Laser-Scan Ltd.

ILINK

Technical Reference

Issue 2.0 - 18-July-2002

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

Document "ILINK Technical" Category "TECHNICAL"
Document Issue 2.0 Andrew C. Morris 0-Dec-1988
Document Issue 1.0 Andrew C. Morris 3-Oct-1986

CONTENTS
---

1 **INTRODUCTION**

This document describes the overall organisation of ILINK, and describes the operation of each of its main subroutines in as much detail as is necessary for someone who is not familiar with ILINK to understand what it is trying to do, without necessarily explaining exactly how it does it, which is best discovered by looking at the code.

Much of what ILINK does is very simple in concept, but the code itself has been complicated by the usual effects of the ever growing requirement for more and more generality and for new qualifiers to deal with special cases.

Features can now be selected for processing by layer and/or by feature code pairs, and/or by FRT, or by layer pairs. Possibly the most awkward modification was to allow ILINK to handle CB entries . Even though it still never uses any but the (X,Y) coordinates from the point attributes for each point during processing, the difficulties that were encountered when it came to retrieving the rest of the point attributes to write to the output file, were considerable.

ILINK is now a large program made up from many highly inter-dependent parts. Some of these parts have grown to a large extent by trial and error rather than by design and may be very easily "upset" by being "fiddled with" without a sufficient insight of exactly what they are trying to do, and very difficult to correct.

Details concerning the practical use of ILINK are to be found in the ILINK User Guide, and documentation concerning the exact function of each subroutine is to be found in the source code. It should be stressed that

> ***this document is for outline technical reference only, and is intended to be read in conjunction with both the ILINK User Guide and the ILINK sources.***

Like most large programs, the overall structure of ILINK is largely a product of step by step evolution rather than overall design. For this reason a brief history of the developement of ILINK is given below.

2 **HISTORY OF DEVELOPEMENT**

ILINK was initially intended for breaking features where they touched or crossed and providing the resulting features with a link-node structure for the purpose of data-base compilation. Much of the code required closely resembled that used by the existing utilities, IFJ, CBA and DRP.

2.1 **IFJ**

The function of LSL IFJ (the IFf Junction-structure creation utility) was simply to create a junction structured IFF file from an existing IFF file, where a junction is where two or more polyline ends meet, but not where polylines cross. As ends often do not not meet exactly, all ends within a given tolerance were brought together in one junction by moving them onto the first end encountered there.

ILINK was required to join polyline ends together more in the way that a human operator might do, that is, with more intelligence. Ends should be brought together either

* one onto the other, or
* to their centre of gravity, or
* by projecting or trunctating end vectors to meet

2.2 **CBA**

The main function of CBA (the MCE Common Boundary Alignment utility) was for polyline alignment, that is, for moving parts of polylines together, one onto the other, along sections where they were seperated by less than a given tolerance.

Automatic feature alignment raises a number of difficult problems, especially near to junctions between three or more features. CBA did not overcome all of these problems and was in need of improvement, but it was highly tailored to the particular requirements of the customer for whom it was originally developed, and this functionality was now required by other customers.

2.3 **DRP**

The function of DRP (the OS Data Reformating Package) was to correct common errors which arise during manual digitising, to a standard that was acceptable to Ordenance Survey. DRP is tailored to OS's high standard of requirements, and spends a lot of time doing things that many people would not worry about. It was felt that it would be nice if ILINK could provide much of the functionality of DRP in a "cheap and chearful" sort of way.

3 **SPECIAL FEATURES**

3.1 **QUADTREE PROCESSING**

Another problem with IFJ was that it was a simple program which compared every polyline end with every other. This process involves order N**2 comparisons, so that when N is large, it can take a very long time. This problem can be overcome by first dividing the image area into a number of sectors and dividing up the set of polyline ends between these sectors, then comparing only ends within the same sector.

This sectoring is usually done in one of two different ways:

* The image area is divided into a fixed number of equal parts, say 10x10. This is easy to code but takes no account of how many points are being divided or how thay are distributed over the image area.

* Ask how many points there area in the image area, and divide this area into just two or four equal parts if there are more than a certain number K of points considered to be worth dividing, and then apply the *same* process recursively to each part until no part contains more than K points. The number of points worth dividing depends on the relative cost of division against the cost of processing the remaining points together in an exhaustive order N**2 scan.

