Chapter 21
IFF Support

IFF files contain information that can be shared between various programs. Graphics and Animation programs on the Amiga save picture information in IFF files. Thus you can save a picture from a paint program and use it in an animation program. Music programs will save sound samples in IFF files The IFF standard was developed by Electronic Arts.

JForth provides three levels of support for IFF files. The lowest allows you to write a program that can read or write various IFF files. The second level provides an easy system for reading and writing ILBM picture files. The top level provides a rudimentary animation language based on the manipulation and display of images read from IFF files.

IFF pictures can even be combined with free form graphics generated using the graphics library for some very special custom effects. An example application, JSHOW, is included to show you how to open a custom screen and display pictures from a file. If you are not interested in the IFF picture files, you may want to skip ahead to the section on the IFF file format.

We do not provide high level support for 8SVX sample files but this could be written using the IFF tools. We do not imagine that this system will serve all the needs of all the people. We have, however, tried to provide tools that are flexible enough so that others can use them. Our main intent here is to provide you with a starting point for developing your own code. All the IFF source code is in the directory JIFF:. I urge you to print it out, study it, modify it and make it your own.

Description of Files in JIFF:

PICTURES - High level system for loading and displaying IFF pictures. This words could be used as the basis for a bitmapped animation system. You can combine these words with the normal graphics calls for custom effects. If you want to save your resulting animation to give to friends, Clone it!

PIC_EFFECTS - Special effects using Pictures - Wipe and Fadein, Fadeout.

PIC_FLIP - Flip a picture about x,y or diagonally.

ILBM_PARSER - Tools for parsing an ILBM bitmap.

ILBM_MAKER - Tools for writing a bitmap as an ILBM IFF file.

SHOW_IFF - Display tools, application for displaying IFF files.

IFF_Support - This file contains the core words for parsing an IFF file. It has tools for reading and writing chunks, and processing special chunks like 'FORM'.

PACKING - Low level support for packing picture data into ILBM form. Support for run length encoding a Bitmap and converting a CTABLE to a CAMP.

UNPACKING - Routines for unpacking data from an ILBM IFF file.

IFF.J - Definition of BitMapHeader structure and the common chunk IDS.

Tutorial 1 - Displaying Pictures

First let's compile the IFF code and display a simple picture. We provide an IFF picture, 2 brushes and an animbrush in the directory JPics:.

Enter:

```
    INCLUDE JIFF:SHOW_IFF
```

```
JSHOW JPICS:MOUNTAINS.PIC
```

JSHOW will read your file, open a screen, and display the picture. Click in the top left corner when you are done looking. (The images provided with JForth were chosen for their small size when compressed and not on artistic merit, as you will see.)

Tutorial 2 - The Picture System

Now let's get fancy with bitmaps. We will use the highest level - the "JForth IFF Picture System". As a first step, let's load a picture to use as our background. We can use the same picture of mountains. Enter:

```
INCLUDE JIFF:LOAD_PIC
GR.INIT   \ Open Graphics and Intuition libraries
PICTURE BACKG  ( declare structure )
" jpics:mountains.pic" BACKG $PIC.LOAD? .
```

The second line declared a picture structure that is used to keep track of pictures that are loaded into the system. You can declare as many of these structures as you need. The third line loaded the graphics from the file "jpics:mountains.pic" and stored it "in" the Picture structure called BACKG. A zero should have been returned from this word if everything went OK. The FIRST picture loaded always causes a screen to open with the proper resolution and depth to display that picture. It is important that this first picture be a full screen picture representative of the resolution you will be using in your program.

You should now see your picture. You can move between screens by hitting these key combinations: <left-Amiga-N> to get the workbench screen, <left-Amiga-M> to get other screens. You might now want to shrink your JForth window and pull your workbench screen down a bit (grab it at the top) so you can see both screens. Click in the JForth window, then enter:

```
PICTURE MYSHIP
" jpics:ship.br" MYSHIP $PIC.LOAD? . ( read bitmap )
```

With the above commands, we read the JPICS:SHIP.BR brush into a bitmap and saved the pointer to the bitmap in the structure MYSHIP.

Let's draw our ship in the screen. Enter:

```
20 30 MYSHIP PIC.BLIT
```

Click over to the screen with the mountains on it. You will probably see a rectangular block near the top left of the screen. Inside the block will be the ship. (The word "BLIT" is computerese for Block Transfer of Pixels. This means one image is drawn in another.)

The black rectangle is not a bug. Bitmaps are rectangular and when you just draw them using PIC.BLIT you get the whole thing. Luckily there is a way to copy a bitmap while keeping the background of the bitmap transparent. This will be more like what you would see when you draw with a brush in a paint program.

To copy a bitmap transparently we need to use a shadow mask of that bitmap. A shadow mask is a special kind of bitmap that has only one real memory plane. It is designed to look as if it has several planes. When you blit another bitmap into it, any color other than 0 will turn on the pixel in the shadow mask. This shadow mask is then used to cut a hole in the destination bitmap. The picture can then be placed into that hole using an OR mode Blit. The picture system will make a shadow mask for you automatically if you try to do a transparent blit.

```
120 40 MYSHIP PIC.TRANS.BLIT
```

If you look at the picture now you should see another ship but without the rectangle.

Drawing a Portion of a Picture

For this next exercise we would like to have two pictures so let’s modify the picture called BACKG to look different, then load another copy of our mountains. To change BACKG, let's draw a big rectangle in it. Enter (exactly):

```
    4 GR.COLOR!
    10 10  310 190 GR.RECT
```

There should now be a big rectangle in the middle of the picture.

