AB33.3.4 REPORT of the Subcommittee on Data Processing and Transput of IFIP Working Group 2.1, held at the Mathemtical Centre, Amsterdam, from August 9th - 11th 1971.

PRESENT: C. H. Lindsey - Convener;
C.H.A.Koster, Mathematical Centre, Amsterdam
A.J.Fox, Royal Radar Establishment, Malvern

Apologies for Absence from:

R.J.Gilinsky, who has now returned to CSI in Los Angeles, but who wishes to be kept in touch with future developments.

A letter had also been received from Phil Fites, now working for the Gas Co. in Edmonton, who wished to be kept up to date with the work of the Subcommittee.

PREVIOUS REPORT:

Items in the previous Report of the Subcommittee were considered. Items (a), (b), (c) and (d) have been passed over to the Improvements Subcommittee.

Item (e) was considered not entirely satisfactory in its present form, and a new proposal for conversion procedures to replace the present int string, dec string and real string is therefore substituted in its place, - see LQ195A.

A question of abbreviated formats more on the lines of the FORTRAN formats was discussed, but it was considered that proposal (h) could be used to include within a format another format identifier which the user had declared to possess the picture which he wished to be abbreviated, therefore no change was recommended.

The wording of proposal (n) was considered and it was decided that the initial contents of a book created by 'establish' should remain undefined, the implementor being trusted to fill it with spaces where appropriate. The last two lines of the proposal were therefore deleted.

Proposal (o) is still unsatisfactory in its present form, but it was not discussed further at this meeting.

The remaining proposals, therefore, still stand as recommendations to the Working Group.

A letter from Professor P.W.Abrahams of the Courant Institute, New York was considered. This pointed out several errors in chapter 10 of the Report. These have been noted by C.H.A.K. as errata for the next edition.

The remainder of the meeting was devoted to a discussion of record transfer, particularly with regard to random access devices and data bases. Most of the discussion centred round a proposal for a new kind of assignation with a new becomes-symbol (<<) the use of which would imply a transfer to some backing medium. The proposal is intended to provide compile time mode checking as is usual in the rest of ALGOL 68.

The proposal as it stands at the moment is described in LQ200, but it should be emphasised that further work remains to be done particularly in specifying how the user-preludes are to be written.

The Report of the Codasyl Data Base Task Group was considered briefly. The concept of data base key in that Report seems to be defined in a way which makes implementation very difficult. Apart from this, there seems no reason why a library prelude could not be written to access such data bases from ALGOL 68.

It is to be noted that the sub-schema of the Codasyl proposal would correspond to a user-prelude of an ALGOL 68 program. A language more akin to ALGOL 68 for describing such sub-schemata would be designed and used to generate such user-preludes.

LQ 195A Style of printing in formatless transput.

Proposal E in the first Report of the Data-Processing sub-committee (WG2.1 (Manchester 11) 151 - also in AB32) met with a somewhat lukewarm response from the Working Group, and we were not over pleased with it ourselves. Here is an alternative proposal for a group of procedures whole, fixed and float. The intention is that the six procedures int string, dec string, real string, string int, string dec and string real, together with their long versions should be removed from the Standard -prelude (they were not over-convenient to use, and had some funny propoerties (see Habay 15).).(By "remove", I of course mean "render in-accessible"). The three procedures proposed here are an adequate replacement for three of them, and proposal O should provide for the other three, which were not likely to be so useful anyway.