Whichever method of sectoring is used, points which are within tolerance of a sector boundary *must* also be included in the neigbouring sector(s). This means that neigbouring sectors must overlap by the given tolerance.

ILINK uses the second form of sectoring (although the IFF junction structure used by ILINK makes use of the first), using code developed from that originally used by CBA, where each sector is recursively divided into quarters, giving rise to the structure commonly referred to as a "quadtree". All of the problems associated with sectoring are now completely hidden from ILINK through the use of the routine QUADSCAN which was standardised within COORDLIB during ILINK developement. All that the QUADSCAN user sees is the subroutine which is passed to it, which it uses to process each point pair which lie within tolerance.

The main *dissadvantage* of using the form of quadtree processing implemented in the routine QUADSCAN is that data relating to every point in the entire file being processed must be stored together in (virtual) memory, so that *the size of the dataset which can be processed by ILINK is restricted by the amount of virtual memory available.* Occasionally this restriction can cause problems, but this is not often the case and the speed advantage in the general case far outweighs this dissadvantage.

3.2 **DYNAMIC MEMORY ALLOCATION**

Because of the way ILINK works, it cannot process a file feature by feature, but must read every feature into memory and process them all together. The amount of memory used is therefore proportional to the size of the file being processed. If a fixed capacity data structure was used this would mean that ILINK would always hog the largest permissible amount of virtual memory even when processing the smallest of files. To avoid this, the file being processed is first scanned to estimate the amount of memory that will be required to process it, and the data arrays are then dimensioned accordingly after the program has started (ie. dynamically).

3.3 MEMORY REQUIREMENT ESTIMATION

The amount of memory required depends not only on the number of features being processed, but also on the particular process being applied (as ILINK now has a number of separate functions). With some of these functions the amount of memory required for processing can be estimated exactly before processing starts, while with others it cannot. For example, if each point requires a fixed amount of memory, and the process involves inserting extra points into features where they cross, then the amount of memory required depends on the number of places where features cross, which is not known until such intersections have been detected during processing. In such cases the user is asked to estimate the average number of crossings per feature, and if this turns

out to be too low, ILINK must be rerun with this estimate increased.

4 **TOLERANCING**

Besides the tolerances supplied by the user (JNTOL and EXTOL) which are in IFF units, ILINK also makes use of a rounding error tolerance which is the smallest distance in IFF units below which coordinate differences may be due to rounding error. VAX REAL*4s have 7 significant denary digits. After a few calculations, such as those involved in finding vector intersections, there may remain only 6 significant digits. The value used is referred to as COORD_TOL. This is set a bit on the cautious (large) side, but smaller values have been found to cause problems. The value used is:-

(maximum absolute coordinate value)/0.5E6.

5 **OUTLINE DATA STRUCTURE**

ILINK makes extensive use of linked lists. Some are one-way connected, others two-way connected. Some are finite, others circular. Finite linked lists are all terminated with the same terminating-pointer value, NIL (which must have a different value from any other legal non-terminator pointer value. The value used is in fact zero, because all pointer values are array indices >= 1).

One-point features are usually treated as zero length one-vector features (i.e. 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 forces the code to be more robust, as non-point features may contain zero-length vectors (duplicate consecutive points) in any case.

In what follows, and in FORTRAN identifiers, features are often referred to as "ftrs", points as "pts" and coordinates as "coords".

Features selected for PRocesessing, as opposed to transfering directly from input to output, or rejecting as invalid, are generally referred to as "PR FTRs". All features in the input IFF file are referred to as "IN" ftrs.

Each IN ftr has data associated with it as follows:

* the feature status = TRANSFER, PROCESS, REJECT or (process as-) SYMBOL_STRING, in IN_FTR_STATUS

* the ftr IFF address is stored in IN_FTR_ADDR, so that this feature can be looked up later when it, or some part of it, is required to be transferred directly to output, or is needed for any other reason.

Each point belonging to a PR ftr is held in a two-way finite linked list of points, which makes it easy to insert and remove points along a feature, while still allowing movement forwards or backwards along the feature. Data associated with each point is as follows:

* its X and Y coords, in PT(1..2). Note that no Z coord, or any other point attribute which may reside in a CB entry toether with X and Y coords, is ever stored in memory. If these are ever required, they are read from the input IFF file, using the IFF address stored in IN_FTR_ADDR.

