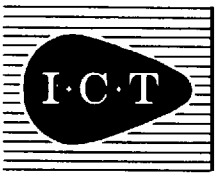


COMPUTER
PUBLICATION

I.C.T.
CONTROL AND SIMULATION MANUAL

INTERNATIONAL COMPUTERS AND TABULATORS LIMITED

I.C.T. HOUSE · PUTNEY · LONDON S.W.15

6/-

First Edition _____ *March 1966*

TL 1251

Issued by Technical Publications

International Computers and Tabulators Limited

Head Office: I.C.T. House, Putney, London S.W.15

Printed in Great Britain

PREFACE

Simulation is a technique for obtaining information about the performance of a system without actually putting that system into operation. A model is constructed so that the results obtained by operating or modifying the model indicate the results to be expected when the corresponding real system is operated or modified.

The technique has a wide range of application in operational research and can be used to study such varied activities as industrial processes, accountancy systems or marketing methods. In particular, the effect of many different approaches or strategies can be evaluated without implementing new systems or modifying those in existence.

Simulation is seldom applied to 'stationary' systems. It is of most interest and value when it is applied to systems that are affected by the passage of time, and in almost all such systems some sort of queueing takes place. A queue is formed when a number of items have to wait before passing through some stage in a process. The rate at which the queue grows depends on the rate at which items join the queue, and the time taken to process each item. Both these factors may vary in a random, or stochastic, way; the problem is to derive information about the way in which the queue size varies. For example, the average length of the queue may be obtained, or the probability of its exceeding a certain length. If there is any degree of complexity in the process - if, for example, there are a number of interrelated queues - a mathematical analysis becomes impossibly complicated.

The simulation technique is to create a model of the system by making lists of items at each stage in the process, and transferring items from one list to another in the correct chronological order. All the information required about the process can then be obtained by observation of the model. This technique can be carried out using a pencil and paper; however, for any but the most simple cases it becomes so cumbersome that the user of a computer is essential.

Languages such as FORTRAN and Algol have only rudimentary listing facilities, and consequently a number of special languages such as SIMON and C.S.L. have been developed. In addition to simplifying the writing of simulation programs, these languages may well find many applications outside the field of simulation, because of the powerful facilities they provide for examining and rearranging the membership of lists.

Part I

SIMON

Part 1

CONTENTS

INTRODUCTION

Simulation is a technique for obtaining information about the performance of a system by constructing a model that behaves in the same way as the system in all essential details.

The technique is usually applied where items passing through a system form queues at certain stages while waiting to be processed. In general, the sizes of these queues and the time taken for items to pass through the system vary in a random way.

The basic requirement for a model of such a system is a means of keeping ordered lists of items, and updating these lists in the correct chronological order. The main purpose of SIMON is to provide special facilities for doing this.

In SIMON, the various operations used in simulation programs are written in Algol, and declared as Algol procedures. The procedures are written in a block of program which is supplied to the user on a reel of paper tape, or a deck of punched cards, and is read into the computer before the simulation program. The normal structure and syntax of Algol are used, and anyone familiar with Algol should find no difficulty in learning and using SIMON. An additional advantage of SIMON is that users may easily supplement or amend the standard set of procedures.

The model formulated by a SIMON program consists of a number of individual components, called entities, each of which may exist in a number of discrete states. These entities are chosen so that by describing the state of each entity at a given time, the state of the model at that time is defined in as much detail as required.

The states of entities are recorded in two ways: each entity may have numerical values associated with it, corresponding to various states; or lists may be kept of all the entities in particular states. The method used depends on how information is to be referenced. For example, suppose we have four entities A, B, C, D, each capable of taking any or all of the four states w, x, y, z. In order to choose a particular entity and determine the states associated with it, we could prepare a table as follows:

ENTITIES:	A	B	C	D
	w	x	w	x
STATES	x	y	z	y
		z		

If, on the other hand, we wish to choose a particular state and determine which entities are in that state, we could prepare a table as follows:

STATES:	w	x	y	z
	A	A	B	B
ENTITIES	B	D	D	D
	C			

Either, or both, of these methods may be useful in a simulation program.

Chapter 1

Having set up a model, by defining the entities and states, and the lists which are to be made, a method is required for defining the way in which the model changes with the passing of time. At any time two classes of entities may exist: those which will change autonomously as a result of the passing of time, and those which will change only as a result of changes in other entities. The former are referred to as time-dependent entities, and the latter as time-independent entities. Each time-dependent entity is assigned a time-value, which is the time at which it is due to change its state. Time is advanced through the simulation by moving forward to the earliest time at which a change occurs. At this time all the consequent changes in the model are carried out.

The interval of time between changes is frequently determined by a number of unpredictable factors, so that an accurate prediction of the interval is impossible. Instead, observations must be made to determine the probability of the time interval falling within various ranges of values. From such observations probability distributions are prepared, which are provided as data for the SIMON program. Time-values are then obtained by sampling from the appropriate distributions.

Because of the random variations which occur in the processes simulated, the results obtained from a simulation will usually have to be subjected to some kind of statistical analysis. To facilitate this, results may be collected and output as histograms.

This manual is arranged as follows:

Chapter 1 describes the type of model formulated in a SIMON program, and the method used for setting up and running the model. The basic terms used in the manual are explained.

Chapter 2 gives a full list of SIMON procedures with details of the use of each.

Chapter 3 describes the preparation of data for SIMON programs and the form of the results which may be obtained.

Chapter 4 gives a simple example program, with the data provided and the results obtained, and an explanation of each step in the program.

Chapter 5 describes the implementation of SIMON for the 1900 series and Atlas computers.

It is assumed throughout the manual that the reader has a working knowledge of Algol and can refer to an Algol manual for specific details.

Chapter 1

BASIC DEFINITIONS

ENTITIES

The component parts of the model are called entities. The choice of entities is an important stage in defining a simulation problem. It is sometimes possible to group together several of the components of the real system into a single entity in the model. The degree to which the model can be simplified in this way will depend on how much detail is required in the results, and what factors are considered important enough to be taken into account in the model.

In SIMON, each entity is represented by a one-dimensional integer array.

The array is set up by a normal array declaration, and defined as an entity by the SIMON procedure: ENTITY.

Each of the elements of the array represents some property or attribute of the entity, and the value assigned to the element indicates the state of that attribute at a particular time. Consider, for example, an entity representing a ship. Suppose two items of information are required about the ship: whether it is in port or at sea, and whether it is loaded or unloaded. The entity could be declared

```
integer array SHIP [0:2]
```

and the second element of the array, SHIP [1], would take one of two values representing "in port" and "at sea", and the third element, SHIP [2], would take one of two values representing "loaded" and "unloaded".

Although the states are stored as integer values in the array, these values may be assigned to variables and the variable names used to describe entity states in the program. Thus we could write

```
LOCATION := 1 ;
ATSEA    := 1 ;
INPORT   := 2 ;
```

and then the statements

```
SHIP [ LOCATION] := ATSEA
```

and

```
SHIP [1 ] := 1
```

both have the effect of assigning to the second element of the array SHIP the value 1, indicating that the ship is at sea.

The element SHIP [0] is reserved for a value assigned automatically by the entity procedure.

A number of entities with the same set of attributes may be defined collectively as a groupentity. A groupentity is represented by a two-dimensional integer array. This array is set up by a normal array declaration, and then defined as a groupentity by the SIMON procedure: GROUPENTITY.

Each row of the array represents a single entity, and each column represents one attribute common to all entities. The first element of each column is reserved for a value assigned by the GROUPENTITY procedure.

For example, a fleet of five ships could be represented by

```
integer array SHIPS[1:5,0:2]
```

and the information

```
SHIPS[1, LOCATION] := ATSEA
SHIPS[3, LOCATION] := INPORT
```

would be stored as shown in diagram 1.

Diagram 1

	Reserved Column	Location	Cargo
Ship 1	[1,0]	[1,1]	[1,2]
Ship 2	[2,0]	[2,1]	[2,2]
Ship 3	[3,0]	[3,1]	[3,2]
Ship 4	[4,0]	[4,1]	[4,2]
Ship 5	[5,0]	[5,1]	[5,2]

Values assigned by program

1

2

SETS

Another way of storing the states of entities is to form lists of entities which are in particular states. These lists are called sets. For example a set called ATSEA could be established to contain the names of all ships currently at sea. Each state, as defined in this way, is necessarily two-valued, since an entity is either in the set or not in the set.

In order to record the change of state of an entity, the entity is added to or removed from the appropriate sets, without itself being altered.

If this method is used, the number of entities with a particular state may be found without examining the attributes of individual entities.

A set is formed, in SIMON, by declaring an integer variable, and then defining the variable name as the name of a set by means of the SIMON procedure: SET. The name defined is then used as a parameter to all the procedures which operate on that set.

QUEUES

Most simulation problems are concerned with queues, that is collections of items waiting, in order, to be processed.

All sets in SIMON have the properties of queues. Items may be added to a set only at the "head" or "tail" of the set, so that an order is established. The item which is first or last in a set at any time may be referenced by its position, and this facility enables queues to be simulated.

ACTIVITIES

The actions taken to change the state of the model are called activities. They fall into two classes referred to as B-phase and C-phase activites.

B-phase activities consist of such state-changes as take place as a direct result of the advance of simulation time. C-phase activities consist of state-changes which take place as a result of other

changes which have been made at that moment in simulation time. Thus C-phase activities are generally associated with tests, the results of which determine the changes that are required.

TIME ADVANCE

A simulation model exists in discrete states, so only the times at which state-changes occur need be considered, and simulation time advances indiscrete jumps.

In SIMON, time is advanced in the following way. Each time-dependent entity is assigned a time-value, which is the time at which the state of the entity will change. All time-dependent entities are listed in a set called TIMESET; this is not done automatically and the programmer must use one of the normal listing procedures. Before any activities can take place the timeset is searched for the entity with the smallest time-value. The time-value of this entity is the earliest time at which any change in the model can take place, so simulation time is advanced to that time. The appropriate activities can then be initiated.

The time-value to be assigned to an entity is obtained by adding to the current calue of the simulation time, a time period which may be a constant, a determinable variable, or a stochastic variable.

Consider the case of unloading a ship. This process would be idealized by assuming that the cargo and the discharging operations do not vary in any way, so that the time to unload is a constant.

A more realistic approach would be to take into account, for example, the amount of cargo carried by the ship. Assuming a fixed relationship between the amount of cargo and the unloading time, the time would be determined by an arithmetic expression.

In reality, the time taken to unload the ship would depend on various unpredictable factors, and so would be a stochastic variable. The time, in this case, would be obtained by sampling at random from a probability distribution of unloading times. Facilities which are provided in SIMON for obtaining values in this way are described below.

SAMPLING

Probability distributions which are to be sampled are provided as data. Pairs of values are read, consisting of a time t, and a probability P where P is the probability that an event will occur before the time t has elapsed. Thus, the values form a cumulative percentage probability distribution. In order to sample a time value from the distribution, a probability value is selected at random by means of a random number generator, and the corresponding time is found from the distribution, interpolating linearly where necessary.

There is, of course, nothing to prevent variables other than time being sampled in this way.

PROGRAM STRUCTURE

Although there is no restriction on program structure imposed by the language, most simulation programs fall naturally into three phases.

Phase-A consists of searching the timeset for the entity with the least time-value, and advancing the simulation time to the time found.

Phase-B consists of making such state-changes as are a direct result of the time-advance.

Phase-C consists of making any further changes that are required because of the new state of the model, set up by phase-B. As soon as changes are made in phase-C, further changes may be required, so phase-C is normally repeated until the model reaches a new "steady state" when no further changes are possible until the time is advanced again.

Besides these three phases, the program must contain statements which set the model in the required initial state, and statements which collect and output data obtained in the simulation.

MONITORING

Input and output are carried out by the usual procedures though some additional facilities are provided in 1900 SIMON. SIMON also provides procedures for the collection of results in histograms, which may be printed out in tabular form when required. The same procedures automatically calculate and print out the mean and variance of the values in a histogram.

Chapter 2

THE SIMON PROCEDURES

INITIATION

PLANTMASTER

This procedure must be called before any of the procedures described below (except the distribution sampling and histogram procedures) may be used. This procedure sets up the chain lists (master lists) in which entities and sets are stored.

ENTITIES AND SETS

ENTITY (NAME, REFERENCE NUMBER)

Specifies that NAME, which must have been declared as a one-dimensional integer array, is to be used to represent an entity. REFERENCE NUMBER is an arbitrary integer chosen by the programmer. Its value may be referred to by the REFNUM procedure which is described below.

The various listing procedures which operate on sets and entities require actual parameters with numerical values, so NAME[0] rather than NAME is used as a parameter to such procedures.

GROUPENTITY (NAME, NUMBER OF MEMBERS, REFERENCE NUMBER)

Specifies that NAME is to be used to represent a group of simular entities. NAME must have been declared as a two-dimensional integer array NAME[1:M, 0:N] where M is the number of members, and N is the number of attributes of each member. REFERENCE NUMBER is an integer chosen by the programmer, and used in the same way as for ENTITY.

SET (NAME)

Specifies that NAME, which must have been declared by an integer type declaration, is the name of a set.

LISTING PROCEDURES

ADDFIRST (NAME) TO: (SET NAME)

Adds NAME to the head of the specified set. NAME may represent an entity (in which case the first element of the array must be used in the call) or it may represent a set. A group-entity may not be added to a set: the members of the groupentity must be added individually.

EXAMPLES:

```
ADDFIRST (S S PIG [0]) TO: (EMPTY SHIPS)
ADDFIRST (SHIP [3,0]) TO: (EMPTY SHIPS)
```

Here, as in other procedures, use is made of the Algol facility that the sequence:

) any string of letters : (

is equivalent to a comma in a parameter list.

For example

) TO : (

is equivalent to a comma in the previous examples.

ADDLAST (NAME) TO: (SETNAME)

adds NAME to the tail of the specified set.

BEHEAD (SET NAME)

deletes the first member from the specified set.

BETAIL (SET NAME)

deletes the last member from the specified set.

DELETE (NAME) FROM : (SET NAME)

deletes the specified member from the specified set. NAME may represent an entity or a set. As usual, an entity must be represented by its first element; for example:

DELETE (S S DOG [0]) FROM : (EMPTY SHIPS)

ROTATE (SET NAME, N)

rotates the specified set, making the first member last, the second first, etc., N times.

IMPLICIT NAMES

HEADOF (SET NAME)

TAILOF (SET NAME)

refer to the first and last members of a set. These are Algol function procedures, and so may be used as parameters to other procedures.

For example:

ADDFIRST (TAILOF(SET 1)) TO: (SET 2)

adds the last member of SET 1 to the head of SET 2, leaving SET 1 unchanged.

REFERENCE TO NUMERICAL VALUES

SIZEOF (SET NAME)

is a function procedure which provides a value equal to the number of members in the specified set. This may be used in expressions, or as a parameter to other procedures. For example:

```
if SIZEOF (SET 1) = 0 then goto LABEL3
else PRINT (SIZEOF(SET 1));
```

MEMNUM(NAME)

is a function procedure which provides the value of the first subscript of one of the members of a groupentity. The parameter NAME will usually be an implicit name. For example, suppose the first member of a set EMPTY SHIPS, is the entity SHIP [5, 0], then we could

write:

X:=MEMNUM(HEADOF(EMPTYSHIPS))

which would assign to X the value 5. This is a useful method of referring to an entity whose identity is unknown. For example we could now write:

SHIP [X, LOCATION] : = ATSEA

REFNUM(NAME)

is a function procedure which provides the value of the reference number that the programmer has associated with the specified entity. NAME will usually be an implicit name. For example, if MEMBER refers to an entity with reference number 5, then

REFNUM(MEMBER)

provides the value 5.

This procedure has one particularly useful application; the REFNUM procedure may be used in place of an arithmetic expression in a switch designator. For example, suppose that a branch is to occur to B1, B2, B3, or C-phase, depending on which entity is selected by the A-phase scan. This could be carried out by the statements

```
Switch B:=B1,B2,B3, CPHASE;
.....
goto B [REFNUM (MEMBER)]
```

where each entity has the reference number 1, 2, 3, or 4, and MEMBER refers to the entity selected in the A-phase of the program.

TIMEVALUES

SETTIME(ENTITY NAME) TO: (VALUE)

stores the given integer value as the time value associated with the specified entity. VALUE may be replaced by an expression giving an integer result. For example:

SETTIME(SHIP[3,0]) TO: (CLOCKTIME + LOADING TIME)

TIMEVALUE(ENTITY NAME)

provides the value of the time associated with the entity. TIMEVALUE is a function procedure.