Now let's reload up our mountains in another PICTURE for these next exercises.

```
    PICTURE MNTNS
    " jpics:mountains.pic" MNTNS $PIC.LOAD? .
```

Note: You will **not** see the new mountains picture. We are still displaying the modified BACKG picture.

We can use the picture system to draw only a portion of a bitmap. Suppose we want to take part of the top left of the mountain scene and draw it to the middle of the background. Enter:

```
    100 80 MNTNS PIC.PUT.WH
```

This sets the width and height of the region to be drawn from. Enter:

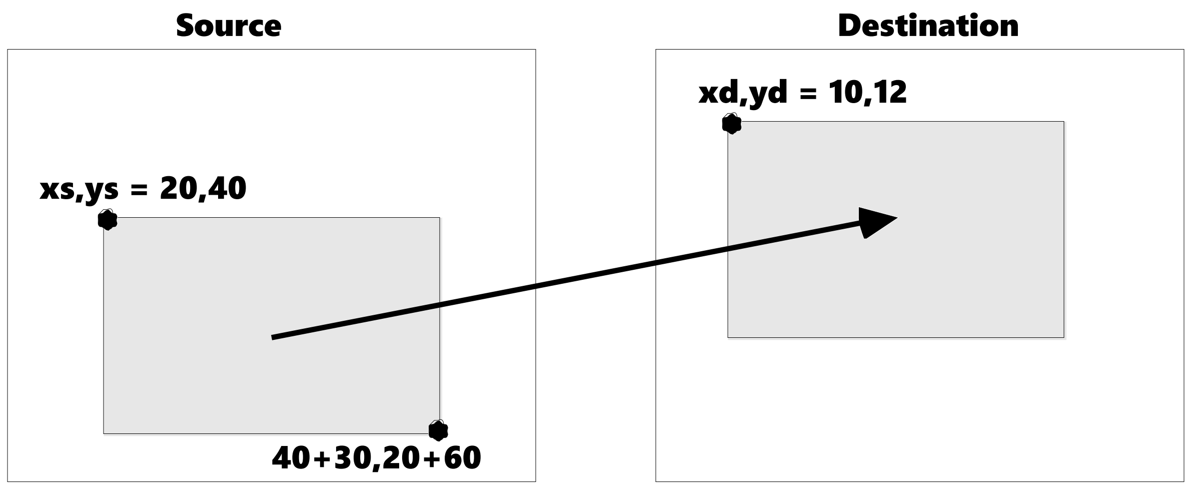
```
    120 70 MNTNS PIC.BLIT
```

You should now see the top left corner drawn in the middle of the screen. We can set the corner of our region anywhere in the picture. Enter:

```
    200 100 MNTNS PIC.PUT.XY
    0 0 MNTNS PIC.BLIT
```

You should now see a 100 by 80 pixel rectangular portion of the lower right of the mountain scene drawn at 0,0 on the screen.

This diagram shows the source region in a picture called Source and the region it is Blitted to in a picture called Destination.



```
20 40  SOURCE  PIC.PUT.XY
60 30  SOURCE  PIC.PUT.WH
10 12  SOURCE  PIC.BLIT
```

Special Effects - Wipes and Fades

We can do a few special effects that might be useful in constructing a video. Remember the composite output of your Amiga can be plugged into a VCR and recorded! To fade to black and then come back with another picture. Enter:

```
4 BACKG PIC.FADEOUT
PIC-START-BLACK ON   ( make MNTNS start black )
MNTNS PIC.DISPLAY
8 MNTNS PIC.FADEIN
```

This is a gentle way to transition between two scenes. The command PIC.DISPLAY makes the specified picture be visible. [Hacker's note - this works by copying pointers to that picture's bit planes to the open screen's bitmap.] The word PIC.FADEOUT and IN take a time parameter and a picture address. The time parameter is the number of frames to wait before each change in brightness.

In this system we distinguish between the picture being displayed and the picture being drawn into. This allows us to do what is called double buffering which is a way to eliminate some of the jitters in animation. We are displaying the MNTNS picture but we are still drawing into the BACKGROUND picture. Let's draw into the BACKGROUND using the Graphics toolbox then rapidly switch display buffers. Enter:

```
23 45 " Peace On Earth" GR.XYTEXT
```

If you look on the screen, you WON'T see this text. Now enter:

```
BACKG PIC.DISPLAY
```

We can do another effect called a wipe that is used as a transition between pictures. You may want to resize the JForth window then pull down the Workbench screen so that you can see the graphics screen as well. Watch the screen and enter:

```
MNTNS PIC.WHOLE ( reset source window to whole picture )
0 0 2 WIPE_RIGHT MNTNS PIC.WIPE
```

This told the Picture system to "wipe" the MNTNS picture into the current picture being drawn to. The 0,0 was the x,y of the top left corner. The 2 was the number of lines to blit per frame. The WIPE_RIGHT parameter was the direction. You can also choose between WIPE_LEFT, WIPE_UP, and WIPE_DOWN. You can wipe with part of a picture by setting the region with PIC.PUT.WH and PIC.PUT.XY.

