PART 3 PROGRAM STRUCTURE AND SYSTEM FEATURES

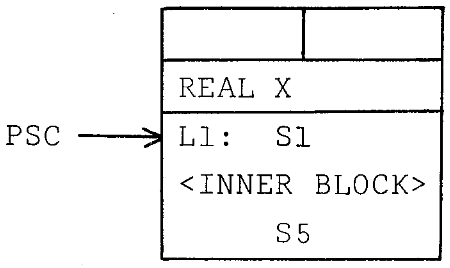
1 PROGRAM STRUCTURE

Without the CLASS concept, the run time structure of a SIMULA program is essentially that of a stack of blocks with the last created block "active". Given the program skeleton

```
BEGIN  REAL X;
L1:    S1;
       BEGIN  BOOLEAN B;
              S2;
              BEGIN  TEXT X;
L3:                  S3;
              END;
L2:           S4;
       END;
       S5;
END
```

(where the S's denote statement sequences) then the structures at run time are

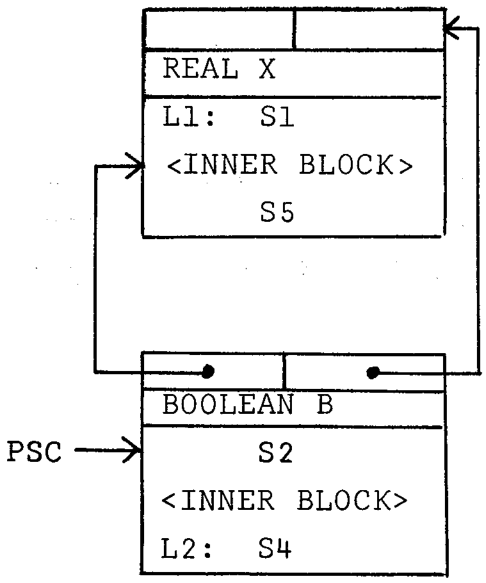
1. during execution of S1 or S5



PSC denotes the current statement

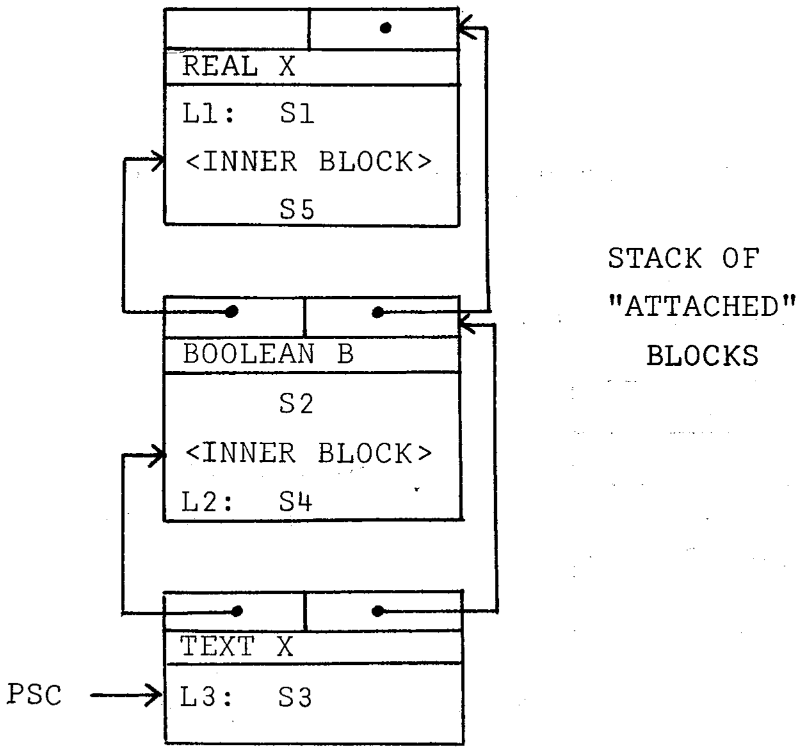
Only one variable REAL X exists.

2. during execution of S2 (or S4)



Two variables REAL X and BOOLEAN B exist.

3. during execution of S3



Three variables exist REAL X, BOOLEAN B, TEXT X but only two are accessible by actions in the active block, for the REAL X of the outermost block has become inaccessible due to an identifier clash with TEXT X of the innermost block.

Note that besides exiting from a block via its final END, it is also possible to exit via a GOTO-statement. If the statement sequence S3 includes the statement

```
GOTO L2
```

then on execution of that statement, the innermost block is deleted (the variable TEXT X will no longer exist).

If the statement sequence S3 includes the statement

```
GOTO L1
```

then both the inner blocks will be deleted when that statement is executed and both BOOLEAN B and TEXT X will no longer exist.

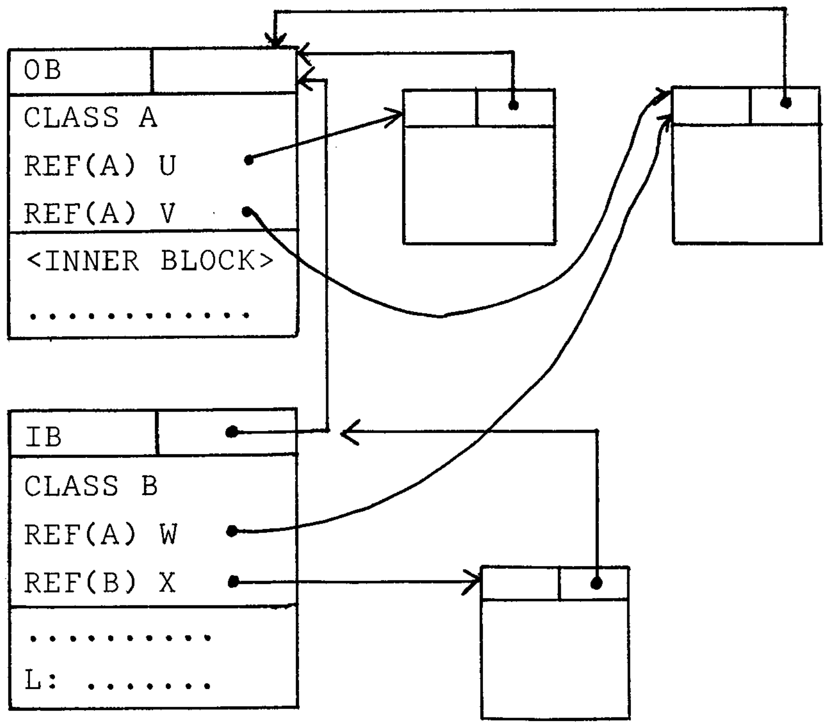
The CLASS concept initially gives the opportunity for data structures to exist in parallel. The program

```
     BEGIN  CLASS A;..........;
            REF(A)U,V;
            U :- NEW A;
            BEGIN  CLASS B;..........;
                   REF(B)X;
                   REF(A)W;
                   V :- W :- NEW A;
                   X :- NEW B;
            L:     ..........
            END ***INNER BLOCK*** ;
            ..........
     END ***PROGRAM*** ;
```

has the representation below at the label L.

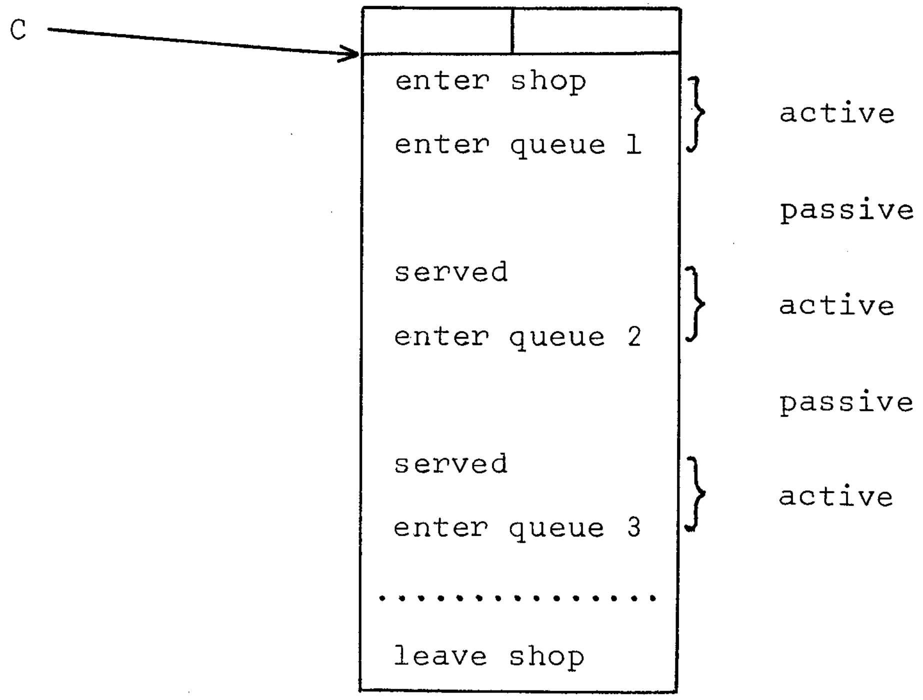
OB: outer block

IB: inner block



When the inner block is deleted, the second A object is still available as it is referenced by V. On the other hand, the B object must be deleted as its declaration and therefore all its references do not exist.

At this level, the actions of each object are executed until exhaustion and then the object is left as an attribute structure in the terminated state. But there is a broad class of problems which cannot be modelled by this mechanism, e.g. when the actions are executed in phases corresponding to the actual concept represented being active or passive. For example, a customer in a shop goes through the stages of queuing (passive) and buying (active), players in a game of cards play, and then are passive until their turn comes again. We could picture a customer object, C, by:



When C is passive its actions may be made active again by a call RESUME(C). The actions of C are resumed from where they were left off last. To mark off this program point objects can be made into program components supplied with a LOCAL SEQUENCE CONTROL (LSC) which marks the current stage of execution of their actions.

A very simple example is that of two players playing a game. Their actions may be loosely described by

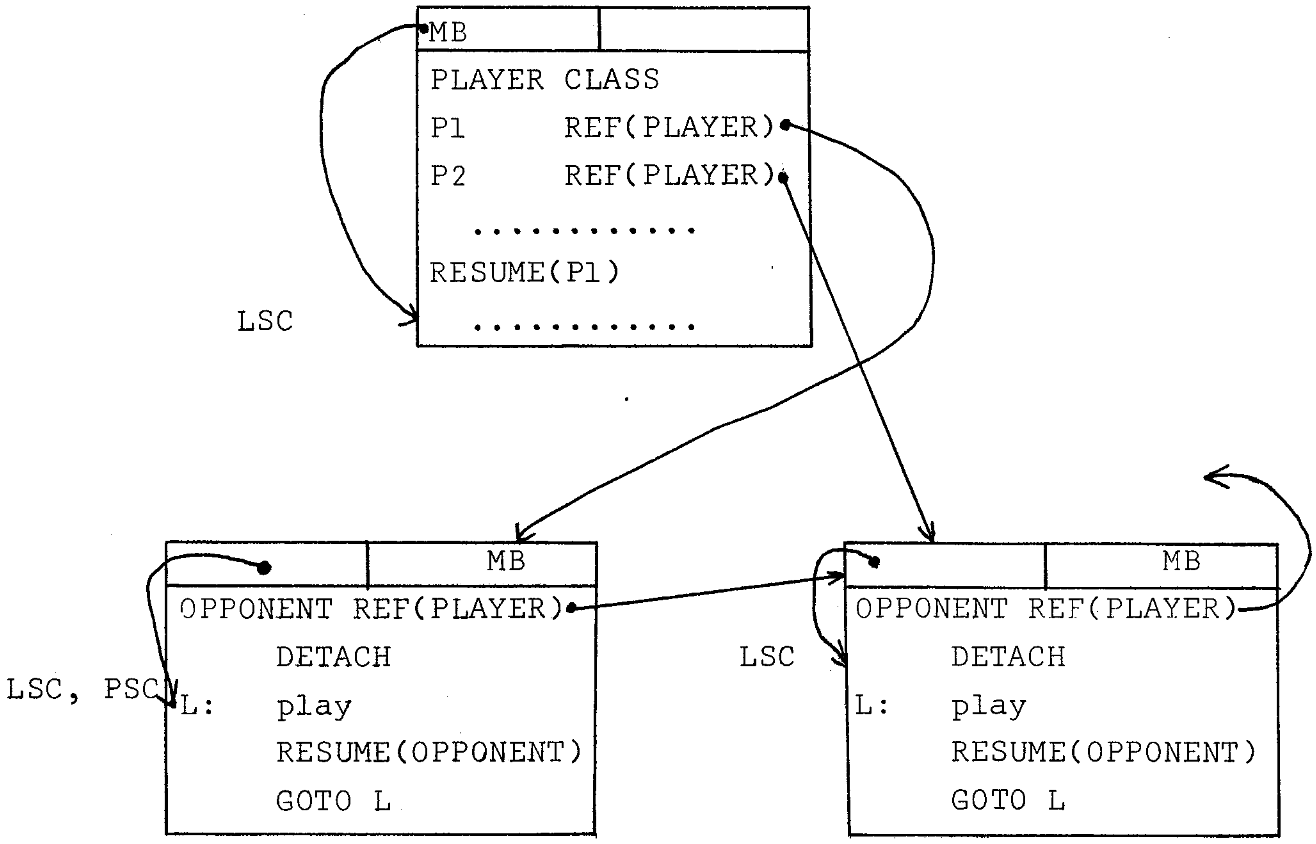
```
         ..........
     L:  play;
         RESUME(opponent);
         GOTO L
```

When the first player is generated, we do not wish to execute these actions as the opponent is not yet generated. So we have to return control to the main block and return a reference to this player, and then create the second player. A second system PROCEDURE DETACH serves this purpose. On meeting DETACH, the object becomes a system component with an LSC referencing the next statement, control is returned to its object generator and with it a reference to the object.