SCAN(TIMESET)FOR:(MEMBER)WITH:(LEASTTIME)

This is a special procedure for carrying out phase-A of the program. TIMESET is a set containing entities to which time values have been assigned. The procedure searches this set and selects the entity with the least time-value. Note that TIMESET is a parameter and in fact any set name could be used. The entity selected by the procedure becomes MEMBER; in other words MEMBER is used in subsequent procedures as an implicit name for the selected entity. The parameter LEASTTIME is assigned the time value of the selected entity. Thus, a typical series of statements would be as follows:

```
SCAN(TIMESET)FOR:(MEMBER)WITH:(LEASTTIME);
DELETE(MEMBER)FROM:(TIMESET);
CLOCKTIME:=LEASTTIME
```

These statements select the entity with least time-value, delete that entity from the timeset, and advance simulation time to the time associated with the entity selected.

DISTRIBUTION SAMPLING

RANDOM (NUMBER)

takes the next value from a stream of uniformly distributed random integers in the range 0-99.

NUMBER specifies which of ten streams is to be used. A call with NUMBER taking the value zero reads a random number from a specified input medium; otherwise, NUMBER must be an integer in the range 1 to 10.

DISTRI (NAME, NUMBER)

reads in the co-ordinates of a cumulative probability distribution in the order 'value', 'percentage probability'.

NUMBER must be in the range 0 to 10, and must also precede the distribution on the data tape. This number is used to check that the right data has been read for the required distribution. It also serves as a stream number when the distribution is sampled.

The first value of the percentage probability must be 0 and the last 100.

SAMPLE (DISTRIBUTION NAME)

takes a value sampled randomly from the specified distribution. This procedure calls the RANDOM procedure, using as stream number the value of NUMBER used as parameter to the DISTRI procedure.

HISTOGRAMS

HISTOGRAM (NAME, LOWBOUND, ZONEWIDTH)

sets up a histogram with the specified name. The histogram will have eleven zones; one below the value given by LOWBOUND, nine within consecutive zone widths above the lowbound, and one above the highest zone width.

ADDTO (NAME, VALUE)

adds one value to the specified histogram. NAME is the histogram name, VALUE the value to be added to the histogram.

WRITEDOWN (NAME, TITLE)

prints out the named histogram in tabular form proceded by TITLE.

The parameter TITLE must be a string, enclosed in string quotes. The mean and variance of the values in the histogram are also printed.

ADDITIONAL PROCEDURES

The following procedures have been added by I.C.T. to the set of procedures specified by the originator of SIMON.

PRIMAS

This procedure prints the contents of the master lists. This will consist only of numerical values, whose significance will not be apparent unless the user is familiar with the method of listing employed in the SIMON procedures. The PRIMAS procedure may, however, be of value in testing or debugging programs.

PRISET (NAME)

prints the entries in the mast lists which correspond to the set name, and members of the specified set.

PRIENT (NAME)

prints the entries in the master lists corresponding to the named entity.

EMPTY (SETNAME)

empties a given set.

BELONG (ENTITY, SET, MARKER)

sets MARKER equal to 1 if the named entity is a member of the specified set, otherwise sets it to -1. MARKER must be declared as an integer variable in the block in which it is used.

GHIST (NAME, LOWERBOUND, N, WIDTH)

declares NAME as a histogram with N zones. One zone is between -∞ and LOWERBOUND, N-2 are of width WIDTH, and the last is between LOWERBOUND + (N-2) *WIDTH and + ∞

NAME must be declared as an integer array dimensioned at least as NAME [0 : N + 5].

GADDTO (NAME, V)

adds the value V to the specified histogram which has been declared by GHIST.

GWRITEHIST (NAME, T)

is exactly analagous to WRITEDOWN but used when the histogram has been declared by GHIST.

Chapter 3

DATA

INPUT

The first item of data provided for any program written in SIMON must be a number which specifies the amount of storage to be allocated for the listing of entities and sets.

The various listing procedures used in SIMON operate on the elements of two-dimensional arrays called master lists. Values are entered in these arrays corresponding to entities, set names, and members of sets, and these entries are linked together by various cross-references.

These arrays are declared and used automatically by the SIMON program, and need not concern the programmer, except that he must specify the number of elements in each of these arrays. This number will be the first item of data read by the program. It may be determined from the formula:

$$SSS = 2E + S + M$$

where SSS is the value required; E is the number of entities defined in the program; S is the number of sets; and M is the maximum number of set members which may exist at any time.

If a particular entity may belong to several sets at the same time, it must be counted once for each of the sets, and once as an entity. For example, suppose a program defines 10 entities and 2 sets, and each of the entities may be added to both sets, then

$$E = 10$$

$$S = 2$$

$$M = 20$$

$$SSS = (2 \times 10) + 2 + 20 = 42$$

If, on the other hand, each entity may only belong to one set at any time, then

$$E = 10$$

$$S = 2$$

$$M = 10$$

$$SSS = (2 \times 10) + 2 + 10 = 32$$

The bulk of the data for simulation programs will usually consist of probability distributions. For each distribution, 21 numbers must be provided. The first number is an integer in the range 0 to 10 which identifies the distribution and also serves as a stream number when the distribution is sampled. The remaining 20 numbers are 10 pairs of values in the order: time, probability. The first value of probability must be 0 and the last 100, but neither time nor probability values need be evenly spaced. All values must be integers.

EXAMPLE

A typical set of data might be as follows:

50 (list size)

1 (identification/stream number)

Time	Probability
10	0
25	15
30	23
35	36
36	48
37	67
38	78
40	95
45	98
60	100

2 (identification/stream number)

180	0
195	13

_____ etc. _____

Distributions are read by means of the procedure DISTRI (NAME, N). NAME is the name given to the distribution by the programmer; N is the identification number associated with the distribution. The number used in the call is compared with the number provided in the data, to check that the distribution read is the one intended by the programmer. For example, the above data might be read by means of the statements:

```
DISTRI (SHORT TIMES, 1)
DISTRI (LONG TIMES, 2)
```

If these calls were made in the wrong order, however, an error message would be printed and the program would be halted.

OUTPUT

Output is handled by the normal Algol output procedures, except in the case of histograms which are output by the procedures WRITEDOWN (NAME, TITLE) and GWRITEHIST (NAME, TITLE), which tabulate the values in the histogram, and also calculate and print their mean and variance. NAME is the name of the histogram, and TITLE is a string which is to be printed as a heading to the histogram. TITLE must, of course, be enclosed in string quotes, and any spaces in the string must be represented by the appropriate symbol.

RANDOM NUMBERS

If random numbers are required in the program, the RANDOM procedure is used. This generates ten streams of numbers, and the parameter to the procedure is a number in the range 1 to 10 which selects the stream from which a number shall be taken.

The method used to generate these numbers in the 1900 and Atlas versions of SIMON are described in Chapter 5. If the standard RANDOM procedure is thought to be unsatisfactory for a particular application, the user may write his own procedure to generate random numbers.

Random numbers may also be prepared externally and read as data by the call RANDOM (0). Random numbers must be integers in the range 0 to 99.

Chapter 4

SIMON EXAMPLE

The following example shows the simulation of a barber's shop. There are two barbers who each take some random time to cut a customer's hair; also, customers enter the shop at random intervals. An output from the simulation is the size of the queue of customers waiting to have their hair cut.

The time for each barber to cut hair and the frequency of entry of customers is generated by randomly sampling three distributions referred to in the program respectively as TIMEABARB, TIMEBBARB, and TIMECUST. When used in conjunction with the procedure DISTRI, they are followed by an integer which acts as a check that the correct distribution is being read in, and as a stream number for the random number generator used by the procedure SAMPLE.

The following data was used for input data for the distributions:

Name	TIMEABARB		TIMEBBARB		TIMECUST	
Stream Number	2		3		1	
Data of distributions	5	0	6	0	0	0
	8	11	6	11	0	11
	9	22	7	22	1	22
	9	33	9	33	1	33
	10	44	10	44	1	44
	10	55	10	55	2	55
	10	66	11	66	2	66
	10	77	12	77	3	77
	15	88	13	88	3	88
	18	100	20	100	4	100

When TIMECUST is sampled, a random number is selected from Stream 1 and the appropriate time before the next customer enters the shop is found from the distribution; for example, if the random number were 76, the time value would be 2; if 77 the value would be 3. The random number selected, say r, is always integer and in the range $0 \leq r < 100$.

The example program omits the listing of the SIMON procedures which precedes the actual simulation program, and starts with a call of the procedure PLANTMASTER.

This is followed by an initialization phase which declares entities and sets, reads in distributions and sets up a histogram which is later to be output.

The program then enters phase-A, phase-B, and phase-C, as described on page 5. For clarity the beginning of each phase is labelled accordingly. Whenever a return is made to phase-A during the simulation, a print of the clocktime and the size of queue is made. The simulation is ended when the clocktime is equal to or greater than 34 units. Then the histogram listing the cumulative frequency of the queue size and the mean and variance of the distribution is printed.

The following is a listing of program and output, which should be sufficiently self-explanatory for someone familiar with Algol to follow it without further elaboration. Procedure names are written in capital letters; all variables in small letters; underlined words are delimiters.

The label STOP and the extra end statements are for the SIMON procedures (not listed) which will precede the main simulation program.

Program and Results

```
comment start of user's program;

PLANTMASTER;
for i:= 1 step 1 until 10 do zzxzz[i]:= i;
comment zzxzz is a global array for the 10 stream random number generator;
inputd:= 3;
comment 3 represents card reader input unit;
begin comment barbers shop with two assistants;
begin  integer queue, freebarbers;
       integer array customer, abarber, bbarber[0:0];
                     timecust, timeabarb, timebbarb, queuesize[0:20];
comment print titles;
WRITE TEXT ("P␣␣CLOCKTIME␣␣QUEUE");
comment define entities, sets and histogram, read in distribution;
ENTITY (customer, 1); ENTITY (abarber, 2); ENTITY (bbarber, 3);
SET (freebarbers); SET (queue); SET (timeset);
DISTRI (timecust, 1); DISTRI (timeabarb, 2); DISTRI (timebbarb, 3);
HISTOGRAM (queuesize, 1, 1);
comment initiate simulation at time zero;
       begin
              ADDFIRST (bbarber[0], freebarbers);
              ADDFIRST (abarber[0], freebarbers);
              SETTIME (customer[0], SAMPLE (timecust));
              ADDFIRST (customer[0], timeset);
              clocktime:= 0
       end;
comment enter phase A of simulation;
phase A: SCAN (timeset) FOR: (member) WITH: (leasttime);
comment enter phase B of simulation;
phase B: clocktime:= leasttime;
       if member = customer[0]
    then
    begin
       if SIZEOF (queue) < 4 then
       ADDLAST (customer[0], queue) else
       begin writetext ("customer␣left␣queue");
             newline (2); end;
       SETTIME (customer[0], SAMPLE (timecust) + clocktime);
       goto phase C;
    end else
```

```
    begin
        ADDFIRST (member, freebarbers);
        DELETE (member, timeset);
        goto phase C;
    end
comment enter phase C of simulation;
phase C: if SIZEOF (queue)> 0 ^ SIZEOF (freebarbers)>0
    then
    begin
        BEHEAD (queue);
        r:= HEADOF (freebarbers);
        if r = abarber [0]
    then begin    s:= SAMPLE (timeabarb);
                  SETTIME (r, s + clocktime);
        end
    else begin    t:= SAMPLE (timeabarb);
                  SETTIME (r, t + clocktime);
        end;
    ADDFIRST (HEADOF (freebarbers), timeset);
    BEHEAD (freebarbers);
    goto phase C;
comment a rescan at same time to ensure that no more action can be carried out during
        phase C;
    end;
comment now phase C is finished the simulation will either terminate or return to phase A
        to advance the time to the point when the next event will take place;
comment print clocktime and size of queue;
PRINT (clocktime, 8, 0); PRINT (SIZEOF (queue), 8, 0);
NEWLINE (1);
ADDTO (queuesize, SIZEOF (queue));
comment terminate if clocktime is greater than or equal to 34, else return to phase A;
if clocktime ≥ 34
then
    begin comment print the histogram and pause;
        WRITEDOWN (queuesize, 'queuesize');
        PAUSE (99);
    end;
goto phase A
end; end; end; stop: PAUSE (99); NEWLINE (2); end
```

An output from one run of the program was as follows:

CLOCKTIME	QUEUE
0	0
1	1
4	2
5	1
6	0
7	1
7	2
9	3
10	4
11	3
12	2
12	3
15	4

CUSTOMER LEFT QUEUE

19	4
19	3
19	2
20	3
23	4

CUSTOMER LEFT QUEUE

23	4

CUSTOMER LEFT QUEUE

23	4

CUSTOMER LEFT QUEUE

24	4

CUSTOMER LEFT QUEUE

26	4
29	3
29	4
31	3
33	4

CUSTOMER LEFT QUEUE

34	4

QUEUESIZE

1	2	3	4	5	6	7	8	9	10	11
3	3	4	7	11	0	0	0	0	0	0	0

MEAN 2.714

VARIANCE 1.847

Note that more than one event may occur at any one time; i.e., a customer entering the shop and a barber finishing cutting hair may coincide.

The histogram has the name 'QUEUESIZE'; the first row of figures represent the frequency ranges, the second the frequency. For example, if q is the queue size and f (q) the frequency of this queuesize

appearing, the table is read as follows:

$0 \leq q < 1$	$f(q) = 3$
$1 \leq q < 2$	$f(q) = 3$
$2 \leq q < 3$	$f(q) = 4$
$3 \leq q < 4$	$f(q) = 7$
.	.
.	.
.	.

Of course, q is integer since fractional people cannot exist.

Underneath the histogram is printed its mean and variance.

Chapter 5

IMPLEMENTATION ON 1900 SERIES AND ATLAS COMPUTERS

SIMON PROGRAMS

The SIMON procedures will be provided on paper-tape or cards. It is planned to add additional procedures from time to time, and a new SIMON program will be made available each time this is done. These programs will be identified by the names ALSIMONA, ALSIMONB, ALSIMONC, etc.

As each version is released a description of it will be issued, giving operating instructions and details of the additional facilities it provides. It is likely that all versions will be upwards compatible.

GLOBAL VARIABLES

Some variables are global; that is, unless they are redefined within a block they will refer to particular variables defined in SIMON.

The names of these variables are as follows:

(i) arrays: zzxzz [1:10]

zzxzz is the array from which the ten streams of random numbers are taken. It must be given initial values, which should preferably be large enough to have a binary representation of 23 bits.

(ii) integer variables: MEMBER; TIME; CLOCKTIME; LEASTTIME; INPUTR; INPUTD; SSS; QUEUE; TIMESET.

In addition, I, J, K, L, M, N, P, Q, R, S, T and V are declared for convenience, but have no special significance.

(iii) sets: TIMESET is also defined as a set when the procedure PLANTMASTER is called.

The global variables INPUTR and INPUTD are used to define the channel numbers of the devices used to inout data for the procedures RANDOM (X) (when X = 0) and DISTRI (A, N).

In 1900 programs, at present, INPUTR and INPUTD must be set to specified values as follows:

card reader = 3
paper-tape reader = 0, or 1

If no values are assigned to INPUTR and INPUTD, the value 0 will be assumed, selecting a tape reader.

The value of SSS is read before INPUTR and INPUTD are defined, so an alternative method of selecting an input device has to be used initially. At present a unit must be selected by setting a switch in word 30 of the object program.

A card reader is selected by typing the message:

ON # name 1

on the console typewriter.

A tape reader is selected by typing:

OFF # name 1

All switches are 'turned off' when a program is loaded, so if no message is typed a paper-tape reader will be selected.

On the Atlas computer the value of SSS is read from the same input device as the program. Subsequently the values assigned to INPUTR and INPUTD will be channel numbers defined in the Job Description. If INPUTR and INPUTD are not defined, channel 0 will be assumed, which will normally be the same device as used for the program.

Thus, the complete input for a simulation, using a single tape-reader, can be in the following form:

JOB
F6000 T. JONES SIMULATION
COMPILER ALGOL
(SIMON procedures)
(Users' Algol and SIMON statements)
(Users' data, commencing with SSS and TRACE)
* * * Z

Details of the full Atlas Job Description facilities will be found in the Atlas Algol Primer, C.S. 379A, page 25.

ERROR STOPS

The 1900 program gives the following error messages:

MASTER LIST EXHAUSTED IN 'procedure name'

The specified procedure cannot be carried out because there is no room for further entries in the master lists. This error is followed by the instruction PAUSE 99 which halts the program and prints a message on the console typewriter. The operator may then restart the program at a different point, or he may insert a recovery program, or output instructions to obtain useful information from the program.

SCAN EMPTY LIST

The A-phase scan procedure has been called when the set TIMESET is empty. This error is followed by PAUSE 99.