Of the three procedures given here, it is expected that fixed will be the most used, being adequate for most int as well as real transput. The other two are, however, provided for the sake of completeness.

```
proc float = (union(⊀ L real ⊁, ⊀ L int ⊁) value, int before, after, exp)string:
     begin string s, bool neg, int p := 0, shift := 0;
     ( ⊀ (L real x, y;
          if   y ::= value
          then if   exp ≠ 0
               then L real g = L 10 ↑ abs before; L real h = g * L .1;
                    x := abs y;
                    while x ≥ g do (x times L.1; shift plus 1);
                    if x>L0 then while x < h do (x times L10; shift minus 1) fi;
                    if (x plus L.5 * L.1 ↑abs after) ≥ g then x := h; shift plus 1 fi
               else x := abs y + L.5 * L.1 ↑abs after
               fi;
               neg := y<0;
               while x>1 or (p=0 and before ≠ 0) or (exp≠0 and p<abs before)
               do    (x times L.1; p plus 1);
               to p do s plus dig char ((int c = S entier (x times L10); x minus c;c
                         ¢ for dig char see 10.5.2.1 ¢
               if abs after ≠ 0 then s plus "." fi;
               to abs after do s plus dig char ( (int c = S entier (x times L10);
                                                  x minus c; c))
          fi ) ⊁,
    ⫷ (L int i, j;
       if j ::= value
       then if exp ≠ 0
            then L int g = L10 ↑ (abs before + abs after); L real h = g * L.1;
                 i := abs j;
                 shift := abs after;
                 if  i ≥ g
                 then real x := i;
                      while x ≥ g do (x times L.1; shift plus 1);
                      i := round x
                 fi;
                 p := - abs after;
                 if i > L0 then while i < h do (i times L10; shift minus 1)
                 elsf i = 0 then shift := 0 fi
            else i := abs j;
            p := - abs after;
            to abs after do
                 ("0" prus s; p plus 1; (p=0 | "." prus s))
            fi;
            neg := j < 0;
            while i ≥ 1 or (p=0 and before≠0) or (exp≠0 and p<abs before) do
                 (dig char(i mod L10) prus s; i overb L10;
                 p plus 1; (p=0 | "." prus s))
       fi ) ⫸ );
    if neg then "-" elsf before<0 or after<0 then "+" else "" fi prus s;
    to abs before - p do "␣" prus s;
    if exp ≠ 0 then s plus "⏨" plus float(shift, exp, 0, 0) fi;
    s
    end;

proc fixed = (union(⫷ L real ⫸, ⫷ L int ⫸)value, int before, after)string:
    float(value, before, after, 0);

proc whole = (union(⫷ L real ⫸, ⫷ L int ⫸) value, int before)string:
        fixed(value, before, 0);
```

Examples:

```
float( 2.718281828,  1,  4,  2);   ¢ yields "2.7183⏨.0" ¢
float(-2.718281828,  1,  4,  2);   ¢        "-2.7183⏨.0" ¢
float(-2.718281828, -1,  4,  1);   ¢        "-2.7183⏨0" ¢
float( 2.718281828, -1,  4,  1);   ¢        "+2.7183⏨0" ¢
float( 2.718281828,  0, -4, -1);   ¢        "+.2718⏨+1" ¢
float( 2.718281828,  2,  6,  0);   ¢        ".2.718282" ¢
float(          99,  3,  2, -1);   ¢        "990.00⏨-1" ¢
float(     9999999,  1,  5,  1);   ¢        "1.00000⏨7" ¢
fixed( 2.718281828,  2,  6);       ¢        ".2.718282" ¢
fixed( 2.718281828,  1,  7);       ¢        "2.7182818" ¢
fixed( 2.718281828,  0,  8);       ¢        "2.71828183" ¢
fixed(          99,  3,  5);       ¢        ".99.00000" ¢
fixed(     9999999,  6,  2);       ¢        "9999999.00" ¢
whole( 2.718281828,  9);           ¢        "........3" ¢
whole(          99,  9);           ¢        ".......99" ¢
whole(         -99,  9);           ¢        "......-99" ¢
whole(         -99, -8);           ¢        ".....-99" ¢
whole(          99, -8);           ¢        ".....+99" ¢
whole(           1,  1);           ¢        "1" ¢
whole(          12,  1);           ¢        "12" ¢
whole(         123,  1);           ¢        "123" ¢
whole(        1234,  1);           ¢        "1234" ¢
whole(           0,  0);           ¢        "" ¢
```

Thus it will be seen that the question of error procedures does not arise, since the procedures produce sensible string for all possible values of their parameters. If the user is worried that the string is longer than he expected, he can always inspect its upb.