The outline of a simple program for a two man game is:

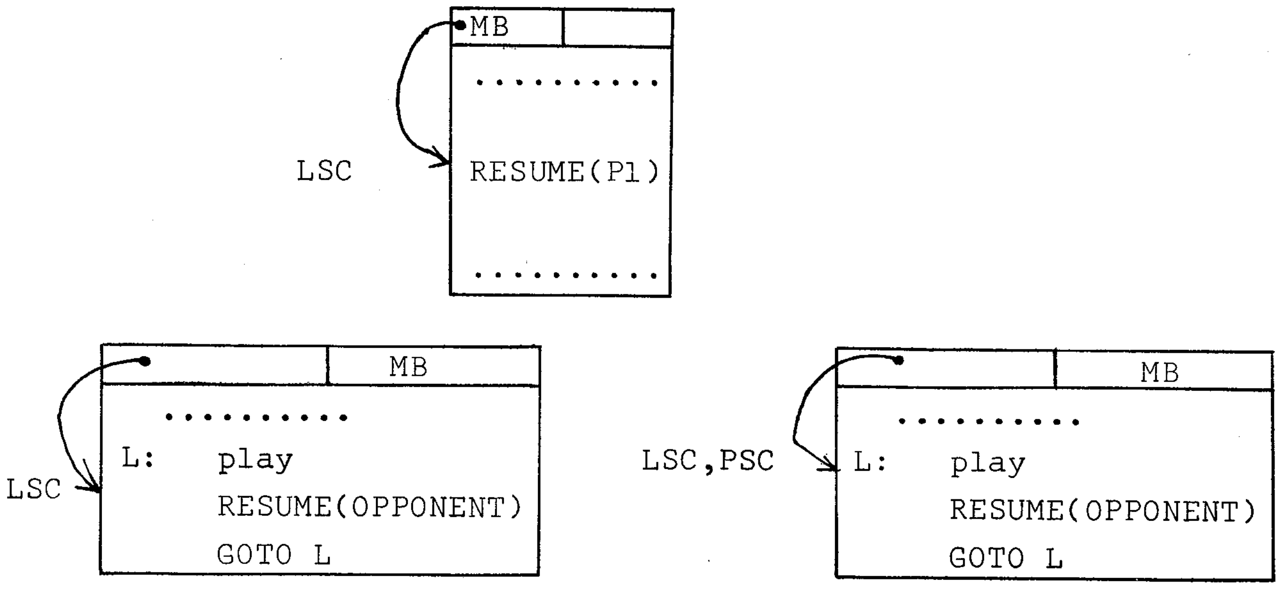
```
BEGIN  CLASS PLAYER;
       BEGIN  REF(PLAYER) OPPONENT;
              DETACH;
       L:     play;
              RESUME(OPPONENT);
              GOTO L
       END;
       REF(PLAYER)P1,P2;
       P1 :- NEW PLAYER;
       P2 :- NEW PLAYER;
       P1.OPPONENT :- P2;  P2.OPPONENT :- P1;
       RESUME(P1);
       ..........
END
```

A snapshot at the point where RESUME(P1) has just been executed is:



The PSC coincides with the LSC of the currently active component.

After P1 has played, P2 is resumed and the new snapshot is:



Note that there are three components here with LSC's: the two players and the program block. The current state of the two objects is "detached" - when they are program components with LSC's. When their actions are exhausted, they lose their LSC's and become "terminated". Control returns to the main program block and continues from its LSC. Only a "detached" object may be resumed.

One question remains: how to transfer control back to the (unreferenced) main program block without terminating an object. This is achieved by a further call on DETACH.

In addition to sequencing PROCEDURES, "DETACH" and "RESUME", there is the PROCEDURE CALL which has one reference parameter which must be a reference to a detached object.

The execution of CALL(Y) from within a block X, will "attach" the detached object Y to X and continue execution of the actions of Y.

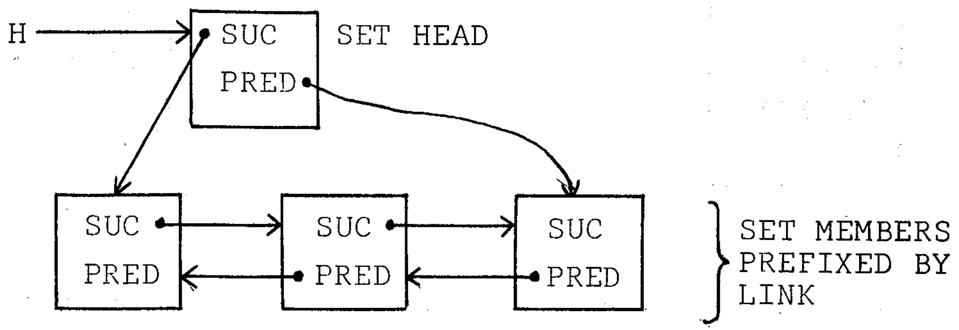
The detailed description of program sequencing given in the "67 Common Base Language" is not repeated here. Further enquiries are directed to that document §9.

2 THE SYSTEM CLASS SIMSET

List processing concepts and list manipulating procedures are declared in the system classes within the CLASS SIMSET. SIMSET can be used as a block prefix or as a prefix to another CLASS at one and only one block level in a program.

In SIMSET are provided the concepts for manipulating two-way lists called "sets". Besides the set members which carry information and are prefixed by LINK, a set also has a HEAD which has attributes giving global information about the set (e.g. how many members it has).

A set is organised on the basis of references SUC and PRED which are common to LINK and HEAD.



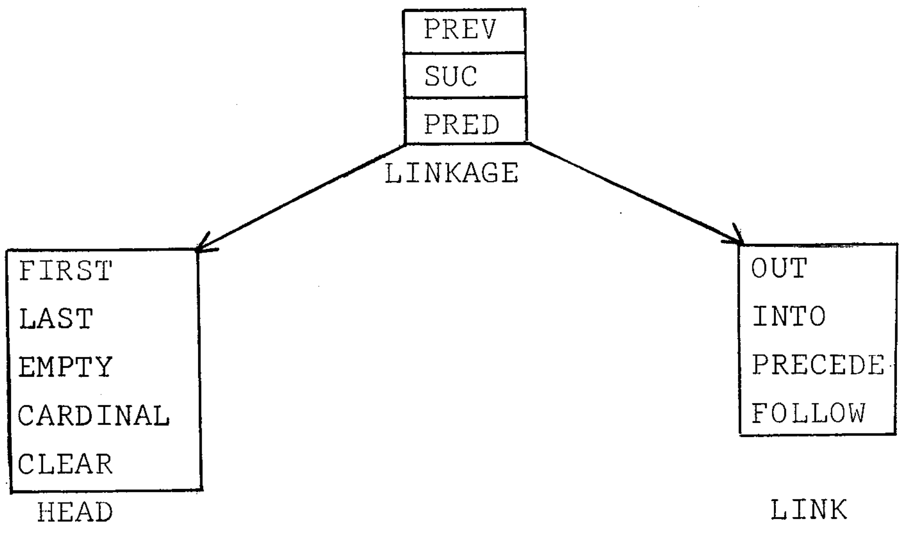
To protect the user from certain kinds of error, SUC and PRED are made REF(LINK) PROCEDURES and so may not be assigned to.

The part common to both HEAD and LINK is declared separately in CLASS LINKAGE which is then used as a prefix to HEAD and LINK.

A skeleton of the CLASS SIMSET is thus

```
CLASS SIMSET;
BEGIN  CLASS LINKAGE...........;
       LINKAGE CLASS LINK.......;
       LINKAGE CLASS HEAD.......;
END ***SIMSET***
```

This hierarchy may be pictorially represented by:



in which the procedures local to each of the classes are denoted by their identifiers.

An outline of the individual classes is now given containing the procedure-headings and a prose description of their actions. Throughout the prose descriptions which are illustrated by representative calls, we assume that the declarations

```
REF(HEAD)HD;
REF(LINK)LK;
REF(LINKAGE)LG;
```

are valid.

CLASS LINKAGE

```
CLASS LINKAGE;
BEGIN  REF(LINK) PROCEDURE SUC;.....;
       REF(LINK) PROCEDURE PRED;....;
       REF(LINKAGE) PROCEDURE PREV;.;
END ***OF LINKAGE*** ;
```

REF(LINK) PROCEDURE SUC;.....;

LK.SUC	returns a reference to the succeeding set member if LK is in a set, and LK is not the last member of the set, otherwise it returns NONE.
HD.SUC	returns a reference to the first set member if the set is not empty, otherwise NONE.

REF(LINK) PROCEDURE PRED;....;

LK.PRED	returns a reference to the preceding set member if LK is in a set and LK is not the first member of the set, otherwise NONE.
HD.PRED	returns a reference to the last set member if the set is not empty, otherwise NONE.

REF(LINKAGE) PROCEDURE PREV;....;

LK.PREV	returns NONE if LK is not in a set, a reference to the set head if LK is first member of a set, otherwise it returns a reference to LK's predecessor.

HD. PREV	returns a reference to HD if HD is empty, otherwise a reference to the last member of the set with head HD.

Note: by following PREV it is possible to give a reference to the head of a set in which a LINK object is a member, as shown in the following procedure (not local to LINKAGE, LINK or HEAD).

```
REF(HEAD) PROCEDURE THESETHEADOF(LK); REF(LINK)LK;
BEGIN  REF(LINKAGE)X;
       IF LK =/= NONE THEN
       BEGIN  X :- LK.PREV;
              WHILE X IN LINK DO
                    X :- X.PREV;
              THESETHEADOF :- X;
       END;
END ***THESETHEADOF***
```

CLASS HEAD

```
          LINKAGE CLASS HEAD;
          BEGIN  PROCEDURE CLEAR;................;
                 REF(LINK) PROCEDURE FIRST;......;
                 REF(LINK) PROCEDURE LAST;.......;
                 BOOLEAN PROCEDURE EMPTY;........;
                 INTEGER PROCEDURE CARDINAL;.....;
          END ***OF HEAD***
```

REF(LINK) PROCEDURE FIRST;.....;

HD.FIRST	is equivalent to HD.SUC

REF(LINK) PROCEDURE LAST;.....;

HD.LAST	is equivalent to HD.PRED

BOOLEAN PROCEDURE EMPTY;.....;

HD.EMPTY	returns TRUE if HD references a set with no members, FALSE if HD references a set with one or more members.

INTEGER PROCEDURE CARDINAL;.....;

HD.CARDINAL	returns how many members the set HD contains (0 if HD is empty).

PROCEDURE CLEAR;......;

HD.CLEAR	removes all members from the set, making it empty.

CLASS LINK

```
          LINKAGE CLASS LINK;
          BEGIN  PROCEDURE OUT;..........;
                 PROCEDURE INTO(H); REF(HEAD)H;..........;
                 PROCEDURE PRECEDE(X); REF(LINKAGE)X;....;
                 PROCEDURE FOLLOW(X); REF(LINKAGE)X;.....;
          END ***OF LINK***
```

PROCEDURE OUT;.....;

LK.OUT	removes LK from a set and re-establishes the SUC, PRED connections between its previously neighbouring members. If LK was not a set member, no action is taken.

PROCEDURE INTO(H); REF(HEAD)H;......;

LK.INTO(HD)	LK.OUT is called first. If HD == NONE no action is taken. If HD =/= NONE, LK goes into the set HD as the new last member.

PROCEDURE PRECEDE(X); REF(LINKAGE)X;......;

LK.PRECEDE(LG)	LK.OUT is called first. If LG == NONE or is not in a set then no action is taken, otherwise LK goes into the same set as LG as the new LG.PRED (LG may reference either a HEAD or a LINK object).

PROCEDURE FOLLOW(X); REF(LINKAGE)X;......;

LK.FOLLOW(LG)	as PRECEDE except that LK becomes the new LG.SUC

The use of reference variables and the fact that an object of any CLASS inner to CLASS LINK may be inserted in a set give the following desirable features:

1) ordered sets can be manipulated by efficient standard procedures

2) both the successor and predecessor of a LINK object are immediately accessible

3) the set members can be objects of different classes.