* the PR ftr index of the PR ftr to which it belongs, stored in PT_PR_FTR

* the distance of the pt along the PR ftr to which it belongs, in PT_DST

* the PT index of the point before and after it, in PRV and NXT

* the PT index of the pt it is nearest to, in NEAREST

Each PR ftr has data associated with it as follows:

* the PT indices of its start and end points, in FTR_PTS(1..2)

* its corresponding IN ftr index, FTR_INFTR

* its original (parent) PR ftr index, FTR_ORIG_PR_FTR, for use when reassembling whole ftrs from the parts they may have been split up into

* a flag, PNT_FTR, indicating whether it is a 1-point ftr.

The internal link-node structure used by ILINK has data structure as follows:

* Each node has (X,Y) coords NPT(1..2), weight NWT, and a pointer into a linked list of node arms, HEADARM. There is therefore no limit to the number of links which can meet at a single node.

* Each link is a ftr and has end node indices in FTR_NDS(1..2).

* Each arm has a signed ftr (link) index, ARM(1), positive for link leaving node, negative for entering, and a pointer to the next arm in the arm-list for this node, ARM(2).

6 **DIVISION INTO SEPARATE FUNCTIONS**

Originally it was envisaged that ILINK would allow any number of operations to be carried out on the input data in one single run. However, as ILINK grew and the problem of deciding what order it was best to perform the various processes in became more difficult, it became clear that the code would be considerably simplified if ILINK only ran one process per run. This transfers The burden of deciding which order to apply these processes in onto the user. This also means that a separate output file is produced for each stage in a multi-stage process, which is very useful for finding at what stage processing is going wrong when it does go wrong.

The separate processes which ILINK now offers are:

1. PPJOIN : Joining feature ends to feature ends
2. LPJOIN : Joining feature ends to features
3. LLJOIN : Aligning features with features
4. MERGE : Merging duplicate feature sections into single features
5. BREAK : Breaking features where they touch or cross
6. SORTARMS : Sorting node arms within link-node structure by arm angle
7. STRUCTURE : Generation of link-node structure

ILINK will only allow one process qualifier per run. The qualifiers used by ILINK and their legal combinations are given below:

'*'=Legal '.'=Illegal	PROCESS QUALIFIER E X								J N	A Y	L	L I	O N I	M O T	P A R	R O J	S H	V E R	V E R	V R
PROCESS	A C P	B P F	T O L	F C C	F C P	F R T	T O L	E R S	L C P	I S T	E S 2	T O R	P A C	E N T	E C T	R F C	T O L	I F Y	T E X	T O L
BREAK	.	*	.	.	*	*	.	*	*	*	*	*	*	*	.	.	.	.	.	.
FREE_ENDS	.	.	.	.	*	*	.	*	*	*	*	*			.	.	.	.	.	.
LLJOIN	.	*	.	.	*	*	*	*	*	*	*	*	*		.	.	.	.	.	.
LPJOIN	.	*	*	.	*	*	*	*	*	*	*	*			.	.	.	*	*	*
MERGE	*	*	.	*	*	*	.	*	*	*	*	*	*	*	.	*		.	.	.
PPJOIN	.	.	*	.	*	*	*	*	*	*	*		.	.	*	.	*	*	.	*
SORTARMS	.	.	.	.	.	.	.	.	.	*		.	.	.	.	.	.	.	.	.
STRUCTURE	.	.	.	.	*	*	.	*	*	*	*	*	.	.	.	.	.	.	.	.

7 **COMMON CODE DESCRIBED**

Although ILINK operates almost as a number of separate utilities, there is a large amount of code shared between different operations. The main subroutines which are used by more than one ILINK process are described below.

7.1 GET_CMDS

Decodes the command line. This is used by every process.

7.2 GET_LOOKUP_FILES

Whenever a large amount of data is needed to tell ILINK exactly what to do, such

as telling it which feature codes are to be intersected with which for example, this data is taken not directly from the command line but from a lookup file specified on the command line. ILINK permits the use of several lookup files (ACP, FCP, LCP, FCC, FRT). GET_LOOKUP_FILES reads in all lookup files specified.

This is used by every process except SORTARMS.

7.3 SCAN_IFF

SCAN_IFF reads through the input file and, using information already obtained from the command line and from lookup files, which determines which features are to be selected for processing, decides the total number of features and the total number of points for which memory must be allocated.

This is used by every process except SORTARMS.

7.4 ALLOCATE_MEMORY

Allocates memory according to the capacities determined by SCAN_IFF.

This is used by every process except SORTARMS.

