Laser-Scan Ltd.

ILINK

Technical Reference

Issue 1.0 - 3-Oct-1986

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

Document "ILINK Technical" Category "TECHNICAL"
Document Issue 1.0 Andrew C. Morris 3-Oct-1986

1 *Introduction*

This document describes in outline the overall organisation and all of the main routines used by ILINK. It is intended to be read in conjunction with the ILINK User Guide.

2 *WHAT ILINK DOES*

ILINK is for the "idealisation" of feature geometry and/or the link/node structuring of the feature topology inherent in graphical data held in IFF format.

There are several routines which transform feature geometry:

1. ALIGN
   Brings close feature sections together into exact alignment

2. JOIN
   Forms junctions where any point comes close to any other

3. LPJOIN
   Forms junctions where feature-ends come close to any other point along a feature

4. PPJOIN
   Forms junctions where feature-ends (or point features) come close to feature-ends (or point features) only

ILINK will only apply one of these geometry transforming routines per run.

The single routine which transforms feature topology is:

1. BREAK
   Breaks features into separate features at every implicit junction (points at which existing features touch or cross).

   BREAK can be selected on its own, or together with any of ALIGN, JOIN, LPJOIN or PPJOIN. When BREAK is used on its own, junctions will be created and features broken wherever features touch or cross. When BREAK is used together with another operation, features will be broken only at junctions determined by that operation.

The qualifier which specifies that IFF output is to be Junction Structured is:

1. STRUCTURE
   Specifies that IFF Junction Structure entries should be included in the IFF output. In this link-node structure, the links are the features and the nodes are the feature junctions, free end-points and free points.

   STRUCTURE can be selected on its own or with any of ALIGN, JOIN, LPJOIN, PPJOIN or BREAK. When STRUCTURE is used on its own, the only nodes in the resulting structure will be at feature ends. In this case

an implicit PPJOIN process will be carried out with a join-tolerance of ZEROPLUS (=1E-4) to determine which end points are shared by more than one feature. When STRUCTURE is used with ALIGN, JOIN, LPJOIN or PPJOIN, but without BREAK, then the nodes in the Junction Structure will be only at junctions determined by that operation. When STRUCTURE is used together with BREAK, the nodes in the Junction Structure will be free feature-ends and junctions determined by BREAK as above.

The sequence in which routines are called is as follows:

```
+-------------------------------- INPUT ---------------------------------+

        GET_CMDS        Get DCL-command lines parameters and qualifiers
                        and read in any data associated with these

        SHOW_CMDS       Show command interpretation

        GET_FTRS        Read through input file setting data into data
                        structure

If no geometry or topology transforming process has been selected then
go straight to PUT_FTRS

+------------------------------ PROCESSING ------------------------------+

If PPJOIN is to be used then

        LN2LKND         find link/node structure with NSTOL = JNTOL

        PPJOIN          determine final node positions

else
        ALIGN or JOIN or LPJOIN (or BREAK, when BREAK is specified on its
        own)            Process feature geometry

        BREAK_FTRS      Break features apart at each node

        LN2LKND         Determine link/node structure
end if

+-------------------------------- OUTPUT --------------------------------+

        PUT_FTRS        Write output file, transferring features not
                        processed directly from input file and writing
                        processed features using both processed data in
                        memory and data from input file, if necessary
                        reconstructing whole features from the parts
                        which they may have been broken up into during
                        processing.
```

3 *SETTING UP THE DATA STRUCTURE*

Most of the operations in ILINK, BREAK for example, require random access simultaneously to every point in every feature being processed.

The simplest way to achieve this is to read every point into on-line (virtual) memory. This does have some disadvantages, such as limiting the maximum number of points which can be processed together to around 100,000 and can involve a considerable amount of page-faulting, even when only a small number of points is actually being processed. However, this is what ILINK *does* because this is what the QUADSCAN routine, which is used extensively by ILINK, requires (see QUADLIB User Guide for details on QUADSCAN).

If it is not required to process every feature in a file, features can be selected for processing by layer (/LAYERS = layer list) and/or by feature code (/FCP=filename, where file contains a list of feature-code pairs, specifying explicitly which features are to be processed with which).