LQ 200 Record transfer.

This paper comprises CHL's interpretation of what the Data-processing and Transput sub-committee was discussing between Aug 9th and Aug 11th 1971.

Reference is made to item P of the first Report of the sub-committee (manchester 11). The present proposal is intended to establish a firmer syntactic and semantic framework for the concept of "record transfer" established there.

<u>Environment.</u>

First, it is necessary to establish as environment in which values sent to a mass storage device by one program may be retrieved by another program.

```
super program : open symbol, super prelude, parallel symbol,
                       program list pack, close symbol.
super prelude : declaration prelude sequence.
```

2.2.4.2.b) The scope of a plain value is the super-program,

{The scopes of routines and names (except those generated by local generators in the super-prelude) are still, however, at most the program. The scope of mass pointers (to be introduced presently) will be the super-program.}

2.3.a) The elaboration of a super-program is . . . .

The super-prelude is intended to contain declarations of internal objects that are accessible to all programs (the date, for example). Perhaps chainbfile (10.5.1.1.c) and the <u>bfile</u>s accessible from it should be kept there rathe than in the standard-prelude as at present. It may well contain <u>sema</u>s to control simultaneous access of <u>bfile</u>s by different programs, and routines to represent those facilities of the operating system which programs are allowed to see.

Note that each program still has its own private copy of the standard-prelude (including its own last randon, for example). In any case, some of the programs may not even be written in ALGOL68.

The programs in the program-list-pack include all the programs that ever have been, or ever will be, run on the particular installation. Although they are all elaborated collaterally, one may envisage that some little gremlins called operators can influence the order of their elaboration by inserting new programs in the list from time to time (whereupon their elaboration may commence) and by removing programs whose elaboration has been completed, or permanently interrupted. Note that there is no super-postlude, because it is not envisaged that the

<u>Transferable modes.</u>

First, a new class of modes will be defined:

TYPE : PLAIN ; format ; PROCEDURE ; reference to MODE ; pointer to RECORD.

where 'RECORD' is the subset of 'MODE' which includes the modes of those values which it is to be allowable to transfer, and which is defined thus:

```
RECORD : MOODREC ; UNITEDREC.
MOODREC : TYPEREC ; STOWEDREC.
TYPEREC : PLAIN ; pointer to RECORD.
STOWEDREC : structured with RECFIELDS ; row of RECORD.
                        {or ROWS of RECORD, following AB 32.2.3.(5)}
RECFIELDS : RECFIELD ; RECFIELDS and RECFIELD.
RECFIELD : RECORD field TAG.
UNITEDREC : union of LMOODRECS MOODREC mode.
LMOODRECS : LMOODREC ; LMOODRECS LMOODREC.
LMOODREC : MOODREC and.
```

Note that 'RECORD' includes all the modes from which <u>outtype</u> (10.5.0.1.b) is united, plus the modes beginning with 'pointer to'.

<u>Mass pointers.</u>

2.2.2.1) Any "name" or "mass pointer", except "nil", refers to one instance of another value. . . . .

2.2.3.5. Names and mass pointers
throughout 2.2.3.5. ≥ name -> name (mass pointer)≥

2.2.4.1.g) The mode of a name (mass pointer) is 'reference to' ('pointer to') followed by another mode; if the name (mass pointer) is not nil, then . . . .

2.2.4.2.b add:
that {i.e. the scope} of a mass pointer is the super-program.

8.5.1.1.a) MODE generator : MODE local generator ; MODE global generator ;
MODE mass generator.

8.5.1.1.d) pointer to MODE mass generator : mass symbol, actual MODE generator.

4.4.?. Context condition

No proper program contains a particular-program which contains a mass-generator {Thus mass-generators are only to be found in the super-, standard-, library- or user-preludes (see LQ 188A for user-preludes) where they are to be regarded, at least for the time being, as a definitional trick.}

Discussion.