Note that a LINK object can only be in one set at a time.

Example on the use of SIMSET

The example is concerned with the dealing of a hand of cards, 13 cards each in rotation, to a table of four players. It shows how the concepts HEAD and LINK of SIMSET may be used as a platform upon which to build more general concepts.

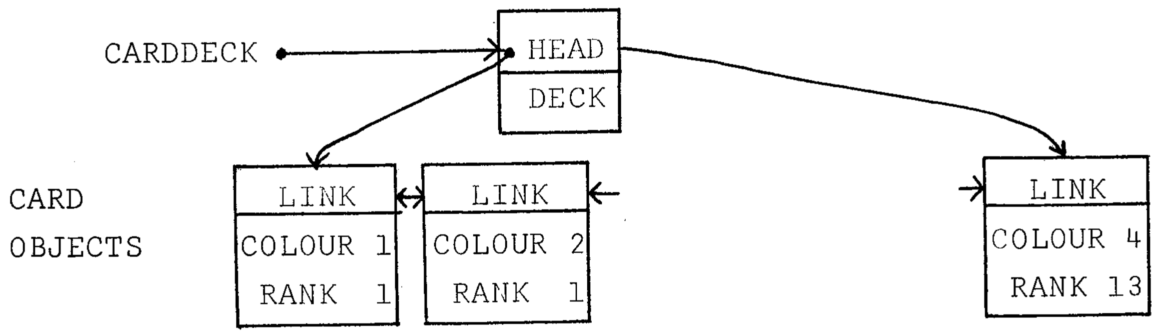
The patterns to be described are

```
CLASS CARD
CLASS DECK
CLASS HAND
```

Initially the CARD objects are created and inserted into a DECK in the order of their generation (which is non-random). Skeletons of these two CLASSES are

```
LINK CLASS CARD(COLOUR,RANK); INTEGER COLOUR, RANK;
BEGIN .................................... END;
HEAD CLASS DECK;
BEGIN ..... END;
```

Immediately prior to the dealing, the situation may be represented by

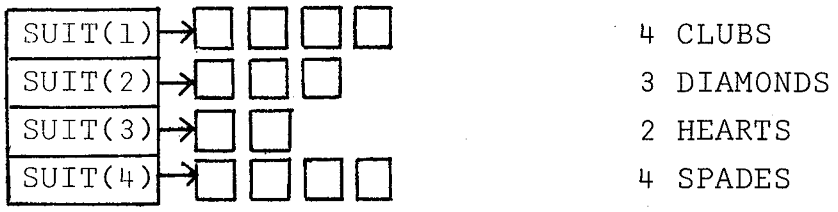


A deal consists of removing the 52 cards in random order from CARDDECK and inserting them into the HANDS in rotation.

It is convenient to describe each HAND by

```
CLASS HAND;
BEGIN  REF(HEAD) SUIT(1:4);
       generate four heads representing the four
            possible suits;
END ***HAND***
```

After generation and the dealing of the CARDS, a HAND object can be visualised by:



Now the CARDS may be ranked in order of their attribute RANK and in the appropriate SUIT. The actions of the program are:

```
generate the deck;
deal the hands;
```

<u>generate the deck;</u>

These actions may be made local to CLASS DECK, which now has the outline

```
HEAD CLASS DECK;
BEGIN  INTEGER I,J;
       FOR J := 1 through 13 DO
           FOR I := 1 through 4 DO
               NEW CARD(I,J).INTO(THIS HEAD)
END ***DECK***
```

On generation of a DECK object, the representation of the CARDS is automatically generated.

<u>deal the hands;</u>

This part consists of randomly selecting the Nth card and placing it in the appropriate SUIT of the current PLAYER. Let J denote the index of the current PLAYER (J = 1,2,3 or 4), then the actions are:

```
J := 0;
FOR I := 52 STEP -1 UNTIL 1 DO
BEGIN  C :- the randomly selected card;
       IF J = 4 THEN J := 1 ELSE J := J + 1;
       COMMENT ***SELECT INDEX OF CURRENT PLAYER*** ;
       place C in PLAYER(J)
END;
```

There remains the tasks of writing the selection procedure, and the procedure to place the CARD C in the current HAND. These are done by PROCEDURE SELECT local to CLASS DECK and PROCEDURE PLACE local to CLASS HAND.

The final program is:

```
SIMSET
BEGIN  HEAD CLASS DECK;
       BEGIN  REF(CARD) PROCEDURE SELECT(N); INTEGER N;
              BEGIN  REF(CARD)X; INTEGER I;
                     X :- FIRST QUA CARD;
                     FOR I := 2 STEP 1 UNTIL N DO
                         X :- X.SUC;
                     SELECT :- X;
              END ***SELECT*** ;
              INTEGER I,J;
              FOR I := 1 STEP 1 UNTIL 13 DO
                  FOR J := 1 STEP 1 UNTIL 4 DO
                      NEW CARD(J,I).INTO(THIS HEAD);
       END ***DECK*** ;

       LINK CLASS CARD(COLOUR,RANK); INTEGER COLOUR, RANK;;
       COMMENT ***COLOUR = 1  REPRESENTS CLUBS
                           2  REPRESENTS DIAMONDS
                           3  REPRESENTS HEARTS
                           4  REPRESENTS SPADES
                    RANK = 1  REPRESENTS ACE
                        2-10  OBVIOUS
                        11  JACK
                        12  QUEEN
                        13  KING*** ;
```

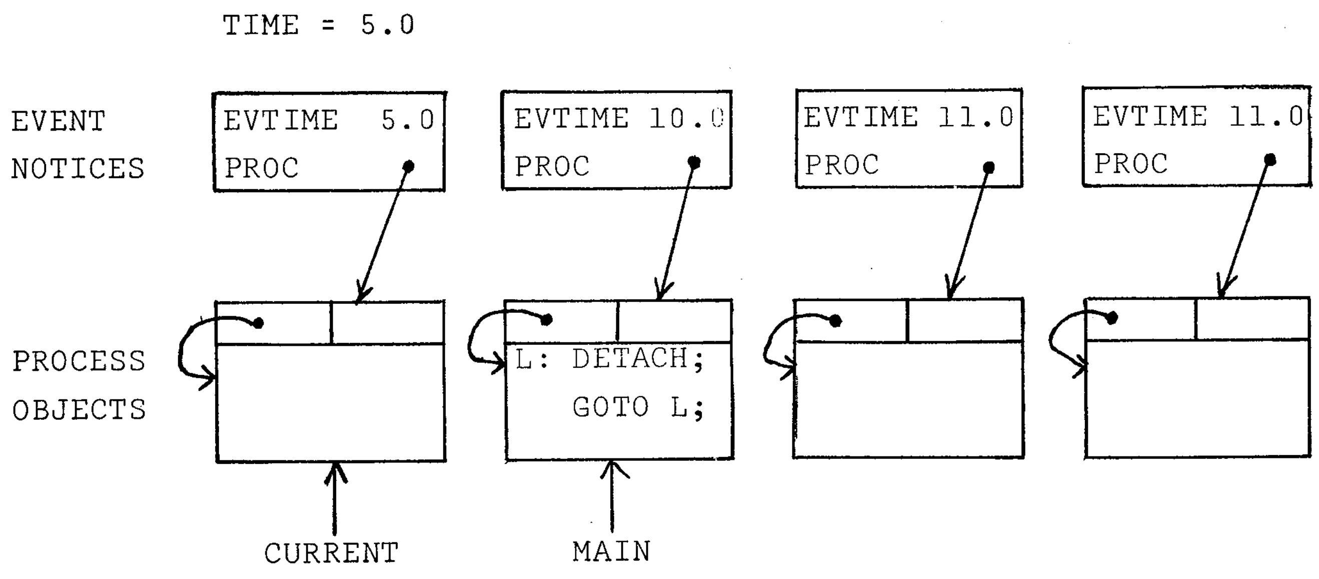
```
CLASS HAND;
BEGIN  PROCEDURE PLACE(C); REF(CARD)C;
       BEGIN  REF(HEAD)S; REF(CARD)X;
              S :- SUIT(C.COLOUR);
              IF ¬ S.EMPTY THEN
              BEGIN  X :- S.FIRST QUA CARD;
                     WHILE X =/= NONE DO
                     BEGIN  IF X.RANK > C.RANK THEN
                               BEGIN  C.PRECEDE(X);
                                      GOTO L;
                               END;
                     END;
              END;
              COMMENT ***WE ENTER HERE IF S IS EMPTY OR
                         IF C.RANK IS THE HIGHEST MET
                         SO FAR*** ;
              C.INTO(S);
   L:  END ***PLACE*** ;

       REF(HEAD) ARRAY SUIT(1:4);
       SUIT(1) :- NEW HEAD;
       SUIT(2) :- NEW HEAD;
       SUIT(3) :- NEW HEAD;
       SUIT(4) :- NEW HEAD;
END ***HAND*** ;
```

```
        REF(CARD)C;
        REF(DECK)CARDDECK;
        INTEGER I,J,U;
        REF(HAND) ARRAY PLAYER(1:4);
        U := ININT; COMMENT ***INPUT THE RANDOM STREAM BASE*** ;
        CARDDECK :- NEW DECK;
        COMMENT ***GENERATES THE WHOLE PACK OF 52 CARDS IN
                        NON-RANDOM ORDER*** ;
        FOR I := 1 STEP 1 UNTIL 4 DO
            PLAYER(I) :- NEW HAND;
        FOR I := 52 STEP -1 UNTIL 1 DO
        BEGIN  C :- CARDDECK.SELECT(RANDINT(1,I,U));
               IF J = 4 THEN J := 1 ELSE J := J + 1;
               PLAYER(J).PLACE(C);
        END;
END ***PROGRAM***
```

3 THE SYSTEM CLASS SIMULATION

The CLASS SIMULATION is prefixed by SIMSET and provides, in addition to SIMSET's set concepts, the notions of a time axis and processes (entities which interact over a period of time).

The time axis consists of a set of event notices which have two attributes, a reference to the PROCESS they represent and the time of its next scheduled event. The event notices are ranked according to the values of the time variable (EVTIME). We can picture the time axis with four scheduled events by:



The first PROCESS represented in the time axis is always referenced by CURRENT and the system time is the value of its scheduled next event (here 5.0). An object of any class prefixed by PROCESS may take an active and passive part in a simulation. The organisation is so framed that the PSC lies within CURRENT and its actions are executed. When the active phase is over, that PROCESS may be rescheduled for a later active phase (for example, by REACTIVATE or HOLD) or removed from the timing tree (by PASSIVATE or WAIT). It is apparent that RESUME is too primitive for this purpose as it involves rescheduling or removing EVENT NOTICES as well as switching the PSC from one PROCESS object to another.

However RESUME and DETACH do form the basis for the scheduling procedures. To prevent the user from destroying system security, event notices may not be explicitly referenced by the user - he must use the system procedures for scheduling or rescheduling. In addition, it is strongly recommended that explicit use of "DETACH", "RESUME" and "ATTACH" be avoided within a SIMULATION block.

There is one special PROCESS object which plays a key role in any SIMULATION - one referenced by MAIN. Whenever MAIN becomes CURRENT, it causes the actions of the SIMULATION block itself to be continued. The corresponding event notice can then be rescheduled (typically by a call on HOLD) and then the action switches from the SIMULATION block to the new CURRENT. Thus the SIMULATION block is itself treated as a program component during the SIMULATION.

The class outline is

```
SIMSET CLASS SIMULATION;
BEGIN  LINK CLASS PROCESS.....;
       REF(PROCESS) PROCEDURE CURRENT;.....;
       REAL PROCEDURE TIME;.....;
       COMMENT ***SCHEDULING PROCEDURES*** ;
       PROCEDURE HOLD;.......;
       PROCEDURE PASSIVATE;..;
       PROCEDURE WAIT;.......;
       PROCEDURE CANCEL;.....;
       PROCEDURE ACTIVATE;...;
       PROCEDURE ACCUM;......;
       REF("the main program")MAIN;
       COMMENT ***HERE FOLLOW ACTIONS WHICH SET UP THE
                  TIME AXIS AT TIME ZERO*** ;
END ***SIMULATION***
```

We now give a prose discussion of the attributes of CLASS SIMULATION:

<u>CLASS PROCESS</u>

```
LINK CLASS PROCESS;
BEGIN  BOOLEAN PROCEDURE IDLE;..........;
       BOOLEAN PROCEDURE TERMINATED;....;
       REAL PROCEDURE EVTIME;...........;
       REF(PROCESS) PROCEDURE NEXTEV;...;
       DETACH;
       INNER;
       PASSIVATE;
END ***PROCESS***
```

An object of a class inner to CLASS PROCESS is a PROCESS object. A PROCESS object has the properties of CLASS LINK and can be manipulated by sequencing statements. Sequencing statements are used to insert or delete a PROCESS object from the time axis. The state of a PROCESS object after generation is "detached" and its LSC is positioned to the first statement of the user defined operations rule. When the actions of the user defined subclass are exhausted, the unfinished actions of the PROCESS level are continued (following the INNER). These remove the object from the time axis (PASSIVATE) and its state becomes "terminated".