Moving a Brush, Restoring the Background

Let's reload the MNTNS picture so that we have a clean slate. Enter:

```
" jpics:mountains.pic" MNTNS $PIC.LOAD? .
MNTNS PIC.DISPLAY
MNTNS PIC.DRAWTO
```

Suppose we wanted to make a brush move smoothly across the screen. We would need to draw it at one location, then draw it again moved slightly in the direction of motion, and so on. If we do this, however, we end up with a trail of brushes. Obviously we must erase one image before we draw the next. How can we do that? There are basically two ways. One is to rebuild the entire image by copying in a fresh image of the background that was saved away, then drawing the brush in the new position. This assumes that you have a copy of the background saved. If that is not that case then you will need method number two.

In this method we save a small portion of the background, just the amount that will be covered when we draw our brush. To erase our brush we can then copy this saved portion back to its original location.

Let's try this with our ship. First we must allocate a bitmap for this backup image. Enter:

```
0 MYSHIP PIC.ALLOC.BACKUP? .  \ must be zero
```

A zero is passed as the first parameter to select between backup image number zero or one. These two backups come in handy when doing double buffering because the brush appears in two backgrounds

Now let's make a backup copy under our brush, then draw the brush. Enter:

```
20 45 0 MYSHIP PIC.BACKUP.NTH  \ no visible change
20 45 MYSHIP PIC.BLIT     \ must be to same X,Y
```

Now to move the brush we must first restore the old background. Enter:

```
0 MYSHIP PIC.RESTORE.NTH
```

Notice that the background is restored.and we can redraw the brush in a new location. Enter:

```
24 53 0 MYSHIP PIC.BACKUP.NTH
24 53 MYSHIP PIC.BLIT
```

By continuing in this manner, a brush can be made to move across the screen.

If you are moving multiple brushes in an image that might overlap, you must follow a simple rule:

For multiple brushes, call PIC.RESTORE.NTH in the opposite order that you call PIC.BACKUP.NTH.

Otherwise you will not restore the image properly.

Cleaning Up

We must now cleanup the allocated memory, close the screen. Enter:

```
MNTNS PIC.FREE
BACKG PIC.FREE
MYSHIP PIC.FREE
```

Whichever picture is currently being displayed will close the screen when freed When finished using the graphics toolbox, we should close the Graphics and Intuition library be entering:

```
GR.TERM
```

For more examples of using the picture system, look at our simple test program: JIFF:TEST_PIC.

Picture System Reference

The Picture System provides easy to use routines for displaying and manipulating images read from IFF ILBM files. It can be loaded by entering:

```
INCLUDE JIFF:LOAD_PIC
```

Error Handling

Many of the routines return an ERROR? flag that is non-zero if there was an error. Your program should always check this flag when it is returned. The most common sources of errors would be if memory could not be allocated for an operation, or if there was a problem with a file. Well written applications should expect these errors to occur periodically and to respond gracefully when they do. Do not simply ABORT because your user might lose valuable work in progress. See the documentation for GOTO.ERROR for tips on handling errors.

Double Buffering

Double buffering is a technique used to achieve smooth animations. The basic sequence of operations is:

```
Display Buffer 0
Draw to Buffer 1
Display Buffer 1
Draw to Buffer 0
Repeat
```

In this manner the viewer always sees a static image while the other image is being drawn.

There are three ways to do double buffering provided by JForth, each with their own advantages and disadvantages. You can also develop your own system if none of these suit your needs.

1) Use the tools in JIFF:DOUBLE_BUFFER. These are demonstrated in JD:DEMO_DBUF. The advantage of these is that they are independent of the PICTURE system, and are very convenient.

2) Use PIC.VIEW to switch displays. This switches very fast because it uses LoadView() and is very convenient when using the PICTURE system. The disadvantage is that it works below the level of Intuition so mouse input may not go where you think it will. Be sure to bring your screen to front before calling this or else your mouse input may go to another screen. To bring the screen that the PICTURE system uses to the front, enter:

```
SIFF-SCREEN @ ScreenToFront()
```

3) Use PIC.DISPLAY to switch displays. This works well with Intuition but is quite slow. It takes about 2-3 frames just to switch displays compared to PIC.VIEW which is virtually instantaneous.

Using your Own Display Screen

The PICTURE system uses the screen pointed to by the variable SIFF-SCREEN. If you load a picture using $PIC.LOAD? and there is no screen, a screen will be opened for you and its address placed in SIFF-SCREEN. If you want to use your own screen, open it before calling $PIC.LOAD? and place your screen address in SIFF-SCREEN.

Clipping with Pictures

There are two types of clipping involved with the Picture system. If you draw to the picture that initiated the opening of the screen, then all graphics operations will be clipped to the BACKDROP window in that screen. This includes calls to PIC.BLIT and other calls like GR.DRAW or GR.TEXT. If, however, you call PIC.DRAWTO to draw to another picture, then only PIC.xxx calls will be clipped. Line drawing using GR.DRAW and similar calls will not be clipped and could result in the trashing of memory. This is because the RastPort of the BACKDROP window has a ClipRect but the RastPort for a picture does not. The PIC.xxx calls are clipped using a custom clipping routine designed for rectangular blits. By setting the variable PIC-CLIPPING OFF you can turn off this custom clipping.

All drawing is directed to the RastPort whose ABSOLUTE address is in the variable GR-CURRPORT. This can be set directly or by using GR.SET.CURWINDOW.