An "area" encompasses those parts of the available mass storage media whose mass pointers can be yielded by the mass-generators contained within one user-prelude. It is envisaged that to each area there corresponds a user-prelude, the inclusion of which within the user's program makes available to that program the area plus, possibly, a set of routines etc to facilitate access to it. The modes of the values that may be stored in a given area are thus limited to those provided for by the mass-generators contained in the corresponding user-prelude, and there is no way provided of transferring values of any other mode into that area.

The intended implementation of mass pointers can now be discussed. A mass pointer has three fields:

area	logical address	displacement

The area field is necessary because a given mass pointer variable may be made to refer to a value stored in any area known to the program.

The "logical address" field is to be used to retrieve the record referred to from wherever on the part of the mass storage device encompassed by the area it has been put. The correspondence between this logical address and the "physical address" on the mass storage device is a matter for the implementor. In general, some conversion algorithm must be invoked. A given implementor may have a repertoire of several such conversion algorithms up his sleeve.

The "displacement" field is to be used to address the start of a component (2.2.2.k) within the record (see modified 2.2.3.5.b,c).

The implementor can choose an algorithm for logical address conversion from a knowledge of the modes permitted for the area (each area, in general, has a different algorithm). The only constraint on his algorithms is that they do not pre-suppose a logical address field longer than the one he proposes to put in his mass pointers. Here are some examples of possible algorithms. In these examples, I use the term "bucket" to mean the unit of data (whose size may either be variable or constrained by the hardware) which is transferred to or from the mass storage device by one hardware instruction. This term is used by ICL and RCA. The corresponding term in other hardwares might be "block" or "trackful" or "1/nth of a trackful".

Examples of conversion algorithms.

1) The only mode of the area is struct([1:10]char name, [1:50]char address)

Here all the records are short (compared with the likely bucket size) and of the same fixed length. Therefore the logical address could be a simple integer which, when divided by the known number of records in a bucket, would give the bucket address.

2) The only mode of the area is [1:50000]compl

Here the record is long, but it (and more importantly its elements) are of the same fixed length. Presumably, the user is not going to call all of it into core at once but, given a mass pointer to a sensible slice of it, the implementor can again do a simple arithmetic transformation on the logical address to obtain the physical address of the required bucket(s).

3) The modes of the area are struct([1:10]char name, [1:50]char address)
and struct(int number, [1:20]char text)

or alternatively the mode is struct(string name, [1:0flex]int payments)

Here the record is presumably shorter than a bucket (the odd one which is not might have to be processed by a special exception routine), but its length is not fixed.Any simple arithmetic transformation will result in wasted space in the first case and be quite impossible in the second. A good algorithm must be capable of packing as many records as possible into the available area space on the mass storage device, having regard to the actual sizes of the actual records being stored at that moment.

Packing of variable length records into a fixed area size is therefore not a trivial problem - but its solution is of the utmost importance to the data-processing community and, hopefully, ALGOL68 can provide a better environment for it than some other programming languages. Clearly, if the worst comes to the worst, the logical address can be transformed into the physical one by means of an index look-up. However, this will require an extra transfer from the mass device each time a record is accessed (since, in general, such an index would be too large to hold all of it in core). Something better is hoped for and the following algorithm, based on a piece of ICL software, is suggested.

It is assumed that a bucket will hold several records of average size (if the occasional record extends over 1 bucket, a special exception routine would have to be called in). It is assumed also that the records contained in the area are more or less uniformly distributed amongst the available buckets, and that the total spac occupied by records still leaves a little free space in the area (10% say). All the records contained in one bucket are compacted towards one end.

The logical address is divided into two fields - the bucket address and the record number within the bucket. The logical address therefore immediately gives the hardware address where the record referred to is expected to be found and, by counting through the records of the bucket (or looking up an index at the start of the bucket), the record can be retrieved. Note that this search can be done rapidly in core after the whole bucket has been transferred.

Now suppose that one of the records in the bucket is rewritten with increased size, or a new record is to be added. If the free space at the end of the bucket is adequate for the extra information, no problem arises. The records are recompacted within the bucket (and the bucket index, if any, is updated). Otherwise, the lastrecord of the bucket is removed to some other bucket, where free space is available, and a pointer to its new position is left in its original place (thus the removed record can still be accessed from its original mass pointer, albeit with an extra mass device transfer).