The FCP file also serves as a means of specifying relative feature priority when the process in use uses feature priority, such as ALIGN which moves lower priority features onto higher ones (unless the MIDPOINT qualifier is active), or ENDS, which moves lower priority feature ends onto higher ones (unless the MIDPOINT qualifier is active). If no FCP file is given then feature priority is in proportion to internal feature sequence number. If an FCP file *is* given then features are only processed (aligned, joined or intersected) together if their feature-codes occur in a pair which appears in the FCP file. The feature given the higher priority is the feature with the code on the right.

Once GET_CMDS has read in every qualifier and any data associated with these, GET_FTRS reads through each feature in the input file.

Every feature found in the input IFF file is referred to as as IN_FTR.

All features selected for processing from the input file are referred to as PR_FTRs.

Features selected for processing may be broken into parts during processing, in which case each part is treated in the data structure in the same way as any original feature selected for processing. For this reason, Such part-features are also referred to as PR_FTRs.

When it is required to distinguish between features originally selected for processing and continuation part-features which arise when these features are broken at junctions, (parent) features with a PR_FTR index <= N_ORIG_PR_FTRS are referred to as ORIG_PR_FTRS.

Each INFTR is tagged

IN_FTR_STATUS(IN_FTR) = TRANSFER, REJECT or PROCESS

and has its file address stored in IN_FTR_ADDR, which will be used during output to relocate input features as necessary.

Each feature selected for processing (every feature from every layer if no /FCP or /LAYERS qualifier was used) are set into the data structure as follows:

1. All of its points are appended to the PT array
   PT(1..2,IPTSTA..IPTFIN) = (x1,y1), (x2,y2), ...

2. The first and last point indices for feature I_PR_FTR are held in
   FTR_PTS(1..2,I_PR_FTR) = IPTSTA, IPTFIN

3. The PT array is set up as a two way linked list
   PRV(IPT) = IPT-1; NXT(IPT) = IPT+1;
   PRV(IPTSTA) = NIL; NXT(IPTFIN) = NIL

4. Each point has access to the feature to which it belongs
   PT_PR_FTR(IPTSTA..IPTFIN) = I_PR_FTR

5. Each processed feature has access to its corresponding input feature index
   FTR_INFTR(I_PR_FTR) = IN_FTR

6. Each PRFTR has access to the original (parent) processed feature.
   FTR_ORIG_PR_FTR(I_PR_FTR) = I_PR_FTR
   This will later be used as an original-processed-feature identifier when a processed feature is broken into separate parts at nodes.

Closed-loop features have first point = last point. All features have PRV for first point and NXT for last point = NIL (the "null pointer" value, actually = 0).

One-point features are treated as zero length one-vector features (ie. they are held as two separate equal points). This might seem wasteful, but it does allow point-features to be processed by the same vector-processing routines which process all other features. This makes the code both more simple and more robust, as non-point features can easily contain zero-length vectors (duplicate points).

Point-features are identified as such on input:

```
PNT_FTR(I_PR_FTR) = .TRUE.
```

Any other features which become single-point features during processing, for any reason, are *deleted*.

4 *GEOMETRY PROCESSING*

```
+---------+
|  ALIGN  |
+---------+
```

Brings close feature sections into perfect alignment as follows:

1. Wherever the closest points on any two vectors eligible for alignment (which may *not* be from the same feature) come within alignment tolerance, the closest point on each vector to the end points on the other vector are inserted into the vector. If the vectors intersect then the intersection point is also inserted into both vectors (points are not inserted where they are very close to points which already exist on a vector).

2. Every point is then pointed to the nearest point within alignment tolerance with the lowest feature priority higher than its own and with which it is eligible to be aligned.

3. Every point is then pointed through any points it may point to which point to further points, to a point which points no further - a "base" point.

4. After some fiddling around, attempting to ensure that the resulting alignment is sensible (which does not always work!), every base point is then pointed back to the point pointing to it which is nearest.

5. After creating new points to pair with base-points which are not yet paired (because it is important that base-points are not deleted), all points which have been pointed to at any stage, or point to other points, but are are not now paired, are *deleted* (except for points which point to a point which is paired with another point, neither of which are in the same feature as the original point, such as occur near to junctions and where more than two features are aligned onto the same section of base feature).