If a PROCESS object is not represented in the time axis, then it is <u>terminated</u> or <u>passive</u> (its actions are not yet exhausted) and the BOOLEAN PROCEDURE IDLE returns TRUE. All PROCESS objects represented in the time axis are said to be <u>suspended</u> except for the first (CURRENT) which is said to be <u>active</u>.

A call on EVTIME for a suspended or active PROCESS object returns the scheduled time of its next event. If the object is passive or terminated, then a call on EVTIME results in a run time error.

PROCESS object	result of procedure call		
state	IDLE	TERMINATED	EVTIME
active	FALSE	FALSE	time of current event
suspended	FALSE	FALSE	time of next event
passive	TRUE	FALSE	run time error
terminated	TRUE	TRUE	run time error

REF(PROCESS) PROCEDURE CURRENT;......;

returns a reference to the currently active PROCESS object. There is one special PROCESS object in the system referenced by MAIN. Every time MAIN becomes CURRENT it causes the actions of the SIMULATION block to be resumed.

REAL PROCEDURE TIME;.....;

always returns the current value of the system time.

PROCEDURE HOLD(T); REAL T;......;

HOLD(N) reschedules CURRENT so that its next active phase will occur at TIME + N. If the value of the actual parameter N is negative, the call is equivalent to HOLD(0). After the rescheduling, the actions of CURRENT are resumed.

Notice that HOLD(T) can be called from the user defined SIMULATION block in which case MAIN will be rescheduled - i.e. the actions of the program block are suspended for N time units.

PROCEDURE PASSIVATE;......;

removes CURRENT from the time axis and resumes the actions of the new CURRENT. A run time error will occur if the time axis is now empty.

PROCEDURE WAIT(S); REF(HEAD)S;......;

```
    PROCEDURE WAIT(S); REF(HEAD)S;
    BEGIN  CURRENT.INTO(S);
           PASSIVATE
    END ***WAIT***
```

WAIT includes the currently active PROCESS object (this could be MAIN) into a referenced set, and then calls PASSIVATE.

PROCEDURE CANCEL(X); REF(PROCESS)X;......;

CANCEL(P) where P is a reference to a PROCESS object will delete the corresponding event notice if any. If P is currently active or suspended, it thus becomes passive. If P is a reference to a passive or terminated PROCESS object or NONE, CANCEL(P) has no effect. Thus CANCEL(CURRENT) is equivalent to PASSIVATE.

PROCEDURE ACTIVATE

For user convenience, calls on the procedure ACTIVATE are written in terms of the corresponding activation-statements.

activation-statement

$$\left\{\begin{array}{c}\text{ACTIVATE}\\ \text{REACTIVATE}\end{array}\right\}\ \text{PROCESS-expression1}\ \left[\begin{array}{l}\{\text{AT}\,|\,\text{DELAY time}\}\text{PRIOR}\\ \{\text{BEFORE}\,|\,\text{AFTER}\}\text{PROCESS-}\\ \qquad\qquad\text{expression2}\end{array}\right]$$

Let X be the value of PROCESS-expression1. If the activator ACTIVATE is used, then the activation-statement will have no effect (other than evaluating X) unless X is passive. If the activator REACTIVATE is used, then X may be active, suspended, or passive (in which latter case, the activation-statement acts as an ACTIVATE statement).

The type of scheduling is determined by the scheduling clause.

Direct activation

$$\left\{\begin{array}{l}\text{ACTIVATE X} \\ \text{REACTIVATE X}\end{array}\right\}$$

X becomes the new CURRENT and the system time is unchanged. The formerly active PROCESS object from where the call was made becomes suspended.

Timing clause

$$\left\{\begin{array}{c}\text{ACTIVATE} \\ \text{REACTIVATE}\end{array}\right\} \text{X} \left\{\begin{array}{c}\text{AT} \\ \text{DELAY}\end{array}\right\} \text{T} \quad [\text{PRIOR}]$$

The timing clause AT specifies the system time of the scheduled active phase. The clause

DELAY T

is equivalent to

AT current-system-time + T

The corresponding EVENT NOTICE is inserted according to the specified time, normally after any EVENT NOTICE with the same system time; the symbol PRIOR may be used to specify insertion in front of any EVENT NOTICE with the same system time.

Default actions

"AT T", when T < the current-system-time, is equivalent to "AT current-system-time".

DELAY T when T < 0 is equivalent to DELAY 0.

Relative activation

$$\left\{\begin{matrix}\text{ACTIVATE}\\ \text{REACTIVATE}\end{matrix}\right\} X \quad \left\{\begin{matrix}\text{BEFORE}\\ \text{AFTER}\end{matrix}\right\} Y$$

If Y is a reference to an active or suspended PROCESS object, then the clause BEFORE Y or AFTER Y is used to insert an event notice for X before or after that of Y and at the same system time.

Default actions

If Y is neither active nor suspended, then the activation-statement is equivalent to

```
          CANCEL(X).
```

If X == Y, then the activation-statement is equivalent to

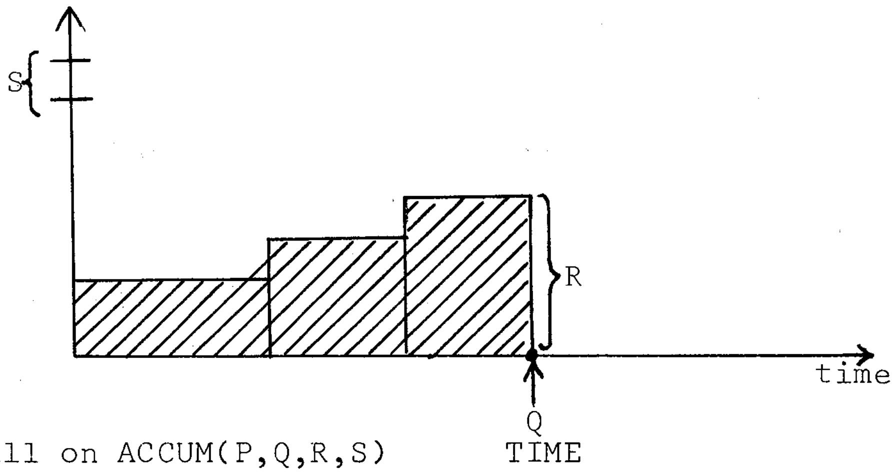
```
          CANCEL(X).
```

PROCEDURE ACCUM

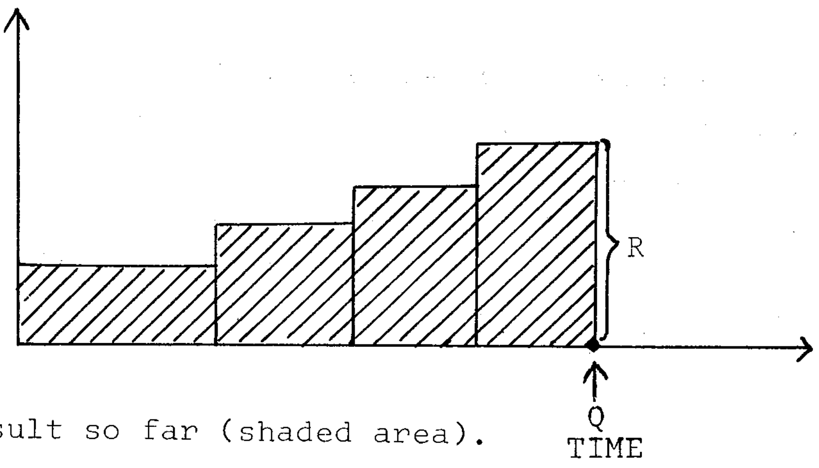
```
PROCEDURE ACCUM(A,B,C,D); NAME A,B,C; REAL A,B,C,D;
BEGIN  A := A + C*(TIME-B);
       B := TIME;
       C := C + D
END ***ACCUM***
```

ACCUM(P,Q,R,S) is used to accumulate the "system time integral" of the variable R. The accumulation takes place in P. The integral is interpreted as a step function of the system time with Q holding the system time when P was last updated. The value of S is the current increment of the step function.

before



after a call on ACCUM(P,Q,R,S) we have



P contains result so far (shaded area).

Example on the use of SIMULATION

The program is a description of a simple epidemic model. A contagious, non-lethal disease is spreading through a POPULATION of a fixed size. Certain countermeasures are taken by a public health organisation. Each individual infection has a given INCUBATION period, during which the subject is noncontagious and has no SYMPTOMS, followed by a contagious period of a given LENGTH.

COURSE OF INFECTION IN DAYS

INCUBATION	LENGTH	thereafter
NON-CONTAGIOUS	CONTAGIOUS	IMMUNE

Each DAY of the contagious period the subject may seek treatment from the public health organisation and get cured. The probability of his seeking treatment is stored in REAL ARRAY PROBTREAT (1:LENGTH). Each person has an expected number of CONTACTS per day. At one such contact the probability of infecting a previously uninfected person is PRINF.

Once cured a person becomes immune. If untreated, the infection ceases after the given period and the person becomes immune.

SICKP (sick persons) appear as PROCESSes in the system. When CURED, or when the disease has run its course, they leave the system. The very first infection is generated by the main program (user block). A person infected by another person is included as a member in a set belonging to the infector (his environment (ENV)). A person can be a member of at most one set.

As people are cured, they are removed from these sets which gradually split up into smaller sets. The latter grow independently, disintegrate further, and so on. As the number of UNINFECTED people decreases, the growth of the contagion slows down until it finally dies out.

The public countermeasures are represented by TREATMENTs which are also PROCESSes. A patient is removed from the environment set to which he belongs, if any. If he has visible symptoms, he is cured. In addition his environment is searched and each member is subjected to a full treatment which may cause other environments to be searched etc. A patient displaying no symptoms is given a mass treatment which has a probability PROBMASS of success. His environment is not searched. In the present model, treatments act instantaneously. The simulation ends after SIMPERIOD units of time.

An outline of the problem description is

```
SIMULATION
BEGIN  PROCESS CLASS SICKP;
       BEGIN  REF(HEAD)ENV;
              PROCEDURE INFECT;
              ......
       END ***SICK PERSON*** ;
       PROCESS CLASS TREATMENT(PATIENT);
                     REF(SICKP)PATIENT;
       BEGIN ...................... END;
       ..........
END ***SIMULATION BLOCK*** ;
```

We may now outline the class actions

```
    PROCESS CLASS SICKP;
    BEGIN  PROCEDURE INFECT;......;
           INTEGER DAY; BOOLEAN SYMPTOMS;
           REF(HEAD)ENV;
           COMMENT ***wait incubation days for symptoms to
                      appear*** ;
           HOLD(incubation period);
           COMMENT ***now the symptoms are apparent.  If
                      "treatment today" is TRUE then a cure
                      is sought which also implies that the
                      environment of this SICK PERSON is
                      examined.  Also a number of contacts
                      are infected*** ;
           SYMPTOMS := TRUE;
           FOR DAY := 1 through LENGTH DO
           BEGIN
                 IF treatment today THEN ACTIVATE NEW TREATMENT;
                 INFECT(todays contacts);
                 HOLD(1);
           END;
    END ***SICK PERSON*** ;
```

```
PROCEDURE INFECT(N); INTEGER N;
 BEGIN  INTEGER I;
        COMMENT ***N gives the number of contacts who can
                   be infected.  N random drawings are made
                   to see if the N contacts are to be in-
                   fected.  If so a NEW SICKP is generated
                   and included in this SICKPs ENV and
                   activated*** ;
        FOR I := 1 through N DO
            IF Ith contact is infected THEN
            BEGIN  NEW SICKP.INTO(ENV);
                   ACTIVATE ENV.LAST;
            END;
 END ***INFECT***
```

```
PROCESS CLASS TREATMENT(PATIENT); REF(SICKP)PATIENT;
BEGIN  REF(SICKP)X;
       INSPECT PATIENT WHEN SICKP DO
       BEGIN  OUT;
              IF SYMPTOMS THEN
              BEGIN  CANCEL(PATIENT);
                     FOR X :- ENV.FIRST WHILE X =/= NONE DO
                         ACTIVATE NEW TREATMENT(X);
              END ELSE IF probmass successful
                          THEN CANCEL(PATIENT)
       END;
END ***TREATMENT***
```

Explanation:

A treatment tests the SYMPTOMS attribute of its parameter PATIENT. If TRUE then the instantaneous successful treatment is given. The patient is removed from the set he is in (OUT) and becomes passive for the rest of the simulation. In addition his environment is searched and a new treatment is activated for each member. If there are no symptoms, the patient is given a cheap pill which has a probability of being successful. If successful the patient is instantaneously cured and takes no further part in the simulation, but his environment is not searched.