7.5 GET_DATA

Reads all data required for processing into memory. For some processes (LPJOIN, LPJOIN, MERGE, BREAK), every point must be read into memory for every feature selected for processing, while for other processes (PPJOIN, STRUCTURE) only the first and last points need be read into memory.

Each feature read into memory has a number of items of data set up, including its feature code and its IFF address. Each point within each feature also has a number of data items set, including its X and Y coordinates, and a pointer to the record for the point before it and the point after it.

This is used by every process except SORTARMS.

7.6 LINKNODE

This routine determines the link-node structure which exists between all of the features currently in memory, treating each feature as a link. A tolerance is used to gather close feature ends together into a single node. The link-node data structure in memory is as follows:

* There is an array of node records, each having a node X and Y coordinate, a node weight, and a pointer into a linked list of node arms.

* Each arm record contains a signed feature index and a pointer to the next arm record in the arm list for this node, or else to a list terminator. The sign of the feature index is positive for features leaving the node, and negative for features arriving there.

* Each feature has a record of the indices of the nodes at both ends.

Any features for which both ends become merged into the same node will be deleted, with a suitable warning, if they are too short to be considered a valid closed loop (less than 3*JNTOL).

7.7 SORT_AND_MERGE

Sorts arms at each node into order by increasing orientation angle. Where two arms have identical orientation, it is assumed that the associated features are exactly coincident along their whole length (think about it!), and these features are merged into a single shared-feature. This is done not only when the MERGE operation is in effect, but for all operations which use this routine.

The idea behind this forced merging is that when features come to be output, if the MERGE process is not in effect then the points along each separate feature sharing the same "base" feature (there is no restriction on the number of features which share the same base feature) must be reconstituted from exactly the same point string, so ensuring that aligned features are perfectly aligned.

Note that angles calculated as zero are treated as 2*PI instead, so that the angle zero can be reserved to signify that the angle is undefined in cases where a vector is a single point having no length and therefore no direction.

7.8 SET_BROKEN

Just marks all features which are not whole original features.

7.9 PUT_FTRS

All output is done by PUT_FTRS, which writes the output IFF file from the input file together with the data now in memory. Features not selected for processing are transffered directly from the input file. Processed features are taken from memory. Features marked for rejection are ignored. Whole features are reassembled where necessary from the part features they may have been broken into, or the shared features they may have been merged into. If the STRUCTURE quallifier is in operation then the in-memory link-node structure is used to write IFF junction structure entries where required.

8 **EACH ILINK PROCESS DESCRIBED**

The way in which ILINK performs each of its different process options is described below.

8.1 PPJOIN - JOINING OF FEATURE ENDS TO FEATURE ENDS

Steps are as follows:

1. GET_CMDS - see above for details

2. GET_LOOKUP_FILES - see above for details

3. SCAN_IFF - see above for details

4. ALLOCATE_MEMORY - see above for details

5. GET_DATA - see above for details
   Records (feature code, layer, feature index) for each feature, but stores no data per point

6. LINKNODE - see above for details
   Sets up link-node structure, reading feature end-point coords from input file

7. PPJOIN
   For each node set up by LINKNODE, gets feature end vector for each arm connectiong to this node, from input file, then finds final node position, which also depends on whether the PROJECT qualifier was used. See figure 7.

8. PUT_FTRS - see above for details
   Sets all feature end points to their corresponding node positions, then creates the output IFF file.

8.2 LPJOIN - JOINING FEATURE ENDS TO FEATURES

Steps are as follows:

1. GET_CMDS - see above for details

2. GET_LOOKUP_FILES - see above for details

3. SCAN_IFF - see above for details

4. ALLOCATE_MEMORY - see above for details

5. GET_DATA - see above for details
   Records (feature code, layer, feature index, first and last point index) for each feature, and (X coord, Y coord, feature index, prev and next point index) for each point.

6. LPJOIN

   1. Calls QUADSCAN to apply LPJSUB to each point pair for all points selected for processing. LPJSUB does nothing if neither point is an end point or a point from a polypoint string. Otherwise it checks to see if one end vector can join onto the other vector (see LPJSUB for details of joining criteria), and if it can, inserts a new point into the vector being joined onto)

   2. Calls A2L to convert linked lists for closed features into circular linked lists, so that first/last points in closed features are treated in what follows as a *single entity*.

