Laser-Scan Ltd.

The Internal Feature Format Library

IFFLIB

Reference Manual

Issue 12.4

Science Park, Milton Road, Cambridge, England CB4 4FY tel: (0223) 420414

Document "IFFLIB Reference"		Category "REFERENCE"	
Original version of PDP11 IFFILE.DOC		Phil Cross	1976
Issue 6	MH & EM entries	Paul Hardy	April 1980
Issue 7	Junction routines and FS entry	Martin Fulford	August 1980
Issue 8	Revised VAX version	Tony J Ibbs	October 1983
Issue 9	New routines (IFFINQ,IFFRLA,IFFLSH)	Paul Hardy	January 1984
Issue 9.1	Revision of AC definitions	Dave Meader	February 1984
Issue 9.2	IFF entry order as Appendix	R. J. Hulme	June 1984
Issue 10	Renamed IFFLIB.RNO, document tidied	Paul Hardy	09-Jul-1984
Issue 10.1	New routines (IFFOPI,IFFID)	Clarke Brunt	29-Oct-1984
Issue 10.2	New routines (IFFCDL,IFFCRN)	Paul Hardy	29-Nov-1984
Issue 10.4	IFFVO withdrawn, IFFZS,IFFHI added	Clarke Brunt	06-Nov-1985
Issue 10.5	IFFUNO, EIHPHI added	Clarke Brunt	10-Jan-1986
Issue 11.0	Junction creation and manipulation	Dave Meader	20-Aug-1986
Issue 11.1	IFFTS, change to EIHPHI, IFFVER IFFMAP, mapped IFF files	Clarke Brunt	24-Oct-1986
Issue 11.2	IS_REAL_AC added	Tim Hartnall	19-Feb-1987
Issue 11.3	DECNET added (file id, mapped files)	Clarke Brunt	30-Mar-1987
Issue 11.4	ACTRTN argument to IFFERM	Clarke Brunt	10-Aug-1987
Issue 11.5	PARS error removed	Clarke Brunt	11-Sep-1987
Issue 12.0	Revision Level and Coordinate Block	Clarke Brunt	5-Jan-1988
Issue 12.1	Modify template IFF program	Tim Adams	13-Sep-1989
Issue 12.2	Allows up to 256 files	Clarke Brunt	12-Dec-1989
Issue 12.3	OS map header defined in MHOSGB.CMN	S Townrow	25-Oct-1990
Issue 12.4	MD entry as defined in MD2DES.CMN	S Townrow	26-Jan-1996

CONTENTS

1 *Introduction*

1.1 *History of IFF*

The Internal Feature Format (IFF) file structure and its associated interface library IFFLIB was written at Laser-Scan in 1975/76 to be used as a compact efficient means of storing graphical data (pictures) in digital form. The initial implementation was for PDP11s running RSX11M. Its primary use was intended to be storage of cartographic data (maps), originally generated from the Laseraid automatic line-following digitiser program. Its design was influenced (slightly) by the need for compatibility with the existing Ferranti MPS magtape format in use at that time at the initial customer site.

The format has been extended several times over the intervening years to cope with the increasing amount of non-graphical attribute information required to describe digital maps. It has also been transferred to VAX11 computers under VMS, retaining file compatibility with the PDP11 version.

1.2 *Characteristics of the file*

An IFF file is a compact means of storing feature oriented data. This data may originate with a digitising program (LAJ, etc), via translation from some external format (O2I, M2I, etc), or by combination of other IFF files (MER, ICE, etc).

At most times, the file is written and read sequentially, but it must be capable of random access when necessary (e.g. for re-ordering points in situ, or for altering details which can be changed without completely redigitising the feature).

1.3 *Notation and conventions*

1.3.1 *FORTRAN subroutine calls and arguments*

Optional arguments are surrounded by square brackets ([and]). Trailing optional arguments may be omitted completely.

An argument having the same name and meaning for a number of different subroutines is not repeatedly described.

1.3.2 *Integers*

Due to its origin on the PDP11, the IFF library tends to assume word storage (i.e. INTEGER*2). To ensure clarity, all variables below are explicitly declared of the required data type.

1.4 *Some definitions*

Certain terms have special meanings in this description and will be introduced here.

1. *Entry* - The file consists of a series of 'entries' end-to-end (a void is a possible entry). There is a class of calls for which an entry is the atom of data transfer.

2. *Feature* - The first unit higher than the entry level is the Feature. A feature is usually a graphical entity such as a building or text on a map. It includes identification and descriptive information as well as graphical detail.

3. *Overlay or layer* - Features can be grouped into overlays to give separation of different classes of feature. Utility programs can then be used to separate and merge overlays from different files.

4. *Section* - The file can be divided into 'sections', each carrying its own calibration data. The principal purpose of sectioning is to provide restart points against the possibility of system failure or other causes of interruption.

   Each section contains a whole number of overlays.

5. *Map* - A file may contain one or more maps, each having its own identification and projection information. Starting a new map implies starting a new section.

1.5 *Levels of access*

In the description which follows, the subroutine calls by which the IFF is manipulated are divided into 'first' and 'second' levels, with a fairly watertight bulkhead between. The first level is implemented mostly in MACRO and allows access to the file in basic units of 16 bit words. The second is mostly in FORTRAN, and imposes the variable length entry structure which is used by most IFF handling programs. Programmers writing IFF applications programs should not need to use the first level data transfer calls, though they will need those for open/close/select etc.

2 *Common Blocks*

2.1 *IFFHAN - General User Interface*

The values in common IFFHAN are those for the currently selected file, and are altered as appropriate when another file is selected using IFFSEL.

```
      COMMON /IFFHAN/ LPOSE, LPOSR, LPOSW, LPOSH, LPOSM, LPOSF,
     &                IERCD, IERAD, IERNM, WATEOF,IFOREL,IRVLEV,ORVLEV
```

```
integer*4 LPOSE - current EOF word  number  (next  free  word,  i.e.  last
                  written word plus one)
integer*4 LPOSR - current  reading  position  (next  word  to   be   read,
                  initially one)
integer*4 LPOSW - current writing  position  (next  word  to  be  written,
                  initially set to LPOSE)
integer*4 LPOSH - position of entry-in-hand (zero if none) - ref 4.2
integer*4 LPOSM - position of marked entry (zero if none) - ref 4.2.4
integer*4 LPOSF - position of start of current feature (zero if none)

                  Of these 'positional' variables, only  LPOSR  and  LPOSW
                  may  be  regarded as writable by the user. Note that any
                  such alteration will be lost over a second-level routine
                  call.

integer*4 IERCD - latest error code (zero if no error, otherwise  2  to  4
                  ASCII character codes)
integer*4 IERNM - VMS numerical error code (for non-internal errors)  Thus
                  set  to 1 (normal successful completion) if no VMS error
                  has occurred. IERCD will always be set  in  addition  if
                  IERNM contains an error.
```

The next three variables are of no concern to the user, and are included here merely for completeness

```
integer*4 IERAD  - locates latest call to IFF  error  handler  (unused  on
                   VAX)
logical*2 WATEOF - TRUE if writing at EOF
integer*2 IFOREL - the number of virtual blocks in the forepart of the IFF
                   file
```

Finally, two variables containing the input and output revision levels of the currently selected file. These should be considered read only - routines IFFIRV and IFFORV are used to set the revision level.

```
integer*4 IRVLEV - input revision level (will be -1, 0, or 1)
integer*4 ORVLEV - output revision level (will be 0 or 1)
```

2.2 *IFFJUN - Junction Structure Interface*

```
      COMMON /IFFJUN/ IFPOSJ, IFSHSZ, IFSHAD, IFSORI, IFSSTP, IFSNUM
```

```
integer*2 IFPOSJ    - position of junction-in-hand (zero if  none)  -  ref
                      4.5
integer*2 IFSHSZ    - number of sectors in header (zero if  no  header)  -
                      ref 4.5
integer*4 IFSHAD    - file address of sector header
real      IFSORI(2) - origin (bottom left-hand corner) of sectors
real      IFSSTP(2) - size of each sector
integer*2 IFSNUM(2) - number of sectors in each axis
```

All of these values should be considered 'read-only' by the user.

3 *First-level calls*

3.1 *Creation, opening, selection, etc*

3.1.1 *Create file/open existing file*

CALL IFFOPN(ILUN, FILNAM, [INILEN], [EXTLEN], [FUNC], [USR])

integer*2 ILUN - unit number (LUN) for accessing the file. This is used to identify which IFF file is to be used (via IFFSEL) when more than one is open. It should be in the range 0-255. Up to 256 files may be accessed simultaneously.

character*(*) FILNAM - name of file to be used, for instance 'SYS$SYSDISK:[FRED.IFF]LAST.IFF;2' The default filename is 'LSL$IF:IFF.IFF' (see below)

integer*4 INILEN - initial length of file in words (if to be created). Default is 100 blocks (of 256 words). If INILEN is explicitly zero, the file will be opened as read-only (and must already exist).

integer*4 EXTLEN - number of words by which the file is to be extended when necessary. Defaults to the value recorded in the forepart of the file by a previous IFFOPN, which in turn defaults to 50 blocks.

character*(*) FUNC - (VAX systems only) string to be entered into the history record describing the action to be performed. Defaults to 'Create', 'Update', or 'Read' as appropriate if not given. (See 4.4 for more details).

integer*4 function USR - (VAX systems only) If supplied, and non-zero, the function is called before $OPEN, and is passed a FAB (with associated NAM) as argument. The function should return a system error code. If this indicates an error, IFF error 'USER' will occur, and the open will be aborted. IFFLIB contains a function IF$MND which can be used here, which sets the end of file to the last allocated block. If this argument is present, IFFLIB will allow files which have been improperly closed to be opened. A null routine may be used just to allow opening of the file. This argument should normally be omitted.

The initial length and extension will be non-contiguous unless explicit and negative (PDP versions only).

The default action is to create a new file. To open an existing file for updating, an explicit version number must be given. The action is then to open the existing file if possible, otherwise create a new file. Explicit version 0 ('find latest') either opens an existing file or fails. Opening a file also 'selects' it, since there is provision for processing more than one file simultaneously.

In summary:

* no version number - create & open a new file
* explicit version 0 - open latest version
* explicit version 'n' - open version 'n'

Note that VAX versions of IFFLIB will not allow a file which was previously open for write and was not properly closed to be opened, unless the USR argument to IFFOPN is supplied. The utility program IMEND should be used under these circumstances, after which normal IFFLIB programs will be able to access the file again.

IFFOPN may be called as an INTEGER*4 FUNCTION (VAX versions) returning the number of blocks allocated to the file. This is only needed for certain specialised applications.

3.1.2 *Open existing file by file identification*

CALL IFFOPI(ILUN, FILNAM, FID, [INILEN], [EXTLEN], [FUNC])

ILUN, INILEN, EXTLEN, and FUNC are the same as for IFFOPN. The file will be opened for updating unless INILEN is explicitly zero. If FUNC is present, then a new history record will be created, otherwise the previous record will be updated, adding the statistics for this opening of the file (See 4.4). Programs which repeatedly open the same IFF file should normally use IFFOPI for subsequent opening, as it is faster than IFFOPN.

character*(*) FILNAM - name of file. The filename given is not used in opening the file, but will be returned by future calls to IFFINQ. It is used by IFFCDL and IFFCRN and if calls to these are to be used, then it *must* be a full filename resulting from a previous call to IFFINQ. N.B. If the file is on another DECNET node, then file identification is not supported, and the name is used to open the file instead.

integer*4 FID(7) - file identification

FID should normally be obtained from a call to IFFID after a previous IFFOPN. It consists of the device identification, file identification, and directory identification fields. If the file is on another DECNET node, then all these fields will be zero, and the filename is used to open the file.