Picture Glossary

JIFF:PICTURE

`$PIC.LOAD? ( $filename picture  -- error? , load IFF picture )`

Load an IFF ILBM picture from a file to a picture. The first one loaded will open an appropriately sized screen for display. Make sure, therefore, that the first picture loaded is a full screen picture that is the right resolution for the rest of your animation. You might want to load a title screen first. Set the variable PIC-START-BLACK to TRUE if you want this one to start black. You can then fade in using PIC.FADEIN .

`$PIC.SAVE? ( $filename picture  -- error? )`

Save the contents of a picture in an IFF ILBM file for use with other graphics applications. The bitmap will be compressed using run length encoding.

`PIC-CLIPPING ( -- addr )`

When this variable is set TRUE, then PIC.BLIT and PIC.TRANS.BLIT will be clipped to the edges of the destination picture. This may be redundant if drawing to a window with Amiga based clipping.

PIC-START-BLACK (-- addr , variable to control color)

If this variable is set TRUE, then $PIC.LOAD? and PIC.DISPLAY will set the screen to all black. You can then use PIC.FADEIN to make the picture visible.

PIC.?BREAK (-- , OBSOLETE, don't use)

PIC.ALLOC.BACKUP? (backup# picture -- error?)

Allocate a bitmap to use with PIC.BACKUP.NTH. Backup# is 0 or 1.

PIC.ALLOC.SHADOW? (picture -- error?)

Allocate a shadow mask bitmap for use with PIC.TRANS.BLIT. You must also call PIC.CAST.SHADOW before calling PIC.TRANS.BLIT.

PIC.ALLOC.VIEW? (picture -- error?)

Allocate a VIEW for PIC.VIEW.

PIC.BACKUP.NTH (dstx dsty backup# pict --)

Backup part of the image that will be overwritten when we do a PIC.BLIT or PIC.TRANS.BLIT at the same DSTX and DSTY. That image can then be restored using PIC.RESTORE.NTH. This is used mostly when using a moving brush. Backup# is 0 or 1.

PIC.BLIT (xd yd picture -- , blit to x,y)

Blit the bitmap of a picture into the current rastport at xd,yd. The current rastport is the one whose absolute address is in GR-CURRPORT . PIC.DRAWTO can be used to select a picture to BLIT into. By BLITting a series of related pictures you can obtain smooth animation effects.

PIC.BUILD (bitmap picture -- , build a picture from scratch)

If you want to use a bitmap that does not come from an IFF file as a picture, use this word. The bitmap should already be initialized and have an image associated with it.

PIC.CAST.SHADOW (picture --)

Create a shadow mask by ORing each plane of a picture into a single plane. You should call this routine anytime you change a picture that you use with PIC.TRANS.BLIT.

PIC.CLOSEBOX (-- , OBSOLETE, don't use)

PIC.COPY (srcpic dstpic -- , copy bitmaps and color table)

Copy the contents of one picture to another. The second picture must be the same size as the first. Use PIC.DUPLICATE to allocate a same sized picture first if needed.

PIC.DISPLAY (picture -- , display picture by copying bitmaps)

Causes this picture to be the current one displayed. Use this with PIC.DRAWTO for double buffering. Loads pointers to this picture's bitmap planes into the SIFF screen's bitmap.

PIC.DRAWTO (picture -- , make this the destination)

Sets the JForth graphics variable GR-CURRPORT to point to this picture's Rastport (absolute address). Now PIC.BLIT, PIC.WIPE, etc. and all GR.xxx commands will draw into this pictures bitmap. You can draw into one picture while displaying another for a double buffering effect.

Warning: Pictures do not have clipping layers in their RastPorts. Thus any line drawing or other graphics operations may extend beyond the edges of the pictures and overwrite memory. PIC.BLIT and PIC.TRANS.BLIT will continue to be clipped as long as PIC-CLIPPING is set TRUE.

`PIC.DUPLICATE?  ( srcpic dstpic -- error? , )`

Take an empty picture and allocate bitmaps for it the same size as the source picture. Then call PIC.COPY to copy the bitmap contents.

`PIC.FREE  ( picture -- , free all parts of picture )`

Free all allocated internal data. This MUST be called when you are completely through using a picture. The picture that is currently being displayed via PIC.DISPLAY will close the SIFF screen when this word is called.

`PIC.GET.DEPTH ( picture -- depth , number of planes )`

`PIC.GET.WH ( picture -- w h , fetch source w and h )`

`PIC.GET.XY ( picture -- x y , fetch source x and y )`

`PIC.GET.XYOFF ( picture -- x y , fetch dest x and y )`

`PIC.MAKE? ( colrtab|0 #colors deep wide high pict -- error? )`

Create bitmaps and other necessary structures for a picture based on input parameters. This is an alternative to $PIC.LOAD?. COLRTAB can be zero or the address of a color table whose contents will be copied to a newly allocated color table.

`PIC.OPEN? ( picture -- screen | 0 )`

Open a screen based on the picture.

`PIC.PUT.WH ( width height picture -- , set source width, height )`

Sets the width and height of a rectangular portion of a picture. This is what will be drawn using PIC.BLIT and PIC.WIPE.

`PIC.PUT.XY ( x y picture -- , set source x and y )`

Set the top,left x,y of a rectangle to draw from. Used with PIC.PUT.WH .

`PIC.PUT.XYOFF ( x y picture -- , set dest x and y )`

When this picture is drawn with PIC.BLIT or PIC.TRANS.BLIT, the **destination** x,y will be offset using these values. This can be used to "center" a picture so that the coordinates you pass to PIC.BLIT determine where a certain part of a bitmap will land other than the top,left corner.