MEMBER NOT PRESENT

The procedure DELETE(NAME) from: (LIST) has been called when the specified set does not contain the specified entity. The program will continue after this error.

The Atlas SIMON program will give approximately the same error messages, but each error will cause the program to be terminated.

RANDOM NUMBER GENERATOR

The RANDOM procedure generates numbers in the sequence given by the formula:

$$U_{i+1} = (3U_i) \text{ Mod } 2^{23}$$

The number U is truncated at both ends to provide a two-digit random number.

ATLAS TRACE FACILITIES

Optional trace facilities are provided in the Atlas SIMON program. Each time a SIMON procedure is called, an abbreviation of the procedure name is printed out. The global variable TRACE is used to determine whether trace messages are to be output, and as a counter to enable twenty messages to be printed per line.

The initial value of TRACE must be provided as the second item of the user's data, following the value of SSS.

If TRACE = 0 no trace messages are printed.
If TRACE = 1 trace messages are printed.

Trace messages can subsequently be stopped by the instruction:

```
TRACE := 0
```

in the program.

TRACE messages are as follows:

Procedure	TRACE message
PLANTMASTER	START
SET (LIST)	S
ROTATE, (LIST, N)	R
HEADOF (LIST)	H
BEHEAD (LIST)	BH
BETAIL (LIST)	BT
SIZEOF (LIST)	SZ
SETTIME (NAME, V)	ST
TIMEVALUE (NAME)	TM
ENTITY (NAME, V)	E
ADDFIRST (NAME, LIST)	AF
ADDLAST (NAME, LIST)	AL
DELETE (NAME, LIST)	D
SCAN (LIST, MEMBER, LEASTTIME)	SCAN
DISTRI (A,N)	DIST
SAMPLE (A)	SA
HISTOGRAM (N,L,W)	HIST
ADDTO (N,V)	ADD
TAILOF (LIST)	TA
GROUPENTITY (E,N,L)	G
REFNUM (E)	RE
MEMNUM (NAME)	M
EMPTY (SETNAME)	EMP
BELONG (NAME, LIST, MARKER)	BEL
GHIST (NAME, L,N,W)	GHIST
GADDTO (NAME, V)	GADD

APPENDIX I

Summary of Simon Procedures

Procedure Call	Description of Parameters	See page
ENTITY(NAME)	NAME: previously declared as integer array.	7
GROUPENTITY(NAME, M, N)	NAME: previously declared as 2-dimensional integer array. M : no. of members of groupentity. N : reference number chosen by programmer.	7
SET(LIST)	LIST: previously declared as integer variable name.	7
ADDFIRST(NAME, LIST) ADDLAST(NAME, LIST)	NAME: entity, defined by ENTITY or GROUPENTITY. LIST: set name, defined by SET.	7 8
BEHEAD(LIST) BETAIL(LIST)	LIST: set name, defined by SET.	8
DELETE(NAME, LIST)	NAME: entity, defined by ENTITY or GROUPENTITY. LIST: set name, defined by SET.	8
ROTATE(LIST, N)	LIST: set name, defined by SET. N : number of rotations required.	8
HEADOF(LIST) * TAILOF(LIST) * SIZEOF(LIST) *	LIST: set name, defined by SET.	8
MEMNUM(NAME) *	NAME: implicit name, referring to a member of a groupentity.	8
REFNUM(NAME) *	NAME: implicit name, referring to an entity.	9
SETTIME(NAME, VALUE)	NAME: entity defined by ENTITY or GROUPENTITY. VALUE: any expression giving an integer result.	9
TIMEVALUE(NAME) *	NAME: entity, which has been assigned a time-value in SETTIME.	9
SCAN(LIST, MEMBER, LEASTTIME)	LIST: set name, usually TIMESET. MEMBER: } global variables, taking values LEASTTIME: } provided by procedure.	9
RANDOM(N) *	N : stream number, integer in range 0 to 10	9

Procedure Call	Description of Parameters	See page
DISTRI(NAME, N)	NAME: name defined as integer array. N : identification/stream number, also punched in data.	10
SAMPLE(NAME) *	NAME: distribution name, defined by DISTRI.	10
HISTOGRAM(NAME, L, Z)	NAME: name defined as integer array [0 : 20]. L : lower limit of the 9 evenly-spaced zones of the histogram. Z : zone width.	10
ADDTO(NAME, VALUE)	NAME: histogram name. VALUE: any expression giving an integer result.	10
WRITEDOWN(NAME, TITLE)	NAME: histogram name. TITLE: string, to be printed out above histogram.	10
PRIMAS PRISET(LIST) PRIENT(NAME)	 LIST: set name, defined by SET. NAME: entity, defined by ENTITY or GROUPENTITY.	10
EMPTY(LIST)	LIST: set name, defined by SET.	11
BELONG(NAME, LIST, MARKER)	NAME: entity, defined by ENTITY or GROUPENTITY. LIST: set name defined by SET. MARKER: takes value 1 or -1 provided by procedure; must be declared as integer variable.	11
GHIST(NAME, L, N, W)	NAME: name declared as integer array, NAME [0:N+5] L : lower limit of the N evenly-spaced zones of the histogram. N : number of zones. W : zone width.	11
GADDTO(NAME, VALUE)	NAME: histogram name defined by GHIST. VALUE: any expression giving an integer result.	11
GWRITEHIST(NAME, TITLE)	NAME: histogram name defined by GHIST. TITLE: string to be printed out above histogram.	11
PLANTMASTER		7

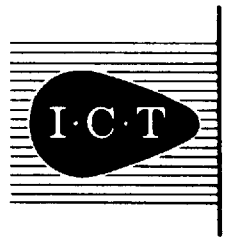
*Asterisk denotes Algol function procedures.

APPENDIX 2

Sizes of Simon Procedures

This table gives the number of words of storage occupied by each SIMON procedure in the object program.

Procedure Name	Number of Words
ENTITY	65
GROUPENTITY	96
SET	37
ADDFIRST	85
ADDLAST	88
BEHEAD	59
BETAIL	91
DELETE	166
ROTATE	40
HEADOF	18
TAILOF	16
SIZEOF	14
MEMNUM	16
REFNUM	16
SETTIME	19
TIMEVALUE	15
SCAN	92
RANDOM	69
DISTRI	58
SAMPLE	129
HISTOGRAM	46
ADDTO	102
WRITEDOWN	193
PRIMAS	83
PRISET	91
PRIENT	71
EMPTY	37
BELONG	60
GHIST	73
GADDTO	124
GWRITEHIST	200
PLANTMASTER	39



AMENDMENT RECORD

TITLE

Amendment Number	Date	Chapter Page	Description

I·C·T

MR. B.D. Willshire

CONTROL AND SIMULATION MANUAL

Part 2 1900 C.S.L.

AMENDMENT RECORD

List Number	*Item Numbers*
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to

Technical Publication 3386

First Edition October 1966

Issued by Technical Publications
International Computers and Tabulators Limited
Head Office: I.C.T. House, Putney, London S.W.15
and printed in Great Britain by
Printing and Stationery Department
Letchworth, Hertfordshire

CONTENTS

3386 (10.66)

PREFACE

Simulation is a technique for obtaining information about the performance of a system without actually putting that system into operation. A model is constructed so that the results obtained by operating or modifying the model indicate the results to be expected when the corresponding real system is operated or modified.

The technique has a wide range of application in operational research and can be used to study such varied activities as industrial processes, accountancy systems or marketing methods. In particular, the effect of many different approaches or strategies can be evaluated without implementing new systems or modifying those in existence.

Simulation is seldom applied to 'stationary' systems. It is of most interest and value when it is applied to systems that are affected by the passage of time, and in almost all such systems some sort of queueing takes place. A queue is formed when a number of items have to wait before passing through some stage in a process. The rate at which the queue grows depends on the rate at which items join the queue, and the time taken to process each item. Both these factors may vary in a random, or stochastic, way: the problem is to derive information about the way in which the queue size varies. For example, the average length of the queue may be obtained, or the probability of its exceeding a certain length. If there is any degree of complexity in the process - if, for example, there are a number of interrelated queues - a mathematical analysis becomes impossibly complicated.

The simulation technique is to create a model of the system by making lists of items at each stage in the process, and transferring items from one list to another in the correct chronological order. All the information required about the process can then be obtained by observation of the model. This technique can be carried out using a pencil and paper; however, for any but the most simple cases it becomes so cumbersome that the use of a computer is essential.

Languages such as FORTRAN and Algol have only rudimentary listing facilities, and consequently a number of special languages such as SIMON and C.S.L. have been developed. In addition to simplifying the writing of simulation programs, these languages may well find many applications outside the field of simulation, because of the powerful facilities they provide for examining and rearranging the membership of lists.

INTRODUCTION

Simulation is a technique for studying and evaluating the performance of a complex system, by setting up a model which behaves in the same way as the system in all essential details.

In a simulation program a model is set up in which a series of variable names represent the components of the system. These components are assumed to exist in discrete states, which may be represented by numeric information assigned to storage associated with the variable names, or by lists to which the names may be added. Logical statements control the way in which the components of the model change their state as time passes.

C.S.L. - Control and Simulation Language - is a problem-oriented language intended to enable the programmer to express the structure of the simulation model in much the same way as he would describe the functioning of the real process being modelled. For example, suppose the system being studied is a barber's shop, with several barbers. Part of the description of the functioning of the shop might be as follows. There is a 'queue' of barbers who are free, and a queue of customers waiting for service. Whenever one or more barbers are free and one or more customers are waiting, the first free barber becomes busy and the number of customers decreases by one.

This could be expressed in C.S.L. by a series of statements such as:

```
CUSTOMERS GT 0
FIND N FREEBARBERS FIRST
BARBER.N FROM FREEBARBERS
CUSTOMERS = CUSTOMERS - 1
```

The components of a model are called *entities*, and each entity is a member of a *class* of similar entities. For example, a simulation representing the movements of ships, docking, loading, and unloading in a port might involve classes called SHIP, BERTH, CRANE, and so on.

An entity is identified by the name of its class followed by a suffix called the *class index* of the entity. For example, entities in a class SHIP would be called SHIP.1, SHIP.2, SHIP.3, ... Associated with each entity may be a number of subscripted variables, for example SHIP.1(1), SHIP.1(2), called *attributes*, which may be used to store numeric information describing the properties or state of the entity.

A much more useful and efficient way of recording the states of entities is to list the entity names in *sets*. A set is an ordered list of entities drawn from a particular class. A set usually represents a particular state, or property, common to all the entities in the set. For example sets called ATSEA, INPORT could be defined within the class SHIP. Sets, because they are ordered, may also represent queues. It is usually possible to define the sets in a model in such a way that the state of each entity at any time is completely defined by the sets to which it belongs. Any change in the state of an entity can then be recorded by moving the entity from one set to another. A number of facilities are provided in C.S.L. for manipulating sets. Entities may be added to sets and removed from sets either one at a time or in groups, and tests may be carried out on the membership of sets.

A C.S.L. program is usually split up into a series of *activities*. Each activity is a series of actions, changing the state of the model, which can be carried out whenever a particular group of conditions is satisfied. The logical facilities of the language are oriented towards this structure; the programmer may specify a series of conditions in a *test chain*, and two possible results may arise: if *all* the conditions are satisfied, then the result of the whole chain is *true*, otherwise the result is *false*. Thus any number of tests may be combined in a single 'decision', such as whether or not a particular activity is to be carried out at a particular time.

A series of tests, specified in a test chain, may also be carried out on each of the members of a set in turn, the required decision being made according to the number of members of the set which satisfy the specified conditions.

It is feasible for the extensive set-handling and logical facilities of C.S.L. to be used to operate on data obtained from a real, rather than a simulated process. In this way a C.S.L. program could be used to control the immediate sequence of operation, or to decide the future policy, of the system under consideration.

A simulation program requires some means of representing the passage of time. In a discrete-state type of model, such as is set up by a C.S.L. program, only those times at which changes of state occur need be considered. Time is therefore advanced in jumps from one such change of state to the next. In order to determine the times at which changes take place, the programmer may define a number of *time-cells*, which may be associated with entities, in the same way as attributes, or may exist as separate variables. As the simulation proceeds, values are placed in these time-cells, each value representing the time that must elapse before some change is due to occur, such as the change of state of an entity, or the start of an activity. The time-advance routine determines the least of these time-values, and then subtracts that value from all the time-cells. A variable called CLOCK is incremented by the chosen value at each time-advance, and therefore represents the total time that has elapsed in the simulation.

The C.S.L. compilation and execution system is intended to make use of the extensive facilities provided by FORTRAN, rather than duplicate them in a new compiler. The C.S.L. translator translates a C.S.L. source program into an intermediate program consisting of statements in 1900 FORTRAN. This FORTRAN program is then compiled by the FORTRAN compiler in the normal way. C.S.L. has a FORTRAN-like structure, and provides most of the facilities of FORTRAN. In particular, C.S.L. contains Input/Output statements and arithmetic statements almost identical to those of FORTRAN, and statements and complete segments written in FORTRAN may be incorporated in C.S.L. programs.

The language described in this manual is based on a language now called C.S.L. 1, which was developed jointly by IBM United Kingdom Ltd. and Esso Petroleum Co. Ltd. Some extensions have been made, but the 1900 C.S.L. translator will accept programs written in C.S.L. 1, including programs punched on cards in the IBM character code.

Where the facilities of C.S.L. are identical to those of FORTRAN, this manual does not give an exhaustive or definitive description of these facilities. The user is therefore recommended to use the 'I.C.T. 1900 series FORTRAN' manual (TL 1167), and the appropriate compiler specifications and operating instructions, in conjunction with this manual.

Some restrictions on the use of the facilities described in this manual are imposed in the first issue of the C.S.L. translator, and some extensions may also be incorporated in later issues. The user should therefore refer to Appendix 2 which contains specifications and operating instructions for the translator.

As extended or amended versions of the translator are issued, new specifications will be published for inclusion in the manual.

Chapter 1

C.S.L. PROGRAMS

A C.S.L. program consists of a series of *statements,* punched on 80-column cards or on paper tape. Each statement occupies one or more *records,* each record being a single card, or a length of paper tape between two newline characters.

Statements are punched in essentially the same format as FORTRAN statements, and consequently C.S.L. programs are normally written on FORTRAN coding sheets.

Each line of a coding sheet is divided into 80 character positions, or columns, and each character of a statement is written in one column. Programs written on coding sheets in this way are punched on cards or paper tape in the manner described below.

PROGRAMS ON CARDS

Each line of the coding sheet corresponds to one complete card, the 80 columns of the line corresponding to the 80 columns of the card. Each character on a line is punched in the corresponding column of a card, and each space on a line corresponds to a blank column on the card.

PROGRAMS ON PAPER TAPE

The line and column structure of the written program is represented on tape by newline, space, and tab characters, as follows.

The end of a line is represented by a newline character. This may be punched immediately following the last character of the line; any spaces following the last character need not be punched.

Spaces appearing before or within a statement are represented on tape by space characters, except when groups of spaces appear in certain positions, when a group of spaces can be replaced by a single tab (horizontal tabulation) character. A tab character signifies to the compiler that no more characters appear until a particular column, the next 'tab position', on that line. For example, statements normally commence at column 7, so column 7 is the first tab position, and whenever a series of spaces precedes column 7 it can be replaced by a single tab character.

The tab positions in C.S.L. are different from those used in PLAN and FORTRAN programs, and are intended to simplify the punching of indented statements (described below). Up to six tab characters are permitted in a line, and the tab positions are columns 7, 9, 11, 13, 15, and 73.

Example

The statements

```
      FOR I = 1,10
        SHIP.I = 5
```

would be punched on tape in the form

```
tabFORVI=1,10newlinetabtab SHIP.I=5
```

STATEMENTS

Each C.S.L. statement must begin on a new line, and may continue on up to two further lines. The statement is written in columns 7 to 72 of each line.

The length of statements is further limited by the rule that a statement may not contain more than 100 'syntactic elements'. In general, each variable name, structural word, or punctuation mark is one syntactic element.

CONTINUATION LINES

The first line of each statement must be blank in column 6. Continuation lines are denoted by any character other than 0 in column 6. None of the lines of a statement may have a completely blank statement field. Constants, names, and structural words must not be split between successive lines. Rules concerning indentation (see below) are not applicable to continuation lines, which may be written anywhere in the statement field.