3.1.3 *Selecting a file (when several are open)*

CALL IFFSEL(ILUN)

The selection thus obtained persists until the next IFFSEL or IFFOPN.

3.1.4 *Inquire file attributes*

CALL IFFINQ(FILENAME,FLEN,CREATED,REVISED,NREV)

Returns information about the currently selected file (VAX systems only). All the arguments are optional, and may be omitted if the information is not required.

FILENAME is a character variable which is set to the full filename.

FLEN is an integer*2 variable which is set to the number of characters written to FILENAME.

CREATED is the creation date of the file as a VAX quadword absolute time.

REVISED is the revision date of the file as a VAX quadword absolute time.

NREV is an integer*2 variable to contain the number of times the file has been modified (revised).

3.1.5 *Display file attributes*

CALL IFFVER([PRTN])

external PRTN - a routine which is called to print the information. It is called by CALL PRTN(STRING) where STRING should be declared as CHARACTER*(*). If omitted, then LIB$PUT_OUTPUT is used.

Displays information about the currently selected file (VAX systems only). Calls IFFINQ (q.v.), then formats and displays the results.

3.1.6 *Inquire file identification*

CALL IFFID(FID)

Returns the device and file identification of the currently selected file, for use in a subsequent call to IFFOPI (VAX systems only). For files on another DECNET node, file identification is not supported, and the returned information will be zero.

integer*4 FID(7) - array for device, file, and directory identification

3.1.7 *Read Look Ahead optimisation*

CALL IFFRLA(ONOFF)

Logical ONOFF

Calling this routine turns on/off Read Look Ahead on the currently selected file. If RLA is turned on (ONOFF=.TRUE.) then the next IFF block will be asynchronously read into a buffer, whenever a new block is accessed. This means that file access will be quicker for sequential access to the file (VAX systems only).

3.1.8 *Input revision level*

CALL IFFIRV(LEVEL)

integer*4 LEVEL - input revision level (-1, 0, or 1)

Set the input revision level for the currently selected file. IFF error 'ARGS' will occur if LEVEL is not 0, 1, or -1. If IFFIRV is not called, the level defaults to 0. IFFIRV must be called *after* the file is opened and repeated calls may be made to alter the input revision level as required. The current input revision level appears in variable IRVLEV in common IFFHAN and is preserved on a per-file basis.

The input revision level of a file controls whether existing entries appear as CBs or ST/ZSs to calling programs. The possible values are: 0 for ST/ZS entries (old style programs), 1 for CB entries (new style programs), or -1 to treat all entries exactly as they are in the file (for debugging, or specialised programs).

3.1.9 *Output revision level*

CALL IFFORV(LEVEL)

integer*4 LEVEL - output revision level (0 or 1)

Set the output revision level for the file created in the next call to IFFOPN. IFF error 'ARGS' will occur if level is not 0 or 1. IFFORV must be called before the call to IFFOPN which creates the file. The call to IFFORV is a one-shot operation, in that it only applies to the next call to IFFOPN. In the absence of a call to IFFORV, the output revision level is taken from the translation of logical name LSL$IFF_OUTPUT_REVISION which must be a single character "0" or "1". If the logical name does not exist, a default of 0 is used. The intention is that, except for testing purposes, programs should allow the logical name to control output revision level. The output revision level of the currently selected file appears in variable ORVLEV in common IFFHAN. Output revision level is stored in the IFF file, so that if a file is subsequently re-opened for update, its original output revision level will be preserved.

The output revision level of a file controls whether new entries added to the file will be CBs or ST/ZSs. The possible values are 0 for old style files containing ST/ZS entries, and 1 for new style files containing CB entries. It should not be possible for a file to contain a mixture of ST/ZS entries and CB entries.

3.1.10 *Memory mapped file option*

CALL IFFMAP(ONOFF)

Logical ONOFF

Calling this routine turns on/off the memory mapped file option for files opened after the call. The default is off. See section 4.6 for details of using memory mapped files. (VAX systems only). The use of mapped files is not supported across the DECNET network.

3.1.11 *Flush Buffers*

CALL IFFLSH

Calling this routine causes any buffers containing modified data belonging to the currently selected IFF file to be written back to disc. It may be called e.g. at end of feature to ensure that data is not lost if the system fails (VAX systems only).

Note that this call is only needed for programs which need to survive system failure, and should not be called in normal processing programs which can easily be restarted as it will cause extra loading due to redundant buffer writes.

3.1.12 *Closing files*

CALL IFFCLO(ILUN, [STATUS], [HIST])

integer*4 STATUS - status code to be written to history record. Default is 1 (success).

byte HIST(80) - array in which to return the final history record (which will also be written to the HI entry of the file if possible). (See 4.4).

The relevant IFF file is closed. Selection becomes null, and another file must be explicitly opened or selected. If attributes have been lost during the writing of the file, due to calls to IFFCB or EIHCPY with attributes other than X,Y,Z when the output revision level is 0, then IFF error 'LOST' will occur and a message will be output. Note that the file is still properly closed, and therefore the appearance of 'LOST' in IERCD should not be treated as a fatal close error.

3.1.13 *Deleting files*

CALL IFFCDL(ILUN, [STATUS], [HIST])

The specified IFF file is closed and deleted. If the file was opened for reading, then it is just closed. Selection becomes null, and another file must be explicitly opened or selected.

3.1.14 *Renaming files*

CALL IFFCRN(ILUN, FILNAM, [STATUS], [HIST])

character*(*) FILNAM - new name of file, for instance 'DRA2:[FRED.IFF]LAST.IFF;2'. Missing parts of the filename will be taken from the original filename.

The specified IFF file is closed and renamed to the given filename. If the file was opened for read, it is just closed. If the rename operation fails (error 'RENA') the file still have been closed. Selection becomes null, and another file must be explicitly opened or selected.

3.2 *Data transfer*

Although the following first level routines for reading and writing files are documented here for completeness, it should be noted that they are not required for normal use of IFF files. All normal IFF utility programs read and write data in terms of entries using the second level calls described later.

3.2.1 *Writing*

CALL IFFW(SRC, [NWDS], [POS])

```
array      SRC   - variable/array to be written to that file.
integer*2 NWDS  - number of words to be written (default 1)
integer*4 POS   - position in the file at which writing is to start (defaults to
                  LPOSW, which is advanced by NWDS words. If LPOSW is then
                  beyond LPOSE, that too is advanced).
```

Similarly:

```
CALL IFFWR( RSRC, [NREALS], [POS] )     write real(s) from RSRC
CALL IFFWI( ISRC, [NINTS], [POS] )      write integer*2(s) from ISRC
CALL IFFWL( LSRC, [NLONG], [POS] )      write integer*4(s) from LSRC
```

Also: CALL IFFWB(BSRC, [NBYTES], [POS])

write byte(s)/character(s) from BSRC (which may be an odd address). If NBYTES is odd a null byte is appended This is not a completely general mechanism for storing bytes, being intended for character strings for which a trailing null is a terminator and can be discarded.

3.2.2 *Reading*

```
CALL IFFR( DST, [NWDS], [POS] )         Read word(s) from the file to DST
CALL IFFRR( RDST, [NREALS], [POS] )     Read real(s)
CALL IFFRI( IDST, [NINTS], [POS] )      Read integer*2(s)
CALL IFFRL( LDST, [NLONG], [POS] )      Read integer*4(s)
```

Arguments and action analogous to those for writing, except that POS defaults to LPOSR.

3.2.3 *Interlocks*

The file handler will ensure that writing something and then reading it back will give the right answer, but there is clearly some onus on higher level routines to behave prudently if simultaneously reading and writing.

4 *Second-level Calls*

At this level, the atom of data transfer is the 'entry', which will be one of several kinds indicated (here and elsewhere) by its 'entry code' of two letters. The structure currently employed to represent entries is implicitly described in 4.1, but programs should not assume structure (and make first-level calls) for any purpose which can be served by making a second-level call or calls. This will avoid at least some bugs due to misunderstanding of structure and preserve first-level flexibility. Where necessary, the provision of additional second-level calls should be requested.

4.1 *Making new entries*

These calls usually make a new entry at the end of the file, and leave LPOSE, LPOSW at the new EOF. Exceptionally, they can also be used to rewrite compressed and/or re-ordered data at a 'marked' void in the file (see 4.2, especially 4.2.4 & 4.2.5).

4.1.1 *Entries necessarily related to a feature*

4.1.1.1 *New Feature (entry code NF)*

CALL IFFNF(INF, ISQ) starts a new feature and makes it 'current'. Any feature already current will be terminated.

integer*2 INF - external feature number, the normal FSN or NF (the NF number of MPS)

integer*2 ISQ - internal sequence number

NOTE that if ISQ=0, the next available number (starting at 1) is generated and ISQ is set to it. Normally the user should be consistent, i.e. EITHER set ISQ=0 every time OR supply their own every time (and ensure that they are unique). Note also that it is not safe to call IFFNF with a constant ISQ, as in:

```
          CALL IFFNF( INF, 0 )
```

as an attempt will be made to overwrite the constant! At best this will cause the program to exit, at worst it will corrupt the constant.

4.1.1.2 *Feature Status (entry code FS)*

CALL IFFFS(ISTAT)

integer*2 ISTAT(4)

ISTAT(1) is interpreted in one of two ways, depending on the degree of attribute coding required.

1. Interpolation Type - a direct indication of how the feature is to be drawn:
```
     IT0  - straight lines
     IT1  - single symbol at each point
     IT5  - cubic interpolation
     IT64 - text feature
```

2. Feature Code (FC) - the value is looked up, either in a Legenda file, or an FRT file to yield a Graphical Type for the code, along with details of line style, symbol definitions, colour, line thickness and so on (see LEGLIB, CTG & SOL documentation). Feature code is sometimes referred to as Graphical code (GC).

ISTAT(2) contains status bits defined currently as follows:
```
        1 bit  1,0    =>closed, open feature
        1 bit  2,0    =>line, edge feature
        1 bit  4,0    =>reversed, normal feature (e.g. anticlockwise closed
                        feature)
        1 bit  8,0    =>feature does, does not need re-ordering (two-part
                        feature)
        1 bit  16,0   =>discard, retain at reprocessing ("paintout-only"
                        feature)
        1 bit  32,0   =>squaring flag set, clear
        1 bit  64,0   =>inverse, normal polarity feature
        1 bit  32768,0=>paint out suppressed, not suppressed during digitising
                        (sign bit)
```

ISTAT(3) contains flags, interpreted according to the value of top two bits:
```
        Bits    Values  Meaning

        d14-15  0       this is a line, circle, area or symbol string feature
                1       this is a symbol feature
                2       this is a text feature
                3       value reserved
        For text features, the rest of the word is interpreted as follows:
        d0-3    0-8     text position code
        d4-5    0-3     type style    (O.S. only)
        d6-11   0-63    name category (O.S. only)
        For other feature types, the rest of the word contains the process
        code.
```

ISTAT(4) contains user dependent data about the feature.

4.1.1.3 *Text Status (entry code TS)*

CALL IFFTS(ISTAT)

integer*2 ISTAT(4)

ISTAT is as for an FS entry, but ISTAT(2) is unused at present.