`PIC.RESTORE.NTH ( backup# picture -- )`

Restore the part of the image saved using PIC.BACKUP.NTH.

`PIC.TRANS.BLIT ( xd yd picture -- , blit transparently )`

Copy a bitmap into the current rastport using a transparent background. This is useful when using brushes from a point program. If you are drawing a head then you just want the round part superimposed over the background. Without transparency, you would get a black rectangle with a head in the middle! This routine uses a shadow mask rastport. The mask is used to punch a hole in the background where there is data in the picture. It then ORs the bitmap with the rastport. The opaque pixels in the bitmap will line up with the black hole in the background. The transparent black pixels in the bitmap will line up with remaining pixels in the background. When you or these you get a nice superimposition of the bitmap over the rastport.

`PIC.USE.COLORS ( picture -- , apply colors to screen )`

Use the color table from a picture in the SIFF screen.

`PIC.VIEW ( picture -- )`

Displays a picture by calling LoadView(). This will call PIC.ALLOC.VIEW? just in case it hasn't been called yet. This is faster than PIC.DISPLAY but Intuition will not realize the display has changed. This can cause unexpected results. If, for example, the Workbench screen was showing before this call, mouse clicks will still go to the Workbench even though this picture is showing in front.

PIC.WHOLE (picture -- , reset bounds to use whole picture)

Resets the source x,y and w,h to the full picture boundaries.

JIFF:PIC_EFFECTS

PIC.BRIGHTNESS (level picture -- , scale colormaps)

Set the SIFF screen brightness by scaling the color map in a picture. The range is zero to 16 where 16 is full brightness.

PIC.FADEIN (frames picture -- , fade in from black)

Call PIC.BRIGHTNESS in a loop from 0 to 16. The frames parameter determines how many video frames to wait between successive level changes. A value of 4 is nice.

PIC.FADEOUT (frames picture -- , fade to black)

PIC.NEXT.WIPE (picture -- done?)

Draw next portion of a wipe based on PIC.SETUP.WIPE call. Call this in a loop until it returns true. It doesn't hurt to call it after it's done.

PIC.ROTATE (-- , rotate siff-screen)

Rotate the plane pointers in the SIFF screen. Causes wild color changes. Do it as many times as there are planes if you want to get back to the same color. Use PIC.GET.DEPTH to find the number of planes.

PIC.SETUP.WIPE (xd yd nlines direction picture --)

Set up a picture for a wipe effect. The data will be taken from the source rectangle and drawn to where xd,yd is at the top,left corner. You can specify the number of lines per wipe pass. The more lines the faster the wipe. Try to make it divide evenly into the number of lines total. The direction parameter can be one of four values:

```
WIPE_LEFT  WIPE_RIGHT  WIPE_UP  WIPE_DOWN
```

WIPE_LEFT will cause to wipe to progress from right to left. You may setup several pictures then call PIC.NEXT.WIPE for each one in a loop to have simultaneous wipes happening in parallel.

PIC.WIPE (xd yd nlines direction pict -- , wipe a picture)

Call PIC.SETUP.WIPE then loop on PIC.NEXT.WIPE until done.

JIFF:PIC_FLIP

PIC.FLIP.X (picture --)

Flip a picture horizontally.

PIC.FLIP.Y (picture --)

Flip a picture vertically.

PIC.FLIP.DIAG (picture --)

Flip a picture about a line drawn from two diagonal corners.

IFF File Support

(For a more detailed description of IFF, please see the ROM Kernal Manual Volume 2)

IFF files are made up of "chunks". A chunk has 3 parts:

```
1) Chunk ID - 4 characters, eg. 'FORM', 'BMHD'
2) Chunk Size - in bytes
3) Chunk Data - whatever
```

The chunk ID tells a program what kind of chunk it is. For example, 'CMAP' means that the chunk is a color map for a picture. A chunk ID consists of 4 character packed into a 4 byte integer. The second part, the chunk size, tells you how many bytes are in the data portion. (If there are an odd number of data bytes, a pad byte will be added so that the next chunk starts on an even boundary.) The data portion is where the actual pictures, samples, note lists, etc. are stored.

There are several Chunk IDs which are considered special. These chunks can contain other chunks in their data portion. They are FORM, LIST and CAT. An IFF file consists of one of these 3 chunks containing one or more sub chunks. FORM chunks are the most common of the 3. The data portion of a FORM consists of a FormType followed by a number of subchunks. One common FormType is 'ILBM' which means that the FORM contains chunks that describe an InterLeaved BitMap, or picture. The chunks that describe a picture include 'BMHD', or BitMapHeaDer, which tells you how many pixels high and wide a picture is, how many bit planes it has, where it is positioned on the screen, etc. Another chunk in an ILBM is the 'BODY' that contains the actual pixels of the picture. These are often compressed to save space on the disk.

How JForth Handles IFF files

When you open an IFF file, you really don't know what kind of chunks you will find inside. To read an IFF file, therefore, you must be prepared to handle anything you find.