   3. Calls SET_PT_DST to set up the distance of each point in every feature from the start of the feature.

   4. Calls UNDO_END_JOINS to "undo" joins to points "too close" (less than JNTOL) to feature ends, since such joins should be made using PPJOIN which uses more suitable criteria for end-point joining.

   5. Calls UNDO_SMALLLOOPS to "undo" joins to points within the same feature which are "too close" (less than 3*EXTOL) to feature ends, since such joins are generally undesirable.

   6. Calls DEL_SURPLUS_PTS to delete inserted points which are not now being joined onto.

   7. Calls MOVE_ENDS_TO_LINES to change to coordinates of end points to those of the point they are being moved onto.

   8. Calls L2A to convert closed features back from circular linked lists to non circular linked lists with end point equal to start point.

   9. Calls BREAK_FTRS to break features into separate part-features at joins. The original reason for doing this was that ILINK used always to break features at joins. Now it will only break features under the BREAK process. The reason features are still broken at this stage is realy only so that LINKNODE will count the correct number of nodes formed by this process. The broken features will

be reassembled into whole features before they are written to output.

7. LINKNODE - see above for details
   Sets up link-node structure, from features broken and with ends repositioned by LPJOIN.

8. SORT_AND_MERGE - see above for details
   To be honest, I don't know why this is called here. I suspect that the reason is historical, because it is possible, though rare, for LPJOIN to pull ends around in such a way that a small amount of "double-digitising" is introduced (which can be fatal to some other utilities, such as IPOLYGON). Here such double-digitised feature sections will be merged into single features, and there was probably a time when they would have stayed merged on output, so eliminating any possible double-digitising. Now only the ILINK MERGE process will leave merged features merged as single features on output. All other processes will separate them into their original constituents before writing to output. However, change this at your own peril!

9. SET_BROKEN - see above for details

10. PUT_FTRS - see above for details

8.3 LLJOIN - ALIGNING FEATURES WITH FEATURES

Steps are as follows:

1. GET_CMDS - see above for details

2. GET_LOOKUP_FILES - see above for details

3. SCAN_IFF - see above for details

4. ALLOCATE_MEMORY - see above for details

5. GET_DATA - see above for details

6. LLJOIN - AAAAAAAAAARRRRRRRRGGGGGGG!!!??###@%^&$$!
   The process of alignment, though simple in concept, is surprisingly difficult to get to work reliably in practice. The following algorithm is probably not yet optimal, but it has been evolved to produce satisfactory results for a very wide range of different alignment problems. Any changes should be made with maximum possible caution, bearing in mind that getting alignment to work properly for some new situation is very likely to cause it to fail in a whole lot of others.

   1. Calls QUADSCAN to apply CHKVEX1 to each point pair for all points selected for processing. CHKVEX1 inserts the nearest points within JNTOL along one vector to the end points on the other vector, and vice versa. Also, where vectors intersect, both vectors have a point inserted at this intersection point.

      By inserting these extra points, the alignment process becomes simply one of comparing distances between points rather than between points and line segments. Points can point to other points, and these to further points, which allows all of the points in the vicinity of a loose junction between any number of features to be "threaded" together into a single point.

   2. Calls A2L to convert linked lists for closed features into circular linked lists, so that first/last points in closed features are treated in what follows as a *single entity.*

   3. Calls QUADSCAN to apply CHKPTS1 to each point pair (indices (P,Q)) for all points selected for processing. Each point has a pointer to another point so that the nearest point so far to P to which P is eligible to join, causing alignment, is given by NEAREST(P). CHKPTS1 points P to Q or Q to P or neither in such a way that in the end every point will be pointed to the nearest point with the lowest higher priority which is within alignment tolerance. See figure 1 for an explanation of why this is necessary, rather than pointing each point simply to the nearest eligible point, or to the nearest eligible point with highest higher priority.

      All points ever pointed to are also marked as having been "seen", so that later on inserted points which are seen but do not take part in alignment can all be deleted.

4. Calls DEL1 to delete all runs of inserted points which are "seen" but do not "point" and are not "pointed to" and are bounded in both directions by points which are pointing or pointed to.

5. Calls SET_PT_DST to set up the distance of each point in every feature from the start of the feature.

6. Calls MARK_ALIGN to mark (with QDAT) all points which are on a feature section that will be aligned to or from according to NEAREST. See code for further details. Also see figures 2 and 3.