6. All points in point/base-point pairs are moved to the same position as the base point.

7. All points which point to a base point but are not paired and have not been deleted are now moved to the same position as the base point.

   Note that this could not be done in the stage before this because if the MIDPOINT qualifier is active the base-point does not have its position fixed until all points in point-pairs have been moved to their mid-point.

8. Each feature is traversed, marking both points in each pair where base-point feature index changes as points which are at nodes. Later on, BREAKFTRS will use these markers to break features apart at every junction.

ALIGN ends here. Further processing is in common with other procedures and is described later.

```
+--------+
|  JOIN  |
+--------+
```

Forms junctions where any point comes close to any other.

The algorithm(s) used by JOIN are closely related to those used by the DRP program written for OS, in which individual vectors in point strings are treated geometrically as features in their own right.

The result of this is that a lot more time and effort is spent trying to get junctions between vectors "right" according to rules specified by OS than in any other ILINK procedures. The processing time and memory requirements necessary to do this are correspondingly high.

JOIN is not suited to processing features which contain a lot of "curves" - ie. point strings with a large number of points which are very close together.

```
+----------+
|  LPJOIN  |
+----------+
```

LPJOIN (Line to Point JOIN) joins point and feature-end points to features, anywhere along the feature length - and carefully avoids joining end-points to end-points, which is a job for PPJOIN, described below. LPJOIN works as follows:

1. Wherever a feature-end vector comes within max(EXTOL,JNTOL) of any other vector from any eligible feature (which *may* be the *same* feature), if the free end of the end-vector can be projected onto the other vector by a distance < EXTOL, or truncated onto it by a distance < min(EXTOL, vector-length), then a new point is inserted into the other vector at this position. Otherwise a point is inserted into the other vector at the nearest position to the free end point (if a point does not already exist at this position on the other vector).

2. Every end-point is then pointed to the nearest point to it (or to the nearest *original* point within JNTOL if the VERTEX qualifier is active).

3. Every end-point is moved to the position of the point it now points to (if any) - except when the point pointed to is on the same feature as the end point, when the end-point will only be moved if the separation *along the feature* between the two points is < JNTOL. All points moved onto are marked as being at a node.

```
+----------+
|  PPJOIN  |
+----------+
```

PPJOIN (Point to Point JOIN) joins points and feature-end points to the same.

After points and end-points have been set into a link/node structure by the LN2LKND process described below, the final position of each node is determined as follows: Any junction formed by the joining of exactly two end-vectors (irrespective of any point-features which may also be drawn into this junction), is located by projecting the end-vectors together if possible (as in LPJOIN). Otherwise the junction is located at the centre of gravity of all the points and end-points meeting there (the position already set by LN2LKND).

5 *TOPOLOGY PROCESSING*

```
+---------+
|  BREAK  |
+---------+
```

If BREAK is specified as a process on its own then this routine is used to breaks features into separate parts at every point where they touch or cross, as follows:

1. At every point where any two eligible vectors touch or cross, a new point is inserted into both vectors (if a point is not already at this position) and is marked for BREAKFTRS as being at a node.

```
+--------------+
|  BREAK_FTRS  |
+--------------+
```

After geometry processing, the next stage is to break features at every point now marked as being at a node - unless the PPJOIN process is in use, in which case this process does not apply as the only nodes in the structure are all at points or feature end points.

BREAK_FTRS works as follows:

1. Each feature is traversed, terminating the current feature and creating a new feature at every point along the original feature which is now marked as being at a node.

   Each feature must start and end with its own independent end points, because a shared point would not be able to link into more than one feature's point list simultaneously.

   So, for each point marked as at a node, a new point is created with the same position as the original node point. This point is used to close the current feature, while the original node point is used to start the new continuation feature.

   Every new feature I_PR_FTR which is created as a part of an original feature ORIG_PR_FTR has the original processed-feature index recorded

   ```
   FTR_ORIG_PR_FTR(I_PR_FTR) = ORIG_PR_FTR
   ```

   This gives access from every feature to the original feature from which it originated. Later on during output this may be used in order to reassemble whole features from the parts they may have been broken into; to provide each part-feature with a unique parent-feature identifier; and to access the parent-feature's input IFF address so that its original contents can be transferred to each part-feature output.