LABELS

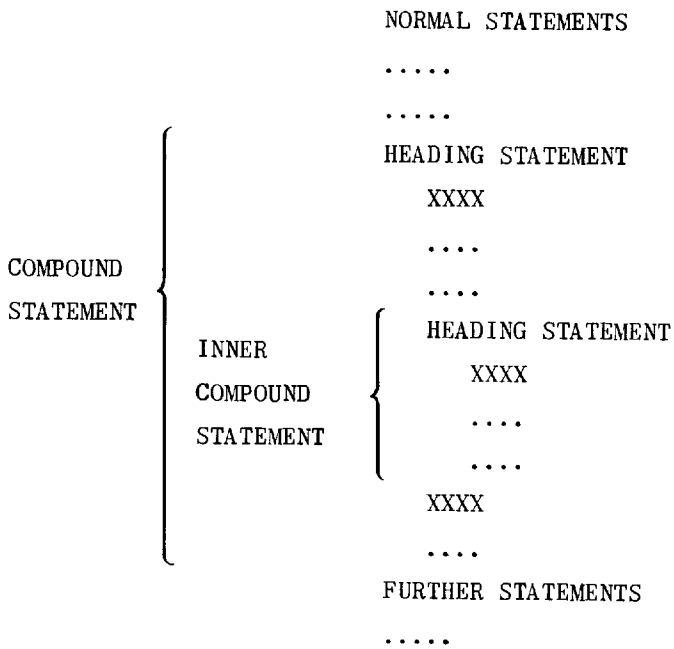
Statements may optionally be labelled, so that other statements can refer to them. A label consists of a number in the range 1 to 25000 written, right-justified, in columns 1 to 5 of the initial line of the statement. Labels must be unique within a program segment but need not be consecutive, or ordered.

INDENTATION

The normal starting position for a statement is column 7 of the initial line. A statement may only be indented to the right of this position when it forms part of a *compound statement.*

A compound statement consists of a heading statement, followed by an indented group. The indented group is a sequence of statements with a common level of indentation greater than that of the heading statement.

A compound statement may contain further compound statements. An inner compound statement consists of a heading statement at the level of the indented group in which it appears, followed by an indented group at a new level of indentation further to the right. This arrangement is shown below.



Nesting in this way may be continued to 20 levels of indentation.

3386 (10.66)

COMMENTS

Comment lines are denoted by a C in column 1 and are not processed by the translator. Descriptive matter may be written anywhere in the remainder of the line, up to column 72, and is reproduced in the source program listing during translation.

PRE-TRANSLATED STATEMENTS

Statements written in FORTRAN may be inserted anywhere, but must be denoted by an F punched in column 1 of the initial card. This does not apply to FORTRAN Input/Output statements which are incorporated in C.S.L.

IDENTIFICATION FIELD

Columns 73 to 80 of a line are not processed or listed, and may contain any desired identification.

NAMES

Names are used to identify variable quantities, and consist of sequences of up to 20 letters. Numeric and special characters are not permitted. A name must not be the same as any of the C.S.L. structural words, which are listed in Appendix 1. In this manual structural words are underlined to distinguish them from variable names.

VARIABLES AND NUMERIC CONSTANTS

All numeric values are held internally in one of two forms, and according to the form in which its value is held, a variable or constant is of type REAL or INTEGER.

A REAL value is a number in the range -5.6×10^{76} to $+5.6 \times 10^{76}$, with a precision of 11 significant figures.

An INTEGER value is a whole number in the range -8,388,607 to +8,388,607.

An INTEGER constant is written in the program as a sequence of digits with or without a sign and without a decimal point.

A REAL constant is written in the program as a sequence of digits with or without a sign, and with a decimal point.

Variables are assumed to be of type INTEGER unless they are defined to be of type REAL in a FLOAT statement (see Chapter 2).

PUNCTUATION

Where names, constants and structural words are not separated by punctuation marks or other special characters they must be separated by at least one space. Names, constants and structural words may not contain spaces.

Chapter 2

DATA STRUCTURE

VARIABLES AND ARRAYS

Names and subscripted variables in C.S.L. may take numerical values and appear in arithmetic expressions in the same way as variables in FORTRAN.

Variables are normally defined by use, and when this is the case are assumed to be of type INTEGER. A name is defined to be an INTEGER variable by its appearance in any statement which assigns it an INTEGER value. Variables that are required to take floating-point values must be of REAL type, and are defined to be of this type by a FLOAT statement. This statement has the form

FLOAT $name_1$, $name_2$, ...

where $name_1$, $name_2$, ... are C.S.L. names.

A series of variables may be defined collectively as an *array*. Each *element* of the array is referred to by the array name, followed by a list of subscripts denoting the position of the element in the array. Subscripts must be separated by commas and the list must be enclosed in parentheses.

Before an array may be used it must be defined in an ARRAY statement. This statement has the form

ARRAY $name_1(n_1,n_2,n_3)$, $name_2(n_4,n_5)$, ...

and defines the C.S.L. names $name_1$, $name_2$, ... to be array names. n_1, n_2, n_3, ... are the dimensions, that is, the maximum values of subscripts, of the corresponding arrays; n_1, n_2, n_3, ... must be positive INTEGER constants.

Subscripts used to refer to particular elements of an array may take the form of numeric constants, variables, or expressions, but must have INTEGER values in the ranges 1 to n_1, 1 to n_2, 1 to n_3, ...

Arrays may have up to 32 dimensions.

CLASSES AND ENTITIES

The basic components of a model are *entities* and are defined in *classes*. Each class contains all the entities of a particular kind; for example, in a simulation of shipping movements, entities might belong to three classes:

SHIPS, PORTS and CARGOES.

Each entity belongs to only one class and is identified by its class name followed by a number which indicates its fixed position in the class. This number is called the *class index* of the entity, and is separated from the class name by a period. Thus a class SHIPS would consist of some fixed number, m, of entities:

SHIP.1 SHIP.2 SHIP.3 ... SHIP.m

An entity may also be referred to by writing in the place of the class index an INTEGER variable, or an expression enclosed in parentheses.

Example

Suppose X = 3 and Y = 5. The fifth entity in the class SHIPS could be referred to by any of the following:

```
SHIP.5
SHIP.Y
SHIP.(X + 2)
```

SETS

Any number of subsets of a class may be defined, and are called *sets*. An entity may be added to and removed from any of the sets defined for the class to which it belongs. An entity may appear in any number of sets, but may appear only once in any one set. Entities may only be added to the beginning or end of a set, and thus an order is established in each set, and is retained in all operations on the set.

Sets serve two main purposes: the membership of sets can be examined and used as a criterion for logical decisions, and, since sets are ordered, they may be used to simulate queuing systems.

ATTRIBUTES

A number of subscripted variables may be associated with each entity of a class. These may be used to store numerical information describing the entities, and are called *attributes* of the entities.

An attribute is referenced as if it were an array element; that is, by an entity name followed by a series of subscripts enclosed in parentheses.

Examples

1 SHIP.5(1)

2 MAN.X(2, Y + 4)

The first example specifies the first attribute of the entity SHIP.5.

TIME-CELLS

A special attribute is used to hold the time-value assigned to an entity in a simulation program. This attribute is called a *time-cell* or T-cell, and is referenced by the prefix T, preceding the entity name.

Examples

1 T.SHIP.4

2 T.MAN.X

The first example refers to the current time-value associated with the entity SHIP.4.

CLASS DEFINITION STATEMENTS

Each class, with its associated sets and attributes, is defined in a single definition statement.

The simplest form of this statement is as follows:

CLASS *cname.m* SET $sname_1$, $sname_2$, ...

This statement defines *cname* to be a class of *m* entities; $sname_1$, $sname_2$, ... are names of sets within the class, each of which may contain any, or all, of the *m* entities in the class. Each entity has a single attribute associated with it and the entity name refers not only to the entity but also to its

single attribute. A class containing entities with more than one attribute is defined by a statement of the form:

CLASS *cname.m* (a_1,a_2,a_3, ...) SET ...

where a_1, a_2, a_3, ... are the dimensions of the 'array' of attributes associated with each entity. Up to 31 dimensions are permitted.

The statement

CLASS *cname.m* (0) ...

defines entities with which no storage is associated. In this case the entities may be added to sets, but may not be assigned attribute values.

If time-values are to be associated with the entities of a class, the word TIME appears, as follows:

CLASS TIME *cname.m* ...

Sets may have sizes less than the class size, in which case they are defined as follows:

CLASS ...SET $sname_1(n_1)$, $sname_2(n_2)$, ...

where n_1, n_2, ... are the maximum numbers of members the sets $sname_1$, $sname_2$, ... may contain.

The general form of the definition statement, therefore, is as follows:

CLASS TIME *cname.* $m(a_1,a_2,a_3, \ldots)$ SET $sname_1(n_1)$, $sname_2(n_2)$, ...

Examples

1 CLASS SHIP.50(5) SET INPORT, ATSEA, REFITTING(10)

2 CLASS MEN.100 SET DECKHAND, ENGINEER(50), COOK(10)

The first example defines a class of 50 entities, each with five attributes. These entities, SHIP.1, SHIP.2, ... may belong to three sets INPORT, ATSEA, and REFITTING. The sets ATSEA, INPORT may contain all 50 entities; the set REFITTING may contain a maximum of 10 entities.

Chapter 3

SIMPLE DATA OPERATIONS

Expressions

C.S.L., like FORTRAN, permits the programmer to write *arithmetic expressions* in a form that closely resembles normal algebraic notation.

Expressions may contain numeric constants, variables, functions, paretheses and arithmetic operators.

The following operators are used:

+	addition (or positive sign)
-	subtraction (or negative sign)
/	division
*	multiplication
**	exponentiation

Parentheses (brackets) can be used, as in normal algebraic notation, to indicate the order in which operations are to be performed.

Standard functions, such as SIN, SQRT, which are available in FORTRAN may appear in expressions. Functions defined by the programmer, in function segments, may also be used. A function name, however, must be preceded by the character £ to indicate to the translator that the name is to be used unchanged in the translated program, and not replaced with a new FORTRAN name generated by the translator.

Any string of numeric characters appearing in the name must be followed by another £.

Functions are described in detail in Chapter 9.

Examples

```
1        £SIN(X)
2        £ABS(£COS(Y + Z))
3        £CSLFUNCTION1£(X,Y)
```

The following rules must be observed in writing expressions in C.S.L. programs.

1 Two operation symbols must not appear next to each other. The expression

```
A * - B
```

is incorrect. It should be written

```
A * (-B) or
-A * B
```

2 When parentheses appear in an expression, that part of the expression within the innermost parentheses is evaluated first, then the part of the expression within the next innermost parentheses is evaluated, and so on.

3 When the order of evaluation of an expression is not completely defined by parentheses, operations are carried out in the following order. First all functions are evaluated, then all exponentiations are carried out, then all multiplications and divisions, and lastly all additions and subtractions. Thus the expression

```
A * B + C/D - E ** F
```

is equivalent to

```
(A * B) + (C/D) - (E ** F)
```

4 Any sequence of operations whose order of evaluation is not determined by rules 2 and 3 is evaluated from left to right. Thus the expression

```
A/B * C
```

is equivalent to

```
(A/B) * C
```

5 Variables and constants in an expression need not be of the same type. If REAL and INTEGER quantities are mixed the result of the expression is REAL.

Examples

Mathematical Notation	*C.S.L. Expression*
$A.B$	`A * B`
$A.-B$	`A * (-B)` or `-A * B`
A^{X+2}	`A ** (X + 2)`
$A^{X+2}.Y$	`A ** (X + 2) * Y`
$A^{X} + 2Y$	`A ** X + 2 * Y`

Arithmetic Statements

ASSIGNMENT STATEMENT

The main type of arithmetic statement is the *assignment statement,* which has the form

variable = expression

The left hand side may be a variable, T-cell, entity attribute, or array element. Its value is replaced by the current value of the expression on the right hand side.

Examples

1 `X = X + 1`

2 `T.SHIP.N = MATRIX(N) + A - B`

3 `MAN.20(2) = 0`

4 `MATRIX(X,Y) = T.MAN.X + SHIP.Y(1)`

The first example means 'replace the current value of X by a new value which is 1 greater'.

The left and right hand sides need not be of the same type; the result of the expression will be converted to the type of the variable on the left hand side.

INCREMENTAL STATEMENT

An arithmetic statement may also take the form of an *incremental statement:*

variable + *expression*

variable - *expression*

The value of the variable on the left hand side is increased, in the first case, or decreased, in the second case, by the current value of the expression on the right hand side. Variable and expression must be of the same type.

Examples

```
1       X + 30
2       T.SHIP.N + MATRIX(N)
3       MAN.X(3) - T.SHIP.X
4       MATRIX(X,Y,Z) + (A ** 2 + B - C)
```

Set Act Statements

These five statements provide facilities for changing the membership of sets by adding and removing groups of entities.

LOSES

This statement has the form

sname$_1$ LOSES *sname*$_2$

where *sname*$_1$ and *sname*$_2$ are names of sets within the same class. The statement removes from the set *sname*$_1$ any entities which are also members of the set *sname*$_2$.

The set *sname*$_2$ remains unchanged.

Example

```
ATSEA LOSES SUNK
```

This statement removes from the set ATSEA any entities which are also members of the set SUNK. The set SUNK is left unchanged.

GAINS

This statement has the two general forms

HEAD *sname*$_1$ GAINS *sname*$_2$

TAIL *sname*$_1$ GAINS *sname*$_2$

where *sname*$_1$ and *sname*$_2$ are names of sets within the same class. The statement, in either form, adds to the set *sname*$_1$ any members of the set *sname*$_2$ that are not already in *sname*$_1$. The set *sname*$_2$ is unchanged.

If the word HEAD appears, the members of *sname*$_2$ are added at the beginning, or head, of *sname*$_1$. If the word TAIL appears the members of *sname*$_2$ are added at the end, or tail, of *sname*$_1$. The words

HEAD and TAIL are optional, and if they are omitted the effect is the same as if TAIL were specified. Entities are added to $sname_1$ in the order in which they appear in $sname_2$.

Example

TAIL ATSEA GAINS LOADED

Suppose that ATSEA consists of the entities:

SHIP.1, SHIP.4, SHIP.2, SHIP.7

and that LOADED contains the entities

SHIP.1, SHIP,3, SHIP.8, SHIP.5

After execution of the statement shown above, ATSEA will contain

SHIP.1, SHIP.4, SHIP.2, SHIP.7, SHIP.3, SHIP.8, SHIP.5

CONVERSE

The statement has the general form

$sname_1$ *cname* CONVERSE $sname_2$

where $sname_1$ and $sname_2$ are names of sets within the class *cname*. The statement first empties the set $sname_1$ and then places in $sname_1$ all entities in the class *cname* which are not members of the set $sname_2$, preserving the class order. The class name *cname* may be omitted without any change in the effect of the statement. The set $sname_2$ is unchanged. The set $sname_1$ may not have a capacity less than the number of entities in the class *cname*.

Examples

ATSEA SHIP CONVERSE INPORT

SUNK CONVERSE FLOATING

Suppose the class SHIP contains ten entities. If INPORT contains the entities:

SHIP.1, SHIP.2, SHIP.5, SHIP.4, SHIP.9, SHIP.10

then the first example will leave in ATSEA the entities

SHIP.3, SHIP.6, SHIP.7, SHIP.8

ZERO

This statement has the form

ZERO $sname_1$, $sname_2$, $sname_3$, ...

where $sname_1$, $sname_2$, ... are set names. The statement empties all the sets named in the list.

Example

ZERO SETA, SETB, ATSEA

This example leaves SETA, SETB and ATSEA with no members.

LOAD

This statement has the two forms

cname LOAD $sname_1$, $sname_2$, ...

where $sname_1$, $sname_2$, ... are names of sets within the class *cname*, and

LOAD $sname_1$, $sname_2$, ...

where $sname_1$, $sname_2$, ...are any set names.

Both forms of the statement have the same effect; each of the sets listed is filled with the entire population of the class to which it belongs, preserving the class order. When the class name appears, however, only sets within that class may be listed. A set may not appear in this statement if it has a capacity less than the number of entities in the class to which it belongs.

Examples

```
1        SHIP LOAD ATSEA

2        LOAD ATSEA, SETA, FREEPORTS
```

The first example places in the set ATSEA all the members of the class SHIP, in the order SHIP.1, SHIP.2, SHIP.3, ...

The second example places in each of the sets ATSEA, SETA, FREEPORTS, all the members of the class to which the set belongs.

Chapter 4

SIMPLE TRANSFER STATEMENTS

The statements described in this chapter enable control to be transferred to a statement elsewhere in the program, which may be specified unconditionally, or selected from a number of alternatives according to the result of a single test.

UNCONDITIONAL GO TO

This statement has the form

GO TO *l*

where *l* is a statement label appearing in the program. Control is transferred to the statement labelled *l*. GO TO may be written as a single word GOTO.

Examples

1 GO TO 250

2 GOTO 250

Both these examples transfer control to the statement labelled 250.

COMPUTED GO TO

This statement has the form

GOTO $(l_1, l_2, l_3, \ldots l_m)$ *vname*

where l_1, l_2, $\ldots l_m$ are statement labels appearing in the program, and *vname* is an INTEGER variable name. Control is transferred to the statement labelled l_i where i is the current value of the variable *vname* $(1 \leqslant i \leqslant m)$.