4.1.1.4 *Ancillary Codes (entry code AC)*

```
CALL IFFAC( ACTYPE, LCODE [,TEXT] [,LENGTH] )
```

integer*2 ACTYPE - type of AC being constructed.

integer*4 LCODE - AC value field (but see below)

byte TEXT(*) - optional text array, null terminated

integer*2 LENGTH - optional length of text array - overrides null

AC types in the range 0 - 100 are reserved for allocation for specific tasks by Laser-Scan. Current allocation of ACs within this range is defined in the IFF User Guide.

AC types in the range 101 - 32767 are allocated (in blocks of 20) for customer AC definition.

The longword field of an AC is usually interpreted as an integer value. However, some AC types in the range 0 - 100 have their longword field interpreted as a real (or "floating point") value. Currently type 3 and type 80 - 99 ACs are interpreted as having a real value longword field.

A function is supplied to enable the user to determine how the longword field is to be interpreted:

```
IS_REAL = IS_REAL_AC( ACTYPE )
```

logical IS_REAL - returns .TRUE. if real value in longword.

integer*2 ACTYPE - type of AC being decoded.

4.1.1.5 *Feature Thickness (entry code TH)*

```
CALL IFFTH( ITHK )
```

integer*2 ITHK - conventionally thickness in microns on the film when digitising. Used for line thickness, or text size.

4.1.1.6 *Coordinate String (entry code ST)*

```
CALL IFFST( STBUF, NPTS, IENDS)
```

real STBUF(2,NPTS) - contains points of the string in order

integer*2 NPTS - number of points. Conventionally never more than 200.

integer*2 IENDS - coded bitwise:

```
                       0,1=>move to 1st point is pen-up, pen-down
                       +2 =>first point is an edge point (not implemented)
                       +4 =>last point is an edge point (not implemented)
                            (all combinations are legal except 3,7)
```

4.1.1.7 *3-Dimensional Coordinate String (entry code ZS)*

CALL IFFZS(STBUF, NPTS, IENDS)

real STBUF(3,NPTS) - contains points of the string in order

integer*2 NPTS - number of points. Conventionally never more than 200.

integer*2 IENDS - coded bitwise as for ST entry

4.1.1.8 *Coordinate Block (entry code CB)*

CALL IFFCB(CBH, CBD)

record /IFF_CBH/ CBH - CB header record

record /IFF_CBD/ CBD - CB data record or e.g.
real*4 CBD(cols,rows)

See the section on Coordinate Blocks and Revision Levels for details of the arguments to IFFCB.

4.1.1.9 *Text Rotation (entry code RO)*

CALL IFFRO(ROT)

real ROT - alignment angle for a text string or symbol (in radians).

4.1.1.10 *Text String (entry code TX)*

CALL IFFTX(STR, [NCH])

byte STR(NCH) - array containing the characters of the text

integer*2 NCH - number of characters (if absent, a null terminates the string)

4.1.1.11 *Junction Pointer (entry code JP)*

CALL IFFJP(LPOSJB, IPOSJ)

integer*4 LPOSJB - address of the junction block (JB) entry containing the junction

integer*2 IPOSJ - offset of the junction within the JB entry

A JP entry is created to point to a junction within a junction block (JB). In practice, IFFJP is often called with its arguments set to zero, as the address and offset cannot be filled in until the junction is created (or until the coordinate data have been 'snapped' to an existing one). Junction creation and manipulation is discussed in section 4.5.

4.1.1.12 *End of Feature (entry code EF)*

CALL IFFEF

The current feature is terminated and 4.1.1 calls (except IFFNF) become illegal until a new feature is started. IFFEF is called automatically if IFFNF, IFFEO, IFFEM, or IFFEJ are called while a feature is open, though relying on this is not recommended. See also 'end of overlay' (4.1.2.7) and 'end of job' (4.1.2.8).

4.1.2 *Entries not related to a feature*

(Though in some cases there may be an association if a feature is current when the call is issued).

4.1.2.1 *New Section (entry code NS)*

CALL IFFNS(SDATA, [NCH])

byte SDATA(NCH) - text array including date, time operator, etc.

integer*2 NCH - length of text array (as for TX, see 4.1.1.9)

4.1.2.2 *New Overlay (entry code NO)*

CALL IFFNO(IOVN, IOVS, [EOPTR])

integer*2 IOVN - overlay number

integer*2 IOVS - overlay status, not currently used - should be zero.

integer*4 EOPTR - pointer to the corresponding EO

This entry indicates that all features following, until further notice belong to overlay IOVN. The EOPTR entry allows fast chaining through the file when searching for a specific layer, and should normally be supplied. The normal way of generating EOPTR involves writing an initial dummy value when the NO entry is first written, and recording the position of the NO entry by remembering LPOSE *before* calling IFFNO. Later when the corresponding EO entry is about to be written, the file can be repositioned to the NO entry using IFFPKE on the saved position, and the EOPTR field rewritten using EIHWL to the current value of LPOSE at which the EO entry is about to be written.

A routine is supplied to perform this commonly used operation:

CALL IFFUNO(LPOSNO) (update NO)

integer*4 LPOSNO - IFF address of the NO to be updated

The routine uses IFFPKE to position to LPOSNO. The current value of LPOSE is then written into the entry (provided that it has space for it). An error is given if the entry at LPOSNO is not an NO. The normal sequence of code should thus be:

```
        ...
        LPOSNO = LPOSE                  ! remember where NO will go
        CALL IFFNO(IOVN,IOVS,0)         ! supply zero EO pointer for now
        ...
        ... fill in layer
        ...
        CALL IFFUNO(LPOSNO)             ! update NO (insert LPOSE in it)
        CALL IFFEO                      ! and write the EO
        ...
```

4.1.2.3 *Calibration Coefficients (entry code CC)*

CALL IFFCC(CFT)

real CFT(10,2) - 20 cubic coefficients in standard LSL order.

These define a transformation between two coordinate systems to be applied by a transformation program (e.g. IPR to remove digitiser distortions).

```
if  X' = a + bX + cY + dXX + eXY + fYY + gXXX + hXXY + iXYY + jYYY
and Y' = k + lX + mY + nXX + oXY + pYY + qXXX + rXXY + sXYY + tYYY
then the matrix would be:-
                                a       k
                                b       l
                                c       m
                                d       n
                                e       o
                                f       p
                                g       q
                                h       r
                                i       s
                                j       t
```

A unit matrix (no transformation) has all terms zero except b and m which are 1.

4.1.2.4 *Corner Points (entry code CP)*

CALL IFFCP(XY)

real XY (4,4) - coordinates of each corner in both input space and output space. 1st suffix. Xin, Yin, Xout, Yout. 2nd suffix: corner number in standard order (NW,SW,SE,NE).

Normal use is to apply a rotation and scaling to the coordinate data. E.g. if IPR is to be used on a file, the 'in' fields describe the 'current' data, and the 'out' fields define the final state required.

Thus the default state, no transformation, has 'in' and 'out' fields equal.
for instance:

corner	Xin	Yin	Xout	Yout
NW	0.0	1000.0	0.0	1000.0
SW	0.0	0.0	0.0	0.0
SE	1000.0	0.0	1000.0	0.0
NE	1000.0	1000.0	1000.0	1000.0

4.1.2.5 *Transmitted Comment (entry code TC)*

CALL IFFTC(STR, [NCH])

Arguments as for IFFTX (see 4.1.1.9). A TC entry is usually associated with the feature immediately following. AC entries are the preferred way of including additional information associated with features.

4.1.2.6 *Literal Character Data (entry code CH)*

CALL IFFCH(STR, [NCH])

Arguments as for TX (see 4.1.1.9). The characters in this case will be copied as they stand; hence this is a mechanism for sending plot commands not otherwise catered for.

4.1.2.7 *End of Overlay (entry code EO)*

CALL IFFEO

No arguments. Ends the current 'overlay'. See IFFNO for information on the NO pointer to corresponding EO (EOPTR)

4.1.2.8 *End of Job (entry code EJ)*

CALL IFFEJ

No arguments. Declares the 'job' complete. Should normally be output at the very end of the IFF file.

4.1.2.9 *Symbol Select (entry code SS)*

CALL IFFSS(ISSN)

integer*2 ISSN - numeric code of selected symbol.

*** This entry is no longer used.

4.1.2.10 *Symbol Library Select (entry code SL)*

CALL IFFSL(ISLN)

integer*2 ISLN - numeric code of selected symbol library (plotter disc or whatever).

*** This entry is no longer used.

4.1.2.11 *Range of Coordinates (entry code RA)*

CALL IFFRA(RXY)

real RXY(4) - minimum and maximum coordinates to be expected, in the order Xmin Xmax Ymin Ymax. This entry should be the first in the file.

4.1.2.12 *Character Size (entry code CS)*

CALL IFFCS(CH, CX)

integer*2 CH - character height
integer*2 CX - character spacing

*** This entry is no longer used.

4.1.2.13 *Map Header (entry code MH)*

CALL IFFMH(MHDR, NWDS)

integer*2 MHDR(NWDS) - contains user-specific data about the following map.

integer*2 NWDS - the length of MHDR in words

This entry specifies a new map comprising all entries up to and including the next EM entry.

Standard map headers are defined in LSL$CMNIFF:

- MHDEF.CMN defines a default map header
- MHDMB.CMN defines the type 2 OS map header (documented within the common file)
- MHMCE.CMN defines the MCE map header (documented within the file)
- MHOSGB.CMN defines the type 3 and 4 OS map header (documented within the common file)

Note that each map header (with the exception of MHOSGB.CMN) starts with a 4-byte descriptor of the form:

	length,	customer,	zero,	zero	
e.g.	174	2	0	0	for OS

where 'length' is the length of the map header in longwords, and 'customer' is currently of value

- 0 for files with an empty mapheader
- 1 for MCE files with a meaningful map header
- 2 for OS files

The 4-byte descriptor of the map header defined in MHOSGB.CMN is slightly different and looks like:

	length,	OS header type,	zero,	zero
e.g.	0	3 or 4	0	0

The size of this map header has been expanded to hold a potential 5000 bytes needed to accomodate new header formats as they are revised. In practice, 5000 bytes are not written to the IFF MH entry but only as much as is required by the header which is dictated by the size in the translation table, LSL$OS_MH_TABLE, as described in the DATA PREPARATION section of the IFFOSTF chapter of the "Convert User Guide".

As the header could be potentially 5000 bytes, its size cannot be stored in the one byte allocated it in the 4-byte descriptor and as consistancy is desired, its size is not stored. Application programs that access type 3 or 4 map headers should use entry-in-hand routines to read the MH entry and return the actual size of the header.

The OS map header types are 3 for OSTF and 4 for CITF.

4.1.2.14 *Map Descriptor (entry code MD)*

CALL IFFMD(MDESC, NWDS)

integer*2 MDESC(NWDS) - variable length array, containing data concerning the map projection and origin. It is only used when performing transformations between standard coordinate systems (by IPR)

integer*2 NWDS - length of MDESC in words

IPR expects the map descriptor to be in the format defined in LSL$CMNIFF:MAPDES.CMN.

By convention, if the map descriptor is unset or not used, then the first word of MDESC is set to -1.

This is called a type 1 Map Descriptor which is now considered obsolete and replaced by type 2 descriptors.

The new type 2 Map Descriptor entries are defined in a common block held in LSL$CMNIFF:MD2DES.CMN. Any application programs which read or write MD type 2 entries should inculde this and the array name should be used in the CALL IFFMD statement.

To use it effectively, the application programmer should study the file LSL$CMNIFF:MD2DES.CMN and see that it contains the array MD2ARR with numerous variables equivalenced onto it.