A complete description of the program now follows:

```
BEGIN  INTEGER POPULATION, LENGTH, CONTACTS, INCUBATION,
               U1, U2, U3, U4;
       REAL PRINF, PROBMASS, SIMPERIOD;
       COMMENT ***THE RANDOM STREAM NUMBERS ARE READ IN*** ;
       U1 := ININT; U2 := ININT;
       U3 := ININT; U4 := ININT;
       POPULATION := ININT;
       INCUBATION := ININT; LENGTH := ININT;
       CONTACTS := ININT;
       SIMPERIOD := INREAL;
       PRINF := INREAL; PROBMASS := INREAL;
SIMULATION BEGIN  REAL ARRAY PROBTREAT(1:LENGTH);
                  PROCESS CLASS SICKP;
                  BEGIN  INTEGER DAY;
                         BOOLEAN SYMPTOMS;
                         REF(HEAD)ENV;
                         PROCEDURE INFECT(N); INTEGER N;
                         BEGIN  INTEGER I;
                                FOR J := 1 STEP 1 UNTIL N DO
                                    IF DRAW(PRINF*UNINFECTED/
                                             POPULATION,U3) THEN
                                    BEGIN  NEW SICKP.INTO(ENV);
                                           ACTIVATE ENV.LAST;
                                    END;
                         END ***INFECT*** ;
```

```
                    IF UNINFECTED > 0 THEN
                       UNINFECTED := UNINFECTED-1;
                    ENV :- NEW HEAD;
                    COMMENT ***NO SYMPTOMS APPEAR UNTIL
                               AFTER INCUBATION DAYS*** ;
                    HOLD(INCUBATION);
                    COMMENT ***NOW SYMPTOMS APPEAR AND THIS
                               SICK PERSON MAY SEEK A CURE
                               AND INFECT OTHERS EACH DAY*** ;
                    FOR DAY := 1 STEP 1 UNTIL LENGTH DO
                    BEGIN  IF DRAW(PROBTREAT(DAY),U1) THEN
                              ACTIVATE NEW TREATMENT(CURRENT);
                           INFECT(POISSON(CONTACTS,U2));
                           HOLD(1)
                    END;
             END ***SICK PERSON*** ;
             PROCESS CLASS TREATMENT(PATIENT);
                     REF(SICKP)PATIENT;
             BEGIN  REF(SICKP)X;
                    INSPECT PATIENT WHEN SICKP DO
                    BEGIN  OUT;
                           IF SYMPTOMS THEN
                           BEGIN  CANCEL(PATIENT);
                                  FOR X :- ENV.FIRST
                                      WHILE X =/= NONE DO
                                      ACTIVATE NEW TREATMENT(X);
                           END ELSE IF DRAW(PROBMASS,U4)
                                       THEN CANCEL(PATIENT);
                    END;
             END ***TREATMENT*** ;

             ACTIVATE NEW SICKP;
             HOLD(SIMPERIOD);
      END ***SIMULATION BLOCK*** ;
END ***PROGRAM***
```

4 TEXT HANDLING FACILITIES

The concept of TEXT is the key to SIMULA's input/output facilities. For example when a card is read in, the internal representation is held as a string of 80 CHARACTERS with a one to one correspondence between the nth column of the card and the nth CHARACTER in the string. Such a string is called a TEXT-value and it is housed in a referenced TEXT-object.

$$T \longrightarrow \boxed{\text{THIS␣TEXT␣VALUE␣LIES␣IN␣A␣TEXT␣OBJECT.}}$$

With TEXTs we have thus a combination of both reference (to the housing object) and value (the string of CHARACTERS) properties. TEXTs resemble objects (of classes) in that they possess attributes which are accessed by the normal remote accessing (dot notation) technique, but TEXTs and their attributes are wholly system defined.

Whereas a CHARACTER ARRAY is oriented towards accessing single characters at a time by direct means (subscripts), the TEXT concept is oriented towards groupings of characters and sequentially accessing these groups.

TEXT variables are declared in the usual fashion.

e.g. TEXT R,S,T

and the initial value of each of these variables is NOTEXT.

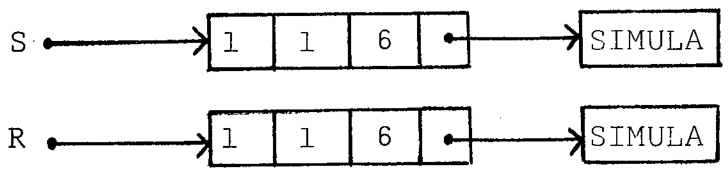
TEXT variables are capable of referencing TEXT objects which may be created by two system defined PROCEDURES - BLANKS and COPY:

```
          T :- BLANKS(N)
```

creates a TEXT object of length N characters, each initialised to the blank character. After creation of the object, its reference value is assigned to T.

```
          S :- COPY("SIMULA");
          R :- COPY(S)
```

COPY will accept either a TEXT value or a TEXT reference expression as parameter, creates a TEXT object with value identical to that of the actual parameter and of the same length, and returns a reference to it. The result of the last two statements may be pictured by



magic box

Each TEXT variable has its own "magic box" which gives information about the start position (SP), end position (EP) and current position (CP) of the object it currently references. The box also contains the reference value of the object itself. The formal pattern of these magic boxes is:

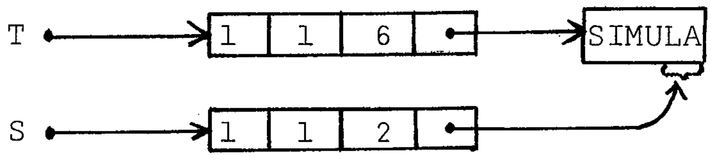
SP	CP	EP	ref.val

A TEXT object may be referenced in subfields by use of the procedure SUB

After T :- COPY("SIMULA")

then S :- T.SUB(5,2)

results in

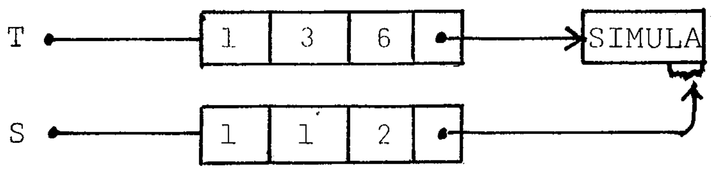


S references the subfield of T beginning from character 5 and of length 2.

The characters may be accessed one at a time by calls on GETCHAR which returns the value of the current character and increments the CP by one.

```
After          C := T.GETCHAR;
               D := T.GETCHAR
```

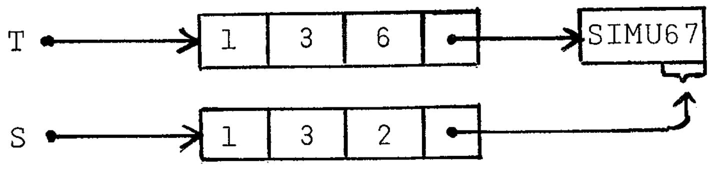
the snapshot is



and the values of C and D are 'S' and 'I' respectively.
As access was made through T, only its C has been incremented (twice). The reverse process of inserting a character value into the current position is achieved through use of PUTCHAR, which also increments the CP.

After S.PUTCHAR('6');
S.PUTCHAR('7')

the snapshot is



Note that the value of T has been changed. The CP of S is now out of range. A further call

```
        S.PUTCHAR
     or S.GETCHAR
```

will result in a run time error. To provide a check, a BOOLEAN PROCEDURE MORE is provided which returns FALSE if the CP is out of range and TRUE otherwise. Currently,

```
        T.MORE = TRUE          S.MORE = FALSE
```

Other useful system defined procedures are:

LENGTH	which returns the length of the currently referenced value (T.LENGTH = 5 S.LENGTH = 2)
POS	which returns the value of the CP (T.POS = 3 S.POS = 3)

SETPOS which resets the CP. (To reset the CP's of T and S back to their initial character, we write

```
          T.SETPOS(1)
          S.SETPOS(1) )
```

Text values may be transferred from one object to another by

```
          S := T
```

or a value into an object by

```
          S := "TEXT␣VALUE"
```

Both are left justified.

The only restriction being that the TEXT object receiving the value must be long enough to accept the value or else a run time error occurs. Any positions not directly copied into are filled with blanks.

Several editing and de-editing procedures are defined within SIMULA. These convert numbers to external form and vice versa. They are designed to operate repetitively across a field and are thus oriented towards formatted output and input.

The further detailed description of the TEXT handling facilities is given under the sub-sections

LENGTH and MAIN
subtexts
character access
text generation
TEXT assignment
TEXT editing

Throughout these subsections X, Y, Z denote TEXT references.

LENGTH and MAIN

INTEGER PROCEDURE LENGTH;

The value of X.LENGTH is the number of CHARACTERS in the TEXT object referenced by X.

```
e.g.     after X :- BLANKS(10), then X.LENGTH = 10
           if  Y == NOTEXT    , then Y.LENGTH =  0
```

TEXT PROCEDURE MAIN;

X.MAIN is a reference to the TEXT object which is or contains the text value referenced by X.

```
e.g.     after X :- BLANKS(20);
               Y :- X.SUB(1,10);
               Z :- NOTEXT;

          then X.MAIN == X
               Y.MAIN == X
               Z.MAIN == NOTEXT
```

The following relations hold for any TEXT reference X

```
               X.MAIN.LENGTH >= X.LENGTH
               X.MAIN.MAIN == X.MAIN
```

SUBTEXTS

TEXT PROCEDURE SUB(I,N); INTEGER I,N;

The call

```
        X.SUB(J,M)
```

designates that part of the TEXT object referenced by X starting in CHARACTER position J and of length M characters.

```
e.g.    after X :- COPY("MAIN␣NOT␣SUB-TEXT");
              T :- X.SUB(10,8);

        then T = "SUB-TEXT"
```

For X.SUB(J,M) to be a legal call, the subtext must be included in X. Thus

```
        J > 0
        J + M-1 <= X.LENGTH
```

If these conditions do not hold a run time error results.

TEXT PROCEDURE STRIP;

STRIP is used to return a reference to a subfield of a TEXT object which differs from the original in that all blanks on the right are ignored. X.STRIP is thus equivalent to X.SUB(1,N) where the remaining CHARACTERS of X (from position N+1 and of length X.LENGTH-N), if any, are all blanks.

CHARACTER access

The CHARACTERS, values housed in a TEXT object, are accessible one at a time. Any TEXT reference contains a "position indicator" which identifies the currently accessible CHARACTER of the referenced TEXT object.

The position indicator of NOTEXT is 1. A TEXT reference obtained by calling any system defined TEXT procedure has its position indicator set to 1. The position indicator of a given TEXT reference may be altered by the PROCEDURES SETPOS, GETCHAR, PUTCHAR, TEXT-reference-assignment and any editing or de-editing PROCEDURE. Position indicators are left unaltered by TEXT reference relations, TEXT value relations and TEXT value assignments.

INTEGER PROCEDURE POS;

X.POS is the current value of the position indicator of X. The following relation is always TRUE.

```
          1 <= X.POS <= X.LENGTH + 1
```

PROCEDURE SETPOS(I); INTEGER I;

The effect of X.SETPOS(M) is to assign the value of M to the position indicator of X, if 1 <= M <= X.LENGTH + 1. If M is out of this range, then the value X.LENGTH + 1 is assigned.

BOOLEAN PROCEDURE MORE;

X.MORE is TRUE if the position indicator of X is in the range 1 through X.LENGTH, otherwise the value is FALSE.

CHARACTER PROCEDURE GETCHAR;

The value of X.GETCHAR is a copy of the currently accessible CHARACTER of X provided X.MORE is TRUE. In addition, the position indicator of X is then increased by one. A run time error results if X.MORE is FALSE.

PROCEDURE PUTCHAR(C); CHARACTER C;

The effect of X.PUTCHAR(C) is to replace the currently accessible CHARACTER of X by the value of C provided that X.MORE is TRUE. In addition the position indicator of X is then increased by one. If X.MORE is FALSE, a run time error results.

Example:

The PROCEDURE COMPRESS rearranges the CHARACTERS of the TEXT object referenced by the actual parameter by collecting non-blank CHARACTERS in the leftmost part of the TEXT object and filling in the remainder, if any, with blanks. Since the parameter is called by reference (and not by name), its position indicator is unaltered.

```
PROCEDURE COMPRESS(T); TEXT T;
BEGIN  TEXT U; CHARACTER C;
       T.SETPOS(1); U :- T;
MOVELEFT:  WHILE U.MORE DO
           BEGIN  C := U.GETCHAR;
                  IF C ¬= '␣' THEN T.PUTCHAR(C);
           END;,
           COMMENT ***WE NOW FILL IN THE RIGHT WITH
                      BLANKS*** ;
           T.SUB(T.POS,T.LENGTH-T.POS+1) := NOTEXT;
END ***COMPRESS***
```

Note the use of a value assignment to T.SUB, and the use of NOTEXT on a right hand side as a neat way of filling a TEXT value to blanks.

```
After       X :- COPY("GET␣RID␣OF␣ALL␣BLANKS");
            COMPRESS(X);

then        X = "GETRIDOFALLBLANKS␣␣␣␣"
            X.STRIP = "GETRIDOFALLBLANKS"
```

TEXT generation

N.B. The PROCEDURES are non-local.