The programmer should ensure that the value of *vname* is within the range 1 to m when the statement is executed, since the result of the statement is undefined for values of *vname* outside the correct range.

Example

GO TO (1,10,55), SWITCH

If SWITCH = 1 control passes to the statement labelled 1, if SWITCH = 2 control passes to the statement labelled 10, and if SWITCH = 3 control passes to the statement labelled 55.

THE IF STATEMENT

This statement has the form

IF (*expression*) l_1, l_2, l_3

where l_1, l_2, l_3 are statement labels appearing in the program. Control is transferred to one of the statements labelled l_1, l_2, l_3, according to whether the current value of the expression in parentheses is negative, zero or positive.

Control is transferred to:

statement l_1 if (*expression*) < 0

statement l_2 if (*expression*) = 0

statement l_3 if (*expression*) > 0

The expression may be of REAL or INTEGER type.

Example

```
IF (B ** 2 + 4 * A * C) 4, 17, 1
```

In this example, control is transferred to the statement labelled 4 if the expression $B^2 + 4AC$ (in algebraic notation) is negative, to the statement labelled 17 if the expression is zero, and to the statement labelled 1 if the expression is positive.

The FOR Statement

The FOR statement provides a means of repeating a series of statements a specified number of times. The series of statements under the control of the FOR statement, the FOR loop, is written immediately after the FOR statement and indented to the right of it, as follows:

```
FOR    .....        FOR   statement
   XXXXX        }
   .....        }   FOR   loop
   XXXXX        }
XXXXX                 Further program
```

The indented statements are also referred to as the *range* of the FOR statement.

Two kinds of FOR statement are provided, one controlling the repetition of the loop by *incremental indexing*, the other by *set indexing*.

FOR - with incremental indexing

This statement has the form

FOR *vname* = m_1, m_2, m_3

where *vname* is an INTEGER variable name, and m_1, m_2, m_3 are unsigned INTEGER constants or INTEGER variables. The effect of the statement is that the index *vname* is used as a count of the number of repetitions of the loop.

m_1 is the *initial* value of *vname*

m_3 is the *increment* of *vname*

m_2 is the *terminal value* of *vname*

For the first execution of the loop the index *vname* takes the value m_1. At the end of each execution of the loop the index *vname* is increased by the value m_3. The index is then tested, and if *vname* is greater than the terminal value m_2, control passes on to the next statement after the FOR loop; otherwise control is transferred back to the beginning of the loop and the loop is repeated.

The index *vname* may appear in statements in the FOR loop, provided its value is not changed. Similarly m_1, m_2, or m_3, where these are variables, may appear in statements in the FOR loop, but must not be changed.

If an incremental value of 1 is required, the parameter m_3, and the preceding comma, may be omitted from the statement.

Examples

```
1       FOR K = 1, 20, 2
           SHIP.K INTO ATSEA
2       FOR I = 1, M
           TERM  =  TERM * X
           SUM   +  TERM
```

The first example places the entities SHIP.1, SHIP.3, SHIP.5, ...SHIP.19 in the set ATSEA.

The second example merely uses I as a counter, and forms the sum of the first M terms of the series

$$X + X^2 + X^3 \ldots$$

assuming SUM and TERM are initially zero.

FOR - with set indexing

This statement has the form

FOR *vname* = *sname*

where *vname* is an INTEGER variable name, and *sname* is a set name. The = sign may be omitted.

The loop is repeated as many times as there are members in the set *sname*. The index *vname* takes in turn the value of the class index of each member of the set.

Example

```
FOR I = ATSEA
   SHIP.I(3) = 0
```

This example has the effect of zeroizing attribute (3) of each of the members of the set ATSEA.

NESTED FOR LOOPS

A FOR statement and its indented range together form a compound statement (as described in Chapter 1) which may appear in the indented range of another FOR statement, or one of the other types of compound statement.

Example

```
FOR I = INPORT
   T.SHIP.I = 0
   FOR K = 1,5
      SHIP.I(K,1) = 0   } Inner Loop
      SHIP,I(K,2) = 1   }
   SHIP.I(6,1) = 100
   SHIP.I(6,2) = 0
```

Outer Loop

In this example the outer loop is executed with the index I taking the values of the class indices of all the ships in the set INPORT. During each repetition, the inner loop is repeated five times. Thus, if the outer loop is repeated ten times, say, the inner loop will be repeated fifty times.

TRANSFER OF CONTROL IN FOR LOOPS

Transfer statements may appear in the range of a FOR statement, and may transfer control to another statement within the range, or to a statement outside the range. The last statement in a FOR loop must not be a transfer statement, or test statement, or part of the indented range of an inner compound statement.

DUMMY STATEMENTS

The above restriction can be circumvented by using a *dummy statement* as the final statement in a FOR loop. A dummy statement consists of either of the single words:

DUMMY

REPEAT

These statements are synonymous. They may appear anywhere in a program, and may be labelled.

Example

```
      FOR I = 1, M
          IF  (MATRIX(I)) 1, 20, 40
    1     DUMMY
```

In this example DUMMY provides a destination for the case where the value of MATRIX(I) is negative. Each of the elements of the array MATRIX is examined in turn, until an element is found which has a positive or zero value. When such an element is encountered control passes out of the FOR loop, the index I retaining the value of the subscript of the element found.

TRANSFER OF CONTROL INTO A FOR LOOP

Control may not be transferred into a FOR loop from a statement outside its range, except in the following case. It is permissible for control to be transferred to a statement outside the FOR loop, for a series of statements to be executed, and then for control to be transferred back into the range of the FOR statement. The programmer must ensure that the values of the index (*vname*) and parameters (m_1, m_2, m_3) are not changed during this process.

Simple Test Statements

A simple test statement transfers control to one of two alternative statements, according to the result of a specified test. A simple test statement consists of a *test expression* followed by a *destination clause.* A test expression may only have two values: *true* and *false.* The destination clause specifies the statements to which control is to be transferred, according to the result of the test expression.

DESTINATION CLAUSE

The destination clause has the form

$$\ldots\ldots l_1 \ @ \ l_2$$

where l_1 and l_2 are statement labels appearing in the program. Control is transferred to the statement labelled l_1 if the preceding test expression is *true*, and to the statement labelled l_2 if the test expression is *false*.

l_1, l_2, or the entire clause may be omitted.

If l_1 is omitted, as follows

```
.....@ l2
```

then in the event of a *true* result control passes to the next statement in the program.

If l_2 is omitted, as follows

```
.....l1 @
```

then in the event of a *false* result, control passes to the beginning of the next activity, usually the next point in the program at which the word BEGIN appears (see Chapter 9). If the next activity after the test statement is the first activity of the program, the word BEGIN may be omitted, but the transfer of control will take place in the normal way as if the word BEGIN were present. If both destinations are omitted the @ symbol must be omitted as well; the transfers of control are as specified above. The destination clause must always be separated from the test expression by at least one blank.

Arithmetic Test Statements

Arithmetic test statements have the form

expression$_1$ relational operator *expression*$_2$ l_1 @ l_2

where *expression*$_1$, *expression*$_2$ are any expressions of the same type, l_1 @ l_2 is a destination clause, and the relational operator is one of the following

GT	greater than
GE	greater than or equal to
EQ	equal to
NE	not equal to
LE	less than or equal to
LT	less than

If the specified relationship holds for the current values of the two arithmetic expressions, then the test expression is *true*, and control is transferred to the statement labelled l_1: otherwise the test expression is *false*, and control is transferred to the statement labelled l_2.

Examples

```
1    A GT 100  101 @ 102
2    B ** 2 NE 4 * A * C  @ 20
3    T.SHIP.I EQ 0
4    SHIP.I(1,2) LT MATRIX(I,1)  101 @
```

In the first example control is transferred to statement 101 if the value of A is greater than 100: if A is less than or equal to 100 control is transferred to statement 102.

In the third example control passes on to the next statement if the value of T.SHIP.I is zero, otherwise control passes to the beginning of the next activity.

Set Test Statements

Set test statements are similar to arithmetic test statements, except that the test expressions that appear test the membership of sets.

IN

This statement has the form

ename IN *sname* l_1 @ l_2

where *ename* is an entity name, *sname* is a set name, and l_1 @ l_2 is a destination clause.

If the entity *ename* is a member of the set *sname* the test expression is *true;* if not, the test expression is *false.* The usual transfers of control take place. The set *sname* is not changed.

Example

```
SHIP.5 IN ATSEA  1 @ 5
```

If the entity SHIP.5 is a member of the set ATSEA, control is transferred to the statement labelled 1, otherwise control is transferred to the statement labelled 5.

NOTIN

This statement has the form

ename NOTIN *sname* l_1 @ l_2

where *ename* is an entity name, *sname* is a set name and l_1 @ l_2 is a destination clause.

If the entity *ename* is not a member of the set *sname* the test expression is *true;* if *ename* is a member of *sname* the test expression is *false.* The usual transfers of control take place. The set *sname* is not changed.

The word NOTIN may be abbreviated to NIN.

Example

```
SHIP.2 NOTIN REFITTING  @ 50
```

If the entity SHIP.2 is not in the set REFITTING, control passes to the next statement in the program, otherwise control passes to the statement labelled 50.

EQUALS

This statement has the form

$sname_1$ EQUALS $sname_2$ l_1 @ l_2

where $sname_1$ and $sname_2$ are set names defined in the same class, and l_1 @ l_2 is a destination clause.

The test expression is *true* if the members of $sname_1$ and $sname_2$ are identical (no account is taken of the order in which they are listed). If an entity is found which is a member of one of the sets but not of the other, the test expression is *false.* The usual transfers of control take place. Both sets are unchanged.

Example

```
ATSEA EQUALS LOADED  20 @ 5
```

If precisely the same entities are recorded in the set ATSEA as are recorded in the set LOADED, control is transferred to the statement labelled 20. If any entity is a member of one set but not of both, control is transferred to statement 5.

WITHIN

This statement has the form

$$sname_1 \text{ WITHIN } sname_2 \quad l_1 @ l_2$$

$sname_1$, $sname_2$ are set names defined in the same class, and $l_1 @ l_2$ is a destination clause.

The test expression is *true* if all the entities which are members of $sname_1$ are also members of $sname_2$. If an entity is found in $sname_1$, which is not in $sname_2$, the test expression is *false*. The usual transfers of control take place. No account is taken of the order in which the entities are listed in the sets. Both sets are unchanged.

Example

```
INPORT WITHIN UNLOADED  10 @
```

If every member of the set INPORT is also a member of the set UNLOADED, control is transferred to the statement labelled 10: otherwise control passes to the next BEGIN statement in the program.

EMPTY

This statement has the form

$$sname_1, sname_2, \ldots \text{EMPTY} \quad l_1 @ l_2$$

where $sname_1$, $sname_2$, ... are set names and $l_1 @ l_2$ is a destination clause. If all the sets listed are empty, the test expression is *true;* if any one of the sets contains an entity, the test expression is *false*. The usual transfers of control take place. None of the sets are changed.

Example

```
LOADED, ATSEA EMPTY  50 @ 20
```

If the sets LOADED and ATSEA have no members, control passes to the statement labelled 50: if either of the sets contain any members control passes to the statement labelled 20.

Chapter 5

COMPLEX CONDITIONAL STATEMENTS

The statements described in this chapter provide facilities for carrying out one or more tests and performing set manipulations and transfers of control according to the results of the tests.

Statements With Implied Tests

These statements are concerned with operations that can only be performed if certain conditions are satisfied. For example, an entity may be added to a set only if it is not already a member of the set, so that a statement which adds an entity to a set implies a test of whether the entity is already in the set. These statements contain a destination clause, and control is transferred to one or other of the specified destinations according to the result of the implied test.

HEAD

This statement has the form

ename HEAD *sname* l_1 @ l_2

where *ename* is an entity name and *sname* is a set name; *ename* and *sname* are defined in the same class. l_1 @ l_2 is a destination clause.

This statement places the entity *ename* at the head of the set *sname*, subject to the condition that *ename* is not already a member of *sname*.

If this condition is satisfied the operation is carried out and control passes to the statement labelled l_1. If the condition is not satisfied the set is left unchanged and control passes to the statement labelled l_2.

Example

```
SHIP.N HEAD ATSEA  @ 45
```

If the entity SHIP.N is not a member of the set ATSEA, it is added to the head of ATSEA and control passes to the next statement in the program. If SHIP.N is already in the set ATSEA, the set is left unchanged, and control passes to the statement labelled 45.

TAIL

This statement has the form

ename TAIL *sname* l_1 @ l_2

where *ename* is an entity name, *sname* is a set name, and l_1 @ l_2 is a destination clause.

This statement has precisely the same effect as the HEAD statement except that the entity is added to the tail of the set, if the implied test is satisfied.

INTO is a synonym for TAIL.

Examples

```
1       SHIP.X TAIL INPORT  1 @ 1
2       SHIP.X INTO INPORT  1 @ 1
```

Both examples have the same effect: if the entity SHIP.X is not already a member of the set INPORT, it is added to the tail of the set. If SHIP.X is already a member of INPORT the set is left unchanged. In either case, control is transferred to the statement labelled 1.

FROM

This statement has the form

ename FROM *sname* l_1 @ l_2

where *ename* is an entity name, *sname* is a set name, and l_1 @ l_2 is a destination clause. *ename* and *sname* must be defined in the same class.

If the entity *ename* is a member of the set *sname*, it is deleted from the set, and control passes to the statement labelled l_1. If *ename* is not a member of *sname* the set is left unchanged and control passes to the statement labelled l_2.

Example

```
        SHIP.X FROM ATSEA  11 @ 10
```

This example tests whether SHIP.X is a member of the set ATSEA. If it is, it is removed from ATSEA and control passes to the statement labelled 11. If SHIP.X is not a member of ATSEA, control passes to the statement labelled 10, and ATSEA is left unchanged.

Test Chain Compound Statements

The conditional statements described so far have been qualified by a single test. It is possible however to combine a number of test statements to give a single *true* or *false* result. This is achieved by means of a compound statement, of the form described in Chapter 1. The indented range of the statement consists of a series of test statements, and is called a *test chain*. Except when the word OR appears (see Disjunctive Test Chains) the result of the test chain is *true* if *all* the tests in the test chain give the result *true*. If any of the tests if *false* the result of the chain is *false*.

Example

```
XXXXXX                          Heading statement
   A GT 10
   SHIP.N NOTIN INPORT          } Test chain
   SHIP.N(1) NE 5
```

In this example the result of the test chain is *true* if A is greater than 10, *and* SHIP.N is not in the set INPORT, *and* the attribute SHIP.N(1) is not equal to 5. If any of these tests gives a *false* result, the test chain is *false.*

The more elaborate forms that test chains may take are described at the end of this chapter.

In general a test chain is repeated, like the range of a FOR statement, under the control of its heading statement. The repetition of the chain is controlled by *set indexing*: the heading statement specifies a set name, and an index variable which takes in turn the value of the class index of each set member.

The final result of the compound statement is determined by the number of set members which satisfy the test chain, different criteria being established by the different types of heading statement described below.

CHAIN

This statement has the form

```
CHAIN  l1 @ l2
    test chain
    .....
    .....
```

The result of a CHAIN statement is simply the result of the test chain. According to whether the result is *true* or *false,* control is transferred to one or other of the destinations specified by the destination clause l_1 @ l_2.

Example

```
CHAIN  50 @ 25
   A LT B
   SHIP.N IN INPORT
```

If A is less than B and SHIP.N is in the set INPORT, control is transferred to the statement labelled 50; otherwise control is transferred to the statement labelled 25.

ALL

This statement has the form

```
ALL vname sname  l1 @ l2
    test chain
    .....
    .....
```

where *vname* is a variable name, *sname* is a set name and l_1 @ l_2 is a destination clause.

The test chain is repeated with *vname* taking in turn the values of the class indices of the members of *sname.* The result is *true* if the test chain is *true* for *every* member of the set *sname.* Otherwise the result is *false.* The usual transfers of control take place.

The index *vname* should, of course, appear in at least one of the tests in the test chain.

Example

```
ALL I ATSEA  @ 30
   SHIP.I IN LOADED
   T.SHIP.I LT 100
```

If every entity, SHIP. I, in the set ATSEA, is in the set LOADED and has a time-value less than 100, control passes to the next statement after the test chain; otherwise control is transferred to the statement labelled 30.

EXISTS

This statement has the form

EXISTS (*expression*) *vname sname* l_1 @ l_2

test chain

.....

.....

where *expression* is any expression giving an INTEGER result, *vname* is a variable name, *sname* is a set name, and l_1 @ l_2 is a destination clause. The test chain is repeated with *vname* taking in turn the values of the class indices of the members of *sname*.

