3. THE KDF9 IMPLEMENTATION.

3.1 General.

In this section the implementation of TMS on KDF9 is covered in detail. The UCA3 routines are described first because appreciation of AMTSIM depends on an understanding of their operation and significance.

3.2 The list processing package.

The subdivision of a KDF9 word known as Q-store format is remarkably suitable for representing a node in a bifurcating tree structure. Since 16 bits suffice to contain any KDF9 word address Q-store format can contain two pointers to subtrees and a description of the node in a 16 bit tag. A simple list processing scheme has been developed to exploit this.

The package consists of four groups of routines which

1) declare new lists,
2) create list structures,
3) fetch and update list components, and
4) perform housekeeping functions.

Each list is composed of a number of cells (KDF9 words in Q-store format) subdivided into the I (counter), J (increment) and K (modifier) parts. The K part of a cell contains either the address of the next cell on the list or else a zero terminator. The J part acts as a tag field which can be considered to be a 16 bit signed integer. If the J part is negative the I part is assumed to contain the address of the first cell of a sublist, while if the J part is positive or zero the I part is assumed to contain a 16 bit integer datum. Thus the I part corresponds to LISP's CAR, the K
part to LISP's CDR and the J part to an "atomic / non-atomic " marker.
If p is a variable containing the address of a list cell, the functions
IP(p), JP(p) and KP(p) yield the I, J and K parts of the cell respectively. Three corresponding subroutines, SETI(p, v), SETJ(p, v) and
SETK(p, v) allow the subfields to be set to the value of v. A new
cell containing specific values in its subfields is created by the
function IJK(i, j, k) whose value is the address of the new cell.
IJK in turn calls NUCELL(rubbish), a function of one, ignored, argument
whose value is the address of a new cell with subfields of indetermin-
ate value. (The dummy argument circumvents the FORTRAN prohibition on
functions of no arguments). IJK is thus an analogue of LISP's CONS.

Two storage areas are required for the purposes of the scheme,
one to contain the list cells and one for administrative use. In order
not to tie these to a fixed size and to avoid the use of COMMON blocks,
always a messy feature, the user allocates these areas as is convenient
to him then notifies the package of their position and extent by
calling the housekeeping routine INITAS (initialise available space).
INITAS is a subroutine with five parameters, the first four being
the bounding entries of the two areas and the fifth being a flag
which, if non-zero, requests the routine to initialise a dump of the
list storage area for output in the event of program failure. A call
to INITAS must be obeyed before any other routine in the package is
used.

During the course of processing, cells often become detached
from lists. Such cells are garbage words, garbage being the picture-
sque American term for inaccessible storage. The package maintains
a list of cells not in use (the freelist) which initially (i.e. after
calling INITAS) contains the whole of the available space. As calls
of NUCELL demand cells, they are removed from the freelist, and the point may eventually be reached where there are no cells left to satisfy the demand and yet a lot of storage is lying inactive in the form of garbage. When this happens NUCELL calls in the garbage collector, a routine whose mission it is to scan the available space, detecting garbage and returning it to the freelist. This process is completely automatic and beyond user control.

It follows that the garbage collector must be able to distinguish between useful cells and garbage cells. Now, useful cells fall into two categories, 1) those on named list structures and 2) "anonymous cells " created during the evaluation of expressions such as \( IJK(IJK(A, B, C), -1, IJK(X, Y, Z)) \). Means are provided to make the protection of useful cells fully automatic.

Consider first the cells on lists based in a variable of the main program. These are protected by notifying the package of the variable used as the base. The garbage collector regards any cell accessible from such "declared lists " as protected and does not sweep them into the freelist. A variable is declared by the call \( \text{DCLST(\text{variable})} \) and a section of an array can be declared by the call \( \text{LSTARR(\text{array, number of lists in section})} \). Thus the calls:

\[
\text{DCLST(NMTREE);} \quad \text{LSTARR(ILF, 1024);} \\
\]

declare a list based in NMTREE and 1024 lists starting at the first element of the array ILF.