Hopefully, while reasonable amounts of free space remain, the vast majority of records will actually be contained within the buckets specified directly by their mass pointers. Eventually, however, as new records are added and old ones altered, the area will become full of overflow pointers and the efficiency will drop below an acceptable level. At this point, an off-line program must be called in to re-organise the area, possibly copying it to a larger area and redistributing the records uniformly. Now, of course, the mass pointer values referring to each record have changed but, since the system knows the modes of this area, and of other areas which contain mass pointers pointing into this one, it knows where all the existing instances of mass pointer values that need changing are to be found.

Note that mass pointer values themselves are not included in outtype, so that the user is unable to keep secret copies of mass pointers beyond the life of his program.

This particular algorithm is, of course, only a suggestion, designed to establish at least the existence of algorithms suited to this purpose.

4) The mode of the area is struct([1:50000]compl a, [1:50000flex]real b)

Here the record is both long and variable. The implementor has got problems.

<u>Databases.</u>

A set of areas, none of which contains mass pointers referring to values contained in areas outside that set, is known as a "database".

A proper program must not contain two user-preludes specifying areas that may contain mass pointers unless those two areas are known by the system to belong to the same database (or unless one of them is a "scratch" area to be discarded after elaboration of the program).

<u>Ordering of mass pointers.</u>

It is the case with presently available random access devices that two consecutive transfers to physical addresses that are close together in some sense (e.g. within the same track, or within the same cylinder) take much less time (by a factor of 10, possibly) than two consecutive transfers to widely spaced physical addresses, where head movement will be involved. It is vital, therefore, that the user should be given the opportunity of organising his access requests so as to take advantage of this situation.

Therefore, an ordering of mass pointers must be defined, the intention being that the ordering of logical addresses should correspond to the ordering of the physical addresses, so far as is possible (this is true, for example, of the algorithm suggested above, provided that the area does not get too disorganised). Thus the user, with a list of the records that he needs to access, can first sort his requests on the basis of their mass pointer values. Then he can retrieve them all in one clean sweep across his disc. If several records are stored to a bucket, he will even get away with fewer mass device transfers than he has records to retrieve.

2.2.3.5.e) For each pair of mass pointers {of not necessarily the same mode} the relationship "to be smaller than" is defined. {However, it had perhaps better not be defined that two different programs, when faced with the same two mass pointers, should be obliged to take the same view of this relationship. This is because the area field of a mass pointer is likely to contain a coded or compressed version of the identification of the area, and different programs, faced with a different subset of the areas from the database, might like to use a different coding for the areas.}

10.2.12.a) <u>mode</u> %<u>maps</u> = <u>c</u> an actual-declarer specifying a mode united from all modes beginning with 'pointer to' <u>c</u> ;

b) <u>op</u> < = (<u>maps</u> a, b)<u>bool</u>: <u>c</u> true (false) if the mass pointer which is the value of 'a' is (is not) smaller than that of 'b' <u>c</u> ;

c) op $\leq$ = (maps a, b)bool: not(b<a);

d) op $\geq$ = (maps a, b)bool: b$\leq$a;

e) op > = (maps a, b)bool: b<a;

{The fact that, of two mass pointers, neither is found to be less than the other does not mean to say that they are the same mass pointer. It merely implies that your implementor considers them to be so close together that it is not worth while trying to order them further (for example, they might refer to different records but within the same bucket). As an extreme case, an implementor who did not care to have his mass pointers ordered at all could define the relationship "to be smaller than" to be false for all possible pairs of mass pointers.

map declarers.

REFPOINT : reference to ; pointer to.

7.1.1.l,m,n (throughout) {reference to MODE declarators}
$\gtrsim$ reference to $\rightarrow$ REFPOINT $\gtrsim$

3.1.1.d) pointer to symbol map

9.c.Γ) add : or transfer symbol {allowing extension of ref m m = loc m $\ll$ skip}

Transfers.