7. Calls GROUND to point all points "through" other points to the highest priority point which points no further *and* is marked (by QDAT) as being on a section of feature that is to be aligned. GROUND then unjoins all joins to and from points not marked by QDAT for aligning. See figure 4.

8. Calls DISCON1 to disconnect runs of pts which point to same point and are bounded in either direction by a non-connected point or a ftr end. See figure 5.

9. Calls DISCON2 to sets PT_SEEN = .FALSE. for runs of pts which are seen but not pointing or pointed to, and are bounded in either direction by a non-seen, non-pointing and non-pointed-to point.

10. Calls QUADSCAN to apply CHKPTS2 to each point pair (indices (P,Q)) for all points selected for processing. CHKPTS2 sets PTSEEN = .TRUE. only for points P which are now within alignment tolerance of some other point Q which is eligible to point to P but is now pointing to some other point.

11. Calls DEL1 to delete all runs of inserted *or original* points which are "seen" but do not "point" and are not "pointed to" and are bounded in both directions by points which are pointing or pointed to.

12. Calls DEL3 to delete all consecutive one-way connected points which point to the same point.

13. Calls CHECK_ALIGNMENT to sort points along each feature so that consecutive point pairs do not "cross pointers" near to aligned junctions. See figure 6 to see why this is sometimes necessary.

14. Calls MARK_NODES to mark every point as "at a node" when it is connected and it is at a discontinuity in connected-to line index or connected-to line contiguity.

15. Calls MOV2 to replace the coordinates of each pointing point with those of the point it is pointing to.

16. Calls L2A to convert closed features back from circular linked lists to non circular linked lists with end point equal to start point.

17. Calls BREAK_FTRS to break features into separate part-features at points marked as at nodes, so that aligned part-features should now have identical coordinates all the way along.

7. LINKNODE - see above for details
   Sets up link-node structure, from features aligned and broken by LLJOIN

8. SORT_AND_MERGE - see above for details

9. REMOVE_2NODES
   Deletes all nodes with 2 arms which belong to the same original feature, unless this is the only node in a closed loop. Such nodes can arise during alignment when the start-end point of a closed loop is amongst a string of points which have been aligned with some other feature and then broken off from the original closed feature at junctions where the alignment ceases.

10. SET_BROKEN - see above for details

11. PUT_FTRS - see above for details

8.4 MERGE - MERGING DUPLICATE FEATURE SECTIONS INTO SINGLE FEATURES

This is exactly as for LLJOIN above, except that feature alignment is assumed to have already taken place. There is no tolerance qualifier with this process. The tolerance actually used is the "very small" positive real value COORD_TOL, as described above.

8.5 BREAK - BREAKING OF FEATURES WHERE THEY TOUCH OR CROSS

Steps are as follows:

1. GET_CMDS - see above for details
2. GET_LOOKUP_FILES - see above for details
3. SCAN_IFF - see above for details
4. ALLOCATE_MEMORY - see above for details
5. GET_DATA - see above for details
6. BREAK
    1. Calls QUADSCAN to apply CHKVEX2 to each point pair (indices (P,Q)) for all points selected for processing. CHKVEX2 inserts points into both vectors leading from given points whenever these vectors touch or cross, and the point to be inserted is not within COORD_TOL of a vector end.
    2. Calls BREAK_FTRS to break features into separate part-features at all points marked as junctions.
7. LINKNODE - see above for details
8. SET_BROKEN - see above for details
9. PUT_FTRS - see above for details

8.6 STRUCTURE - GENERATION OF LINK-NODE STRUCTURE

Steps are as follows:

1. GET_CMDS - see above for details
2. GET_LOOKUP_FILES - see above for details
3. SCAN_IFF - see above for details
4. ALLOCATE_MEMORY - see above for details
5. GET_DATA - see above for details
6. LINKNODE - see above for details
7. PUT_FTRS - see above for details

8.7 SORTARMS - SORTING OF NODE ARMS BY ORIENTATION WITHIN LINK-NODE STRUCTURE

Steps are as follows:

1. GET_CMDS - see above for details
2. SORT_ARMS
   Operates on the input IFF file in-situ to sort the arms in each node into order anti-clockwise from 3 o'clock, by calling the IFFLIB junction structure manipulation routine IFJSCN for each node sector, to apply the ILINK routine SORTARMS to each node in that node-sector.

figure 1

In the process of polyline alignment, any number of different polylines may come together to share some part in common. As each point pair P and Q which is within alignment tolerance and is otherwise eligible for alignment, is encountered and examined, effectively in random order, P is pointed to Q by setting NEAREST(P) = Q, or Q to P, or neither in such a way that in the end every point will be pointed to the nearest point with the lowest higher alignment priority. After all pointers have been set, each point is then pointed "through" to a point which points no further, with the result that points in the vicinity of junctions between three or more features are generally "threaded" togther successfully onto a single point, as in fig 1(a).