Anonymous cells are even more vulnerable than cells on main lists, and their protection is completely automatic, requiring no effort by the user. To see this, consider again the expression above, and suppose that only two cells remain upon freelist.
IJK(A, B, C) and IJK(X, Y, Z) remove them. Now the outer IJK demands a new cell and none is available so the garbage collector is entered. But the cells created by the inner calls of IJK are not yet safely on a declared list - how can they be protected? The answer is that NUCELL, before returning the address of the new cell it is about to deliver, also places that address on a table, in fact the very table used to remember declared lists. The garbage collector regards any such cell as safe from its depredations.

It will be evident that a call of NUCELL now consumes two words, one in the listspace and one in the table of declared lists. Clearly some means must be provided of retrieving the latter if the scheme is not to consume inordinate amounts of storage space. Before introducing it however we first make what may initially appear to be a digression.

Consider the following skeleton of an Algol program.

```algol
begin integer l; integerarray a[1:1000];
initas(a[1], a[50], a[50], a[1000], 0);
begin integer m; dclst(m);

Y: m= ijk(3,0,0);

end;

X: end
end
```

If the garbage collector is called in at X it will regard as protected any cell accessible from the word in core which was used by the Algol program to hold the variable m (by virtue of the call of DCIST in the inner block). But since these cells are now inaccessible to the Algol program via m they should not be protected on that account. More seriously, the contents of that word in
core may have been corrupted by an extension of the Algol stack
after exit from the inner block (on account of a procedure call for
example) and any attempt to follow the content of a corrupted word
by the garbage collector would probably be disastrous. To prevent
this the declaration of m must be cancelled in some way before exit
from the inner block. In other words the declarations of lists to
the package must follow the same dynamic course as the declarations
in the Algol program, and a stack structure is required.

The scheme does in fact provide such a facility, which is
employed in the following manner. All calls of DCLST or I-STARR
within a block must be preceded by a call of the routine BGNIST,
which pushes down the declared lists table and thus serves to com-
plement the begin of the block. DCLST and I-STARR may then be used
as required, mirroring the Algol declarations. Finally, immediately
before leaving the block and certainly before making any further
use of the scheme outside the block, one must call ENDIST which
complements the Agol end. ENDIST pops up the declared lists table,
thus removing protection from all the cells declared implicitly or
explicitly since the last call of BGNIST. In this way the amount
of storage required for the declared lists table is minimised, solving
the problem posed before this 'digression' and the protection afforded
by the system remains absolute where and only where it is needed.
The routine CLSTCK clears the declared lists stack, thus cancelling
all declarations and should be called immediately after INITAS. An
extra call of BGNIST after this CLSTCK will ensure that the store
is not corrupted by an excess of obeyed ENDISTs over obeyed BGNISTs.
The Algol program skeleton now reads as follows.
begin integer n; integer array A[1:1000];

initas(A[1], A[50], A[51], A[1000], 0);

clstck; bgnlst; bgnlst; dclist(n);

begin integer m; bgnlst; dclist(m);

begin integer i; bgnlst; dclist(i);

m := ijk(3,0,0);

endlst;

end;

X:

endlst;

end;
3.3 The transput routines.

In programs designed for interactive use it is important to keep the core requirement to a minimum (thus reducing disc swapping overheads). This consideration prompted the decision not to employ the Algol or FORTRAN I/O packages in TMS since they are comparable in size with TMS itself. Concomitant advantages of this approach were (1) greater efficiency (the means being tailored to the end) and (2) greater flexibility. The primary disadvantage was that AMTSIM became machine-dependent to the extent that explicit numerical equivalents of the characters had to be used. To replace the standard scheme a very basic system was constructed to work on a character-by-character technique. It consists of a single input routine (P89), two routines to effect output (P1000 and P1001) and a short routine to request dumps of the buffers, all being written in UCA3.