JForth provides a word called IFF.SCAN that reads the chunk headers and eats its way through an IFF file to see what chunks are there. It uses the chunk size to move from one chunk to the next. Once it has the chunk ID and size it passes these to the deferred word IFF.PROCESS.CHUNK. IFF.PROCESS.CHUNK can then check to see if it is a special type of chunk, ie. a 'FORM', 'LIST' or 'CAT'. This can be done by calling IFF.SPECIAL? which will process the special chunk if it is one. IFF.SPECIAL? returns a flag that tells IFF.PROCESS.CHUNK whether the chunk has already been processed. If not IFF.PROCESS.CHUNK can do whatever it needs to for that chunk. You can set this word to be anything you want and thus control how the IFF file is processed. There are several chunk processors to parse ILBM files, or to print an outline of chunks in the file.

Tutorial 3 - Vectored Parsing of IFF Files

In the previous tutorial, we used the existing ILBM parser to display an IFF picture. Let's now write our own custom parser.

(The parsing of IFF files is done using deferred words. If you are not familiar with DEFER, please see the section on DEFER in this manual.)

IFF.PROCESS.CHUNK is the most important deferred word in this system. Its stack diagram is:

```
IFF.PROCESS.CHUNK ( size chkid -- )
```

Printing Chunk Headers

It is called by IFF.SCAN which is called by IFF.DOFILE. We can write a word to be executed when

IFF.PROCESS.CHUNK is called. Let's first write a simple word to print out the header of a chunk. We have a word called .CHKID that will print a packed 4 character chunk ID so let's use it. Enter:

```
: SHOW.CHUNK  ( size chkid -- )
        .CHKID SPACE . CR
;
20 'BMHD' SHOW.CHUNK
```

Now let's use this to examine our MOUNTAINS file. Make sure you are in the same directory as your MOUNTAINS file then enter:

```
' SHOW.CHUNK IS IFF.PROCESS.CHUNK
20 'BMHD' IFF.PROCESS.CHUNK
IFF.DOFILE MOUNTAINS
```

Notice that the chunk in the file is a FORM chunk. Where are the other chunks, the BitMapHeader (BMHD) or the pixels (BODY)? They are nested inside the FORM chunk. To parse an IFF file we need to have a recursive parser. This is easier than it sounds. We have a special word for handling chunks like FORM, LIST and CAT called IFF.SPECIAL? It's stack diagram is:

```
IFF.SPECIAL? ( size chkid -- handled? )
```

If the chunk is a special recursive chunk, this word will handle it and return TRUE. Other wise it will return FALSE. To handle a special chunk IFF.SPECIAL? calls IFF.SCAN which in turn calls IFF.PROCESS.CHUNK . Enough talk, let's show we can use this in our code. Enter:

```
: NESTED.SHOW ( size chkid -- )
        2DUP IFF.SPECIAL?
        IF 2DROP  ( we can ignore it )
        ELSE SHOW.CHUNK
        THEN
;
' NESTED.SHOW IS IFF.PROCESS.CHUNK
IFF.DOFILE MOUNTAINS
```

We should now see all the chunks in the file listed. We have a word already written that does the above using IFF-NESTED to show the recursive nature of the file. ("Now he tells me!") Try:

```
IFF.CHECK MOUNTAINS
```

This word can be cloned if you want it.

Parsing ILBM FORMs

JForth provides tools specifically for parsing an ILBM form. The word $ILBM.PARSE.FILE will scan a file for chunks. The BitMapHeader is copied to a structure called ILBM-Header. This contains information used to decipher the rest of the file. The packed BODY chunk and the CMAP chunk are read into allocated memory, and their pointers left in the variable ILBM-BODY and ILBM-CMAP. GRAB chunks and CAMG chunks are read directly into variables called ILBM-GRABXY and ILBM-CAMG. When the parser is finished you can pull values from these storage locations and build a display. Look in the file JIFF:SHOW_IFF for examples of how this is done.

Also take a look at the file JA:DumpIFF.f which prints the contents of an IFF file for analysis.

IFF Support Glossary

JIFF:ILBM_PARSER

$ILBM.PARSE.FILE? ($filename -- error? , parse an IFF file)

Parse an ILBM file based on whatever is in IFF.PROCESS.CHUNK. Uses $IFF.DOFILE. Use

ILBM.PARSE.INIT to initialize IFF.PROCESS.CHUNK if you have changed it and want to use the original ILBM parser.

`HEADER>BITMAP ( bitmapheader -- bitmap | 0 , allocate bitmap)`

Allocate a properly sized bitmap based on the contents of the BitMapHeader structure.

These next few variables and structures are set by the ILBM Parser as it reads an IFF file. Look in here for the information found.

`ILBM-BODY  ( -- var-addr , holds pointer to allocated 'BODY' )`

`ILBM-BSIZE ( -- var-addr , holds size of 'BODY' )`

`ILBM-CAMG  ( -- var-addr , holds viewmodes from any 'CAMG' chunk)`

`ILBM-GRABXY ( -- var-addr , holds packed 16 bit x,y from 'GRAB' )`

`ILBM-CMAP  ( -- var-addr , holds pointer to allocated 'CMAP' )`

`ILBM-CMSIZE ( -- var-addr , holds size in bytes of CMAP )`

`ILBM-HEADER  ( -- addr , handy BitMapHeader structure )`

The ILBM Parser fills this structure with information from the 'BMHD' chunk. Read the IFF.J file for a list of members.

`ILBM.ALLOC.BITMAP ( -- bitmap | 0 )`

Allocate a bitmap of the appropriate size and depth based on the information in the ILBM-Header. This bitmap will be used to receive the bitmap as it is unpacked from the 'BODY' chunk. Calls HEADER>BITMAP. Returns zero if the bitmap could not be allocated.