TEXT PROCEDURE BLANKS(N); INTEGER N;

The reference value is a new TEXT object of length N, filled with blank CHARACTERS.

The value of the actual parameter, M, is restricted to

$$0 <= M <= 2^{15}-20 = 32748$$

otherwise a run time error results.

TEXT PROCEDURE COPY(T); VALUE T; TEXT T;

The referenced value is a new TEXT object which is a copy of the TEXT value which is (or is referenced by) the actual parameter.

Example:

```
          T :- COPY("360SIMULA");
```

is equivalent to,

```
          T :- BLANKS(9);
          T := "360SIMULA";
```

TEXT assignment

a) TEXT-reference-assignment

A TEXT-reference-assignment causes a copy of the TEXT-reference obtained by evaluating the right part to be assigned to the left part variable - this includes a copy of its position indicator.

```
e.g.     after    X :- COPY("ABCD");
                  X.SETPOS(3);
                  Y :- X;

          then    X.POS = 3
                  Y.POS = 3
```

In general, after

```
                  X :- P;     where P is a TEXT reference,

             then X == P
                  X = P
                  X.POS = P.POS
```

are all TRUE.

b) TEXT-value assignment

Consider the value assignment

```
        T := P;
```

let the length of T be Ll, and the length of the right part be a TEXT value of length Lr. There are three cases to consider:

Ll = Lr:	the character contents of the right part TEXT are copied to the left part TEXT
Ll > Lr:	the character contents of the left part are copied into the leftmost Lr characters of the left part TEXT, whose remaining Ll-Lr CHARACTERS are filled with blanks.
Ll < Lr:	a run time error results.

```
After   T :- COPY("EIGHT␣CHARS");
        T := "WRONG:11";

 then   T = "WRONG:11␣␣␣"
```

Note that

```
        T := NOTEXT;
```

would set all the character positions of T to blanks.

In a multiple TEXT value assignment

```
        T1 := T2 := ..... TN := P;
```

then

T_J.LENGTH >= T_{J+1}.LENGTH

for $J = 1,2,...,N-1$

Text editing and de-editing

TEXT editing and de-editing procedures are provided to transform binary values into field data and vice versa. The syntax for numeric-text-values (external data) follows:

numeric-text-values

$$\left\{\begin{array}{l}\text{grouped-item}\\ \text{real-item}\\ \text{integer-item}\end{array}\right\}$$

grouped-item

```
sign-part[[groups].]groups
```

groups

```
[digits blank]... digits
```

real-item

$$\text{sign-part}\ \left\{\begin{array}{l}\text{[[digits].]digits[E sign-part digits]}\\ \text{E sign-part digits}\end{array}\right\}$$

integer-item

```
sign-part digits
```

sign-part

$$\text{[blank]...}\ \left[\begin{array}{c}+\\-\end{array}\right]\ \text{[blank]...}$$

where 'E' represents an exponent sign. This CHARACTER may be altered by the user by use of the PROCEDURE LOWTEN (see Appendix B).

A numeric-text-value is a character sequence under the above rules.

De-editing procedures

A de-editing procedure operating on a given TEXT reference X operates in the following way:

1) the longest numeric item of the given form is located, contained within X and containing the first character of X. If such error can be found, a run time error results.

2) the numeric item is interpreted as a number. If it is outside the accepted range (see PART 2, section a run time error results.

3) the position indicator of X is made one greater than the last character of the numeric item.

N.B. Unless otherwise stated, the de-editing procedures are illustrated in the context:

```
    T :- COPY("1234.5+7.3&4AB");
    S :- T.SUB(7,6);
    R :- T.SUB(5,2);
```

INTEGER PROCEDURE GETINT;

Locates an integer-item.

```
    T.GETINT = 1234
    S.GETINT = 7
    R.GETINT   causes a run time error
```

<u>REAL PROCEDURE GETREAL;</u>

locates a real-item

```
T.GETREAL = 1234.5
S.GETREAL = 73000.0
R.GETREAL = 0.5
```

<u>INTEGER PROCEDURE GETFRAC;</u>

Locates a grouped item. In its interpretation, any number of blanks, commas, and one decimal point are ignored and the resulting value is an INTEGER.

```
After     T := COPY(1␣013.42");

 then     T.GETFRAC = 101342
```

Editing procedures

Editing procedures in a given text reference X convert arithmetic values to numeric items. After an editing operation, the numeric item obtained is right adjusted in the TEXT X preceded by padding blanks. The final value of the position indicator is X.LENGTH+1.

A positive number is edited with no sign. If X == NOTEXT then a run time error results, otherwise if X is too short to contain the numeric item, an edit overflow is caused (X is filled with asterisks) and a warning message is given at the end of program execution.

```
Let     T :- BLANKS(10);
```

PROCEDURE PUTINT(I); INTEGER I;

T.PUTINT(VAL) converts the value of the parameter to an integer-item of the designated value.

```
     T.PUTINT(-37)          ␣␣␣␣␣␣␣-37
     T.PUTINT(118.8)        ␣␣␣␣␣␣␣119
```

PROCEDURE PUTFIX(R,N); REAL R; INTEGER N;

T.PUTFIX(VAL,M) results in an integer-item of M=0, or a real-item (with no exponent) if M>1 with M digits after the decimal point. It designates a number equal in value to VAL rounded to M decimal places. A run time error results if M<0.

```
     T.PUTFIX(18,0)          ␣␣␣␣␣␣␣␣18
     T.PUTFIX(-1375,4,3)     ␣-1375.400
```

PROCEDURE PUTREAL(R,N); REAL R; INTEGER N;

T.PUTREAL(VAL,M) results in a real-item to M significant places with an exponent

```
          X.XXXXXXXXE±XX
          M figures
```

If M<0, a run time error results
If M=0, the exponent is preceded by a sign-part
If M=1, the exponent is preceded by an integer-item of one digit.

```
     T.PUTREAL(16,0)                E+01
     T.PUTREAL(-25.32,1)          -3E+01
     T.PUTREAL(-0.001472,3)    -1.47E-03
```

PROCEDURE PUTFRAC(I,N); INTEGER I,N;

T.PUTFRAC(VAL,M) results in a grouped-item

```
           XXX XXX.XXX XXX
```

If M=0, there is no decimal point. If M>0, there are M digits after the decimal point. Each digit group consists of 3 digits except possibly the first and the last. The numeric item is an exact representation of $I*10^{-M}$.

```
          T.PUTFRAC(10012416,3)          10␣012.416
```

The editing and de-editing procedures are oriented towards "fixed field" text manipulation.

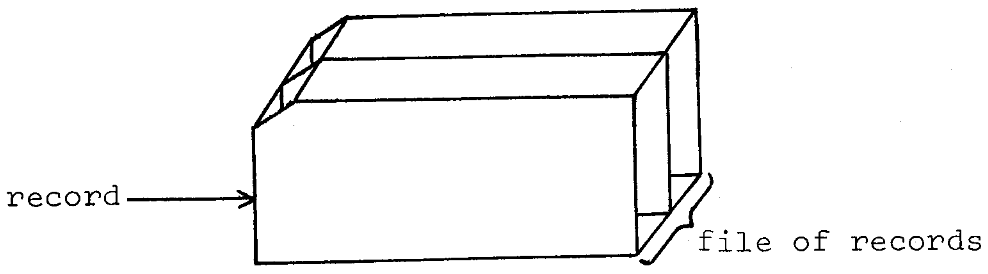
Example:

```
TEXT TR,TYPE,AMOUNT,PRICE,PAYMENT;
INTEGER PAY,TOTAL;
TR :- BLANKS(80);
      TYPE :- TR.SUB(1,5);
      AMOUNT :- TR.SUB(20,5);
      PRICE :- TR.SUB(30,6);
      PAYMENT :- TR.SUB(60,10);
..........
IF TYPE = "ORDER" THEN
BEGIN  PAY := AMOUNT.GETINT*PRICE.GETFRAC;
       TOTAL := TOTAL + PAY;
       PAYMENT.PUTFRAC(PAY,2);
END;
..........
```

5 THE SYSTEM CLASS BASICIO

Files or data sets are collections of data external to a program. They may be organised in a sequential manner (a batch of cards) or direct access manner (collection of items on a disc where each item is specified directly).

A file is composed of several records each of which is an ordered sequence of CHARACTERS.



The internal representation of a record is naturally held in a TEXT object, but TEXT handling facilities alone are not enough for treating input and output to secondary storage. We need in addition

a) means for tying the external medium to the internal representation,

b) for transferring information (record-by-record) either from the external file or to the external file, and

c) either interpreting the information in the internal TEXT object in a sequential manner, or else filling the TEXT object in a sequential manner.

A SIMULA system provides system classes for these purposes. The system classes have the hierarchy

$$\begin{tikzcd}
 & \text{file} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\text{INFILE} & \text{OUTFILE} \arrow[d, dash] & \text{DIRECTFILE} \\
 & \text{PRINTFILE} &
\end{tikzcd}$$

The identifier "file" is not accessible by the user - it defines the parts common to the subclasses.

The four types of defined file are:

INFILE	a sequential input file which transfers data from an external file to the program
OUTFILE	a sequential output file which transfers data from the program to an external file
PRINTFILE	(a subclass of OUTFILE) a sequential file with special extra facilities for transmitting information to a line printer
DIRECTFILE	a direct file with facilities for input <u>and</u> output

Each file object has a TEXT parameter called "name" - again this is NOT accessible by the user. When the file object is created, the external file associated with this file object is the file name appearing in a data set control card. The actual parameter must be a valid DDNAME of up to eight CHARACTERS.

The CLASS file has the declaration:

```
     CLASS file(name); VALUE name; TEXT name;
                       VIRTUAL : PROCEDURE OPEN, CLOSE;

     BEGIN  TEXT IMAGE;

            PROCEDURE SETPOS(I); INTEGER I;
                 IMAGE.SETPOS(I);

            INTEGER PROCEDURE POS;
                 POS := IMAGE.POS;

            BOOLEAN PROCEDURE MORE;
                 MORE := IMAGE.MORE;

            INTEGER PROCEDURE LENGTH;
                 LENGTH := IMAGE.LENGTH;

     END ***file***
```

The variable IMAGE references a TEXT object value which acts as a buffer containing the information currently being processed.

The PROCEDURES SETPOS, POS, MORE and LENGTH defined local to file operate on the buffer IMAGE. Given a reference to X to an object belonging to a subclass of file, then it is now possible to write the more convenient

```
          X.MORE           X.LENGTH           ..........
```

Instead of (the still valid)

```
          X.IMAGE.MORE     X.IMAGE.LENGTH     ..........
```

The PROCEDURES OPEN and CLOSE, which are specified as VIRTUAL but have no matching declaration at the "file" level, complete the definition of CLASS file. The matching PROCEDURES declared in the subclasses of "file" conform to the patterns below with possible minor variations depending upon the subclass. The variations are listed in the appropriate following sub-sections.

The PROCEDURE outlines are:

```
     PROCEDURE OPEN(BUF); TEXT BUF;
     BEGIN  IF OPEN THEN ERROR;
            IMAGE :- BUF;
     END
     PROCEDURE CLOSE;
     BEGIN ..........
            IMAGE :- NOTEXT;
     END
```

No information can be processed through a "file" object until it has not only been generated but also opened. This can only be achieved by a call on the PROCEDURE OPEN whose actual parameter is assigned to IMAGE and acts as the buffer. A call on OPEN when a "file" is already open gives a run time error.

The PROCEDURE CLOSE closes a file and releases the buffer (by the assignment IMAGE :- NOTEXT). No information may be transmitted through a closed "file" object, but it may be opened again by a further call on OPEN.

CLASS INFILE

```
file CLASS INFILE; VIRTUAL : PROCEDURE INIMAGE;
                             BOOLEAN PROCEDURE ENDFILE;

BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;.....;
       PROCEDURE CLOSE;......;
       BOOLEAN PROCEDURE ENDFILE;.........;
       CHARACTER PROCEDURE INCHAR;........;
       BOOLEAN PROCEDURE LASTITEM;........;
       INTEGER PROCEDURE ININT;...........;
       REAL PROCEDURE INREAL;.............;
       INTEGER PROCEDURE INFRAC;..........;
       TEXT PROCEDURE INTEXT(W); INTEGER W;......;
END ***INFILE***
```

PROCEDURE OPEN

conforms to the pattern listed with CLASS file but in addition positions the current position indicator to "LENGTH+1".

PROCEDURE CLOSE

conforms to the pattern listed with CLASS file.