P89 has three entry points named MXTCH, INPCH and RESINP. MXTCH is an integer function of no arguments which delivers the Egdon card code equivalent of the next character in the input stream, advancing the stream by one character. In order to signal "end-of-card" to the calling sequence P89 maintains a count of the number of card columns of the current card which remain to be read, subtracting one from this count each time a character is delivered. Whenever the routine is entered with this count zero a new read is initiated, the count is reset to 80 and a newline character is returned as the value of the function. By the simple expedient of declaring the count in a Vstore to be zero on loading, the routine was made self-initialising.
INPCH is also an integer function of no arguments, but it has no side effects and delivers the value of the last character delivered by NXTCH. By this means a small measure of "look-behind" is provided. Should INPCH be called before NXTCH has initialised itself, it delivers a dummy character as its value (77). RESINP extends the range of "look-behind" by allowing the user to backtrack to the start of the current card so that the next call of NXTCH would yield the first column over again. RESINP is a subroutine of no arguments.

No attempt was made to buffer input efficiently for two reasons (1) the Egdon director's "fetch card image" routine is effectively single-buffered, thus causing a bottleneck in the flow from the input device and (2) multiple-buffering would mean that the user would have to supply redundant lines at the end of his typed input to ensure that all the relevant input was swept through the buffers before the next interaction. In background use however the existence of the "fetch card image" buffer in addition to the program's buffer means that, in effect, the input is double-buffered (although using the less efficient "transfer buffer" scheme and not the better "alternate buffer" scheme).

By contrast, the output routines do employ double (transfer) buffering, since the user is in control of the lineprinter even under Egdon. The two buffers are known as the "assembly" buffer and the "device" buffer, new output being assembled in the first, character by character, before being moved to the latter for transmission to the output device (whence the names).
The output scheme consists of two routines, P1000 and P1001. Generally speaking, P1000 is responsible for servicing the assembly buffer and P1001 for initiating transfers via the device buffer. This division of responsibility was adopted to permit the actual device type to vary (by changing P1001) without substantial recoding, because while the routines were being written it was not known if online use would necessitate special output arrangements (it does not). As a result of this flexibility the scheme has been employed in a number of other programs without any alteration.

P1000 has three entry points named OUTCH, PUTCH and DEFIOP. Unlike OUTCH, OUTCH is not self-initialising and a call of DEFIOP (define output procedure) must be obeyed before OUTCH or PUTCH is used. The first of DEFIOP's two arguments (if non-zero) requests OUTCH/PUTCH to terminate the run with a core dump request in the event of an output failure. The second argument (if non-zero) should be a routine entryname which is nominated as an "emergency procedure" to be called on failures instead of dumps. If both parameters are zero no action whatever is taken over failures, and an exit from OUTCH or PUTCH is effected without comment. DEFIOP stores its parameters in V stores of P1000, initialises the buffer housekeeping and clears the assembly buffer to all spaces (all bit-zeroes on KDF9, so that new characters can be ORed into position). A 15-word buffer is used, sufficient for 120 characters of output.

OUTCH is the normal entry point to P1000 and should be called with the character to be printed in the least significant bits of the parameter. These seven bits should contain a value less than 65 (the rest of the word is irrelevant), being the printer code or
64 to force the immediate output of an incomplete buffer-load. This last facility is used to ensure that all the results destined for the online desk reach it before a demand for input can be generated. It thus solves the inverse of the problem mentioned above in connection with input buffering. Should the character presented be greater than 64, say \( x \), then failure 1000 + \( x \) occurs. If an emergency procedure has been nominated it is entered with \( x \) in the top cell of the nest and the return link to the routine which called P1000 in the top cell of the SJNS. If dumps have been requested instead the error routine is called and error 11000 + \( x \) transpires, and the program suffers a dump termination. If neither is wanted, an immediate exit from P1000 is effected without in any way altering the state of the output system. It should be noted that a device failure may also cause these steps to be taken, but in this case \( x \) would be less than 64. If the argument is 64, a "buffer full" condition is faked, to force the current assembly buffer contents to be output just as soon as the previous transfer terminates (see below).