The result is *true* if the test chain is *true* for *at least as many* members of *sname* as the current value of the specified expression; otherwise the result is *false*.

The usual transfers of control take place.

The expression and its enclosing parentheses may be omitted, in which case its value is taken to be unity.

EX is a synonym for EXISTS.

Example

```
EXISTS (X + 3) I LOADED  101 @ 102
   SHIP.I NIN ATSEA
```

If at least X + 3 members of the set LOADED are not members of the set ATSEA control is transferred to the statement labelled 101; otherwise control is transferred to the statement labelled 102.

UNIQUE

This statement has the form

UNIQUE (*expression*) *vname sname* l_1 @ l_2

test chain

.....

.....

where *expression* is any expression giving an INTEGER result, *vname* is a variable name, *sname* is a set name, and l_1 @ l_2 is a destination clause. This statement is similar to the EXISTS statement, but gives the result *true* if the test chain is *true* for *exactly as many* members of *sname* as the current value of the expression.

UN is a synonym for UNIQUE.

Example

```
UNIQUE (5) I INPORT  101 @ 102
   X + Y LT 100
   SHIP.I(3) NE 0
```

If (X + Y) is less than 100 and exactly five members of the set INPORT have a non-zero value in attribute (3) then control is transferred to the statement labelled 101. Otherwise control is transferred to the statement labelled 102.

COUNT

This statement has the form

```
COUNT vname sname
    test chain
    .....
    .....
```

where *vname* is a variable name and *sname* is a set name. The test chain is repeated with *vname* taking in turn the values of the class indices of the members of the set *sname*. A count is made of the number of members of the set *sname* for which the test chain is *true*. The result of the count is assigned to the variable *vname*, and is available for use in subsequent statements. No transfer of control takes place.

The test chain may be omitted, in which case the statement determines the number of members in the set.

Example

```
COUNT N INPORT
    SHIP.N IN LOADED
    SHIP.N(3) NE 0
```

When this statement has been executed, the value of N will be the number of members of the set INPORT which are also in the set LOADED, and have a non-zero value in attribute (3).

SUM

There are two forms of this statement, one using set indexing, the other incremental indexing.

SUM - with set indexing

This statement has the form

```
SUM (expression) vname = sname
    test chain
    .....
    .....
```

where *expression* is any expression giving an INTEGER result. *vname* is a variable name, and *sname* is a set name. The test chain is repeated with *vname* taking in turn the values of the class indices of the members of *sname*. For each member of the set which satisfies the test chain, the expression is evaluated. The values obtained are summed and at the end of execution of the statement the accumulated total is assigned to the variable *vname*, and is available for use in subsequent statements.

No transfer of control is made.

The test chain may be omitted, in which case the expression is summed over the entire set.

If the test chain fails for every member of the set, the result recorded in *vname* is zero.

The = sign may be omitted.

Example

```
SUM (TABLE(X)) X LOADED
    SHIP.X IN ATSEA
    TABLE(X) GE 10
```

A value is found in the array TABLE for each member of the set LOADED which is also in the set ATSEA.

The sum of all such values, except those which are less than 10, is the final value of X.

SUM - with incremental indexing

This statement has the form

SUM (*expression*) *vname* = m_1, m_2, m_3

where *vname* is an INTEGER variable name; m_1, m_2, m_3 are either unsigned integers or INTEGER variable names; and *expression* is any expression giving INTEGER results.

The index *vname* takes values between m_1 and m_2, with increments of m_3, as in a FOR statement (see 'FOR with incremental indexing', Chapter 4.) The expression is evaluated for each value of the index, and at the end of execution of the statement the sum of the results is assigned to *vname* and is available for use in subsequent statements. The parameter m_3 may be omitted, in which case its value is taken as one.

Note that this form of the SUM statement may *not* be qualified by a test chain.

Example

```
SUM (MATRIX(I)) I = 1, 9, 3
```

The final value of I will be:

```
MATRIX(1) + MATRIX(4) + MATRIX(7)
```

SPLIT - general form

This statement has the general form

SPLIT *vname* $sname_1$ INTO $sname_2$ ELSE $sname_3$

where *vname* is an INTEGER variable name, and $sname_1$, $sname_2$, and $sname_3$ are set names defined in the same class.

The sets $sname_2$ and $sname_3$ are zeroized initially. The test chain is then carried out for each entity in the set $sname_1$, with *vname* taking the value of the class index of each entity in turn. If the result of the test chain is *true*, the entity is added to $sname_2$: if the result is *false*, the entity is added to $sname_3$. The set $sname_1$ is left unchanged.

A test chain must appear, and at least one test in the test chain should involve the index *vname*.

Example

```
SPLIT I INPORT INTO LOADED ELSE UNLOADED
   SHIP.I(1) NE 0
```

The sets LOADED and UNLOADED are first zeroized. Each ship in the set INPORT is placed in LOADED if attribute (1) is not zero, and in UNLOADED if attribute (1) is zero.

The set INPORT is left unchanged.

SPLIT - partial forms

The SPLIT statement may be written in the following short forms:

1 SPLIT *vname* $sname_1$ INTO $sname_2$

```
   test chain
   .....
   .....
```

2 SPLIT *vname* $sname_1$ ELSE $sname_2$

```
   test chain
   .....
   .....
```

The first form of the statement places in $sname_2$ those members of $sname_1$ for which the test chain is *true*. The second form places in $sname_3$ those members of $sname_1$ for which the test chain is *false*.

Examples

```
1       SPLIT X INPORT INTO LOADED
           SHIP.X(3) NE 0
2       SPLIT X INPORT ELSE UNLOADED
           SHIP.X(3) NE 0
```

The first example places in the set LOADED those members of the set INPORT whose attribute (3) has a non-zero value. The second example places in the set UNLOADED those members of the set INPORT whose attribute (3) is zero.

SPLIT - HEAD and TAIL forms

The SPLIT statement may also be written in the following forms:

```
SPLIT vname sname1 INTO { HEAD sname2 } ELSE { HEAD sname3 }
                        { TAIL        }      { TAIL       }
   test chain
   .....
   .....
```

In this form of the statement the sets $sname_2$, $sname_3$ are not zeroized. When a member of $sname_1$ is placed in $sname_2$ or $sname_3$ it is added at the beginning of the set if HEAD is specified, and at the end of the set if TAIL is specified. If the entity is already in the appropriate set, the set is left unchanged.

Example

```
SPLIT I INPORT INTO HEAD UNLOADED ELSE HEAD LOADED
   SHIP.X(3) NE 0
```

In this example each ship in the set INPORT is added to the head of the set UNLOADED if its attribute (3) is non-zero, and to the head of the set LOADED if its attribute (3) is zero.

The INTO and ELSE parts of the statement may be qualified by HEAD or TAIL, or by neither, independently of one another. Also, HEAD and TAIL may appear in the partial forms of the statement. Thus the following forms are among those permitted:

```
SPLIT ..... INTO HEAD ..... ELSE TAIL .....
SPLIT ..... INTO HEAD ..... ELSE .....
SPLIT ..... INTO ..... ELSE HEAD .....
SPLIT ..... INTO HEAD .....
SPLIT ..... ELSE TAIL .....
```

SPLIT - with QUALIFY

In order to ignore certain entities completely in the SPLIT statement, the QUALIFY statement may be used. The QUALIFY statement may appear *only* in the test chain of a SPLIT statement. It has the form

```
QUALIFY
   test chain
   .....
```

Each member of the set $sname_1$ must satisfy the test chain of the QUALIFY statement in order to be allocated to $sname_2$ or $sname_3$.

Example

```
SPLIT X INPORT INTO LOADED ELSE UNLOADED
   SHIP.X(3) NE 0
   QUALIFY
      X GT 5
```

In this example the entities SHIP.1, SHIP.2, ...SHIP.5 will not be added to the sets LOADED and UNLOADED in any circumstances.

RANK

This statement has the form

RANK *vname sname* (*expression*)

where *vname* is an INTEGER variable name, *sname* is a set name, and *expression* is any expression, of either type. For each member of the set *sname*, the variable *vname* takes the value of the class index, and the expression is evaluated. The members of the set are re-ordered so that the corresponding values of the expression are in descending order of magnitude. Thus, the entity for which the expression has its largest value becomes the first member of the set, and so on.

This statement may *not* be qualified by a test chain.

Example

```
RANK I INPORT (TONNAGE(I))
```

The ships in the set INPORT are arranged in order of decreasing tonnage, the ship with the largest tonnage being the first member of the set.

Test Chains

The following types of statement may appear as tests in a test chain:

1 A simple test statement without its destination clause,

e.g. GT, EQ, LT, IN, NOTIN

2 A statement with an implied test, without its destination clause,

e.g. HEAD, TAIL

When a statement of this type appears the specified operation is carried out, as well as the implied test.

3 A test chain compound statement, withouts its destination clause,

e.g. CHAIN, ALL, UNIQUE

Note that QUALIFY is an exception, and may only appear in the test chain of a SPLIT statement.

4 A FIND compound statement, without its destination clause (see Chapter 4). When a compound statement such as CHAIN or FIND appears in a test chain, the usual indentation rules apply (see 'Indentation', Chapter 1). Nesting of test chains may be continued up to 20 levels of indentation.

DISJUNCTIVE TEST CHAINS

A disjunctive test chain is a series of simple test chains connected by the word OR, as follows:

```
XXXXXXXXX
   test 1
   .....
   test 2
   OR test 3
   .....
   test 4
   OR test 5
   test 6
   .....
```

Each section of the chain terminated by an OR is called a *disjunct.* Thus in the diagram above, the section

```
   test 1
   .....
   test 2
```

is a disjunct; the section

```
   OR test 3
   .....
   test 4
```

is a disjunct; and the section

```
   OR test 5
   test 6
   .....
```

is a disjunct.

If any of the disjuncts in a test chain gives the result *true* then the result of the whole test chain is *true.* If at least one test in *every* disjunct is *false* then the result of the test chain is *false.*

Example

```
CHAIN  5 @ 6
   A EQ B
   SHIP.N NIN ATSEA
   OR SHIP.N(2) EQ 0
   OR A GT 10
   B GT 15
```

In this example, if A is equal to B and SHIP.N is not in the set ATSEA, *or* if SHIP.N(2) is equal to zero, *or* if A is greater than 10 and B is greater than 15, the test chain is *true.* If each of the disjuncts is *false,* then the test chain is *false.*

INTERSPERSED ACTS

Statements that are not tests (act statements) may appear in a test chain, and when encountered are executed in the normal way. Frequently, however, the result of the test chain is determined before all the tests have been considered, and any remaining tests, and act statements, are not executed.

In a simple test chain the result is *false* as soon as a test is found which is *false,* and any ensuing tests are ignored. Consequently an act statement in a simple test chain is only carried out if all the preceding tests are *true.*

Similarly, in a disjunctive test chain the result is *true* as soon as a disjunct is found which is *true,* and any remaining disjuncts are ignored. Consequently an act statement may be ignored because a preceding disjunct is found to be *true,* or because a preceding test is found to be *false.* A useful application of these effects is that by inserting suitable diagnostic output statements, such as CHECK statements, in a test chain, the programmer can detect the point at which the test chain was found to be *false.*

Chapter 6

FIND COMPOUND STATEMENTS

FIND statements, like the statements described in the previous chapter, carry out a series of tests on members of a specified set. An index variable takes in turn the value of the class index of each member. The FIND statement carries out the additional operation of selecting, from the subset of members which satisfy the test chain, one particular entity which satisfies some further criterion. For example, one form of the FIND statement selects the *first* entity which satisfies the test chain.

FIND - general form

The FIND statement has the following general form

FIND *vname sname criterion* l_1 @ l_2

test chain

.....

.....

where *vname* is a variable name, *sname* is a set name, l_1 @ l_2 is a destination clause, and *criterion* is one of the following

ANY

FIRST

LAST

MAX (*expression*)

MIN (*expression*)

These criteria are described in detail below. The test chain is carried out on each member of the set *sname*, with the variable *vname* taking the value of the class index of the member, until either an entity is found which satisfies the test chain and the specified criterion, or the complete set has been examined.

If the test chain is found to be *false* for every member of the set *sname*, or if there are no entities in *sname*, then control passes to the statement labelled l_2, leaving the index *vname* with an indeterminate value. Otherwise, after execution of the statement, *vname* is left with the value of the class index of the selected entity, which is available for use in subsequent statements, and control passes to the statement labelled l_1.

The test chain may be omitted, in which case an entity will always be selected, unless the set *sname* is empty.

FIND... ANY

This statement is written

FIND *vname sname* ANY l_1 @ l_2

test chain

.....

.....

and selects *at random* an entity which satisfies the test chain. A random number is generated and used to select a member of the set. A search is carried out, starting with this member, until an entity is found for which the test chain is *true* or until every member of the set has been considered.

In the generation of a random number a stream variable is used (see Chapter 7). Normally a standard stream variable is used for the FIND...ANY statement, but in order to make the random selections of two such statements independent, the programmer may specify a stream variable in the statement. The statement is then written:

FIND ... ANY (*stream*) l_1 @ l_2

where *stream* is an INTEGER variable name, which should have been assigned some value previously.

Example

```
FIND X ATSEA ANY  10 @ 20
   SHIP.X NOTIN LOADED
```

This example selects at random a ship which is at sea and not loaded. The final value of X is the class index of the ship selected.

FIND... FIRST

This statement is written

FIND *vname sname* FIRST l_1 @ l_2

test chain

.....

.....

The entity selected is the *first* entity in the set for which the test chain is *true*. Starting with the first entity in the set, each entity in turn is tested until one is found for which the test chain is *true*.

Example

```
FIND Y INPORT FIRST  10 @ 20
   SHIP.Y NOTIN LOADED
```

This example selects the first ship in the set INPORT which is not in the set LOADED. The final value of Y is the class index of the ship selected.

FIND... LAST

This statement is written

FIND *vname sname* LAST l_1 @ l_2

test chain

.....

.....

The entity selected is the *last* member of the set for which the test chain is *true*. Starting with the last member of the set *sname*, and working through the set backwards, each entity in turn is tested until one is found for which the test chain is *true*.

Example

```
FIND X ATSEA LAST
SHIP.X FROM ATSEA
```

In this example the FIND statement has no test chain and no destination clause. The last entity in the set ATSEA is removed from the set. If the set ATSEA is empty control passes to the next BEGIN statement in the program.

FIND... MAX

This statement is written

```
FIND vname sname MAX (expression)  l1 @ l2
    test chain
    .....
    .....
```

where *expression* is any expression, of either type. Each entity in the set is tested, and for each entity for which the test chain is *true* the expression is evaluated. The entity for which the value of the expression is the maximum value obtained is the one selected.

FIND... MIN

This statement is written

```
FIND vname sname MIN (expression)  l1 @ l2
    test chain
    .....
    .....
```

This statement has the same effect as FIND...MAX except that the entity selected is the one for which the value of the expression is the minimum value obtained.

Example

```
FIND X ATSEA MAX (TONNAGE(X))  10 @ 20
    TONNAGE(X) LT 25000
```

This example selects the ship from the set ATSEA which has the largest tonnage under 25000 tons.

In the MAX and MIN forms of the FIND statement the expression in parentheses must appear, and must involve the index variable *vname*, so that a distinct value is obtained for each member of the set for which the expression is evaluated. The expression may be of either type, but REAL values are rounded to the nearest integer before comparison. If the maximum or minimum value occurs for more than one entity, the last entity for which that value is obtained is selected.

NESTED FIND STATEMENTS

A FIND statement, with a blank destination clause, may appear as a test within a test chain. The usual application of this facility is a FIND within another FIND, as shown in the following example.

Example

```
FIND N INPORT FIRST
   SHIP.N IN LOADED
   FIND B VACANT FIRST
      BERTH.B(3) EQ SHIP.N(2)
   DUMMY
SHIP.N INTO BERTHED
BERTH.B FROM VACANT
```

In this example, a ship is selected which is in the set LOADED, and for which a suitable vacant berth is available. At the same time the class index of the first vacant berth suitable for the selected ship is recorded.

The selected ship is then added to the set BERTHED, and the selected berth is removed from the set VACANT.

In some cases, for example when FIND appears in the test chain of a FIND...MAX or FIND...MIN statement, the required result of the inner FIND will be overwritten in subsequent repetitions of the test chain. In these cases the programmer must store each result of the inner FIND and extract the required result when execution of the outer test chain is complete.

Example

```
FIND X INPORT MAX (TONNAGE(X))
   FIND B VACANT FIRST
      BERTH.B(3) EQ SHIP.X(2)
   STORE(X) = B
B = STORE(X)
BERTH.B FROM VACANT
```

In this example an array called STORE is used to hold the results of the inner FIND. When the required ship, SHIP.X, is found, the corresponding vacant berth, BERTH.B, is found from the array STORE by taking the value of B held in the element STORE(X).