PROCEDURE ENDFILE

returns TRUE before the INFILE is opened (by OPEN), if the end of external file marker has been met, and if the INFILE has been closed (by a call on CLOSE).

PROCEDURE INIMAGE

transfers an external file record into the TEXT IMAGE. A run time error will occur if the TEXT object referenced by IMAGE is too short to contain the record. If the record is shorter than IMAGE, it is left adjusted and the remainder of IMAGE is filled with blanks. Finally the position indicator of IMAGE is set to 1. When the last record has been read in, and INIMAGE is called again, a call on ENDFILE will return TRUE. Any further call on INIMAGE, INCHAR, INTEXT, ININT, INREAL or INFRAC will result in a run time error.

BOOLEAN PROCEDURE LASTITEM;

returns FALSE only if the external file contains more information (non-blank CHARACTERS). It scans past all blank CHARACTERS (calling INIMAGE if need be). If LASTITEM returns FALSE then the currently accessible CHARACTER of IMAGE is the first non-blank CHARACTER. If ENDFILE returns TRUE, a call on LASTITEM also returns TRUE.

CHARACTER PROCEDURE INCHAR;

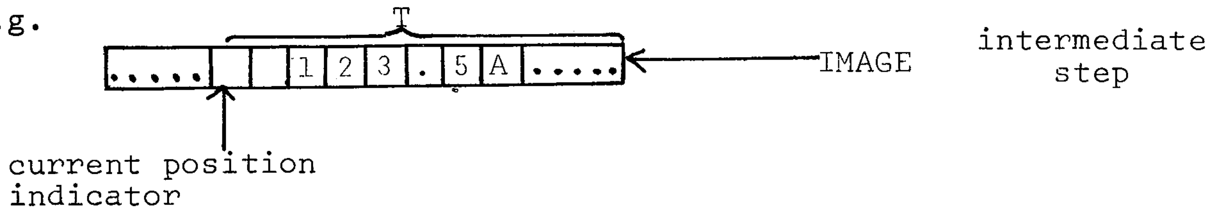
gives access to the next available CHARACTER and scans past it. If IMAGE.MORE is FALSE, the INIMAGE is called once and the value of the call is the first CHARACTER of the new image. INCHAR gives a run time error if an attempt is made to read past the last record in the file.

TEXT PROCEDURE INTEXT(W); INTEGER W;

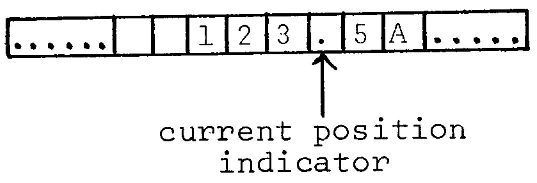
INTEXT(M) creates a copy of the next M CHARACTERS (which may be spread over several records) and returns a reference to this copy. If M < 0 or M > 2^{15}-20 then a run time error results. A run time error will also result if the file does not contain M more CHARACTERS, i.e. an attempt is made to read past the last record.

The remaining PROCEDURES treat the file as a continuous stream of records. They scan past any number of blanks (calling INIMAGE if need be) and then de-edit a numeric item lying in one image. This is done by calling LASTITEM (which scans past the blanks) and then referencing the remainder of the current IMAGE by a temporary TEXT variable, say T. The value of the "IN***"-PROCEDURE call is the value of the corresponding call on T."GET***". On exit, the current position indicator is updated to reference past the de-edited field, i.e. to reference the first CHARACTER which is not a part of the de-edited numeric item.

e.g.



T.GETINT = 123



ININT = 123

Run time errors will result if the remaining CHARACTERS in the file are blanks (LASTITEM = TRUE) or if the item is not numeric.

An outline of ININT is:

```
INTEGER PROCEDURE ININT;
BEGIN  IF LASTITEM THEN ERROR;
       T :- IMAGE.SUB(POS,LENGTH-POS+1);
       ININT := T.GETINT;
       SETPOS(POS+T.POS-1);
END *** ININT***
```

INREAL and INFRAC follow the same pattern.

CLASS OUTFILE

```
file CLASS OUTFILE; VIRTUAL : PROCEDURE OUTIMAGE;
BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;....;
       PROCEDURE CLOSE;.................;
       PROCEDURE OUTIMAGE;..............;
       PROCEDURE OUTINT(I,W); INTEGER I,W;.............;
       PROCEDURE OUTFIX(R,N,W); REAL R; INTEGER N,W;....;
       PROCEDURE OUTREAL(R,N,W); REAL R; INTEGER N,W;...;
       PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;.........;
       PROCEDURE OUTTEXT(T); VALUE T; TEXT T;...........;
       PROCEDURE OUTCHAR(C); CHARACTER C;...............;
END ***OUTFILE***
```

PROCEDURE OPEN(BUF); TEXT BUF;

Follows the pattern set by the PROCEDURE OPEN listed with CLASS file.

PROCEDURE CLOSE;

Conforms to the pattern set by the PROCEDURE CLOSE listed with CLASS file but in addition checks the value of POS. If POS ¬= 1 then presumably extra information has been copied into IMAGE since the last call on OUTIMAGE.

Accordingly, if POS ¬= 1, OUTIMAGE will be called once before the OUTFILE is closed.

PROCEDURE OUTIMAGE;

OUTIMAGE transfers the contents of IMAGE to the external file creating a copy as a new record. IMAGE is then cleared to blanks and its current indicator set to one.

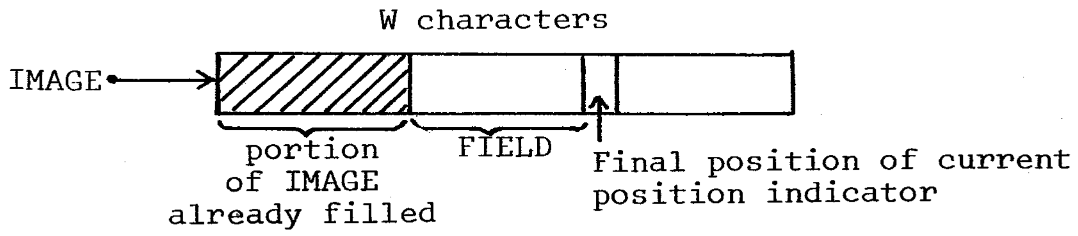
<u>PROCEDURE OUTTEXT(T); VALUE T; TEXT T;</u>

A copy of the CHARACTER sequence represented by the actual parameter is edited into IMAGE from the current position. If the remaining length of IMAGE is insufficient, INIMAGE is called and the editing process proceeds. Thus the TEXT value may be split over several external records.

<u>PROCEDURE OUTCHAR(C); CHARACTER C;</u>

Outputs the value of C into the current position of IMAGE (if MORE = FALSE, then OUTIMAGE is called first). In either case, the current position indicator is then incremented.

The remaining PROCEDURES are all based upon the PUT-PROCEDURES local to TEXTs. The corresponding PUT-PROCEDURES are augmented by an extra parameter W which specifies the field width.


W characters
IMAGE
portion of IMAGE already filled
FIELD
Final position of current position indicator

The editing PROCEDURE commences by establishing a temporary TEXT reference (FIELD) to the next sequence of W CHARACTERS lying in one IMAGE. If the current IMAGE has not enough space left, INIMAGE is called. Then the value is edited by calling FIELD."PUT***" where "PUT***" is the PUT-PROCEDURE corresponding to the OUT-PROCEDURE. Finally the current position indicator is increased by W to reference past FIELD - past the just-edited field.

```
PROCEDURE OUTINT(I,W); INTEGER I,W;
     FIELD(W).OUTINT(I);

PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;
     FIELD(W).PUTFRAC(I,N);

PROCEDURE OUTREAL(P,N,W); REAL P; INTEGER N,W;
     FIELD(W).PUTREAL(P,N);

PROCEDURE OUTFIX(P,M,W); REAL P; INTEGER M,W;
     FIELD(W).PUTFIX(P,M);
```

CLASS PRINTFILE

```
     OUTFILE CLASS PRINTFILE;
     BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;...............;
            PROCEDURE CLOSE;.....;
            PROCEDURE LINESPERPAGE(N); INTEGER N;........;
            INTEGER PROCEDURE LINE(N); INTEGER N;........;
            PROCEDURE SPACING(N); INTEGER N;.............;
            PROCEDURE EJECT(N); INTEGER N;...............;
            PROCEDURE OUTIMAGE;..........................;
     END ***PRINTFILE***
```

CLASS PRINTFILE further orients the prefixing CLASS OUTFILE towards line printer output. The PROCEDURES OPEN and CLOSE take the pattern of those local to OUTFILE but in addition OPEN positions to the top of the next page.

PROCEDURE LINESPERPAGE(N); INTEGER N;

Initially the number of printable lines per page is fixed at some value (V) dependent upon the installation. A call LINESPERPAGE(M) will alter this figure to allow only M printable lines per page. A run time error results if M < 0 or M > V.

PROCEDURE SPACING(N); INTEGER N;

Initially the spacing is 1 and successive images are printed on successive lines. A call SPACING(M) will alter this to separate successive lines by M-1 blank lines. This becomes effective _after_ the next call on OUTIMAGE. If M > "current value of lines per page", or M < 0, then a run time error results. If M = 0, overprinting will occur - successive images being printed on the same physical line.

PROCEDURE EJECT(N); INTEGER N;

This PROCEDURE skips to a certain line on the page - (it avoids calling OUTIMAGE several times). EJECT(L) will position to line L on this page if this is further down the current page (if L > LINE), or else skip to LINE L of the next page if L <= LINE.

A run time error occurs if L < 0. If L > LINESPERPAGE, EJECT(L) is equivalent to EJECT(1).

INTEGER PROCEDURE LINE;

This PROCEDURE returns the INTEGER value of the line number which indicates the next line to be printed. Thus EJECT(LINE+3) will skip three lines and not alter spacing. After each call on OUTIMAGE, the line number is incremented by the current spacing.

PROCEDURE OUTIMAGE;

This PROCEDURE acts like the OUTIMAGE of OUTFILE but in addition increments the line number by spacing, and will position to the top of the next page if the current page is filled.

Example:

This example shows the use of three types of file and how to open and close them. The example was chosen to demonstrate these features and how formatting is available by use of the sub-text concept. The logic of the example is particularly simple. A file of transactions has been punched on cards in the format

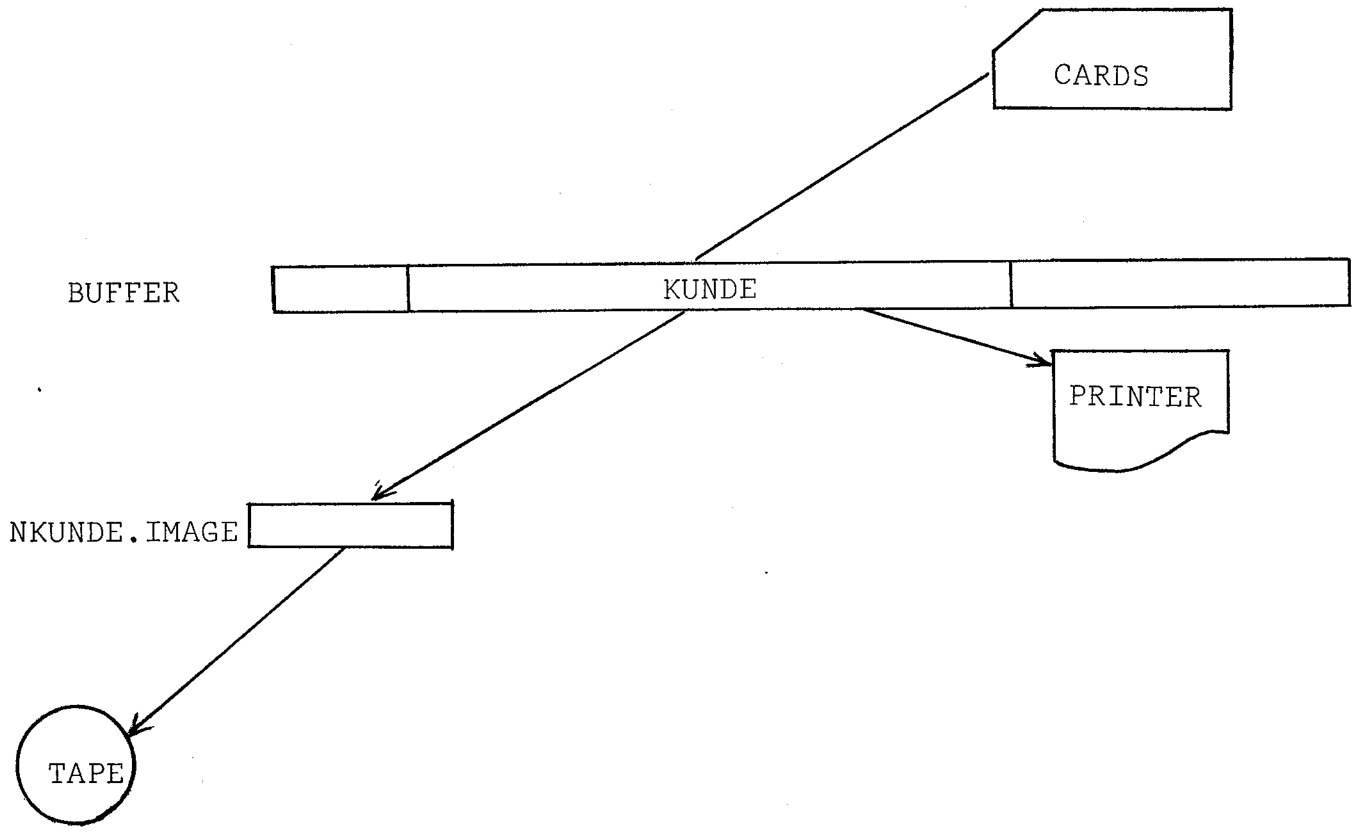
col 1-5	customer number	KNR
col 7-16	sum of debit transactions	DB
col 19-28	sum of credit transactions	KR

Each transaction is on a fresh card. The information is to be compressed and recorded on a new sequential file on tape. The length of each tape record is 15 CHARACTERS, and its format is:

5	10
KNR	KR - DB
customer number	total sum of transactions

Checks are made that the customer number is valid (1-3-7 digit check) and that the card is validly punched. If not, a copy of the card is printed on a line printer, and the scan continues.