Having verified that the character is valid, OUTCH then stores it in V1 (to give a one-character look-behind) and also plants it in its place in the buffer. If it is a newline the newline count is incremented by one (this is a statistic required by the lineprinter OUT). A test is now made for "end of buffer" (i.e. all 120 positions full). If the buffer is full the pointers are reset, the return link is saved in a V store to avoid SJNS overflow, and the newline counter, buffer address and look-behind word are placed in the nest. The newline counter is then zeroised and P1001 is called to transfer the information. On return from P1001, the link is restored and control returns to the calling routine.
PUTCH is a subsidiary entry to P1000 which allows the use of "pseudo-octal" arguments (for mnemonic ease). It simply pre-processes its parameter then jumps back into OUTCH to perform the rest of its task. Thus the calls OUTCH(33) and PUTCH(41) both output an A.

P1001 is the only device-dependent part of the output system. Its job is to move the contents of the assembly buffer to the device buffer (being held up by a lockout if the previous transfer has not yet terminated), clear the assembly buffer to all spaces and then initiate the actual output transfer. It has two non-obvious features. First, if P1001 has been entered as the result of (actual or faked) buffer exhaustion the last character may not be a format effector. Now it is a hardware characteristic of the KDF9 lineprinter that a format effector character \( (02\text{ or }03) \) must be transmitted to it to prompt the printing of the data which has been transferred to it. Thus, if the last character in the device buffer is not a format effector an indeterminate amount of output would be lost by a straight forward write operation. To avoid this, P1001 examines the lock behind word supplied by P1000. If it does not contain a newline the area to be output is extended by one word, which is set to contain a newline and 7 dummy characters as packing. In this way loss of output is avoided.

The second feature of P1001 is that, to cut down printing time at an online desk, trailing spaces in the device buffer are not included in the area of the buffer to be transferred. This is achieved by scanning from right to left, decreasing the area to be printed by one word each time an all-space word is found until the first non-space word is encountered, when the scanning stops. This is why the phrase
"the area to be output" was used above instead of "the device buffer".

Having done all this, P1001 initiates the device transfer, sets zero in the nest (to indicate that the previous transfer has not failed - the lineprinter OUT returns no indications to the user program, but a paper tape version of P1001 would have to be able to signal parity failure) and returns to P1000.

The final routine in the scheme is the trivial DUMBUF (P1002), which merely requests dumps of the I/O buffers and the simulated TM tape to be set up.
3.4 The tape handling routines.

Four operations are essential to the success of the simulator - overwrite a cell, read a cell, move the head one cell to the right and move the head one cell to the left. For reasons which will become apparent later, it is more convenient to regard the latter pair as - move the tape one cell to the left and move the tape one cell to the right, respectively. In an ideal situation we would have the following:

However, efficiency considerations (not to mention the finite size of core memory!) militate against this and the following, poor approximation is adopted:

The scanned cell is indicated by the scanned cell pointer, which shows the character in the scanned word which contains the scanned cell. Moving the tape adds or subtracts one from the scanned cell pointer. Whenever such a move oversteps the confines of the scanned word, the scanned word buffer is written back to its place in the YT area at the point indicated by the scanned word pointer. The scanned word pointer is then updated as appropriate and, unless the YT area has been exceeded, the new scanned word is fetched to the scanned word buffer. This two-level approach is adopted for speed. Without the scanned word buffer, the read and write character routines would have to fetch both pointers at every call, then perform a modified fetch and a modified store of the scanned word. With the present arrangements, the scanned word has a fixed location (the scanned word buffer) and only the scanned cell pointer need be fetched. Even greater advantages would accrue if the scanned word buffer were a
Q store, but the conventions of the Egdon system forbid this.

Clearly, this implementation of the TM tape leaves infinitely much to be desired. In practice, however there can be no indefinitely extended computations and the resources of infinite store are not called upon. Nonetheless, a better approximation can be developed and is now described. We begin by recalling once again the analogy between the TM tape and the large magnetic tape backing stores of practical computers. We can take advantage of this by adding another hierarchy to the two-level system described above, and consider the YT area in which the tape is stored as being merely a core storage copy of a section of the actual simulated tape which is held on reels of magtape. This restores our "ideal situation" above, except that we now have the whole of the YT area accessible to the simulator (not to the TM's).