The final example, below, shows a FIND within another kind of test chain statement.

Example

```
SPLIT X INPORT INTO BERTHED
   FIND B VACANT FIRST
      BERTH.B(3) EQ SHIP.X(2)
   SHIP.X(3) = B
   BERTH.B FROM VACANT
```

This statement places in the set BERTHED every ship in the set INPORT for which a suitable vacant berth can be found. At the same time, for each ship, SHIP.X, which is placed in BERTHED, the number of the berth selected for that ship is stored in the attribute SHIP.X(3), and that berth is removed from the set VACANT.

Chapter 7

DISTRIBUTION SAMPLING FUNCTIONS

Each of the four functions described in this chapter provides a value which is obtained by sampling from a frequency distribution. Samples may be taken from the following distributions:

(a) Rectangular

(b) Normal

(c) Negative Exponential

(d) Distributions provided as input data by the programmer.

Each of these functions may only appear on the right hand side of an assignment statement of the form:

variable = sampling function

The value provided by the function is assigned to the variable on the left hand side of the statement, which may be a simple variable, array element, T-cell, or entity attribute. It may be of REAL or INTEGER type.

RANDOM NUMBERS

All four sampling procedures make use of a procedure which generates a stream of pseudo-random numbers. Each of the sampling functions has, as one of its parameters, a *stream variable* specified by the programmer. Each time the sampling function appears, the current value of the stream variable is operated on by the random number procedure to produce the next value in the sequence of random numbers, and a new value of the stream variable. Each stream variable should be assigned some initial value by the programmer, before it appears in a sampling function; thus for each separate stream variable an independent sequence of random numbers is provided.

The initial value of the stream variable may be any positive integer. If no value is assigned to the stream variable in the program, a standard value is assigned to it by the random number routine.

RANDOM

This statement has the form

variable = RANDOM (*stream*, *range*)

where *variable* and *stream* are as described above, and *range* may be any expression having an INTEGER value. A value is sampled from a Rectangular distribution: the result is a random number in the range 1 to *range*.

Examples

```
1       A = RANDOM (STREAMA, 101)

2       T.SHIP.N = RANDOM (STREAMB, MATRIX(N))
```

The first example assigns to the variable A, a random value in the range 1 to 100. STREAMA is the stream variable used.

DEVIATE

This statement has the form

variable = DEVIATE (*stream*, *deviation*, *mean*)

where *variable* and *stream* are as described above, and *deviation* and *mean* may be any expressions having INTEGER values. A value is sampled from a Normal distribution whose mean is the current value of *mean* and whose standard deviation is the current value of *deviation*.

Examples

```
1       P = DEVIATE (STREAMA, 2, 10)

2       T.VAR = DEVIATE (RAN(5), X, 10 * X)
```

The first example assigns to P a value sampled from the Normal distribution with mean 10 and standard deviation 2. STREAMA is the stream variable used. No number will be selected outside the range (*mean* − 5 × *deviation*) to (*mean* + 5 × *deviation*).

NEGEXP

This statement has the form

variable = NEGEXP (*stream*, *mean*)

where *variable* and *stream* are as described above, and *mean* is any expression having an INTEGER value. A value is sampled from a Negative Exponential distribution whose mean is the current value of *mean*.

Examples

```
1       N = NEGEXP (STREAMC, 50)

2       TABLE(N) = NEGEXP (BASE(N), TABLE(N))
```

In the first example, the value assigned to N is sampled from the Negative Exponential distribution with mean 50. STREAMC is the stream variable used.

SAMPLE

This statement has the form

variable = SAMPLE (*vname*, $dist_1$, $stream_1$, $dist_2$, $stream_2$,...)

where *vname* is a variable name, $dist_1$, $dist_2$,... are names of distributions supplied as input data, and $stream_1$, $stream_2$,... are stream variables, as described above.

The value of *vname* indicates the position in the parameter list of the name of the distribution to be sampled, and the stream variable to be used by the sampling procedure. Thus, if *vname* takes the value 3, the function samples from the distribution $dist_3$, using stream variable $stream_3$, where $dist_3$, $stream_3$ is the third pair of names in the list. The method of specifying distributions to be sampled in this way is described below.

Distributions Specified by the Programmer

A frequency distribution consists of a series of pairs of value n_t a_t, where n_t is the number of observations, or relative frequency of occurrence, of the value a_t.

The frequency distribution is stored in a two-dimensional array of dimensions $2 \times (m + 1)$, where m is the number of values a_t recorded.

The first column of the array contains the following pair of values:

row 1 : total number (check sum) of observations = $\sum_{t=1}^{m} n_t$

row 2 : number of values observed = m

Each of the remaining columns of the array contains one pair of values, n_t a_t.

The array is shown in diagrammatic form below:

Check sum	n_1	n_2	n_3	n_4	n_5	n_6	.....
m	a_1	a_2	a_3	a_4	a_5	a_6	.....

The following statements must appear in the program to define a distribution *before* a SAMPLE statement appears:

1 An ARRAY statement defining a $2 \times (m + 1)$ array to hold the distribution data.

2 Input statements (see Chapter 8) to input the distribution data, and assign it to the defined array.

3 A DIST statement, which designates the array as a distribution.

DIST

This statement has the form

DIST *aname*$_1$, *aname*$_2$, *aname*$_3$,...

where *aname*$_1$, *aname*$_2$,... are names of arrays holding distribution data. The DIST statement converts the frequency distributions stored in the specified arrays into *cumulative* frequency distributions, to enable samples to be taken. Consequently the original values of n_t read into the arrays are no longer available to the programmer.

EXAMPLE OF DISTRIBUTION STATEMENTS

This example shows a complete set of statements defining and sampling a distribution.

```
      ARRAY A(2,11)
      READ (1,10)A
   10 FORMAT (22I3)
      DIST A
      STREAM = 199
      X = SAMPLE (1, A, STREAM)
```

The ARRAY statement sets up an array A of dimensions 2×11. The READ statement (see Chapter 8) reads data from input unit 1, in the format specified by the statement labelled 10.

Successive values are assigned to the elements of A in 'column order', that is, in the order A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3),...

The FORMAT statement (see Chapter 8) defines the format in which the data is punched in the input medium.

In this case the data takes the form of 22 integers, each appearing in a field of three punching positions.

The DIST statement defines the array A to be a distribution, and converts the data to cumulative form.

The SAMPLE statement obtains a sample from distribution A, using the stream variable STREAM, and assigns the value obtained to the variable X.

The initial value of STREAM is 199.

The table below shows a list of data in a suitable form to be read and sampled by the above statements.

n_t	a_t
75	10
2	55
3	60
4	65
6	70
10	75
15	80
14	85
11	90
7	95
3	100

This data would be punched on tape or cards in the form

∇75∇10∇∇2∇55∇∇3∇60... etc.

where ∇ indicates a space.

After execution of the DIST statement the data would have the following form:

n_t	a_t
75	10
2	55
5	60
9	65
15	70
25	75
40	80
54	85
65	90
72	95
75	100

Chapter 8

INPUT AND OUTPUT OPERATIONS

The entire FORTRAN Input/Output system is available in C.S.L. The basic facilities will be described in this chapter for the benefit of readers not familiar with FORTRAN. Details of the full range of facilities may be obtained from the I.C.T. 1900 series FORTRAN manual, Part 3.

Some additional statements are provided in C.S.L. The statements READ INPUT TAPE, WRITE OUTPUT TAPE are provided for compatibility with other versions of C.S.L. Statements are provided for the formation and output of histograms. The CHECK statement is used to monitor the values of variables at intervals during the running of a program.

Summary of FORTRAN Input/Output Facilities

RECORDS

All data and results are handled in the form of records, and two kinds of record are recognized: formatted and unformatted. The form taken by a record depends on the medium on which it is held, and on whether it is formatted or unformatted. Examples of formatted records are one punched card, one line of print, and a string of characters on paper tape terminated by a newline character. A formatted record is considered to be split into *fields*, each field representing the value of one variable, or array element, or containing a string of text. An unformatted record consists essentially of information copied directly onto the output medium in its internally stored form. Unformatted records are usually output onto backing store media, such as magnetic tape, for subsequent re-input to the computer.

READ AND WRITE STATEMENTS

Records are input and output by READ and WRITE statements which may be formatted or unformatted. Each of these statements normally specifies the names of variables, array elements or complete arrays, whose values are to be transferred.

FORMAT INFORMATION

Each formatted READ or WRITE statement will have associated with it information about the external format of the data it is to input or output. This information is normally provided in a FORMAT statement which specifies, for each of the items named in the corresponding READ or WRITE statement, such information as the number of characters in the external field, and the positions of any decimal points.

PERIPHERAL UNITS

Each input and output statement refers, by a number chosen by the programmer, to a specific input or output peripheral. For example, if a programmer wished to use one card reader, one paper tape reader, and one line printer for a program, he might choose to refer to these by the numbers 1, 2 and 3 respectively. He would associate these numbers with actual types of peripherals by special statements within the 1900 FORTRAN Program Description (see Chapter 10).

CONTROL OF PERIPHERALS

FORTRAN also provides three auxiliary input/output statements: REWIND, BACKSPACE and ENDFILE. As their names suggest, these statements are more appropriately used to control some media, such as magnetic tape, than others. Four input/output subroutines, ALLOT, RLEASE, RUNOUT, DISENG, are provided in 1900 FORTRAN to control dynamically the allocation of peripherals to the program. However, these subroutines and statements will not be described in this manual. Details can be found in the I.C.T. 1900 series FORTRAN manual.

READ and WRITE Statements

INPUT/OUTPUT LISTS

READ and WRITE statements normally contain a list of the items whose values are to be transmitted. The list has the form

$$e_1, e_2, e_3, \ldots e_n$$

where each *e* is a *list element* and may be a variable, an array element, an array name, or a DO-implied list (described below).

When the READ or WRITE statement is executed, the value of each list element in turn is input or output. If an array name appears, the values of all the elements of that array are transmitted, in column order, that is, in the order obtained by varying the first subscript most rapidly and the last subscript least rapidly.

Also permitted as list elements, in C.S.L., are entity attributes, and T-cells.

Examples

```
1    A,
2    I(J,K),
3    Y(2 * L + 3),
4    MATRIX,
5    T.VARA,
6    SHIPS.5(1),
7    T.SHIPS.2
```

An element of the list may be a DO-implied list. The form and effect of a DO-implied list closely resemble those of a FOR statement (a DO statement in FORTRAN).

A simple DO-implied list has the form

$$(e_1, e_2, e_3, \ldots e_n, vname = m_1, m_2, m_3)$$

where each e is a list element, as defined above, and *vname*, m_1, m_2, m_3 have the same form and meaning as in the FOR statement (see Chapter 4). The READ or WRITE statement is executed repeatedly, operating on the list e_1, e_2, e_3, ... e_n, while the index *vname* takes values ranging from m_1 to m_2 in increments of m_3. As any list element e may itself be a DO-implied list, the DOs may be nested to any depth.

Examples

```
1        (A(I), I = 1, 10)
2        ((A(I,J), I = 1, 10),B(J), J = 1, 10)
```

The first example is equivalent to the list

```
A(1), A(2), ... A(10)
```

The second example is equivalent to the list

```
A(1,1), A(2,1), A(3,1), ...A(10,1), B(1),
A(1,2), A(2,2), A(3,2), ...A(10,2), B(2),
.....
A(1,10), A(2,10), A(3,10), ...A(10,10), B(10)
```

FORMATTED WRITE STATEMENT

A formatted WRITE statement has one of the forms

```
WRITE (u,f) k   or
WRITE (u,f)
```

where u is an INTEGER constant or variable identifying a particular peripheral unit, f is the label of a FORMAT statement, and k is an input/output list. When the statement is executed, the values of the items in the list are output in the order in which the items appear in the list, to create one or more new records on the unit u, according to the format specified by the FORMAT statement labelled f. The FORMAT statement may appear anywhere in the segment of the program containing the WRITE statement.

The second form of WRITE statement, with no list, may be used to write blank records, or records containing a fixed string of characters.

Examples

```
10 FORMAT (2I5)
   .....
   .....
   WRITE (3,10) X, Y
```

The WRITE statement outputs the values of X and Y in the format specified in the FORMAT statement, labelled 10, and on the peripheral unit identified by the number 3.

```
20 FORMAT (17H EXAMPLE OF TITLE)
   .....
   .....
   WRITE (4,20)
```

This example outputs on the unit identified by the number 4 a record consisting of the string of characters specified in the FORMAT statement labelled 20. The record output would consist of the words

```
EXAMPLE OF TITLE
```

FORMATTED READ STATEMENT

A formatted READ statement has one of the forms

READ (u, f) k or

READ (u, f)

where u is an INTEGER constant or variable that identifies a particular peripheral unit, f is the label of a FORMAT statement and k is an input/output list.

Each field in the external record should be presented in the format specified by f, and in the order required by the list k. When the statement is executed the reading of a new record is initiated. Any fields at the end of a previous record which have not been read will be ignored. Further records may be read, as required by the format specification. Each field read will be converted to the internal form of the appropriate type and assigned to the next element of the list k.

The FORMAT statement may appear anywhere in the segment containing the READ statement.

The second form of the READ statement, with no list, may be used if the next record or records on a particular device are not required. As many fields and records as are specified in the FORMAT statement are read, but no assignment takes place.

Examples

```
1          READ (1, 50) A,B,C
2       50 FORMAT (3I7)
```

The READ statement causes three values to be input from the unit identified by the number 1, and assigns these values to A, B and C. These values should appear on the input medium in the format defined by the FORMAT statement, labelled 50.

UNFORMATTED WRITE STATEMENT

An unformatted WRITE statement has the form

WRITE (u) k

where u is an INTEGER constant or variable that identifies a particular peripheral unit, and k is an input/output list. When the statement is executed, the values of the items in the list k will be output, in the order in which they appear in the list, to form a new record on unit u. The record will be in internal machine form, normally for later re-input.

Each unformatted WRITE will create one new record, irrespective of the number of items in the list k.

UNFORMATTED READ STATEMENT

An unformatted READ statement has one of the forms

READ (u) k or

READ (u)

where u is an INTEGER constant or variable that identifies a particular peripheral unit, and k is an input/output list. The external record should be in internal machine form and will normally have been created by a previous unformatted WRITE statement. When the statement is executed the whole of the next record from the unit u will be read and values will be assigned, in order, to the elements of the list k. The number of elements in the list k may be less than or equal to the number of values in the record, but may not exceed this number. The second form of the statement, with no list, may be used if the next record is not required. Although the record is read, no assignment takes place.

FORMAT Statements

Each formatted READ or WRITE statement refers to a FORMAT statement which describes the external character representation of data to be transmitted.

GENERAL FORM

A FORMAT statement has the form

```
FORMAT (s)
```

where *s* is a format specification.

A FORMAT statement must be labelled, but the label may only be referred to by input/output statements, and must *not* appear in any kind of transfer statement.

FORMAT SPECIFICATIONS

A format specification consists of a series of *field descriptors*. Field descriptors are separated by a comma or a slash; a slash also signifies the end of a record. The whole specification must be enclosed in parentheses, and parentheses may also be used to group together a number of fields within the specification. The parentheses at the beginning and end of the specification may be considered to initiate a new record and terminate a record respectively.

Example

```
FORMAT (I4, I7, E7.2/I6, E5.3)
```

This specification associated with a WRITE statement, will cause a record consisting of two INTEGER numbers followed by a REAL number, followed by another record consisting of one INTEGER and one REAL number, to be output. A number of blank records may be output, or a number of input records ignored, by writing a series of slashes.

Example

```
FORMAT (I4////I4)
```

This statement, associated with a WRITE statement, will output a record consisting of a single INTEGER number, followed by three blank records, followed by another record containing a single INTEGER number.

A single field descriptor, or a group of field descriptors enclosed in parentheses, may be repeated by writing in front of the descriptor, or group of descriptors, a repeat count.

Examples

```
FORMAT (3I5)
```

is equivalent to

```
FORMAT (I5, I5, I5)
```

Similarly

```
FORMAT (2 (I7, E5.3))
```

is equivalent to

```
FORMAT (I7, E5.3, I7, E5.3)
```

A group of descriptors may contain further groups in parentheses, with or without repeat counts. Groups may be nested to any depth.

Example

```
FORMAT (3I5, 2 (E5.3, 3I5, 5(I5, 3X)))
```

ACTION OF FORMAT SPECIFICATION

When a formatted input or output statement is executed, each action performed depends on information jointly provided by the next element in the input/output list (if one exists) and the next field descriptor obtained from the format specification.