If the application program is writing a MD entry, these variables can be set before the CALL IFFMD statement is invoked and the EQUIVALENCE statement will mean that the correct field within the MD2ARR array will be set. If the application program is reading an MD entry, an EIHR (entry-in-hand-read) routine should be used to read the descriptor into MD2ARR and fields such as the local origin or scale can be examined as variables MD2LOC and MD2SCL etc. will be set automatically.

4.1.2.15 *End Map (entry code EM)*

CALL IFFEM

No arguments. Ends the current map (pairs with MH).

4.1.2.16 *History (entry code HI)*

CALL IFFHI

No arguments. Create a blank history entry in the file. This entry should be second in the file (after the RA). The entry will be filled in automatically by IFFLIB (see 4.4).

4.1.2.17 *Sector Header (entry code SH)*

CALL IFFSH(RXO, RYO, RXS, RYS, NX, NY)

```
real       RXO, RXO - origin (bottom left-hand corner) of sectored area
real       RXS, RYS - size of each sector
integer*2   NX, NY  - number of sectors in each axis
```

Create a sector header and update the corresponding variables in common IFFJUN. The SH entry (which is only necessary if the file is to contain junction data) should occur after the RA and HI entries. Junction creation and manipulation is discussed in section 4.5.

4.1.2.18 *Junction Block (entry code JB)*

CALL IFFJB(NWDS)

integer*2 NWDS - size of junction block in words

This call is only included for completeness, as JB creation is automatically performed by IFJCR (section 4.5.1). Users should not create JBs explicitly, as the necessary pointers will not then be set up.

4.2 *Finding and modifying existing entries*

These subroutines modify or operate on the entry-in-hand. It should be noted that none of the EIH calls may access material outside the entry-in-hand.

4.2.1 *Find next entry*

CALL IFFNXT(IECODE, IELEN)

integer*2 IECODE - entry code for the next IFF entry

integer*2 IELEN - entry length, in words

The next entry in the file is taken 'into hand' and its entry code returned in IECODE (2 letters in A2 format). The number of words of data in the entry is returned in IELEN. LPOSH is set appropriately, and LPOSR, LPOSW are set to the first word of data in the entry.

N.B. IELEN may be zero, and will be set to -1 if there are no more entries in the file (but note that IELEN is an unsigned integer and may therefore appear to be negative if an extraordinarily large entry in encountered).

There are three places where the above operation can start:

1. The current entry in hand (or start of file if none).

2. The start of the file, after opening it, or a call to IFFRWD (see 4.3.1)

3. A previously marked entry, by means of CALL IFFRWM (see 4.3.2).

4.2.2 *Find next entry of given code*

CALL IFFNXC(IECODE, IELEN)

As above, but skips over entries not matching the given IECODE. If nothing is found, IELEN is set to -1.

Example: CALL IFFNXC('NF', IELEN) finds the start of the next feature.

4.2.3 *Position to known entry*

CALL IFFPKE(IECODE, IELEN, POS)

integer*4 POS - position in IFF file

POS is assumed to be either a copy of LPOSH taken *after* a previous 'find', or a copy of LPOSE *before* writing an entry. The file is positioned to the corresponding entry, and IECODE, IELEN returned as for IFFNXT.

4.2.4 *Mark file at current entry/remove marker from file*

CALL EIHMK(N)

integer*2 N - such that

N=1 => 'Mark' the file at the entry-in-hand (error if none). Any previous marker is removed.
N=0 => Remove marker (if any).

The marked position can be exploited in two ways:

1. As a position from which to find further items (by CALL IFFRWM - see 4.3.2).

2. As a position from which to rewrite data by means of section 4.1 calls, after creating a 'void' (see 4.2.5)

4.2.5 *Create void at marked position*

CALL IFFVOM

No arguments. Replaces all entries between (inclusive) the marked entry and (exclusive) the current entry in hand with a single void entry. Once this has been done, until the marker is removed, section 4.1 calls will make entries into this void instead of making them at end of file. When an entry is thus made, the void is contracted appropriately and the marker shifted to the new start of it. There are some restrictions on this process:

1. The void is not permitted to cross feature or section boundaries.

2. Entries which would overflow the void are not permitted.

4.2.6 *Reading from the entry-in-hand*

CALL EIHR(DST, NWDS, IWNO)

integer*2 DST(NWDS) - destination array

integer*2 NWDS - number of words to read

integer*2 IWNO - start word number

Reads NWDS words of data from the entry-in-hand to DST, starting at word number IWNO in the entry. The first word of data in an entry is numbered 1.

Similarly:
CALL EIHRR(RDST, NREALS, IWNO) read real(s)
CALL EIHRI(IDST, NINTS, IWNO) read integer(s)
CALL EIHRL(LDST, NLONGS, IWNO) read longword integer(s)

CALL EIHRS(NPTS, IENDS) read point-string details

integer*2 NPTS - set to the number of points in the point-string (ST), or 3-dimensional point string (ZS), or to 0 if the entry is not of type ST or ZS.

integer*2 IENDS - see 4.1.1.6

The actual point string may then be read using EIHRR. Note that NREALS has a maximum sensible value of NPTS*2 for ST, or NPTS*3 for ZS.

4.2.7 *Writing to the entry-in-hand*

```
CALL EIHW(  SRC,   WORDS, IWNO )
CALL EIHWR( RSRC, NREALS, IWNO )
CALL EIHWI( ISRC, NINTS,  IWNO )
CALL EIHWL( LSRC, NLONGS, IWNO )
```

Behaviour analogous to the read entry-in-hand mechanisms, but overwriting the value of the entry-in-hand.

4.2.8 *Copying the entry-in-hand to another (open) file*

CALL EIHCPY(ILUN)

integer*2 ILUN - unit number of (open) destination file. The whole entry, including entry code etc, is copied to the other file (at EOF). The destination file must *not* be the same as the source - this may appear to work but can cause the program to hang.

NOTE that EIHCPY does not work correctly for NF (and thus EF) entries, as it does not set the internal 'in feature' flags. IFFNF and IFFEF should thus be used explicitly on the destination file. Also beware copying NO entries with EO pointers - unless a straight copy of the entire file is performed, the value of the EO pointer will be incorrect. The same is true of the junction entries SH, JP and JB, all of which contain pointers to other entries in the file.

4.2.9 *Junction-in-hand*

Each Junction Block (JB) entry will usually contain several junctions. The following calls enable the user to examine and manipulate these without requiring a detailed understanding of the junction block structure. Junction creation and manipulation is discussed in section 4.5.

4.2.9.1 *Find next junction*

CALL IFJNXT(RPOSX, RPOSY, NARMS)

real RPOSX,RPOSY - position of junction

integer*2 NARMS - number of arms

This routine assumes that the current entry-in-hand is a junction block (JB), typically located using IFFNXT. The next junction in the JB is taken 'into hand' and its position and number of arms is returned. If there are no more junctions in the current JB, the number of arms returned is -1.

4.2.9.2 *Position to known junction*

CALL IFJPKJ(RPOSX, RPOSY, NARMS, LPOS, IPOS)

real RPOSX,RPOSY - position of junction

integer*2 NARMS - number of arms

integer*4 LPOS - position of current entry (JB)

integer*2 IPOS - offset of known junction within JB

IFJPKJ is similar to IFJNXT, but moves directly to a particular junction (defined by IPOS) within a particular JB (defined by LPOS). The JB becomes the entry-in-hand. LPOS is assumed to be the position of a junction block entry (usually a copy of LPOSE or LPOSH in common IFFHAN which was taken at the appropriate time), while IPOS will be a copy of IFPOSJ in common IFFJUN taken *AFTER* the junction was created using IFJCR (section 4.5.1). The junction position and the number of arms will be returned as for IFJNXT. If the number of arms is -1, then either the junction has been deleted (see JIHDEL below) or IPOS points into an unset area of the JB.

4.2.9.3 *Deleting the junction-in-hand*

CALL JIHDEL

The whole of the junction-in-hand is deleted from the junction block. Note that no modification is made to any junction pointer (JP) entries which address the junction, or to any ST entries which constitute arms of that junction. IFJNXT will no longer find the junction, and an attempt to IFJPKJ to it will yield an arm count of -1.

4.2.9.4 *Reading junction arms*

CALL JIHR(NARM, LSTRP, NVERTX)

integer*2 NARM - number of required arm

integer*4 LSTRP - address of ST entry for arm NARM

integer*2 NVERTX - vertex number of junction position within ST

Having taken a junction 'into hand', JIHR and its complementary call JIHW allow the user to obtain or update information about a particular junction arm, NARM. LSTRP is the returned position of the ST entry corresponding to that arm, and NVERTX is the vertex number of the junction within that ST. NVERTX will either be 1 or the number of points in the ST (a junction may not occur in the middle of an ST).

4.2.9.5 *Updating junction arms*

CALL JIHW(NARM, LSTRP, NVERTX)

integer*2 NARM - number of required arm

integer*4 LSTRP - address of ST entry for arm NARM

integer*2 NVERTX - vertex number of junction position within ST

Having created a junction using IFJCR (section 4.5.1), JIHW may then be used to fill in the details of the arms (see section 4.5 for details of the junction structure). LSTRP should be the remembered position of the ST entry corresponding to NARM, and NVERTX should be the vertex number of the junction position coordinate within that ST (see also JIHR above). Often it will be the case that junctions are created with one or more arms 'unknown', and these are subsequently filled in as processing continues. Under these circumstances, LSTRP should be initialised to zero for the unknown arms, and NVERTX is then available for holding temporary information about those arms (e.g. Laseraid uses it to hold the arm direction in degrees).

4.3 *Miscellaneous operations*

This section is a compendium of calls mostly introduced elsewhere.

4.3.1 *Rewind to start of file*

CALL IFFRWD

The file is repositioned to its beginning. CALL IFFNXT will find the first entry.

4.3.2 *Rewind to marked entry*

CALL IFFRWM

The file is repositioned to the entry which was in hand at the last CALL EIHMK(1). This entry is again taken into hand.

4.3.3 *Delete current feature*

CALL IFFDEL

If a feature is currently being built (i.e. had NF but no EF), it is abandoned and the file repositioned to the start of it. 4.1.1. calls (except IFFNF) become illegal.

4.3.4 *Clear whole file*

CALL IFFCLR

The end of file is set back to the beginning so that the file becomes empty and can be reused. N.B. The length as recorded by the filing system (which appears on directory listings) is unchanged.

4.3.5 *Update private positions*

CALL IFFUPP(POSF, POSM, POSH, POSE)

The internal values for the file pointers for the current feature, marked position, entry in hand, and end of file are updated to the given longword values. The values in common IFFHAN are updated also. Trailing arguments may be omitted, in which case the corresponding values are unchanged. LPOSF, LPOSM, LPOSH, and LPOSE (in common IFFHAN) should *not* be used as arguments in positions other than 'their own'. This routine, which is used internally by IFFLIB, should only be used in exceptional circumstances by those with a detailed understanding of the working of IFFLIB.

4.4 *History entry*

VAX versions of IFFLIB provide a mechanism for automatically recording statistics in an IFF file each time it is updated, so that it may be determined which users and programs contributed to the final state of the file.

The information is stored in an HI (history) entry within the file. This entry is of fixed length (4001 words). The first word contains a count of the number of filled 'history records', and is followed by space for 100 80-byte ASCII records each with the following format:

```
Date        Time  Username     Program   Function     Elapsed  CPU      STATUS
23-JUL-1985 12:22 CLARKE       TWOTVES   Output       01:31:34 00:09:05 00000001
```

In order that the mechanism can work, a blank history entry must be inserted in files created from scratch. This is done by a call to IFFHI. The entry should come second in the file, immediately following the RA entry. If the file is reprocessed to produce an output file, then EIHCPY should be used on the existing HI entry.