8.3.0.1.a) MODE confrontation : MODE assignation ; MODE conformity relation ;
MODE identity relation ; MODE cast ; MODE transfer.

8.3.5.1. Syntax

a) reference to RECORD transfer : reference to RECORD destination,
transfer symbol, pointer to RECORD source.

b) pointer to RECORD transfer : pointer to RECORD destination,
transfer symbol, RECORD source.

c) pointer to RECORD destination : firm pointer to RECORD tertiary.

3.1.1.c) transfer symbol $\ll$

{In which case, < must not be allowed as a monadic operator.}

8.3.5.2. Semantics

A transfer is elaborated in the following steps:

Step 1: Its destination and source are elaborated collaterally;

Step 2: If the mode enveloped by the original of the transfer begins with 'reference to' {inward transfer} then the value referred to by {the mass pointer which is} the value of its destination is assigned to {the name which is}the value of its source; otherwise, {outward transfer} the value

Throughout 8.3.1.2.c, replace all relevant occurrences of "name" by "name (mass pointer)".

Discussion.

Clearly, the basic transfer operation could have been specified in other ways. For example:

1) Invent a demapping coercion, and use an ordinary assignation. This would certainly be possible syntactically but, since transfers to backing store are likely to be 10000 times as slow as assignations within core, it was thought better to force the user to write an explicit transfer-symbol, so that he could be under no illusions as to what he was asking for.

2) Use a different symbol for inwards and outwards transfers (e.g. ≪ and ≫). However, this would mean that for one direction of transfer the source would be on the left and the destination on the right, and the analogy with assignations would be lost. What, moreover, would be the value yielded by the transfer as a whole when its source was on the left?

3) Define ≪ (and/or ≫) as an operator.

In the reach of	<u>ref</u> <u>real</u> rx = <u>skip</u>, <u>map</u> <u>real</u> px = <u>skip</u>;
then as proposed	rx ≪ px ≪ 2.0
means	rx ≪ (px ≪ 2.0)
or	px ≪ 2.0; rx ≪ px

whereas, with ≪ defined as an operator

it would mean	(rx ≪ px) ≪ 2.0
or	rx ≪ px; ?? ≪ 2.0

which is obviously why the becomes-symbol is not an operator in the present language

An operator ≫ with the destination always on the right could sensibly be defined, but again the analogy with assignations is lost.

Identity relations and nil.

8.3.3.1.a (throughout) {identity-relations}
≥ reference to → REFPOINT ≥

8.3.3.2.Step2+1 ≥ names → names or mass pointers ≥

8.2.7.1.d {nihils}
≥ reference to → REFPOINT ≥

{Thus mass pointers may be compared using identity-relations, and a good mass pointer value is <u>nil</u>.}

adopted (Manchester 11.C), then the program should be able to handle pointers to modals as well as references to them.

Mass pointers to components.

REFPOINTETY : REFPOINT ; EMPTY.

8.5.2.1.a (throughout),	{selections}
8.6.1.1.a (throughout),	{slices}
LQ 181.8.2.9.1.a,b,c,d,e	{decompositions}

≥ REFETY -> REFPOINTETY ≥

Obvious changes in the associated semantics.

{I resisted the temptation to do the same trick in 8.6.2.1 (rowed coercends).}

Thus mass pointers to components of values already on backing store may be obtained. Whether such mass pointers should be unassignable in the sense of LQ 181 is open to debate. Implementation would be simpler if they were (and indeed, if they refer to components of flex values on the disc, or if they arise via LQ 181 (decomposition), then they will be unassignable anyway).

Example.

