Electronic Arts is a company that deserves credit for helping make life easier for both programmers and end users. By establishing **Interchange Format Files** (ie, IFF) and releasing the documentation for such, as well as C source code for reading and writing IFF type of files, Electronic Arts has helped make it easier for programmers to develop "backward compatible" and "extensible" file formats. IFF also helps developers write programs that easily read data files created with each others' IFF compliant software, even if there is no business relationship between the developers. In a nutshell, IFF helps minimize problems such as new versions of a particular program having trouble reading data files produced by older versions, or needing a new file format everytime a new version needs to store additional information. It also encourages standardized file formats that aren't tied to a particular product. All of this is good for endusers because it means that their valuable data isn't locked into some proprietary standard that can't be used with a wide variety of hardware and software. Above all else, endusers don't want their work to be held hostage by a single, corporate entity over whom the enduser has no direct control, but that's exactly what happens whenever an enduser saves his data using a program that produces a proprietary, unpublished file format. IFF helps to break this needlessly proprietary stranglehold that developers have exerted upon endusers' works.

An IFF file is a set of data that is in a form that many, unrelated programs can read. An IFF file should not have anything in it that was intended specifically for just one, particular program. If a program must save some "personal" (ie, proprietary) data in an IFF file, it must be saved in a manner which allows another program to "skip over" this data. There are several different types of IFF files. ILBM and GIFF files store picture data. SMUS files store musical scores. WAVE and AIFF files store sampled sounds. Each of these files must start with an ID which indicates that it is indeed an IFF file, followed by an ID that indicates which type of file. So what is an ID? An ID is four, printable ascii characters (ie, 8-bit bytes). If you use a file viewer (capable of displaying each byte as an ascii character) to look at an IFF file, you will notice that every so often you will see 4 "readable" characters in a row. These 4 characters are an ID. Every IFF file must start with one of the following 3 IDs. (I've enclosed each ID in single quotes).

```
'FORM'

'LIST'

'CAT '
```

If the first 4 chars (bytes) in a file are not one of these, then it is not an IFF file. These IDs are referred to as **group ID**s in EA literature because each is like a "master ID" after which there may follow more IDs (ie, chunks) that are grouped under that master ID.

Note that the last character in the 'CAT ' ID is a blank space (ie, ascii 32).

After this group ID, there is an UNSIGNED LONG (ie, 32-bit binary value) that indicates how many bytes are in the entire file. This count does not include the 4 byte group ID, nor this ULONG. This

ULONG is useful if you wish to load the rest of the file into memory to examine it. After this ULONG, there is an ID that indicates which type of IFF file this is. As mentioned earlier, "ILBM", "WAVE", and "AIFF" are 3 types of IFF files. There are many more, and programmers are always inventing new types for lack of better things to do. Here is the beginning of a typical ILBM file.

'FORM'	OK. This really is an IFF file because it has one of the 3 defined group IDs.
13000	There are 13000 more bytes after this ULONG.
'ILBM'	It is an ILBM (picture) file.

All IFF files start with something similiar to the above, 12 byte "header", except that instead of 'FORM', the group ID can be 'LIST' or 'CAT '. Of course, the ULONG size and file type ID may be different in various files, but nevertheless, a 12 byte header always appears at the beginning of an IFF file. For example, here's an example AIFF header:

'FORM'	OK. This really is an IFF file because it has one of the 3 defined group IDs.
4000	There are 4000 more bytes after this ULONG.
'AIFF'	It is an AIFF (digital audio) file.

What you find after the header depends on which type it is (ie, From here on, an ILBM will be different than an AIFF).

One thing that all IFF files do have in common after the group ID, byte count, and type ID, is that data is organized into chunks. OK, more jargon. What's a chunk? A chunk consists of an ID, a ULONG that tells how many bytes of data are in the chunk, and then all those data bytes. For example, here is a CMAP chunk (which would be found in an ILBM file).

'CMAP'	This is the 4 byte chunk ID.
6	This tells how many data bytes are in the chunk (ie, This is the chunkSize).
0,0,0,1,1,4	Here are the 6 data bytes.

Notice that the chunk size doesn't include the 4 byte ID or the ULONG for the chunk Size.

So, all IFF files are made up of several chunks (ie, groups of data). Each group of data starts with a convenient ID (so that a program can ascertain what kind of data is in the chunk) and a ULONG size (so that a program can ascertain how many bytes of data are in the chunk). There are a few other details to note. A chunk cannot have an odd number of data bytes (such as 3). If necessary, an extra zero byte must be written to make an even number of data bytes. The chunk Size doesn't include this extra byte. So for example, if you want to write 3 bytes in a CMAP chunk, it would look like this:

'CMAP'	
3	Note that chunk Size is 3.
0,1,33,0	Note that there is an extra zero byte.