In addition, programs may set the 'function' string using the optional argument to IFFOPN or IFFOPI (this defaults to 'Create', 'Update', or 'Read' as appropriate), and may set the final status using the optional argument to IFFCLO, IFFCDL, or IFFCRN (this defaults to 1 - success). Another optional argument is provided for the 'close' routines to return the final history record, so that the program may for instance print it out.

The normal action is to add a new history record to the HI entry each time the file is opened (final values being written in when it is closed). This may be undesirable for programs which repeatedly open and close the same file. Such programs should use IFFOPI for subsequent openings - if the 'function' argument is omitted, the latest history record will be updated, adding the times to those already present. If there is no space in the history entry to add the new record, then all but the first record are discarded, a record saying 'HI overflow. Records lost.' is placed second, and the new record is placed third.

When a file is opened for write, a 'prototype' history record, with blank elapsed and CPU fields, and a status of 0, is written to the HI entry and also to the forepart of the file. In the event that the file is never properly closed, this record can be examined (possibly using the DUMP utility) to determine which operation had failed. It will not be possible to open such a file with IFFLIB until utility program IMEND (or another suitable program using the USR argument to IFFOPN) has been used.

Normal programs should *not* write anything into the history entry, though IFFLIB does not attempt to prevent this. Programs must ensure that there is one, and only one, HI entry in a file, and that it is the second entry.

A utility routine is provided to print the contents of a history entry so that the programmer need not be aware of the structure of the entry:

CALL EIHPHI([PRTN], [SUPH]) (entry-in-hand print history)

external PRTN - a routine which is called to print the header and records. It is called by CALL PRTN(STRING) where STRING should be declared as CHARACTER*(*), but its length will in fact always be 80. If omitted, then LIB$PUT_OUTPUT is used.

logical SUPH - if present and true, then no header line is printed.

This routine prints out the contents of a history entry, optionally preceded by a header line to title the columns. It should be called with an HI entry in hand.

4.5 *Junction creation and manipulation*

Within the IFF structure a junction is defined as a single coordinate together with one or more 'arms'. An arm consists of a pointer to the position in the file of an ST (string) entry, together with a vertex number within that string to indicate which end of the string is attached to the junction. The vertex number will either be 1 or the number of points in the string (junctions may not occur in the middle of an ST).

To enable fast location of junctions within the file, all junctions whose identifying points lie within a given rectangular area of the coordinate range of the file (a sector) are gathered together in one or more Junction Block (JB) entries. All JB entries for a given sector are chained together, and the head of each such chain is referenced in a Sector Header (SH) entry near the beginning of the file. The number of sectors may be specified when the file is created by means of a call to IFFSH (section 4.1.2.17). This size is available to users as variable IFSHSZ in common IFFJUN. In the case of an IFF file with no junction structure this variable will contain zero.

The sector concept may be visualised by means of the following diagram:

```
                             ------
                             | SH |
                             ------
                               |
        ----------------------------------------------- - - -
        |            |            |           |
     ------       ------       ------      ------
     | JB |       | JB |       | JB |      | JB |
     ------       ------       ------      ------
        |            |            |           |
     ------       ------       ------      ------
     | JB |       | JB |       | JB |      | JB |      ---------------------
     ------       ------       ------      ------  /---| XY, arm, arm, arm |
        |            |            |           |   /    ---------------------
     ------       ------       ------      ------/      ---------------------
     | JB |       | JB |       | JB |      | JB |------| XY, arm, arm, arm |
     ------       ------       ------      ------      ---------------------
        |            |            |           |                    |
                                                                ------
        |            |            |           |                 | ST |
                                                                ------
```

The ST pointers within the junction block enable the junction arms to be easily related back to the coordinate strings which form them (and also, although much less easily, to the features which contain the arms). In order to refer to the junction data from within a feature, the Junction Pointer (JP) entry is provided (see section 4.1.1.11). Note that although the position of the JP entry within a feature is not formally defined in IFF, in order that processing programs should not need to be too complex a logical ordering should be adopted with JPs between STs as appropriate.

For example, the geometrical arrangement:

```
     A                                B
     --------*------------*-----------
             |            |
             |            |
             |            |
```

should have the horizontal feature AB represented by the IFF sequence:

    NF FS ... ST JP ST JP ST EF

rather than, for example:

    NF FS ... ST ST ST JP JP EF

Similarly, although JB entries may in theory occur anywhere in the IFF file, to accord with current practice they should not appear within features. This means that when the feature is created, all junctions implicated in that feature must be remembered until the EF has been generated, after which IFJCR etc. can be called the requisite number of times.

It should be noted that this scheme preserves the integrity of the coordinate data and the file as a whole. The junction structure can be stripped out at any time and the resulting file will be geometrically complete; only the relationship between the junction arms will be lost.

4.5.1 *Junction creation*

CALL IFJCR(ISN, RPOSX, RPOSY, NARMS)

integer*2 ISN - sector in which junction lies

real RPOSX,RPOSY - position of junction

integer*2 NARMS - number of arms to reserve for junction

IFJCR creates a junction descriptor in an appropriate JB entry, creating that also if necessary. The junction in then 'in hand' and may be referenced by JIH calls to set up the arm information (see section 4.2.9.5). The 'entry-in-hand' is the JB, and the offset within the JB is contained in variable IFJPOS in common IFFJUN. The sector number ISN can be obtained via a call to IFJSEC (see below).

4.5.2 *Obtaining sector number*

CALL IFJSEC(RPOSX, RPOSY, ISN)

real RPOSX,RPOSY - position of junction

integer*2 ISN - sector in which junction lies

Given the position of a proposed junction, IFJSEC returns the number of the sector in which it lies, or -1 if the position is outside the sectored area. ISN can then be used to create the junction via IFJCR (see above).

4.5.3 *Scanning a sector*

CALL IFJSCN(ISN, IFJFN)

integer*2 ISN - sector to be scanned

external IFJFN - function to be called for each junction

IFJSCN scans sector number ISN, calling external function IFJFN with each junction in the sector in turn as the junction-in-hand. IFJFN should be declared as:

```
      LOGICAL FUNCTION IFJFN( RPOSX, RPOSY, NARMS )

      real      RPOSX,RPOSY - position of junction

      integer*2 NARMS       - number of arms
```

IFJFN should return a logical value 'true' to continue scanning or 'false' to stop. This mechanism can be used, for example, when searching for the nearest junction to a given point. The user's IFJFN can remember the closest junction so far (handing this information back via common), and possibly give up the search when a junction is found which meets the required criteria.

4.6 *Memory mapped files*

VAX versions of IFFLIB allow IFF files to be accessed as memory mapped section files instead of the usual asynchronous block input/output to disc. Routine IFFMAP should be called *before* files are opened to specify whether each file should be memory mapped or not. Note the files on another DECNET node may not be accessed by the mapped method.

The use of mapped files can be much more efficient in terms of CPU and disc I/O used, but naturally uses larger amounts of virtual memory than the normal IFF access method. Because of this, care must be exercised in its use. Particular problem areas are:

1. Mapping very large files may exceed the system limit VIRTUALPAGECNT, whereas the normal access method would have coped with such files.

2. The files are mapped by expanding the program virtual memory at the top end, above other code and data. When the file is closed, the address space used is deleted. This will contract the program address space provided that no memory has been allocated at higher addresses since the IFF file was opened, otherwise a 'hole' will be left in the address space which cannot be re-used, yet which still contributes towards the VIRTUALPAGECNT limit. To attempt to avoid this problem, mapped IFF files should be closed in a 'last-opened, first-closed' order. This only really matters if the program intends to open more IFF files subsequently.

3. Extending a mapped IFF file involves un-mapping it, extending the disc file, and then re-mapping it. For the same reasons as above, this may result in fragmentation and wasting of address space. To avoid this, files should be created large enough to start with (using the INILEN argument to IFFOPN) if possible. Failing this, one should attempt to ensure that an IFF file being extended is the last one to be opened, so that it is mapped at the top of the program's address space.

4.7 *Coordinate Blocks and Revision Levels*

The IFF coordinate block (CB) entry is intended to supercede ST and ZS entries, allowing multi-dimensional coordinate strings. In order to provide a degree of compatibility with existing programs, the concept of IFF Revision Levels allows CB entries to "pretend" to be STs or ZSs and vice-versa.

4.7.1 *Revision levels*

Two new fields have been added to the IFFHAN common block. These always contain the input (IRVLEV) and output (ORVLEV) revision level of the currently selected file.

The output revision level of a file controls whether new entries added to the file will be CBs or ST/ZSs. The possible values are 0 for old style files containing ST/ZS entries, and 1 for new style files containing CB entries. The output revision level for a file is fixed when the file is created and may not be altered subsequently, thus it should not be possible for a file to contain a mixture of ST/ZS entries and CB entries.

See routine IFFORV for details of setting the output revision level.

The input revision level of a file controls whether existing entries appear as CBs or ST/ZSs to calling programs. The possible values are: 0 for ST/ZS entries (old style programs), 1 for CB entries (new style programs), or -1 to treat all entries exactly as they are in the file (for debugging, or specialised programs).

See routine IFFIRV for details of setting the input revision level.

4.7.2 *Coordinate block (CB) entry*

The CB entry provides a variable number of rows (data points) together with a variable number of columns (point attributes). Any point attributes which are constant for all the points of a CB may be stored as a 'fixed attribute' in the header part. The header of a CB provides fields for the number of rows and columns, together with a 'flags' field (which at present contains the same as the IENDS word of an ST or ZS), and the graphical type to be associated with the points in the CB. These are followed by 'column header' fields identifying the contents of each column, and then the fixed attributes (each containing a code to identify the attribute, and the value).

It is not intended that the user should directly read or write (using the EIHR... routines) CB entries. A set of "third level" routines is provided for this purpose. These routines expect Fortran Records as arguments. The structure of these records is defined in the include file LSL$CMNIFF:CB.STR.

```
C define some parameters for now
C
        INTEGER*4       IFF_MAXCBCOL            ! max number of columns
        PARAMETER       (IFF_MAXCBCOL=20)
        INTEGER*4       IFF_MAXCBROW            ! max number of rows
```

```
        PARAMETER       (IFF_MAXCBROW=200)
        INTEGER*4       IFF_MAXCBDATA           ! max longwords of data
        PARAMETER       (IFF_MAXCBDATA=IFF_MAXCBCOL*IFF_MAXCBROW)
        INTEGER*4       IFF_MAXCBATT            ! max fixed attributes
        PARAMETER       (IFF_MAXCBATT=20)
        INTEGER*4       IFF_ABSENT              ! absent value
        PARAMETER       (IFF_ABSENT='80000000'X)
C
C Parameters defining column headers for x,y,z for convenience
C These must agree with those in the default ACDs
        INTEGER         IFF_C_X,IFF_C_Y,IFF_C_Z
        PARAMETER       (IFF_C_X=91)
        PARAMETER       (IFF_C_Y=92)
        PARAMETER       (IFF_C_Z=93)
C
C attribute structure
        STRUCTURE       /IFF_CBATT/
          INTEGER*4     ATTC            ! attribute code
          UNION
            MAP
              INTEGER*4 IATTV           ! attribute value (integer)
            END MAP
            MAP
              REAL      RATTV           ! attribute value (real)
            END MAP
            MAP
              CHARACTER*4 CATTV         ! attribute value (char*4)
            END MAP
          END UNION
        END STRUCTURE
C
C define the record structure to hold Coordinate Block (CB) header
C
        STRUCTURE       /IFF_CBH/
          INTEGER*4     FLAGS                   ! flags (from IENDS now)
          INTEGER*4     GTYPE                   ! graphical type
          INTEGER*4     NROW                    ! number of rows
          INTEGER*4     NCOL                    ! number of data columns
          INTEGER*4     COLH(IFF_MAXCBCOL)      ! column headers
          INTEGER*4     NATT                    ! number of attributes
          RECORD        /IFF_CBATT/ATT(IFF_MAXCBATT)    ! attributes
        END STRUCTURE
C
C and a record structure to hold Coordinate Block (CB) data
C - treats the whole thing as a 1-dimensional array
C
        STRUCTURE       /IFF_CBD/
          UNION
            MAP
              INTEGER*4         I(IFF_MAXCBDATA)        ! integer value
            END MAP
            MAP
              REAL              R(IFF_MAXCBDATA)        ! real value
            END MAP
            MAP
              CHARACTER*4       C(IFF_MAXCBDATA)        ! char value
```

```
            END MAP
          END UNION
        END STRUCTURE
C
C and a structure for use in declaring CB row arrays
C - for instance
C               RECORD  /IFF_CBITEM/    ROW(IFF_MAXCBATT)
C
        STRUCTURE       /IFF_CBITEM/
          UNION
            MAP
              INTEGER*4         I       ! integer value
            END MAP
            MAP
              REAL              R       ! real value
            END MAP
            MAP
              CHARACTER*4       C       ! char value
            END MAP
          END UNION
        END STRUCTURE
C
```

