it has been closed, are put in the default context. That is, they will be the local items found only after all other context nodes have been searched. Items in the default context are also immune to being purged until the IFFHandle struct itself is deleted with FreeIFF(). This means that handlers installed in the root context will still be there after an IFFHandle struct has been opened and closed. (Note that this implies that items stored in a higher context will be deleted when that context ends.)

#### INPUTS

iff - pointer to IFFHandle struct.  
 item - pointer to LocalContextItem struct to insert.  
 position- where to store the item (IFFSLI\_ROOT, \_TOP or \_PROP).

#### RESULT

error - 0 if successful or an IFFERR\_#? error code if not successful.

#### EXAMPLE

#### NOTES

#### BUGS

#### SEE ALSO

FindLocalItem(), StoreItemInContext(), EntryHandler(), ExitHandler()

## 1.41 iffparse.library/WriteChunkBytes

#### NAME

WriteChunkBytes -- Write data from a buffer into the current chunk.

#### SYNOPSIS

```
error = WriteChunkBytes (iff, buf, size)
    d0                a0  a1  d0
```

```
LONG                error;
struct IFFHandle *iff;
UBYTE               *buf;
LONG                size;
```

#### FUNCTION

Writes "size" bytes from the specified buffer into the current chunk. If the current chunk was pushed with IFFSIZE\_UNKNOWN, the size of the chunk gets increased by the size of the buffer written. If the size was specified for this chunk, attempts to write past the end of the chunk will be truncated.

#### INPUTS

iff - pointer to IFFHandle struct.  
 buf - pointer to buffer area with bytes to be written.  
 size - number of bytes to write.

#### RESULT

---

error - (positive) number of bytes written if successful or a  
(negative) IFFERR\_#? error code if not successful.

EXAMPLE

NOTES

BUGS

SEE ALSO

PushChunk(), PopChunk(), WriteChunkRecords()

## 1.42 iffparse.library/WriteChunkRecords

NAME

WriteChunkRecords -- Write records from a buffer to the current  
chunk.

SYNOPSIS

```
error = WriteChunkRecords (iff, buf, recsize, numrec)
      d0                      a0  a1      d0      d1
```

```
LONG          error;
struct IFFHandle *iff;
UBYTE         *buf;
LONG          recsize, numrec;
```

FUNCTION

Writes record elements from the buffer into the top chunk. This  
function operates much like ReadChunkBytes().

INPUTS

iff - pointer to IFFHandle struct.  
buf - pointer to buffer area containing data.  
recsize - size of data records to write.  
numrec - number of data records to write.

RESULT

error - (positive) number of whole records written if successful  
or a (negative) IFFERR\_#? error code if not successful.

EXAMPLE

NOTES

BUGS

SEE ALSO

WriteChunkBytes()

---