In practice, it is better to regard the YT area as a set of buffers used cyclically to buffer the TM tape. Whenever the scanned word pointer is changed we now check to see if the new scanned word is "below" or "above" the YT area. If this is the case, the buffers are rotated once in the appropriate sense and the content of one is written to one of two sets of magtapes which contain the information "spilled over" from each side of the buffers. The converse operation must be performed to read back in information previously written out when overflow in the opposite direction takes place. If the corresponding tape from which this information would be read is empty (no such data having been written, or it all having been re-read) the simulator must clear the buffer to all so. Since the computer operators can remove full magtapes from the computer and supply fresh ones in their place, or restore filled tapes as their contents become needed again, the vistas of infinite storage are opened up (at least in theory).
Only the two-level has in fact been implemented, but the magtape scheme can easily be added at any time. As the programs stand, the four basic tape operations are effected by \texttt{WCHIT} (write character to tape), \texttt{RCHIT} (read character from tape), \texttt{MTL} (move tape left) and \texttt{MTR} (move tape right) respectively. The coding of these routines is trivial. A fifth routine, \texttt{CISEART}, clears the tape to all so and resets the scanned word and scanned cell pointers at the start of each simulation.

3.5 Miscellaneous routines.

Two \texttt{UCA3} subroutines remain to be described - \texttt{SPLIT} / \texttt{ICOMB} and \texttt{ITIME}. \texttt{ITIME} is an integer function of no arguments whose value is the program run time consumed so far in this run. This is obtained (to the nearest second) from the Egdon director by means of \texttt{OUT 122}. \texttt{OUT 122} is not serviced by the current implementation of the COTAN EXECUTE command so a dummy \texttt{ITIME} which always returns zero is used for the interactive version of the program.

The "action" part of each quadruple is stored in the rules table in 16 bits, 10 bits giving the new state and 6 bits indicating either the new symbol or the direction of the move. \texttt{ICOMB} is an integer function of two parameters, the state and the symbol, whose value is a 16-bit packed form for use in the table. \texttt{SPLIT} is the converse routine, deriving the state and the symbol from the packed form as retrieved from the table (supplied as the first parameter).
Finally, we come to AMTSIM. This, the central control routine of the simulator, falls into three parts:

1) the input procedures
2) the output procedures
3) the simulation procedures.

The great bulk of AMTSIM is occupied by these procedure declarations, the rest consisting of an infrequently executed initialisation phase and the main simulator loop itself. Since the simulator is programmed in so modular a fashion, this is almost transparent and a description of the various procedures suffices to explain the functioning of AMTSIM.

The input procedures are built around the list-processing and character input schemes described above. The primary requirement was a means of recognising the various input commands. Now the command language (as can easily be verified) has a type 3 grammar. Such languages are efficiently recognised by finite-state machines (or programmed equivalents thereof). However the complexity of the language mirrors that of the machine and any changes to the language specification can require drastic reprogramming. To avoid this, the step was taken of replacing the finite-state machine by a "table-driven" type 3 language recogniser. (This is the procedure TREESW). The "table" is embodied in a list based in the variable NMTREE. This list contains a "decision tree", at each of whose nodes resides a character. If the input character equals it, the left fork is taken, if not, the right fork is taken instead. Should the right branch be null, the input is not a string in the command language and TREESW jumps to the label FAIL in the main loop. On the other hand, if the left branch leads to a mode whose character equals a space, the input has been recognised.
as the command whose type number is given in place of the left branch.

TREESW returns this value to the calling sequence as the value of TREESW, which is then used to index the switch MNSW in a goto which leads to a short routine to process that type of command. When no command is recognised we have a free-format line. At FAIL, a test is made to determine if a tape expression is being read (TEMAIL true). If this is the case, the input is added to the TM tape by a call of the procedure ADTAPE. If not, the rest of the line is ignored and a new line is read.