After including the file LSL$CMNIFF:CB.STR, the records are declared by e.g.

```
        RECORD /IFF_CBH/ CBH   for a header record, or
        RECORD /IFF_CBD/ CBD   for a data record
```

The fields are then accessed as for example:

```
        CBH.NCOL = 3
        CBH.COLH(3) = IFF_C_Z
        CBH.ATT(2).IATTV = 2
        CBD.R(23) = 1.234
```

4.7.3 *CB entry conventions*

In order that the majority of Laser-Scan programs can deal successfully with CB entries, the following points should be observed:

1. A CB entry should always contain X and Y columns. These should preferably not be fixed attributes.

2. A feature (i.e. all its CBs taken together) should not contain more than IFF_MAXCBATT attributes (including X and Y, and whether fixed or varying).

3. A CB must not contain more than IFF_MAXCBROW rows.

4. The attribute value IFF_ABSENT (='80000000'X) should be assumed to mean exactly the same as if the attribute was not present at all. Note that this value is an INTEGER, and tests for it must be made using an INTEGER variable (equivalenced onto a REAL if required).

4.7.4 *Creating a CB entry*

In order to create a new CB entry, the header fields must be set up as required, in particular FLAGS, GTYPE, NROW, NCOL, COLH(1:NCOL), NATT, ATT(1:NATT) (each ATT is itself a record, and both its fields ATT.ATTC and ATT.IATTV or .RATTV should be set). The data fields must be set up treating the arrays as though they were dimensioned (NCOL,NROW). The /IFF_CBD/ structure declares one-dimensional integer, real, and character arrays. The routine IFS_CB_WRITE (q.v.) is provided to assist in loading up the data part of the CB structure, though the user may create the structure themselves, or pass an ordinary array (see EIHRCB_DATA for the format).

Parameters for the X,Y, and Z column headers are provided as IFF_C_X (and Y, and Z) in the LSL$CMNIFF:CB.STR file for convenience and to avoid having to use FRTLIB to obtain these.

Routine IFFCB is used to create the CB entry from the header and data structures. See documentation of IFFCB.

If operating at output revision level 0, then the CB must contain x, y, (and z) values so that an ST/ZS can be created in the file. Failing this, IFF error 'LETC' will occur. Any other attributes will be discarded - if this happens, then the error 'LOST' will occur when the file is closed.

4.7.5 *Reading a CB entry*

After taking a CB entry into hand (via IFFNXT or IFFPKE), it may be read using the routines EIHRCB_HEAD and EIHRCB_DATA. The usual EIHR... routines should not normally be used.

```
CALL EIHRCB_HEAD(cbh)

out - record/IFF_CBH/ cbh        CB header record
```

The header part of the CB entry is read into the supplied record.

```
CALL EIHRCB_DATA(cbd)

out - record/IFF_CBD/ cbd        CB data record
```

The whole data part of the CB entry is read into the supplied record.

The format of the data is a sequence of 4-byte longwords (usually REAL*4 or INTEGER*4) arranged with columns in the order of the column headers (CBH.COLH). It thus could be considered as a Fortran array dimensioned as ARRAY(CBH.NCOL,CBH.NROW).

Note that it is permissable to pass an ordinary array in place of the record. It is then the user's responsibility to ensure that the CB will fit into the array, and that the columns are arranged as expected.

4.7.6 *Amending a CB entry*

A CB entry in hand may be amended in-situ using the routine EIHWCB. The usual EIHW... routines should not normally be used.

```
     CALL EIHWCB(cbh,cbd)

     in  - record/IFF_CBH/ cbh        CB header record
     in  - record/IFF_CBD/ cbd        CB data record
```

The entire CB entry is re-written from the supplied records. The new entry must occupy the same space as the old, so certain fields may not be altered, in particular NROW, NCOL, and NATT. If it is required to write a shorter CB back in-situ, then the old entry must be voided using IFFVOM, and the new entry replaced using IFFCB.

4.7.7 *Manipulating CB data*

Routines are provided to assist in extracting/inserting data values in CB data records. These routines do not access the IFF file.

```
     INTEGER*4 FUNCTION IFS_CB_READ(cbh,cbd,buf,ncol,colh,first,last)

     in  - record/IFF_CBH/ cbh        CB header record
     in  - record/IFF_CBD/ cbd        CB data record
     out - array           buf        space for output values
     in  - long            ncol       number of columns required
     in  - long array      colh(ncol) headers of required columns
     in  - long            first      first row to read
     in  - long            last       last row to read
```

The NCOL columns specified by the COLH array are extracted from the CBD structure into the simple array BUF. The rows (points) extracted begin at FIRST and end at LAST. BUF is treated as though it were dimensioned (NCOL,LAST-FIRST+1). If one of the requested columns is a fixed attribute, then the constant value is returned for each of the requested rows. If a requested column is not present, the absent value IFF_ABSENT (='80000000'X) is returned.

The function returns one of three codes with symbolic names in the parameter file LSL$CMNIFF:IFFMSG.PAR. These are IFF__MISSING (one or more columns requested were not present in the CB at all), IFF__FIXATT (one or more columns requested were present as fixed attributes), or IFF__NORMAL (all columns requested were present as varying columns). The first two are 'warning' codes (and may be tested as .FALSE.), while the latter is a 'success' code (and may be tested as .TRUE.).

```
     INTEGER*4 FUNCTION IFS_CB_WRITE(cbh,cbd,buf,ncol,colh,first,last)

     in  - record/IFF_CBH/ cbh        CB header record
     out - record/IFF_CBD/ cbd        CB data record
     in  - array           buf        input values
     in  - long            ncol       number of columns required
     in  - long array      colh(ncol) headers of required columns
     in  - long            first      first row to write
```

```
in  - long            last       last row to write
```

BUF is treated as though it were dimensioned (NCOL,LAST-FIRST+1). The data in the BUF array is inserted into the NCOL columns of the CBD structure specified by the COLH array, starting at row FIRST and ending at row LAST. All other rows and columns are left unchanged. If one of the requested columns is a fixed attribute, then the value is inserted into the fixed attribute. If a requested column is not present, the data for it is ignored.

The function returns the same codes as IFS_CB_READ.

```
CALL IFS_CB_COMPRESS(cbh,cbd)

in/out - record/IFF_CBH/ cbh        CB header record
in/out - record/IFF_CBD/ cbd        CB data record
```

The CB structure is compressed. The procedure may be considered to consist of the following steps:

1. Any columns with a constant value (including the absent value IFF_ABSENT) are moved to a fixed attribute and then removed. X and Y columns however are always left as they are. If the fixed attribute already exists, then its value is overwritten by the constant column value.

2. Any fixed attributes now containing the absent value are removed.

3. Any fixed attributes also present as varying columns are removed.

4.7.8 *Scanning CB entries in an IFF feature*

A routine is provided to assist in finding the total number of CB entries, rows, and columns, in an IFF feature or text component.

```
INTEGER*4 FUNCTION IFS_CB_SCAN(ncols,cols,ncbs,nrows,simple)

out - long            ncols       total number of columns
out - long array      cols(ncols) the column headers
out - long            ncbs        total number of CB entries
out - long            nrows       total number of rows
out - logical         simple      only x,y,(and z) present?
```

The function should be called with the first CB entry of interest in hand and returns with this entry still in hand. It scans the file, ignoring void (VO) and junction pointer (JP) entries until an entry which is not a CB is found, and returns the total number of distinct columns (including fixed attributes) found, their column headers, the total number of CB entries, the total number of rows, and a logical which is set to true if only X, Y, and possibly Z columns were found.

The function returns one of three codes with symbolic names in the parameter file LSL$CMNIFF:IFFMSG.PAR. These are IFF__NORMAL if all is well, or IFF__TOOMANY if more than IFF_MAXCBCOL distinct columns were found, or

IFF__POSERR if an error occurred positioning back to the original CB or the original entry was not a CB. The latter two are 'error' codes and may be tested as .FALSE..

4.7.9 *Compatibility between CB and ST/ZS*

Since the CB entry is intrinsically more versatile than the ST/ZS entries, there are limitations to the ability of IFFLIB to emulate ST/ZS entries when the actual entries in the file are CB, or to emulate CBs when the actual entries are ST/ZSs.

IFFST or IFFZS with output revision level 1 will create CBs with FLAGS set to IENDS, GTYPE set to 1, 2/3 columns (x,y,z), and no fixed attributes. This also applies if EIHCPY is used to copy a ST/ZS to a file with output revision level 1.

IFFCB with output revision level 0 will only be able to emulate as ST/ZS if the CB contains x,y (and z for ZS) columns. If it does not, the error 'LETC' will occur. Any extra columns are ignored. The emulation is more efficient if the CBs contain only 2/3 columns, and these are in the order x,y,(z). Note that IENDS for an ST/ZS has only 3 significant bits - any other bits in the CB FLAGS will be discarded. Note also that if EIHCPY is used so copy a CB to a file with output revision level 0, then any extra columns, or fixed attributes will be lost.

Reading an actual CB at input revision level 0 will only be able to emulate as ST/ZS if the CB contains x,y (and z for ZS) columns. If it does not, the error 'BINC' will occur. Any extra columns are ignored as far as the user is concerned. The emulation is more efficient if the CBs contain only 2/3 columns, and these are in the order x,y,(z). The EIHW routines (usually EIHWR on an ST/ZS) will work provided that the data read is a whole number of real data values (which it ought to be to be sensible anyway).

Reading actual ST/ZS entries at input revision level 0 will return CBs with FLAGS set to IENDS, GTYPE set to 1, 2/3 rows (x,y,z), and no fixed attributes. EIHWCB will work, again with the limitations that the amended CB must remain compatible with ST/ZS emulation.