LN2LKND (LiNe to LiNk-NoDe structure conversion) locates nodes and creates a link/node structure of data giving access from each link to the nodes at both ends, and from each node to all of the links which join there, and to the direction in which they run.

The tolerance used here, NSTOL (Node Separation TOLerance), is ZEROPLUS (= 1E-4) for (BREAK, ALIGN, JOIN or LPJOIN), and JNTOL for PPJOIN.

LN2LKND works as follows:

1. The position of each point and feature-end point is checked for being within NSTOL of the position of every node record so far created.

   The image is divided into 10x10 node sectors to speed up this N**2 process, though this is complicated internally by the fact that nodes within NSTOL of more than one node sector must be included in every such sector - ie. all sectors must effectively overlap by NSTOL.

2. If no existing node is found within NSTOL within the sector into which the point or end-point falls, then a new node record is created with the position of this point and inserted into this sector (and also into any other sector overlapping this sector which it may also fall within). Otherwise, if an existing node is found within NSTOL of this point in this sector, then this node will have its position adjusted to be the centre-of-gravity of all of the points falling into it so far.

3. Whether the node is new or not, the node index for the start and end of each feature is recorded in FTR_NDS(1..2,I_PR_FTR), and as the node is identified for each feature-end, the feature index is recorded in a linked-list of arms connected to this node. This arm list is accessed from each node by a pointer to the head of the list, HEADARM. Each arm in the arm list is a record with two fields. ARM(ARM_FTR,IARM) is the processed-feature index, signed positive if the feature is leaving the node; negative if entering. ARM(ARM_NXT,IARM) is a pointer to the next arm in the arm list for this node. The list is terminated by a nul-pointer (NIL).

   Note that point features are connected to a node by *one arm only*.

This results in a structure in which every feature has access to the node at its start and end (point- and closed- features pointing to the same node at each end), and every node has access to all of the features which join there - and whether they are arriving or leaving.

6 *SORTING NODE-ARMS AND DUPLICATE-FEATURE IDENTIFICATION*

After the link/node structure has been created by LN2LKND, the arms for each node are always sorted so that the angle which each arm leaves the node (measured anticlockwise from the positive X-direction), is increasing.

Angles are calculated from the vector which joins to the node. Angles of zero are treated as 2*PI instead, so that the angle zero can be reserved to signify that the angle is undefined when the vector is a single point and had no direction.

Duplicate features are identified and merged during this process, as described below.

```
+-------------+
|  SORT_ARMS  |
+-------------+
```

Node arms are sorted and duplicate-features identified as follows:

1. For each node the (signed) feature indices in the arm list for this node are set into a local array. The angle (as defined above) at which each of these features leaves this node is then set into another array. These two arrays are then sorted together so that the angle array is sorted into increasing order.

2. Whenever the sorted list contains arms angles which are *exactly* equal, this is taken to mean that the corresponding two features must be geometrically identical along their entire length - because if they were not, they would have to part at some point, at which there should already be a node, between this node and the next. If features have already been broken into separate parts at every node implicit in their geometry, then there cannot be such a node along any feature.

   This test for duplicate point-strings is only valid when features have been broken at *every implicit node*, which is not the case when the processing routine used was LPJOIN or PPJOIN, so these processes will not identify any duplicate feature sections which may be present in features input for processing, or which may arise during processing - and will allow such sections to overlap.

   Where two features are found to be identical, the feature with lower feature-priority is "taken into" the feature with higher priority, and its arms are deleted from the arm lists for the nodes at both of its ends. This is effected by providing each feature with a linked list of features which share it. The list is accessed from the feature via HEADSHR. Each sharing feature record has two fields. SHR(SHR_FTR,ISHR) holds the processed-feature index; SHR(SHR_NXT,ISHR) holds a pointer to the next SHR record. The list is terminated by a nul-pointer.

   The feature which is taken in to the other is marked with
   ```
   FTR_PTS(1,I_PR_FTR) = NIL
   ```

   Having sorted these two arrays together - eliminating from them any reference to features which are now taken into other features - the

arms list for this node is reassembled from these sorted arrays.

7 *OUTPUT*

All output is done by PUT_FTRS. Each input-feature, which is now marked either for TRANSFERring, REJECTing, or having been PROCESSed, is processed in turn.