The reason for this extra "pad byte" for odd-sized chunks has to do with Motorola's 68000 CPU requiring that LONGs be aligned to even memory addresses. IFF files were first used on 68000 based computers, and padding out odd-sized chunks made it easier to load and parse an IFF file on such a computer (ie, if you load the entire file into a single block of RAM starting upon an even address, all of the chunk IDs and Sizes will conveniently fall upon even memory addresses).

In the preceding example, the group ID was 'FORM'. There are 2 other group IDs as well. A 'CAT ' is a collection of many different FORMs all stuck together consecutively in 1 IFF file. For example, if you had an animation with 6 sound effects, you might save the animation frames in an ANIM FORM, and you might save the sound effects in several AIFF FORMs (one per sound effect). You could save the animation and sound in 7 separate files. The ANIM file would start this way:

FORM	
120000	Whatever the size happens to be (this is expressed in 32 bits).
ANIM	

Each AIFF file would start this way:

FORM	
8000	whatever size.
AIFF	

If the user wanted to copy the data to another disk, he would have to copy 7 files. On the other hand, you could save all the data in one CAT file.

CAT	
4+120008+8008+2028+...	The total size of the ANIM and the 6 AIFF files.
' '	Type of CAT. 4 spaces for the type ID means "a grab bag" of IFF FORMs are going to be inside of this CAT. If it just so happened that all of the enclosed FORMs were 1 type, such as ILBM, then this type ID would be 'ILBM'.

FORM

120000

ANIM	
	...all the chunks in the ANIM file placed here. (Note: ANIMs have imbedded ILBM FORMs. The guy who devised the ANIM type of IFF file broke the rules by mistake, and nobody caught his error until it was too late).
FORM	
8000	
AIFF	
	...all the chunks in the first sound effect here.
FORM	
2020	
AIFF	
	...all the chunks in the second sound effect here.
	...etc. for the other 4 sound effects.

To further complicate matters, there are LISTs. LISTs are a lot like CATs except that there is an optional, additional group ID associated with LISTs. That ID is a PROP. LISTs can have imbedded PROPS just like an ILBM can have an imbedded CMAP chunk. A PROP header looks very much like a FORM header in that you must follow it with a type ID. For example, here is an ILBM PROP with a CMAP in it.

PROP	Here's a PROP.
4+14	Here's how many bytes follow in the PROP.
ILBM	It's an ILBM PROP.
'CMAP'	Here's a CMAP chunk inside of this ILBM PROP.
6	There are 6 bytes following in this CMAP chunk.
0,0,0,1,1,4	

LISTs are meant to encompass similiar FORMs (i.e. several AIFF files stuck together). Often, when you have similiar FORMs stuck together, some of the chunks in the individual FORMs are the same. For example, assume that we have 2 AIFF sound effects. AIFF FORMs can have a NAME chunk which contains the ascii string that is the name of the sound effect. Also assume that both sounds are called "car crash". With a CAT, we end up having 2 identical NAME chunks in each AIFF FORM like so:

CAT	We put the 2 files into 1 CAT.

4+1008+508	
AIFF	It's a CAT of several AIFF FORMs.

FORM	Here's the start of the first sound effect file.
1000	
AIFF	
	...other chunks may be inserted here.
NAME	Here's the name chunk for the 1st sound effect.
9	
'car crash',0	
	...other chunks may be inserted here.

FORM	Here's the start of the 2nd sound effect file.
500	
AIFF	
	...other chunks may be inserted here.
NAME	Here's the name chunk for the 2nd sound effect. Look familiar?
9	
'car crash',0	
	...other chunks may be inserted here.

With a LIST, we can have PROPs. A PROP is group ID that allows us to place chunks that pertain to all the FORMs in the LIST. So, we can rip out the NAME chunks inside both AIFF FORMs and replace it with one NAME chunk inside of a PROP.

LIST	Notice that we use a LIST instead of a CAT.
4+30+990+490+...	
AIFF	

PROP	Here's where we put chunks intended for ALL the subsequent FORMS; inside a PROP.
22	
AIFF	Type of PROP.
NAME	Here's the name chunk inside of the PROP.
9	
'car crash',0	

FORM	Here's the start of the first sound effect file.
982	Size is 18 bytes less because no NAME chunk here.
AIFF	
	...other chunks may be inserted here, but no NAME chunk needed.

FORM	Here's the start of the 2nd sound effect file.
482	
AIFF	
	...other chunks may be inserted here, but no NAME needed for this guy either.