In order that a user can tell whether the data in a file is in simple ST or ZS format (even though it may actually be a CB, or a CB is being emulated by IFFLIB), the common block LSL$CMNIFF:IFFSTR.CMN is provided. This contains two logical variables SIMPLE_XY and SIMPLE_XYZ. When a CB, ST, or ZS entry is taken into hand (using IFFNXT or IFFPKE), these are set as follows:

SIMPLE_XY is set to true if the entry actually is an ST, or it is a CB with only X and Y columns in that order (and no fixed attributes). This means that (in the case of a CB) EIHRCB_DATA can be used to read directly into a user array formatted as X1,Y1,X2,Y2,X3,Y3... (as for old STs).
SIMPLE_XYZ is set to true similarly for a ZS entry or a CB with only X, Y and Z columns in that order.

5 *Error handling*

The action taken when an error is detected by IFFLIB is as follows:

IERCD will be set to the appropriate ASCII code (see section 5.3 below) for all errors, thus it is sufficient for programs to test for IERCD non-zero to check for errors. In addition, IERNM will be set to a VMS error code, if appropriate (otherwise to 1, i.e. 'normal successful completion').

Unless IFFERM (q.v.) has been used to suppress error message generation, an error message of the form:

```
%IFF-E-OPEN, error opening IFF file
%IFF-E-MEND, open failed as IFF file was improperly closed, needs mending
%IFF-E-CREA, error creating IFF file
%IFF-E-NIFF, file is not an IFF file
%IFF-W-LOST, attributes lost due to IFF output revision level
or for all other IFF errors
%IFF-E-IFFERR, IFF error IERCD on LUN n
```

will be output using $PUTMSG, possibly followed by a system error message.

If IFFTDY has been called specifying an error handling routine, it is called.

If logical name LSL$DEBUG_TRACE exists, then LIB$SIGNAL is called with system error IERNM or with SS$_ABORT, to provoke a traceback (or even image exit for fatal errors).

Control is returned to the user program.

5.1 *Error recovery*

CALL IFFTDY([RTIDY])

external RTIDY - a subroutine name

This routine sets up a routine to be called whenever an IFF error occurs. When it has finished, RTIDY may either RETURN control to the user program, or exit. If it has called any IFF library routines, then it is strongly recommended that it exit, to avoid the possibility of recursive calls of the library.

5.2 *Error message control*

CALL IFFERM(ONOFF [,ACTRTN])

Logical ONOFF

This routine sets on (ONOFF = .TRUE.) or off IFF error message generation. The initial default is message generation is on. This routine does not clear IERCD or IERNM, and cannot result in an error. ACTRTN is an 'action routine' for the $PUTMSG system service. If supplied, then it is called with a character string argument containing the error text. The routine should return 0 to suppress

$PUTMSG outputting the message itself, or 1 to allow the message to be output. If ACTRTN and the comma are omitted, then any previous action routine is retained. If the comma is present, but ACTRTN absent (equivalent to %VAL(0)), then the action routine is cancelled and $PUTMSG reverts to its default behaviour.

CALL IFFERR([ERCD], [ERNM])

Character ERCD - error letter code
Integer*4 ERNM - system error number

Generates standard IFFLIB error message (provided that messages are enabled). ERCD is copied into IERCD, and ERNM into IERNM. If both arguments are omitted, then the present values of IERCD and IERNM are used. If just ERCD is given, then IERNM is set to 1 (SS$_NORMAL).

IFFERR is used internally in IFFLIB, but a likely use in programs is to allow an open error to be preceded by the program's own error message i.e.

```
        CALL IFFERM(.FALSE.)            ! turn off error messages
        CALL IFFOPN(1,'filename')       ! open file
        CALL IFFERM(.TRUE.)             ! error messages on again
        IF (IERCD.NE.0) THEN            ! error occurred
           Produce appropriate error message
           CALL IFFERR                  ! add IFFLIB message
        ENDIF
```

5.3 *Error code summary*

IERNM contains the VMS error code (as a longword) for system service and I/O errors, or 1 ('normal successful completion') for IFF internal errors.

IERCD contains 0 if no error has occurred, otherwise one of:

BA	bad argument/compulsory argument absent
BJ	bad junction (e.g. already deleted)
BS	bad sector number (outside sectored area)
BW	bad word count in entry (normally due to reading an unwritten part of the file)
DS	directive error (numerical code in IERNM)
EM	disc almost full ('end of medium')
FN	filename syntax error
IE	internal consistency check failed
IO	I/O error (numerical code in IERNM)
IV	unable to create a void
LC	unknown 2-letter entry code
LU	invalid LUN/no file on this LUN
MK	no marked position
NE	no entry in hand
NF	no current feature
NJ	not a junction (entry in hand not JB)
NM	no more (simultaneously open) files allowed
NS	no file selected
OE	requested transfer is outside entry-in-hand
OF	old file not IFF
OV	requested entry would overflow void

RB	reading beyond EOF
UI	UIC error (directory not found)
ARGS	bad arguments to call
BINC	unknown binary code in file
CLOS	unable to close file
CONN	unable to connect to file
CREA	error creating IFF file
HIST	error obtaining statistics for history record
LETC	bad letter code (IFFLIB internal error)
LOST	attributes lost due to IFF output revision level
LUNI	LUN already in use (file already open on this unit)
MEND	open failed as IFF file was improperly closed, needs mending
NIFF	file is not an IFF file
NOCB	the entry is not a CB
NOIF	no IFF file on this LUN
OPEN	error opening IFF file
READ	error reading the file
RENA	error renaming file
SLOT	maximum number of files already open (currently 256 for VAX)
USER	the user open routine (call by IFFOPN) has returned an error
WRIT	error writing the file

6 *File Layout*

The file structure is shown schematically below, broken down into Maps, Sections, Overlays, Features, and Entries.
Asterisks are used to indicate entries at a particular level.

```
+-+-+-+---------+-+-+---------+-+-+
 R H M           E M           E E             File Level
 A I H           M H           M J
+-+-+-+---------+-+-+---------+-+-+
 * *  (...Map...)   (...Map...)  *
+-+-+-+---------+-+-------------+-+
 M M N           N               E             Map Level
 H D S           S               M
+-+-+-+---------+-+-------------+-+
 * *  (.Section.) (...Section...)*
+-+-+-+-+------+-+-+-----------+-+-+
 N C C N        E N             E E            Section Level
 S C P O        O O             O M
+-+-+-+-+------+-+-+-----------+-+-+
 * * *  (Overlay)  (..Overlay..)
+-+-+----------+-+-+-----------+-+-+
 N N            E N             E E            Overlay Level
 O F            F F             F O
+-+-+----------+-+-+-----------+-+-+
 *  (.Feature..)   (..Feature..)  *
+-+-+-+-+--------------+-+
 N F T S                E                      Feature Level
 F S H T                F
+-+-+-+-+--------------+-+
 * (..Entries.........) *
```

The order in which IFF entries occur within each level is given below.

Entries at File Level

```
        RA - coordinate RAnge information.
        HI - HIstory of IFF file
o       SH - Sector Header
        EJ - End Job marker (end of file).
```

Entries at Map Level

```
        MH - Map Header (map-specific information).
        MD - Map Descriptor (map projection information).
        EM - End Map marker.
```

Entries at Section Level

```
        NS - New Section identification.
        CC - Cubic Coefficients for coordinate transformation.
        CP - Corner Points for coordinate transformation.
```

Entries at Overlay Level

```
        NO - New Overlay number and status.
        EO - End Overlay marker.
```

There is no specific order for the following entries, which occur within overlays but outside features.

```
o +     TC - Transmitted Comment.
o +     CH - Character data.
o +     CS - Character Size and spacing.
o +     SS - Symbol Select.
o +     SL - Symbol Library select.
```

Entries at Feature Level

```
        NF - New Feature sequence numbers.
        FS - Feature Status (includes feature type).
o +     AC - Ancillary Code
o       TS - Text Status                        \
o       TH - THickness or Text Height            |
  +     ST - STring of coordinates               |
  +     ZS - 3-dimensional string of coordinates | repeat for composite text
  +     CB - coordinate block, supercedes ST/ZS  |
o +     JP - Junction Pointer                    |
o       RO - text ROtation                       |
o       TX - TeXt string                        /
        EF - End Feature marker
```

Other Entries

The following can occur anywhere in a file :-

o + VO - VOid
o + JB - Junction Block

o - entry is optional
+ - several occurences of entry are permissible

7 *Annotated File Listing*

The following is an annotated listing of a brief OS style IFF file showing layout and order of entries.

```
RA   0.00    500.00    0.00    500.00   (coordinate RAnge)
HI - History                            (statistics for IFF file)
MH - Map Header                         (non-graphic map info)
         1001      201000      101000        1250           0
            0        4097           0     1310722         ...
MD - Map descriptor                     (holds projection information)
NS IFF file layout demo sheet           (New Section identification)
CC - bicubic transform for non-linear corrections
     .00000000E 000   .00000000E 000
     .10000000E 001   .00000000E 000
     .00000000E 000   .10000000E 001
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
     .00000000E 000   .00000000E 000
CP - 4 point transform                  (scale, translation, and rotation)
     0.0000 500.0000   0.0000 500.0000
     0.0000   0.0000   0.0000   0.0000
   500.0000   0.0000 500.0000   0.0000
   500.0000 500.0000 500.0000 500.0000
NO 1  0                                 (start New Overlay)
NF 1                                    (start New Feature)
FS 11                                   (Feature code - 11 is linear)
AC 3 100.5                              (Ancillary Coding)
AC 4 34 Cambridgeshire                  (Ancillary Coding)
AC 5 34 Bedfordshire                    (Ancillary Coding)
TH 0                                    (size unset)
ST 4 0                                  (STring of coords)
 137.2988  144.9971
 137.1202  156.9030
 150.4982  156.8733
 150.8999  146.3822
EF                                      (End Feature)
NF 2                                    (start New Feature)
FS 25                                   (Feature code -
                                         25 is an unoriented symbol)
TH 20                                   (size of symbol)
ST 1 0                                  (locating point)
 147.3486  257.3202
EF                                      (End Feature)
NF 3                                    (start New Feature)
FS 69                                   (Feature code -
                                         69 is an oriented symbol)
TH 40                                   (size of symbol)
ST 1 0                                  (locating point)
 169.6900  252.4772
RO     0.835                            (ROtation angle)
EF                                      (End Feature)
```

```
NF 4                                    (start New Feature)
FS 49                                   (Feature code -
                                         49 is a scaled symbol)
TH 0                                    (size unset)
ST 2 0                                  (two points give angle and size)
 149.4567  346.4330
 156.7132  355.5345
EF                                      (End Feature)
NF 5                                    (start New Feature)
FS 28                                   (Feature code - 28 is a text)
TH 12                                   (point size)
ST 1 0                                  (locating point)
 117.0385  144.7751
RO     0.869                            (ROtation angle)
TX Garden House                         (the text string itself)
EF                                      (End Feature)
EO                                      (End overlay 1)
NO 11  0                                (start overlay 11 - grid)
NF 9980                                 (start New Feature)
FS 398                                  (Feature code is linear)
TH 0                                    (size)
ST 2 0                                  (String of coords)
   0.0000  0.0000
 500.0000  0.0000
EF                                      (End Feature)
..                                      (rest of grid)
..
EO                                      (End Overlay)
EM                                      (End Map)
EJ                                      (End Job)
```