If it is to be rejected then it is just ignored

If it is to be transferred then TRANSFER_FTR uses the INFTR IFF address to locate the feature in the input IFF file, from where all of its original entries are transferred, except that it gets a new NF entry and AC-type-9 entries (parent-feature identifiers from a previous run of ILINK) and JP entries are ignored, as these are no longer meaningful.

If the feature has been processed then it is output by WRITE_FTR.

```
+-------------+
|  WRITE_FTR  |
+-------------+
```

WRITE_FTR accesses all of the parts into which the original feature may have been broken by starting with the original processed-feature index to access the first part, and then using GET_NEXT_PT_FTR to search from N_ORIG_PR_FTRS+1 upwards for a continuation feature with a matching original-feature index recorded in FTR_ORIG_PR_FTR (all continuation features arising from original feature I_PR_FTR have processed-feature indices which are contiguous and start above N_ORIG_PR_FTRS).

If the original feature is to be output whole then WRITE_FTR reassembles a feature from its parts, accessed as above, using APPEND_FTR, and the uses WRITE_WHOLE_FTR to output the whole feature. Otherwise it passes each part of this original feature to WRITE_PART_FTR.

```
+------------------+
|  WRITE_PART_FTR  |
+------------------+
```

First WRITEPARTFTR checks to see whether the feature has been taken into another feature because it shares the same point string:
FTRPTS(1,IPRFTR).EQ.NIL or has been deleted:
FTRPTS(2,IPRFTR).EQ.NIL If either is the case then the feature is ignored and WRITEPARTFTR returns.

Then the point string for this feature is set into LINKPTS(1..2,1..NLPTS) by the routine APPENDFTR. If there are less than two points in the link and the feature is not an original point-feature, then the feature is deleted and WRITEPARTFTR returns.

Now it is safe to start to output this feature. This is done as follows:

1. NF entry
   A new NF entry is written

2. FS entry
   Look to see how many features share this feature. If none share then the feature-code to be used for this feature is taken from the input

file.

If it is shared then, if the FCC (Feature Code Combination) qualifier has been used then an array containing the parent feature codes from each of the sharing features is set up, sorted into increasing order with duplicates removed and then matched against all of the feature-code combinations taken from the FCC file. If the combination matches then the feature-code specified for this combination of shared features is used. If the combination does not match, or if the FCC qualifier has not been used, then the shared-feature feature code SHRFC set by the SHRFC qualifier (default=999) is used.

3. AC entries
   If the feature is not shared, but has been broken, then an AC-type-9 entry is (optionally) written next (if the /PARENT qualifier is active) holding the parent-feature processed-feature index as a unique identifier for the parent feature from which this broken section of feature arises.

   If the feature is shared then for each sharing feature (including this feature) an AC-type-1 (secondary feature-code) entry is written holding the feature code from the parent feature from which the shared feature originated; followed (optionally) by an AC-type-9 parent identifier entry (as above); followed by every AC entry which may have been present in the original feature - except for AC-type-9's which are ignored as no longer meaningful.

4. JP entry
   If the /STRUCTURE qualifier is active then, before the ST entry, a JP (Junction Pointer) entry is written. This will eventually point to the Junction Block and give the offset from the start of this block of the node which the start of this feature is connected to. But as the Junction Block is not written until after the last feature has been written, the JP entry here is used only to reserve space for this data which can then be filled in later on, and to hold in the mean time the node index for this node in memory.

5. ST entry
   Next the ST entry is written. The output file IFF address of the start of this ST entry is stored in FTR_DAT(FTR_ST,I_PR_FTR) = LPOSE, and the number of points in the ST is stored in FTR_DAT(FTR_NP,I_PR_FTR) = NLPTS, so that when it comes to write the JB (Junction Block) entries, the data to be inserted in the JB is available.

6. JP entry
   If the feature is not a point-feature and the /STRUCTURE qualifier is active then another JP entry is written after the ST which will eventually point to the node which the end of this feature is connected to.

7. EF entry
   Then an EF entry is written as usual.

Finally, a summary of process statistics, including link/node structure details, is written to SYS$OUTPUT, and a check is made on the link/node structure generated so that if the number of node arms belonging to non-point features is not exactly twice the number of such features, a warning is given that the structure is invalid.