If each point had simply been pointed to the nearest eligible point with highest higher priority, then the final result for the case in fig 1(a) would have been as indicated in fig 1(b).

figure 2

Very short aligned sections, which occurr wherever lines cross, must be left alone.

figure 3

Points "seen" during alignment, but not involved in final alignment, must be removed in order to guarantee complete alignement - but not until after pointers have been pointed through to points which point no further.

figure 4

GROUND must be careful to point points through other points only as far as the furthest point which has been marked by MARK_ALIGN for alignment

figure 5

Points pointing to other points during alignment, but which are bounded in one or both directions by points along the same feature not pointing or pointed to, should be left alone, so that not more points than is desirable are dragged onto other features in the vicinity of junctions where alignment ceases.

figure 6

When the acute corner of one feature is aligned with some other feature, it is possible "for pointers to get crossed" in which case the resulting alignment will not be satisfactory as the corner will be pinched flat and "double digitising" will occurr. The routine CHECK_ALIGNMENT detects such cases by moving along each feature checking for local reversals in the point order along the feature being aligned onto. Where these are found, pointers are rearranged so as to eliminate these crossovers.

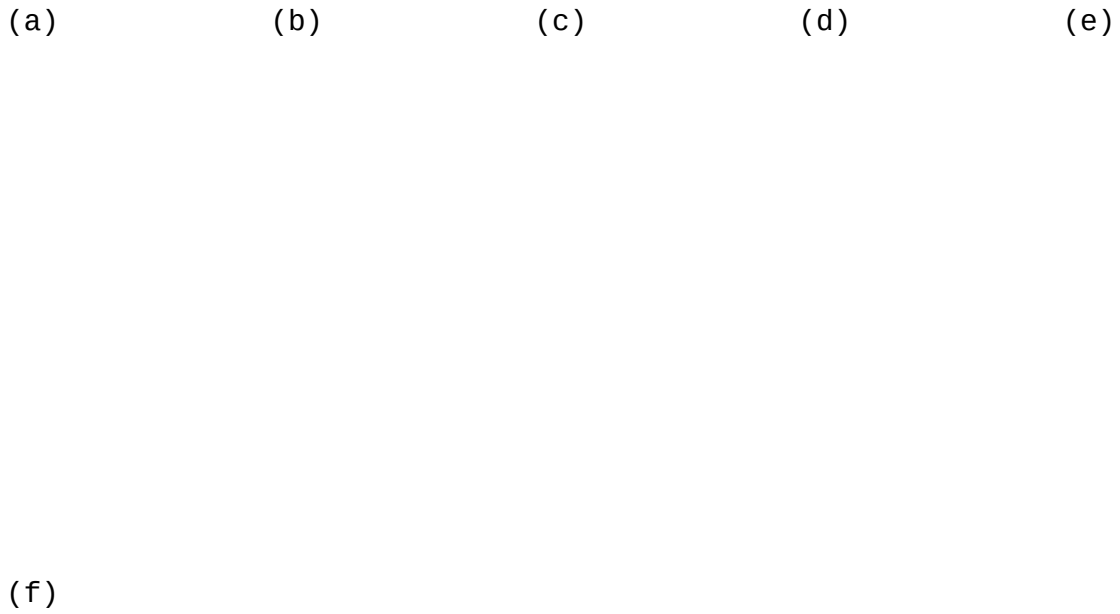

figure 7

LPJOIN may join polyline ends to polylines either by moving the end to the nearest original vertex within tolerance, or to the nearest position along the polyline, or to the position along the polyline which the end vector projects or truncates onto. Which is the case depends on the position of the end vector relative to the polyline it is being moved onto, the values of JNTOL and EXTOL, and whether or not the PROJECT qualifier was used.

Whichever case applies, the polyline end is joined to the other polyline simply by changing the position of the end point. There is no attempt to "propagate" this change in position along the polyline whose end is being moved. The cost of this simple rule is that it is possible for cases such as (b), (c) and (d) above to arise.