Where integers have to be read, this service is performed by the integer procedure ININT, whose coding is largely self-explanatory. The integer procedure NEXTCH simply delivers the next input character which is not a space. ADTAPE (as explained above) adds the current line of input to the TM tape. To achieve this, it must first recover the characters which have already been input and this is effected by RESINP. That done, it reads successive non-space characters from the line, writing them into successive positions on the TM tape and each time incrementing INCNT by one. INCNT has the job of tallying the number of characters which have been added to the TM tape since the /TAPE command was issued, in order that the simulator may find its way back to the proper place in the tape at the resumption of simulation.

The output procedures consist of OUTINT, PTERM and PSTAT. OUTINT is an analogue of ININT and is called whenever an integer is to be printed. PTERM is the routine with responsibility for printing the instantaneous description of the simulated TM, being called PTERM because this is usually done only in a terminal state. PSTAT on the other hand prints either the rule which the simulator is about to obey, or if no such rule exists (indicated by FND being false) then the warning message "/G15j NOT FOUND".
INPUT SCANNING TREE

RULES TABLE FORMATS

(a) No rule: \( ILF(i) = 0 \):

(b) One rule: \( ILF(i) \neq 0 \):

(c) Many rules: \( ILF(i) \neq 0 \):

\[
\begin{array}{c}
\text{\( S_1 \)} \\
\text{\( S_2 \)} \\
\text{\( S_3 \)} \\
\end{array}
\]
**INSERT** is the sole routine of the third group. Its function is to place a new rule in the table, printing a warning if a rule with the same prefix (QiSj--) already exists. To understand this routine it is necessary to understand the structure of the rules table itself, and the nature of the solution to the problem of how to ensure a fast table look-up when obtaining the next rule to be obeyed.

The rules table has been implemented as a forest of trees based in the array ILF. The tree based in the element ILF(n) contains all the rules which begin QnS--. In this way (assuming an equal number of rules per state) the search time is immediately cut by the number of states used in a given TM, as compared with a simple linear scan. Further, each such tree is arranged as a balanced, binary sort tree. The symbol part of the rule is used as the sort key, so that a search within the tree can be conducted as a fast logarithmic (binary) scan, looking for the node which has the desired symbol in its J-part. The I-part of that node then contains the 16-bit packed form of the action to be taken and the new state. The action to be performed is coded as an integer less than 62 if the symbol represented by that integer is to be written to the TM tape. A code of 62 indicates a type 2 rule and a code of 63 a type three rule. Since a tree cannot contain more than 64 rules, the maximum search length is 6 (log₂ 64) node comparisons. This is all depicted graphically on the previous page. One slight complication of this picture is occasioned by the desire to save store. If a node in the tree has no descendants it is represented by a single list cell whose K-part is zero. However, if a node has descendants, a second word must be used to hold the left and right link to them. Only at most one of these link will ever be null. Thus the storage required for each node is kept at the minimum needed for the information it must carry.
INSERT reflects this preoccupation with storage saving. Its first action is to form the packed 16-bit form of the "action." It then looks at the place where this is to be inserted and if this is zero, returns as its value a new single-word node. The calling sequence can then insert this at the appropriate point (if the calling sequence is the main loop, it will be stored in a previously-zero element of ILF.) It then examines the K-part of the place where it is to be inserted. If this is zero, we have the single-word node as the parent of the new node and a second word is constructed for the single-word node which contains a pointer to the new cell in either its I-part or its K-part according as the symbol of the new rule is less than or greater than that of its parent. If it is equal to that of its parent, the "GisJ replaced" warning message is printed and INSERT overwrites the old I-part with the new action (this uses no new store at all).

Should, however, the node be a fully-fledged two-cell type, then insert routine calls itself recursively to place the new rule beneath the current node. For the same reason that the search length never exceeds 6, the depth of recursion in this routine can never exceed 6 and thus using a recursive technique is economical.