Notice that the PROP group ID is followed by a type ID (in this case AIFF). This means that the PROP only affects any AIFF FORMs. If you were to sneak in an SMUS FORM at the end, the NAME chunk would not apply to it. Also, if you included a NAME chunk in one of the AIFF FORMs, it would override the PROP. For example, assume that you have a LIST containing 10 AIFF FORMs. All but 1 of them is named "Snare Hit". You can store a NAME chunk in a PROP AIFF for "Snare Hit". Then, in the one AIFF FORM whose name is not "Snare Hit", you can include a NAME chunk to override the NAME chunk in the PROP.

It should be noted that you can take several LISTs and squash them together inside of a CAT or another LIST. Personally, I have never seen a data file with this level of nesting, and doubt that it would be of much use.

In the above examples, psuedo code was used to represent the headers. Let's look at how a hex file viewer might display the actual contents of an IFF file (in hex bytes). First, an IFF header for a FORM AIFF, psuedo code.

FORM
4096
AIFF

Now here's a view of the actual data file.

46 4F 52 4D	FORM
00 00 10 00	hex 00001000, or 4096 decimal
41 49 46 46	AIFF

Note that the ULONG byte count is stored in Big Endian order (ie, the Most Significant Byte is first, and the Least Significant Byte is last). This is how the Motorola 680x0 stores long values in memory (ie, the opposite order of Intel 80x86). IFF files use Big Endian order for all 16-bit (ie, SHORT) and 32-bit (ie, LONG) values.

Microsoft decided that IFF was a good idea, but since Windows is traditionally tethered to Intel CPUs, a version of IFF was needed which stored LONG or SHORT values in Little Endian order. So, MS decided to create some new group IDs. MS took the FORM ID and created a Little Endian version of it known as RIFF. For example, the WAVE file format has a RIFF group ID. All of the SHORT and LONG values in the file are stored in Little Endian order. Let's take a look at an example header for a WAVE file. Assume that there are 258 bytes of data after the byte count.

52 49 46 46	RIFF
02 01 00 00	hex 00000102, or 258 decimal
57 41 56 45	WAVE

Note that the ULONG byte count is stored in Little Endian order (ie, the Least Significant Byte is first, and the Most Significant Byte is last). Good old backwards-thinking Intel.

Now, there's some real justification for creating a RIFF group ID, if you're working with an Intel CPU. But Microsoft couldn't stop there. True to their "not made here, so if we're going to accept it, we have to inflict our brutish, unneeded brand upon it" mentality, Microsoft created another group ID called RIFX. What's an RIFX file? It's simply a FORM with RIFX replacing the FORM ID. So, if you want to turn a FORM AIFF into a RIFX AIFF, you just change the first 4 bytes to RIFX. Needless to say, nobody has ever used the RIFX group ID, and it will undoubtably suffer a justifiably ignoble disappearance.

Just like everyone else, programmers make mistakes. As mentioned before, the Amiga's ANIM file format was a mistake. It puts FORM headers inside of a FORM group ID. That's not supposed to happen. You can put FORM headers inside of a CAT or LIST, but not another FORM. A mistake was also made with the MIDI file format. The programmer who devised it didn't put a proper IFF header on the file. It should be:

FORM	group ID. Indicates an IFF file that contains one type of data.
3000	whatever size the file happens to be.
MIDI	type of data. What follows will be chunks as defined by the MIDI type of IFF file.

But the programmer omitted the FORM group ID, and simply put the MThd chunk first. So, a MIDI file starts as so:

MThd	Chunk ID.
6	size of MThd chunk.

Another deviation from the standard occurs with padding out odd-sized chunks with an extra byte. Some programmers didn't bother doing this when devising new IFF type files, and occasionally, one will come across some specification for a new IFF type that allows odd-sized chunks.

Unfortunately, these programmers released their work based upon these aberrations before getting that work reviewed by other programmers who might have offered good reasons why the aberrations should be corrected. It makes it that much harder for software to read and write files if it has to deal with aberrations of the IFF standard. There's no reason for that, particularly when a strict adherence to the standard sacrifices almost nothing in the way of quality and efficiency over an aberration. But try to tell that to a paranoid programmer who thinks that if he shows anyone what he's doing before his product is shrink-wrapped, someone will steal his soul... well, IFF does give the computer industry a means for resolving needless hassles with data file formats, and it has worked very successfully in a number of instances, although occasionally people don't always use the standard wisely, or don't quite grasp EA's altruistic notion that there is no good reason why a file format should ever be proprietary or unpublished. (I urge consumers to avoid products where that is the case).