Note that the INFILE and PRINTFILE share the same buffer.



```
//TAPE   DD   DSN=A,DISP=(NEW,CATLG),LABEL=(1,SL),UNIT=TAPE,VOL=SER=TAPE1

//PRINTER   DD   SYSOUT=A
//CARDS     DD      *

    <cards>

/*
```

```
BEGIN  TEXT KNR, DB, KR, NR, T1, T2, T3, T4, KT, TSUM,
            TS, BUFFER;
       INTEGER SUM, SALDO;
       REF(INFILE) KUNDEKORT;
       REF(PRINTFILE) PRINT;
       REF(OUTFILE) NKUNDE;

       BOOLEAN PROCEDURE NONNUMERIC(T); TEXT T;..........;
       COMMENT ***THIS PROCEDURE RETURNS TRUE IF THE TEXT
                  PARAMETER CAN NOT BE NUMERICALLY INTER-
                  PRETED FROM THE LEFT*** ;
       BUFFER :- BLANKS(132);
       BUFFER := "ERROR";
       COMMENT ***CONSTRUCT THE CARDFILE*** ;
       KUNDEKORT :- NEW INFILE("CARDS");
       KUNDEKORT.OPEN(BUFFER.SUB(10,80));
       KNR :- BUFFER.SUB(1,5);
           NR :- KNR.SUB(1,4);
              T1 :- NR.SUB(1,1);
              T2 :- NR.SUB(2,1);
              T3 :- NR.SUB(3,1);
              T4 :- NR.SUB(4,1);
           KT :- KNR.SUB(5,1);
       DB :- BUFFER.SUB(7,10);
       KR :- BUFFER.SUB(19,10);
```

```
COMMENT ***CONSTRUCT PRINTFILE*** ;
PRINT :- NEW PRINTFILE("PRINTER");
PRINT.OPEN(BUFFER);

COMMENT ***CONSTRUCT TAPEFILE*** ;
NKUNDE :- NEW OUTFILE("TAPE");
NKUNDE.OPEN(BLANKS(15));

COMMENT ***CONSTRUCT WORKING TEXT TSUM*** ;
TSUM :- BLANKS(3);
TS :- TSUM.SUB(3,1);
```

```
       INSPECT NKUNDE DO
       BEGIN
              KUNDEKORT.INIMAGE;
              WHILE ¬KUNDEKORT.ENDFILE DO
              BEGIN  IF    NONNUMERIC(KNR)  OR
                           NONNUMERIC(DB)   OR
                           NONNUMERIC(KR)
                     THEN  ERROR:  PRINT.OUTIMAGE
              ELSE  BEGIN  COMMENT ***137 DIGIT CHECK*** ;
                           SUM := 7*(T1.GETINT + T4.GETINT) +
                           3*T3.GETINT + T2.GETINT;
                           TSUM.PUTINT(SUM);
                           IF TS¬= KT THEN GOTO ERROR;

                           COMMENT ***OUTPUT TO TAPE*** ;
                           SALDO := KR.GETINT - DB.GETINT;
                           OUTTEXT(KNR);
                           OUTINT(SALDO,10);
                           OUTIMAGE;
                     END;
                    KUNDEKORT.INIMAGE;
              END;
       END ***INSPECT NKUNDE*** ;

SLUTT: KUNDEKORT.CLOSE;
       NKUNDE.CLOSE;
       PRINT.CLOSE
END
```

CLASS DIRECTFILE

```
file CLASS DIRECTFILE;
     VIRTUAL: PROCEDURE LOCATE, ENDFILE, INIMAGE, OUTIMAGE;
BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;...................;
       PROCEDURE CLOSE;................................;
       INTEGER PROCEDURE LOCATION;.....................;
       PROCEDURE LOCATE(I); INTEGER I;.................;
       BOOLEAN PROCEDURE ENDFILE;......................;
       PROCEDURE INIMAGE;..............................;
       CHARACTER PROCEDURE INCHAR;.....................;
       BOOLEAN PROCEDURE LASTITEM;.....................;
       INTEGER PROCEDURE ININT;........................;
       REAL PROCEDURE INREAL;..........................;
       INTEGER PROCEDURE INFRAC;.......................;
       TEXT PROCEDURE INTEXT(W); INTEGER W;............;
       PROCEDURE OUTIMAGE;.............................;
       PROCEDURE OUTCHAR(C); CHARACTER C;..............;
       PROCEDURE OUTINT(I,W); INTEGER I,W;.............;
       PROCEDURE OUTFIX(R,N,W); REAL R; INTEGER N,W;......;
       PROCEDURE OUTREAL(R,N,W); REAL R; INTEGER M,W;.....;
       PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;...........;
       PROCEDURE OUTTEXT(T); VALUE T; TEXT T;.............;
END ***DIRECTFILE***
```

A direct file represents an external file in which individual records are addressed by indices (ordinal numbers). The index of the current record is returned by a call on LOCATION. The current record may be copied into the program by a call on INIMAGE, or overwritten by a call on OUTIMAGE. In either case, the sequentially next record is then taken as the current record. This sequential accessing may be altered at any time a call LOCATE(M) which will locate the Mth external record and make it the new current record.

PROCEDURE OPEN

conforms to the pattern of OPEN in CLASS file but in addition locates the first record.

PROCEDURE CLOSE

conforms to the pattern of CLOSE in CLASS file.

PROCEDURE ENDFILE

is FALSE if the current index locates a record in the file.

Calls on the PROCEDURES INIMAGE and OUTIMAGE will cause run time errors if ENDFILE is TRUE otherwise they conform to these of the same identifiers in INFILE and OUTFILE but in addition increment the index of the current record by one.

The remaining PROCEDURES are analogous to the corresponding PROCEDURES of INFILE and OUTFILE.

CLASS BASICIO

The system defined file facilities are grouped together in the CLASS BASICIO whose skeleton reads:

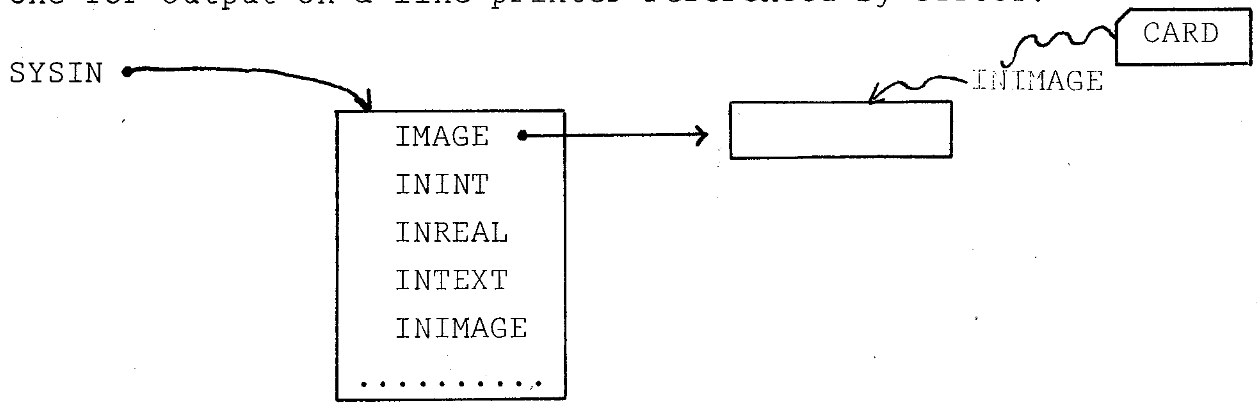
```
CLASS BASICIO(LINELENGTH); INTEGER LINELENGTH;
BEGIN  CLASS file....................;
       file CLASS INFILE............;
       file CLASS OUTFILE...........;
       file CLASS DIRECTFILE.........;
       file CLASS PRINTFILE.........;
       REF(INFILE)sysin;
       REF(PRINTFILE)sysout;
       REF(INFILE) PROCEDURE SYSIN; SYSIN :- sysin;
       REF(PRINTFILE) PROCEDURE SYSOUT; SYSOUT :- sysout;
       sysin :- NEW INFILE("SYSIN");
       sysin.OPEN(BLANKS(80));
       sysout :- NEW OUTFILE("SYSOUT");
       sysout.OPEN(BLANKS(LINELENGTH));
       INNER;
       sysin.CLOSE; sysout.CLOSE;
END ***BASICIO***
```

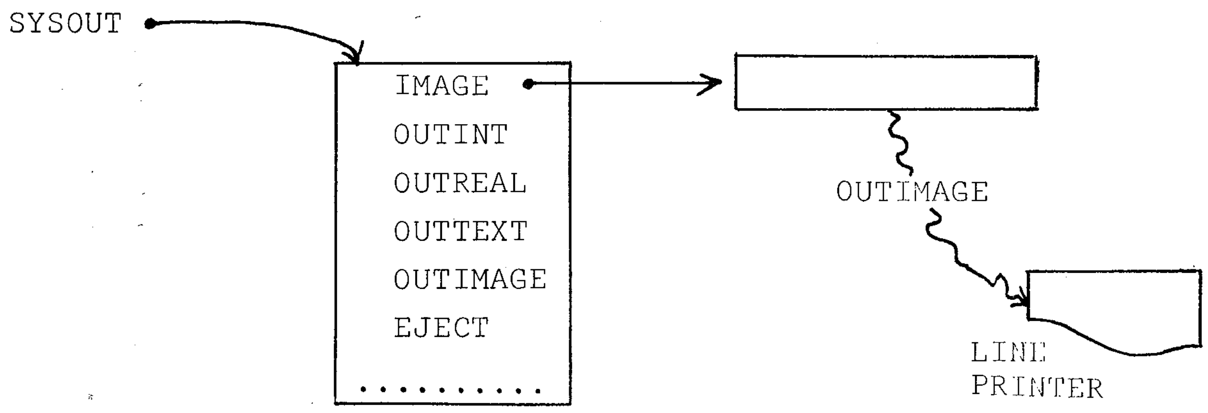
BASICIO contains actions to generate an INFILE (SYSIN for cards), and a PRINTFILE (SYSOUT for line printer). These objects are accessible only through PROCEDURES which copy the values of certain identifiers (sysin, sysout) which are otherwise not accessible by the user.

A user's program behaves as though it is enclosed as follows:

```
BASICIO(132)  BEGIN  INSPECT SYSIN DO
                       INSPECT SYSOUT DO
                         <program>
              END;
```

When a user program begins the system automatically generates two files - one INFILE for card input referenced by SYSIN, and one for output on a line printer referenced by SYSOUT.





When the actions of the user defined program are exhausted, control returns to the prefix level of the BASICIO object and continues after the INNER. The following three statements close the three system generated files.

The inspect statements enclosing the program allow the user to write ININT, INIMAGE,..... instead of SYSIN.ININT, SYSIN.IMAGE and OUTREAL, OUTIMAGE,.... instead of SYSOUT.OUTREAL, SYSOUT.OUTIMAGE. There are attribute name clashes

```
OPEN  }  which should never be used for
CLOSE }  SYSIN or SYSOUT
IMAGE
SETPOS
POS
MORE
LENGTH
```

When these occur they are naturally bound to SYSOUT and the corresponding attributes of SYSIN may be obtained by writing SYSIN.SETPOS, SYSIN.IMAGE etc. Alternatively, an input section may be written as

```
INSPECT SYSIN DO
BEGIN
        input - in this block occurrences IMAGE, SETPOS,
        POS, MORE and LENGTH are bound to SYSIN
END;
```