A user wants to construct an area containing records which can be retrieved on the basis of a string key. The area must also therefore contain an index. First he must write a user-prelude which will achieve the same effects as the following:

```
mode data = string;  ¢ say ¢
mode %indexline = struct ( string key, map indexline more, less, map data item )

map indexline %firstindex = mass indexline << ("", nil, nil, nil);
    ¢ this line should perhaps really have been in the super-prelude, since
      the intention is that it should be elaborated only once when the area
      is first created ¢

proc seek = (string key) map data:
    begin
    map indexline indexptr := firstindex; indexline copyindex;
    map data dataptr;
    int moreless := +1;
```

```
  repeat: if   (map indexline ex indexptr) isnt nil
          then copyindex << indexptr;
               moreless := (key <key of copyindex |-1 |: key =key of copyindex|0|+1);
               indexptr := (moreless+2 |less of copyindex,
                                        goto found,
                                        more of copyindex);
               goto repeat;
     found: item of copyindex
          else (moreless+2 | less of indexptr, skip, more of indexptr)
                    << mass indexline
                         << (key, nil, nil, dataptr := mass data);
               dataptr
          fi
          end;

     file f; open(f, "identification", dischannel);
          ¢ this is intended to associate the area with a file for administrative
            purposes; in particular, it makes available a set of error procedures
            for use when exception conditions arise (such as no more mass pointers
            available within the area)¢
```

Note that this user-prelude contains mass-generators for the modes data and indexline, and these are therefore the modes of the area.

Within the context of this user-prelude, the user may now write a particular-program such as the following:

```
     begin
     seek("LINDSEY") << "UNIVERSITY.OF.MANCHESTER";
     seek("KOSTER") << "MATHEMATICAL.CENTRE";
     proc address = (string name)string:
          (string s << seek(name); s);
     print(address("KOSTER"));
     seek("KOSTER") << "MATHEMATISCH.CENTRUM";
     print(address("KOSTER"))
     end
```

Discussion of example.

It will be observed that the particular-program cannot generate mass pointers for itself. It must employ the procedure seek for this purpose. Since seek cannot yield mass pointers of mode map indexline, it also follows that the particular-program cannot on its own overwrite the index, or attempt to generate new index entries. It cannot even inspect the existing index for itself because the identifier firstindex is secret. Therefore the area must always be in the state that every index entry leads to a data record, and there exists no data record that is not properly linked in to the index; the particular-program can never cause the area to cease to be in this desirable state.

It will be observed, however, that this user-prelude gives no guarantee that any particular record will be at any particular physical location in the file (the efficiency would, for example, have been much improved if it could have been arranged that all the indexline records were stored close to one another). Also, there is no means provided for removing a data record and its corresponding index entry from the area. What this indicates is that a mass-generator is really too crude a tool for generating mass pointers in real life situations, although as a definitional tool it is satisfactory.

What a particular user would like to see would be routines in his user-prelude for yielding a generated mass pointer close to some other mass pointer, or at a given percentage of the way through the area, or calculated by a hash-coding technique from some key, or immediately adjacent to the last mass pointer that was generated. Likewise he might like routines for giving him all the mass pointers (of some given mode) within the area in turn, or for returning some no-longer-required mass pointer to the pool (it is to be noted than conventional garbage collection is a grossly inefficient way of recovering no-longer-required mass pointers on random access devices - far better for the user to say when a record is no longer required, and let him write his own garbage collector if he really cannot think of a better method).

As things stand, it is up to the implementor to provide the means to include such routines within a user-prelude. Probably he can provide some helpful routines in his library-prelude and even, if it can be agreed what facilities should be generally available, in the standard-prelude. This is an area of the present proposals which has been deliberately left vague at the moment and in which considerable further thought is needed.

It should be noted that routines such as those mentioned can easily be defined at present, but not in a way in which one would like to implement them. For example, one simply generates a vast stock of mass pointers in the super-prelude and as required.

It will be pbserved that the user-prelude in the example contained one line that ought really to have been in the super-prelude, except that one feels the definition of the identifier which was to possess the initial mass pointer of the index was a matter for the user and not for the super-prelude writer. This matter also will have to be considered further, as will the relationship between _files_ and areas and the treatment of exception conditions.

Please do not therefore take the user-prelude in this example too literally. Its purpose is to illustrate the facilities that are required, rather than to sugges a seqqence of symbols that an actual user might actually write down.