8 *Template IFF Program*

The following is a template program for reading data from an IFF file. It uses the LSL library IFFLIB.

```
        PROGRAM IFFEXAMPLE
C
***     MODULE  IFFEXAMPLE
***     IDENT   18MY83
C
C       Copyright Laser Scan Laboratories, Cambridge, England.
C       Author  : Paul Hardy
C       Created : 18/May/1983
C
C example file for IFFLIB programming
C
        IMPLICIT NONE
C
        INCLUDE 'LSL$CMNIFF:IFFHAN.CMN' ! common block definition
C
C w/s
        CHARACTER*40    FILENM          ! IFF filename
        INTEGER*2       IECODE          ! IFF entry code
        INTEGER*2       IELEN           ! IFF entry length
        INTEGER*2       FSN             ! Feature Serial Number
C
C code
C
        FILENM='SYS$DISK:[]IFFEXAMPLE.IFF'
C
C let's open the IFF file
C
        CALL IFFOPN (1,FILENM,0)                ! open IFF on unit 1
        IF (IERCD.NE.0) THEN                    ! check for errors
           TYPE *,'Can''t open IFF file ',FILENM
           TYPE *, 'Error code ',IERCD,' number ',IERNM
           GOTO 999
        ENDIF
C
C print IFF version information
C
        CALL IFFVER
C
C ok - the file is open - read entries from it
C
100     CALL IFFNXT (IECODE,IELEN)              ! gets next entry
        IF (IERCD.NE.0) THEN                    ! check for errors
           TYPE *, 'Error reading IFF file', IERCD
           TYPE *, 'Error code ',IERCD,' number ',IERNM
           GOTO 990
        ENDIF
        IF (IELEN.EQ.-1) THEN                   ! entry length negative
           TYPE *, 'Unexpected end of IFF file'
           GOTO 990                             ! unexpected end of file
        ENDIF
C
C check for normal end of job
```

```
C
        IF (IECODE.EQ.'EJ') GOTO 990
C
C check for start of feature and print feature number
C
        IF (IECODE.NE.'NF') GOTO 100            ! not an NF entry
        CALL  EIHRI(FSN,1,1)                    ! read 1 word from entry
        TYPE *, 'Feature Serial number is ',FSN
        GOTO 100
C
C all done - close up and exit
990     CALL IFFCLO (1)                         ! close IFF file
999     CALL EXIT
        END
```

Running the program as shown above produces the following responses:

```
M3A$
M3A$ ! IFFEXAMPLE.TXT
M3A$ ! example compile, link, and run of IFF library example program
M3A$
M3A$ fort iffexample
M3A$ link iffexample,lsl$library:ifflib/lib
M3A$
M3A$ r iffexample
IFF file - LSL$USER:[PAUL.EXAMPLES]IFFEXAMPLE.IFF;1
Created on  11-AUG-1988 13:00:31.41
Feature Serial number is     389
Feature Serial number is     396
Feature Serial number is     397
Feature Serial number is     435
Feature Serial number is     307
Feature Serial number is     308
Feature Serial number is     311
Feature Serial number is     313
Feature Serial number is     312
Feature Serial number is     309
Feature Serial number is     481
Feature Serial number is     303
Feature Serial number is     305
Feature Serial number is     436
Feature Serial number is     444
Feature Serial number is     319
Feature Serial number is     304
Feature Serial number is     318
Feature Serial number is     302
Feature Serial number is     315
Feature Serial number is     317
Feature Serial number is     351
Feature Serial number is     443
Feature Serial number is     310
Feature Serial number is     437
Feature Serial number is     442
Feature Serial number is     438
Feature Serial number is     301
```

```
Feature Serial number is     314
Feature Serial number is     316
Feature Serial number is     306
Feature Serial number is     320
Feature Serial number is     627
Feature Serial number is     439
Feature Serial number is     446
Feature Serial number is     631
Feature Serial number is     630
Feature Serial number is     626
Feature Serial number is     628
Feature Serial number is     625
Feature Serial number is     629
Feature Serial number is     581
Feature Serial number is     440
Feature Serial number is     445
Feature Serial number is     719
Feature Serial number is     736
Feature Serial number is     633
Feature Serial number is     632
Feature Serial number is     594
Feature Serial number is    9995
Feature Serial number is    9984
Feature Serial number is    9985
Feature Serial number is    9974
M3A$
M3A$ rena iffexample.iff xxx.iff
M3A$
M3A$ r iffexample
%IFF-E-OPEN, error opening IFF file
-RMS-E-FNF, file not found
Can't open IFF file SYS$DISK:[]IFFEXAMPLE.IFF
Error code   'OPEN' number        98962
M3A$
```

9 *Technical notes on differences from PDP IFFLIB*

1. Design constraints were to preserve file compatibility with the existing PDP11/RSX11M library and as much compatibility as possible at the subroutine call level.

2. The idea of using RMS record access to IFF files was considered but rejected because of the difficulty of rewriting entries of differing lengths (e.g. ST and VO to replace several STs). Hence Block I/O was used through RMS and the original IFFLIB scheme of Asynchronous read/writes into multiple block buffers was kept.

   Note however that this means that if a program collapses whilst in an AST, then IFFLIB may not be used to close any open IFF file, as ASTs may not be used from within an AST.

3. File open/close and basic I/O obviously had to be rewritten to use FABs/RABs rather than FCS FDBs. However the logic is similar and subroutine calls are compatible apart from the file descriptor handed to IFFOPN which is now a string descriptor rather than a zero byte terminated string.

4. The same layout of IFCB file control blocks and IBCB buffer control blocks was used but with the following changes:

   1. All addresses now LONG not WORD.

   2. FDB replaced by FAB, NAM, RAB.

   3. F.LUN(FDB) replaced by I.LUN(IFCB).

   4. Defined using $DEF and _VIELD rather than .ASECTs

   5. Other minor changes to ensure longword alignments.

5. Layout of COMMON/IFFHAN/ is also similar but several WORDS have become LONGs, notably IERCD and IERNM (see later for description of changes to error handling).

6. The second level interface to IFFLIB (Entry level) which is written mainly in FORTRAN required very little changes except:

   a) Error handling

   b) Variable lengths (WORD->LONG)

   c) String handling.

7. The first level calls required almost total rewriting to allow for the differences in VAX/PDP assembler languages and calling conventions. In particular Register usage was completely changed, together with procedure calls and argument handling.

8. Routines which are called from FORTRAN are defined as .ENTRY statements with an appropriate register save masks. In many cases these set up registers and use a JSB entry to common code routines which may be called direct from MACRO. This allows error cases to use JMP to common error handling code without worrying whether they were called from FORTRAN or MACRO.

9. Most FORTRAN entries call standard initialise routines IF$INI or IF$1ST which clear error conditions and set up a 'get next argument' mechanism.

10. The PDP version SAVR02 Coroutine type register save calls have largely been replaced by the .ENTRY register save masks. For MACRO entries PUSHR and POPR have been used and routines rewritten to use common exits to allow this.

11. Register usage is different because of the different machine architectures. The general register allocation is shown below but varies slightly between routines.

    PDP VERSION

    R0 Address of current FDB
    R1 Workspace/pointer
    R2 Buffer address/workspace
    R3 File virtual block number/word address high order
    R4 Byte offset in block/word address low order
    R5 Procedure call argument pointer.
    R6 Stack pointer (SP)
    R7 Program counter (PC)

    VAX VERSION

    R0 Temporary Workspace (not preserved across calls).
    R1 Temporary workspace (not preserved across calls).
    R2 Pointer to byte within buffer/workspace
    R3 File VBN/word address.
    R4 Byte offset in block/temporary pointer.
    R5 Pointer to current buffer.
    R6 Event flag number/flag word.
    R7 Unused
    R8 Unused
    R9 Pointer to current IFCB
    R10 Address of FAB within IFCB
    R11 Number of arguments left
    R12 CALL argument pointer (AP)
    R13 CALL frame pointer (FP)
    R14 Stack pointer (SP)
    R15 Program counter (PC)

12. IFF error handling is different because of the differences in operating system error codes. Under RSX error codes are byte negative, while under VMS they are longwords and a mechanism is supplied for generating error message texts from error code numbers. IERNM is now therefore a longword containing the VMS error code for system service and I/O errors or 1 ('Normal successful completion') for IFF internal errors.

IERCD is zero if no error has occurred and a two or four letter code defining the error if one does occur. The majority of error codes are the same or similar to the previous two letter codes but new codes OPEN, CLOS, CONN, CREA, READ, WRIT etc. have replaced the previous IO code for open, close, connect, create, read and write errors.

10 *IFF File Internal Description*

Knowledge of the internals of an IFF file is not generally required by people writing IFF applications utilities. However there are times when it may be useful, e.g. for programmers carrying out IFFLIB library maintenance, or in the investigation of elusive bugs. The following description, although sketchy, should be sufficient for these requirements.

10.1 *Overall Structure*

An IFF file consists of a series of 512 byte blocks addressed as a linear array of 16 bit words. The file is divided into a forepart of length one block, and a data part of variable length. The forepart contains saved information about the file as a whole, and is described below. The data part contains the user data as a series of variable length entries, and the entry structure is also described in a section below.

Note that an IFF file does not have RMS or FCS record structure, and can only be accessed by system utilities in terms of blocks. A block dump by the dump utility in terms of hexadecimal words is a useful tool in understanding the structure of an IFF file.

10.2 *Forepart Structure*

The layout of the forepart of an IFF file is defined within the IFFLIB library sources in module IFFDCL, but the first few fields might be as follows.

```
Hex value  Layout         Description
3936      .WORD IFF     ; IFF nameplate contains 'IFF' in RADIX50 (X3936)
0032      .BLKW 1       ; Default amount to extend file.
04AB      .BLKW 1       ; highest used feature internal sequence number
0001578E  .BLKL 1       ; End of file position (LPOSE)
0000016A  .BLKL 1       ; Reading position (LPOSR)
0000016A  .BLKL 1       ; Writing position (LPOSW)
00000000  .BLKL 1       ; Entry-in-hand position (LPOSH)
00000000  .BLKL 1       ; Marked position (LPOSM)
00000000  .BLKL 1       ; Start of feature position (LPOSF)
```

10.3 *Entry Structure*

The forepart takes up the first block of the IFF file, and the entry data starts in block 2. The first word of the first data block is unused and has the value 0 (zero). Thereafter there is a succession of entries all of which start with a 1 or 2 word entry descriptor. The high byte of the first word of an entry descriptor contains a binary entry code, described below. The low byte contains an entry length in words, including that of the descriptor itself. A special case is if the length byte is zero, when the next word contains the entry length, also in words, and also including both words of the descriptor.

```
e.g. (in hex)
4709 d d ...       code=47 (RA), length=9 words (8 data + 1 descriptor)
4D00 0160 d d ...  code=4D (MH), length=160 words (15E data + 2 descriptor)
```

10.4 *Entry codes*

Name	Decimal	Hex	description
ST	0-7	00-07	; 2D string (bottom 3 bits are IENDS)
ZS	16-23	10-17	; 3D string (bottom 3 bits are IENDS)
NF	32	20	; new feature
FS	33	21	; feature status
TH	34	22	; feature thickness
TX	35	23	; text within feature
JP	36	24	; junction pointer
RO	37	25	; text rotation
AC	38	26	; ancillary code
TS	39	27	; text status
CB	40	28	; coordinate block
EF	-1	FF	; end of feature
EO	-2	FE	; end of overlay
EJ	-3	FD	; end of job
EM	-4	FC	; end of map
VO	-128	80	; void
NS	64	40	; new section
CC	65	41	; calibration coefficients
CP	66	42	; corner points
TC	67	43	; transmitted comment
CH	68	44	; literal characters (to user's format)
SS	69	45	; symbol select
SL	70	46	; symbol library select
RA	71	47	; range of coordinates
CS	72	48	; character size / spacing
NO	73	49	; new overlay
MD	74	4A	; map descriptor
JB	75	4B	; junction block
SH	76	4C	; sector headers
MH	77	4D	; map header
HI	78	4E	; history

INDEX