`ILBM.CLEANUP ( -- , free any data allocated )`

Free BODY and CMAP chunk memory allocated by ILBM.HANDLER.

`ILBM.FILL.BITMAP ( bitmap -- bitmap | 0)`

Unpack the body pointed to by ILBM-BODY into the bitmap. Returns 0 if there is an unpacking error.

`ILBM.PARSER ( size chkid -- , default handler used to parse ILBM)`

IFF.PROCESS.CHUNK is set to call this word by ILBM.INIT. BMHD chunks are copied to ILBM-HEADER , a BitMapHeader structure. BODY and CMAP chunks are read into an allocated memory area whose address is stored in ILBM-BODY or ILBM-CMAP. GRAB chunks are read into the ILBM-GRABXY variable. CAMG chunks are read into the ILBM-CAMG variable. Any other chunks are passed to the deferred word ILBM.OTHER.HANDLER for custom processing.

`ILBM.INIT ( -- , set vectors )`

Set IFF.PROCESS.CHUNK to ILBM.HANDLER and set ILBM.OTHER.HANDLER to IFF.NOT.PROC .

`ILBM.MAKE.BITMAP ( body bsize bmheader -- bitmap | 0 )`

Allocate and fill a bitmap based on body and BitMapHeader.

`ILBM.MAKE.CTABLE  ( -- ctable num_colors  )`

Allocate a color table based on the information in the CMAP chunk. Return two NULLs if there was no CMAP or it couldn't be allocated.

`ILBM.OTHER.HANDLER  ( size chkid -- , handle other chunks)`

Deferred word called by ILBM.HANDLER when it sees a chunk it doesn't handle, 'CRNG' for example.

JIFF:ILBM_MAKER

`ILBM.HEADER.SETUP  ( bitmap bmapheader -- , set w,h and depth )`

Setup BitMapHeader structure values based on the bitmap.

`ILBM.WRITE.BITMAP?  ( bitmap -- error? )`

Write as a BODY chunk to the currently open IFF file.

`ILBM.WRITE.ILBM?  ( bmap ctable ctable# -- error? )`

Write a bitmap and a ctable to an IFF file. You must call $IFF.OPEN first then call IFF.CLOSE. This is to be considered as an example program. You will probably want to make a copy of this in another file and modify it to suit your purposes.

`$SCREEN>IFF?  ( screen $filename -- error? )`

This handy word pulls the bitmap and colortable from a screen and writes an IFF file.

JIFF:SHOW_IFF

Most of these words use the screen pointed to by the SIFF-SCREEN variable.

`$IFF>BITMAP ( $filename -- bitmap | 0)`

Read a file, allocate a bitmap and load it with the picture in the file. You can use this bitmap as a brush or whatever. You must use FREE.BITMAP from JU:GRAPH_SUPPORT to free this bitmap when done.

`$IFF>DISPLAY ( $filename -- bitmap | 0 , display iff on screen )`

Read an IFF file, open an appropriate screen and display the picture. The screen address will be in the variable SIFF-SCREEN . When done you should call SIFF.CLOSE and then use FREE.BITMAP to deallocate the bitmap you have been given. Look at the source code for JSHOW.

`IFF>BITMAP ( <filename> -- bitmap | 0 , read IFF file )`

Reads filename from input stream and calls $IFF>BITMAP.

`IFF>DISPLAY ( <filename> -- bitmap | 0 , open screen and display)`

Reads filename from input stream and calls $IFF>DISPLAY.

`JSHOW  ( <filename> -- )`

Read the IFF file and display the picture. Take down the picture when you click the topleft corner or hit a key. This can be cloned for a handy IFF display program.

`SIFF-SCREEN ( -- addr , variable holding address of screen )`

`SIFF-WINDOW ( -- addr , variable holding address of window )`

`SIFF.BLACKOUT ( -- , black out colors on screen )`

`SIFF.SHOWIT ( -- , Put window display in front for closebox.)`

`SIFF.USE.CMAP ( cmap cmsize -- , use CMAP directly from IFF)`

Set colors in SIFF screen based on CMAP.

`SIFF.USE.CTABLE ( ctable #colors -- , use Amiga CTABLE )`

`SIFF.CLOSE ( -- , Close SIFF screen and window.)`

`SIFF.WAIT ( -- , Wait until CLOSEBOX or Keyboard is hit.)`

Low Level Support

JIFF:IFF_SUPPORT

`$IFF.DOFILE? ( $filename -- error? )`

Process file using deferred words. Open the file whose name is on the stack, pull out the chunks and call IFF.PROCESS.CHUNK after verifying that it is an IFF file.

`$IFF.OPEN?  ( $filename -- fileid | 0 )`

Open a file for IFF.READ and IFF.WRITE. Set IFF-FILEID variable to file-pointer.

`.CHKID ( chkid -- , print a chunk id as 4 characters )`

`IFF.BEGIN.FORM?  ( type -- start-position error? )`

Start writing an IFF 'FORM' chunk. An example of type is 'ILBM'. The start position is saved for IFF.END.FORM.

`IFF.CHECK ( <filename> -- , print chunks )`

Set IFF.PROCESS.CHUNK to execute IFF.PRINT.CHUNK then call IFF.DOFILE. This results in a list of the chunks that are in an IFF file. This is a handy tool that could be cloned.

`IFF.NOT.PROC ( size chkid -- , default for ILBM.OTHER.HANDLER )`

Prints a message that a chunk was not processed.