To find the rule corresponding to a given configuration, the simulator employs the UCA3 integer function FIND which is treated here, as being logically part of this context (indeed, it was originally written in Algol and was transcribed to UCA3 for operating speed). The value of FIND(symbol, statetree) is the 16-bit packed form of the rule, if it exists. If not, the common store FIND is set false.

The remainder of AMTSIM consists of an initialisation phase which is obeyed only once per background run (never in an online run) and which sets up the input scanning tree, requests any dumps, etc.,
and a series of short routines dealing with each type of input command. This should be easy to understand, as it consists simply of applications of the battery of routines previously described. One of them, however, requires special mention - the routine labelled DISC, for this caters for a command which has not yet been described. The /DISC command instructs the simulator to write itself up to the disc in binary image form. It is this binary image which is used by the current implementation of the COTAN EXECUTE command. The writing is performed by the UCA3 routine UPDISC which is my version of the SUSPEN routine of (4). If the /DISC command is obeyed in the online environment, at present, the run will fail. When the new EXECUTE command becomes available, this will be redundant and the routine UPDISC will be replaced by a dummy which simply returns at once to the calling sequence.
4 Practical details.

Due to the lack of disc space the simulator exists in only two distinct versions - (1) TMS, an RLB disc program and (2) TMSIMBIN, an absolute binary (designed for interactive use). Because of this the disc program has been built with a view to adaptibility and it presents the user with several options which he exercises via a FRONTSHEET and a PREDATA in the job deck. These options are:

1) the length of the simulated tape
2) the number of states allowed
3) the size of the rules table
4) the size of the list stack
5) whether core dumps are to be output in the event of failure.

TMSIMBIN caters for a 1024 cell tape, 1024 states, about 500 rules and a 50 word list stack. (The question of dumps is not relevant in the interactive environment).

To vary these specifications one must understand the significance of the FRONTSHEET and the PREDATA. The FRONTSHEET is used solely to reserve storage for the simulated tape (it goes into YT stores).

Thus, to allot 8m cells to the tape, one employs a FRONTSHEET of the form

```
*FRONTSHEET
P TMS, YTn, YUO, END,
```

The other four options are the province of the PREDATA. To request i as the highest state (less than 1024), j as the size of the list stack and k as the size of the rules table, with m equal zero or one according as dumping is not or is required then the PREDATA takes the form

```
*PREDATA
COMMON FND, LIL, LSL, LAL, LDIP
```
LIL = i
LISL = j
LAL = k
LDIP = m

Note that j must be greater than about 10 and less than k (suitable values are about k/10 for small k and k/20 for large k). For the &STORAGE card the third field should be quoted as roughly 

(2600 + i + k + m) and the second field as 1760.

The error routine may signal an execution error. Those with numbers less than 10,000 are the standard failures documented in the Egdon system manual (see 3). An array subscript error is probably the result of using an invalid state in a quadruple card. Failures with numbers greater than or equal to 10,000 are reported by the simulator's own P91 and are to be interpreted as follows.

<table>
<thead>
<tr>
<th>FAILURE</th>
<th>REASON</th>
<th>CURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>invalid integer in input</td>
<td></td>
</tr>
<tr>
<td>10022</td>
<td>attempt to move past right of tape</td>
<td>increase n</td>
</tr>
<tr>
<td>10032</td>
<td>attempt to move past left of tape</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>10101 to 10603</td>
<td>list processing failure</td>
<td>contact author</td>
</tr>
<tr>
<td>11130 to 11400</td>
<td>list stack too small</td>
<td>increase j</td>
</tr>
<tr>
<td>11500</td>
<td>rules table too small</td>
<td>increase k</td>
</tr>
</tbody>
</table>

References cited in this report:

(1) Davis - Computability and Unsolvability.

(2) Turing - On computable numbers, with an application to the Entscheidungsproblem.

(3) ICL - Egdon system reference manual.

(4) Documentation of the EXECUTE command.