`IFF.PRINT.CHUNK  ( size chkid -- , print chunk id and size )`

`IFF.PROCESS.CHUNK ( size chkid -- )`

This deferred word is called from IFF.SCAN when a new chunk is encountered in the file. You can set it to your own word for customized parsing of IFF files.

`IFF.PROCESS.FORM ( size -- )`

This reads a 'FORM' chunk and processes all of the chunks it finds by calling IFF.PROCESS.CHUNK .

`IFF.READ ( addr #bytes -- #bytes , read from open IFF file)`

This uses the fileid obtained using $IFF.OPEN .

IFF.READ? (addr #bytes -- error?, read from open IFF file)

Calls IFF.READ and returns ERROR? true if the number of bytes read does not the number of bytes requested.

IFF.READ.CHKID (-- size chkid | 0 0)

Read the next 8 bytes in file assuming it is a chunk header. Return 0 0 if an error occurs.

IFF.READ.DATA (dsize -- addr | null , allocate space)

Read DSIZE bytes from the IFF file into an allocated memory area. Return NULL if couldn't allocate. This is handy if you encounter a big chunk.

IFF.READ.TYPE (-- typeid | 0)

Read the next 4 byte from the IFF file. Used by words like IFF.PROCESS.FORM to check the FORM type. Return zero if an error occurs.

IFF.SCAN (-- size , read chunk header and doit)

Read the next chunk header and pass it to IFF.PROCESS.CHUNK then move the file pointer past that chunk's data.

IFF.SEEK (position -- , move file pointer, "seek")

The next read or write will occur at this new position.

IFF.SPECIAL? (size chkid -- done?)

Check to see if the chunk is one of the special type, ie. 'FORM', 'LIST', or 'CAT'. If so process it and return true. This calls IFF.SCAN which can result in recursion. Increments IFF-NESTED to indicate depth of recursion.

IFF.WHERE (-- current_pos , in file)

Where are we currently positioned in file?

IFF.WRITE (addr #bytes -- #bytes , write to open IFF file)

Write data to file opened by $IFF.OPEN .

IFF.WRITE? (addr #bytes -- error?)

Calls IFF.WRITE then checks to make sure the number of bytes written matches the requested number. Note: the stack diagram for this word has changed since JForth V2.0. See the note on incompatibilities at the end of this chapter.

IFF.WRITE.CHKID? (size chkid -- error? , write chunk header)

Write an 8 byte chunk header using IFF.WRITE.

IFF.WRITE.CHUNK? (address size chkid -- error?)

Write complete chunk to current file.

JIFF:UNPACKING

BODY>BITMAP (bodyptr bsize bmap compr -- bmap | NULL)

Unpack a body into a bitmap using given compression mode.

CMAP>CTABLE (cmap ctable #entries -- , unpack)

Convert an IFF CMAP to an Amiga CTABLE array.

UNPACKROW (src dst #src #dst -- src' dst' #src' error?)

Unpack a run length encoded row from source to destination.

JIFF:PACKING

This is a new version of PACKING provided by Martin Kees. It uses a virtual file system to write BODY chunks to a file as they are created.

CTABLE>CMAP (ctable cmap #entries -- , pack)

Convert a CTABLE array to an IFF CMAP.

ILBM.MAKE.BODY (bmap compr -- bodyptr bsize | -1)

Allocate a body memory area then pack the bitmap into it using the desired compression mode. Return -1 if an error occurs. Uses the virtual file system to write to a RAM: based file, then reads it back.

WRITE.BITMAP.BODY { bmap ifffile compr -- bodysize | 0 }

Write a bitmap to a file as a BODY chunk. It will be run length encoded if COMPR = 0.

JIFF:PACKING_OLD

This is the old version of the packing code that had problems with highly randomized pictures whose compressed form was larger than the original form. Obsolete.

BITMAP>BODY (bmap bodyptr bsize compr -- bsize'|-1)

Pack a bitmap into a body using given compression mode.

PACKROW (src dst src# dst# -- dst' dst# error?)

Pack a row of data using run length encoding. This could be used for other than picture data!

Incompatibilities with JForth V2.0

There have been some changes that may make code written using JForth 2.0 incompatible with JForth V3.0. In version 2.0, when an error occurred, the deferred word IFF.ERROR was called which typically caused an abort. This is fine when debugging but is totally unacceptable for a finished application. A properly written application should test for possible errors and handle them gracefully. The previous version did not allow programmers to do that. We felt it was better to correct this problem than to perpetuate a mistake. Unfortunately, some words have been changed. Words that might fail due to insufficient memory or problems with a file now return an error flag. We added a '?' at the end of their names to distinguish them from their original versions and to indicate that they return something of interest. Examples are ILBM.WRITE.BITMAP? and $IFF.DOFILE?. One notable exception to this is IFF.WRITE? which existed in V2.0 but did not return a flag. The stack diagram for this word was actually changed so that it now returns an error flag. I hated doing this but felt it was justified for the purpose of consistency.

Another incompatibility is that when $PIC.LOAD? is first called, the current RastPort is set to that of the backdrop window. The advantage of this is that clipping is active for all graphics. If you then call PIC.DISPLAY for another picture, graphics output will got to that picture as if you had called PIC.DRAWTO for that picture. If you want graphic output to go to a specific picture you must call PIC.DRAWTO explicitly. Be aware that only PIC.xxx calls are clipped after a call to PIC.DRAWTO. See the section in this chapter on clipping for more information.