Except for the effects of repeated groups, the format specification is interpreted from left to right. A new record is commenced at the start of execution of the input or output statement and additional records are commenced as the format specification requires.

If there is an input/output list, at least one field descriptor other than *w*H or *w*X must appear in an associated FORMAT statement (*w*H and *w*X descriptors are not associated with list elements). When an item in the input/output list is selected, the next available field descriptor is examined. If it is *w*H or *w*X appropriate action is taken and the next field descriptor is examined. The process is repeated until a descriptor other than *w*H or *w*X is found, and the value is then transmitted. The next item is then selected from the input/output list, and the process is repeated.

When all the elements in the input/output list have been operated on, execution ceases, unless the next field descriptor is *w*H or *w*X. Further *w*H or *w*X descriptors are acted upon, until a descriptor other than *w*H or *w*X is encountered, when execution ceases. A special case of the above is when the format specification contains only *w*H and *w*X descriptors, and there is no input/output list.

If the format specification has been completely scanned and there are still items left in the input/output list, then a new record is commenced and the scanning is repeated as follows:

1 If there are no internal parentheses then scanning is repeated from the beginning of the specification.

2 If there are internal parentheses then scanning is repeated from the left parenthesis corresponding to the right-most internal right parenthesis.

Examples

```
1    FORMAT ↑(.........)
2    FORMAT  (... ↑(.....))
3    FORMAT  (... ↑(...(....)...).....)
4    FORMAT  (....(...)..... ↑(...(.)..(.))...)
```

The arrows show where scanning would re-commence. The last example should clarify rule 2 above.

Field Descriptors

Nine different kinds of field descriptors are available in 1900 FORTRAN and C.S.L., distinguished by one-letter *conversion codes*. Only five of these will be discussed here, the I, E, F, H, and X codes.

INTEGER FIELDS

INTEGER data (the most usual form of data for C.S.L. programs) is transferred by means of the conversion code I. The descriptor has the form

I*w*

where *w* is the field width, an integer indicating the number of characters in the external representation of the number. With an output statement the descriptor causes the value of the corresponding list element to be output as an INTEGER number, right justified and occupying *w* character positions in the external record. A negative sign precedes the first digit if the number is negative; positive numbers are not signed. Leading blanks appear, if necessary, to make up *w* character positions. The number of characters to be output must not exceed the field width.

Examples

Descriptor	*Internal Number*	*External Number*
I5	+3659	b3659
	–987	b–987
	+3	bbbb3

A blank is represented here, and subsequently, by b. With an input statement the descriptor causes the next *w* characters in the external record to be read and converted to INTEGER form. The value is assigned to the corresponding list element.

Blanks before the number are ignored, but must be included in the character count. Any other blanks in the field are treated as zero. A field of all blanks is treated as zero. The field may contain a sign, but must not contain a decimal point or exponent.

Examples

Descriptor	*Internal Number*	*External Number*
I5	+393	b+393
	+23	bbb23
	–5868	–5868
	+2300	b23bb
	0	bbbbb

REAL NUMBER FIELDS

REAL values are transferred by means of the conversion code E. The descriptor has the form

$Ew.d$

where *w* is the field width, indicating the number of characters in the external representation of the number, and *d* is the number of digits in the fractional part of the number.

With an output statement the descriptor causes the value of the corresponding list element to be output as a decimal fraction with an exponent.

The fraction, f, is in the range

$$0.1 \leqslant f < 1$$

and rounded to d decimal places. A zero precedes the decimal point if the field is wide enough.

The exponent has one of the forms

$Eb d_1 d_2$ or
$E{-} d_1 d_2$

where d_1 and d_2 are digits. The number is right justified in the field. Both fraction and exponent are signed only if negative. Leading blanks appear, if necessary, to make up the *w* character positions. The number of characters to be output must not exceed the field width.

Examples

Descriptor	*Internal Number*	*External Number*
E14.5	+12345678	bbb0.12346Eb08
	–1.23	bb–0.12300Eb01
	+.000123	bbb0.12300E–03
	–.003	bb–0.30000E–02

With an input statement the descriptor causes the next *w* characters in the external record to be read and converted to REAL form. The value is assigned to the corresponding list element.

The *w* characters of the external field consist of an optional sign, followed by a string of digits optionally containing a decimal point, and optionally followed by an exponent part.

The exponent part may be one of the forms:

1 A signed INTEGER constant.

2 E or D followed by a signed INTEGER constant.

3 E or D followed by an unsigned INTEGER constant.

The exponent field has a maximum width of four characters.

Blanks preceding the first digit and blanks within the exponent are ignored. Blanks elsewhere are treated as zero. A field of all blanks is treated as zero. If the field contains a decimal point, then this will override the implied point specified by .*d* in the descriptor.

Examples

Descriptor	*Internal Number*	*External Number*
E7.3	+7654.321	7654321
	+.000137	b+137-3
	+123.4	1.234E2
	-12.3	-123E02

An alternative to the E code is the F code. The descriptor has the form

F*w*.*d*

where *w* is the field width and *d* is the number of digits in the fractional part of the number.

The form and effect of the external field on input is exactly the same as for the E conversion code.

With an output statement, however, the F code causes the value of the corresponding list element to be output as a decimal fraction, rounded to *d* decimal places. The number is right justified, and signed if the number is negative. Leading blanks may appear, to make up the *w* character positions, and trailing zeros may appear, to make up the *d* decimal places. If the number has no integral part, a zero precedes the decimal point, if the field is wide enough.

CHARACTER STRINGS

The conversion code H is used to transfer a fixed string of characters held in the format specification. The descriptor has the form

$wHh_1h_2h_3 \ldots h_w$

where *w* is the field width and each *h* is a character in the 1900 printing character set. The field width is the same as the number of characters in the descriptor.

No list element is associated with an H descriptor. When associated with an output statement the descriptor causes the *w* characters following the H code to be written as part of the output record.

Example

```
FORMAT (12H THE TIME IS)
```

The three spaces as well as the nine letters are included in the field width.

BLANK AND IGNORED FIELDS

The conversion code X is used to skip characters in the input record, or to output blank characters.

The descriptor has the form

*w*X

where *w* is the field width, indicating the number of characters to be skipped, or the number of blanks to be output. No list element is associated with an X descriptor.

With an output statement the specification will cause *w* blank characters to be output on the external record.

With an input statement the specification will cause the next *w* characters of the input record to be skipped.

USE OF LINE PRINTER

When a line printer is used for output, the first character of the line to be output is not printed, but is interpreted as a paper control character. Thus a format specification associated with line printer output should begin with an H descriptor of the form

1H*c*

where *c* is the required paper control character. The control characters are as follows.

space	one line feed
0	two line feed
1	throw to top of new page
+	no advance

Any other character is treated as a space, i.e. one line feed, except the characters 2 to 7 which assign control to channels 2 to 7 of the line printer's control loop (for further details see I.C.T. 1900 series FORTRAN manual, Part 2, Chapter 5).

READ INPUT TAPE AND WRITE OUTPUT TAPE

The statements READ INPUT TAPE and WRITE OUTPUT TAPE are exactly synonymous with READ and WRITE, and are permitted in order that programs written in IBM C.S.L. 1 may be translated by the 1900 C.S.L. translator. Each of these statements must appear as three separate words.

Histograms

A single run of a simulation program will normally consist of a large number of repetitions of the simulation procedure, and the sets of results produced will show random variations. It is therefore convenient to record by means of *histograms* the number of times that the results fall within certain ranges of value, rather than to print out the actual values obtained by each repetition.

HIST

The HIST statement defines one or more histograms. It has the form

HIST $name_1(n_1, n_2, n_3), name_2(n_4, n_5, n_6), \ldots$

where $name_1$, $name_2$, ... are C.S.L. names, and n_1, n_2, ... are unsigned INTEGER constants.

The first histogram is given the name $name_1$, and has n_1 cells. The first cell holds entries in the range $-\infty$ to n_2, the next (n_1-2) cells hold entries in successive ranges of width n_3 and the final cell holds entries from the upper limit of the previous cell up to $+\infty$.

Example

```
HIST HISTA(10,0,2),HISTB(6,100,5)
```

This statement sets up two histograms HISTA and HISTB, HISTA has the form shown diagrammatically below:

HISTA

Observations										
Value Range	$-\infty$ to 0	1 to 2	3 to 4	5 to 6	7 to 8	9 to 10	11 to 12	13 to 14	15 to 16	17 to ∞

The number of observations in each range is initially zero. The current value of an expression is recorded in a histogram by means of an ADD statement which adds 1 to the count of the number of observations in the appropriate range.

ADD

Simple form

There are two forms of the ADD statement. The simple form is

```
ADD expression, hname
```

where *hname* is a histogram name, and *expression* is any expression having an INTEGER result.

A count of one is added to the histogram *hname* in the cell whose range of values contains the current value of *expression*.

Examples

```
1   ADD A, HISTA
2   ADD T.SHIP.1, HISTOGRAM
3   ADD SHIP.N(3) - CLOCK, QTIME
```

In the first example one of the observation counts in the histogram HISTA, corresponding to the value of A, will be increased by 1.

Multiple form

This statement has the form

```
ADD (i,j, imax, jmax, expression1, expression2,... hname1, hname2, ...)
```

where i, j are INTEGER variables or unsigned INTEGER constants, $imax$, $jmax$ are unsigned INTEGER constants, $hname_1$, $hname_2$, ... are histogram names, and the expressions $expression_1$, $expression_2$,...have INTEGER values.

This statement makes an entry in one of the histograms, corresponding to the current value of one of the expressions. The expression involved is selected by the current value of i: thus if $i = 2$ an entry is made corresponding to the value of $expression_2$. The entry $imax$ specifies the maximum value which i may take, in other words the number of expressions in the list. The histogram in which the entry is made is selected by the current value of j. Thus if $j = 3$ an entry is made in the histogram $hname_3$. The entry $jmax$ specifies the maximum value that j can take, in other words the number of histogram names in the list.

Example

```
ADD (I, J, 2, 2, SHIP.N(3), SHIP.N(2), HISTA, HISTB)
```

In this example if I = 1 and J = 2, say, then an entry corresponding to the value of SHIP.N(3) would be made in the histogram HISTB.

OUTPUT

This statement has the form

OUTPUT $hname_1$, $hname_2$, ...

where $hname_1$, $hname_2$, ... are histogram names.

The contents of the histograms listed in the statement are output on a 'standard' output unit, which is a unit selected in the Program Description for the output of all diagnostic information from the object program.

A histogram is printed out in the following form.

One line is printed for each value range, or cell, of the histogram.

Each line has the form

n_1 m_1 TO m_2

where n_1 is the number of observations recorded in the range m_1 to m_2.

The two columns of information are headed by the words COUNT and RANGE. No histogram identification appears in the output; histogram names, if required, must be output by means of a normal WRITE statement preceding the histogram output.

Examples

```
1        OUTPUT HISTA
2        OUTPUT HA, HB, HC
```

If HISTA were of the form defined in an example earlier in this chapter, then the result of the first example might be the following output.

Count	Range
0	TO 0
0	1 TO 2
2	3 TO 4
7	5 TO 6
15	7 TO 8
27	9 TO 10
11	11 TO 12
4	13 TO 14
3	15 TO 16
1	17 TO

CLEAR

This statement has the form

CLEAR $hname_1$, $hname_2$, ...

where $hname_1$, $hname_2$, ... are histogram names. This statement clears the histograms listed, resetting all the observation counts to zero. The structure of the histograms is retained and after clearing they may be used again without further definition.

Example

```
CLEAR HISTA, HISTB, HISTC
```

This example resets to zero all the observation counts in the histograms HISTA, HISTB, HISTC.

Note that although a histogram resembles a frequency distribution, the data contained in a histogram may not be sampled.

Diagnostic Output

CHECK

This statement has the form

CHECK $name_1$, $name_2$, ...

where $name_1$, $name_2$, ... are names of variables, array elements, entity attributes, or time-cells, defined elswhere in the program. For each item mentioned in the statement, up to eight characters of the name, followed by the current value of the item, are output on the 'standard' output unit, which is the unit selected in the Program Description for the output of diagnostic information.

Example

```
CHECK A, T.SHIP.A, SHIP.A(5), TONNAGE(3)
```

This statement might output the lines

```
A          EQUALS 30
T.SHIP.A   EQUALS 355
SHIP.A(5   EQUALS 0
TONNAGE(   EQUALS 24,000
```

The CHECK statement is executed only if the Diagnostic Control switch K1 is 'on', i.e. set to a positive value. See Chapter 10 for full details of Diagnostic Control, and other forms of diagnostic output.

Chapter 9

PROGRAM STRUCTURE

Activities and Time Advance

The part of a program that represents the dynamic operation of the model consists of a list of *activities*. Each activity is a section of program that makes a series of changes in the model whenever certain initial conditions are satisfied. An activity is therefore normally headed by one or more test statements which determine whether the activity can be carried out. If these tests are not satisfied control normally passes to the next activity, and this transfer is usually carried out automatically by means of blank destination clauses.

One run of a simulation program normally consists of a large number of 'passes' through the list of activities. At each pass all the activities are entered in turn, but each activity will only be carried out if the initial conditions are satisfied.

Most systems studied by simulation are time-dependent; C.S.L. therefore provides a mechanism for representing the passage of time. Although a real system may change continuously, in a C.S.L. model it is assumed that changes take place instantaneously at discrete points in time, and that in the intervening time intervals no changes occur. Thus time is advanced in discrete steps, in the following way.

Initially, and at intervals during execution of the program, one or more of the time-cells defined in the model are assigned values, each of which represents the time that must elapse before an entity is due to change its state, or some process is due to begin. The time-advance routine examines all the time-cells and selects the smallest positive value, ignoring zero and negative values. This value represents the time that must elapse before any change in the model may take place, so 'simulation time' is advanced by this amount. Simulation time is represented by a variable called CLOCK, which does not have to be defined in the program, but may be used in the program if required. CLOCK is initially set to zero, and at each time-advance the selected time value is added to CLOCK, and subtracted from all the time-cells, making the selected one zero.

This time-advance is carried out at the *end* of each pass through the activities, unless either of the statements EXIT or RECYCLE has been encountered during the execution of the activities. The effect of these instructions is described below.

The activities structure may be seen in the example programs in Chapter 11.

ACTIVITIES

The statement

```
ACTIVITIES
```

defines the start of the section of program containing activities. This statement instructs the compiler to insert the time-advance and repetition statements. The remainder of the program, up to the final END statement, is regarded as the list of activities, and is executed repeatedly.

BEGIN

The start of each activity is defined by the statement

BEGIN *identification*

The main purpose of this statement is to provide a destination for any blank destination clauses in the previous activity. The word BEGIN may be followed by up to 18 characters (including blanks) of identification. This identification may be output as diagnostic information each time the activity is entered at run-time. The BEGIN statement may be omitted from the first activity.

RECYCLE

It is possible for an activity to make changes in the model which permit a previous activity to be carried out. If this is likely to occur the instruction

RECYCLE *identification*

may be inserted in the activity. If one or more RECYCLE statements are encountered during any execution of the activities, then a further execution of the complete list of activities is carried out before the next time-advance.

RECYCLE may appear in the program any number of times. Repetition of the list of activities is continued until a repetition is made during which no RECYCLE statement is encountered. It is then assumed that no further changes can be made, and a further time-advance is carried out.

RECYCLE may be followed by up to 18 characters (including blanks) which may be output as diagnostic information each time the statement is encountered.

EXIT

The statement

EXIT

is used to terminate the execution of a program (see also Chapter 10).

Example

The final statements of a program might be:

```
     .....
     .....
     CLOCK GE 1000  @100
     EXIT
100  DUMMY
     END
```

If the value of CLOCK is greater than or equal to 1000, control passes to the next statement after the test, which is EXIT; thus execution of the program will halt when simulation time reaches 1000. If CLOCK is less than 1000, control passes to statement 100, so that the EXIT statement is not encountered. Control then passes back to the beginning of the list of activities, and execution continues.

Program Segmentation

A C.S.L. program may be written and compiled in *segments*, in the same way as a FORTRAN program.

Segments are of four kinds: MASTER segment, SUBROUTINE segment, FUNCTION segment, and BLOCK DATA segment.

Segments may be written in FORTRAN or C.S.L. A BLOCK DATA segment may only be written in FORTRAN, and this type of segment will not be dealt with here.

MASTER SEGMENT

Every C.S.L. program must contain one, and only one, MASTER segment. This segment starts with the statement

MASTER *name*

where *name* may be any valid FORTRAN name. Like all other segments, the MASTER segment must end with the statement

END

The MASTER segment is the controlling segment of a program, and it may be the only segment. Execution of the program starts with the first executable instruction of the MASTER segment. Control may subsequently be transferred to other segments by CALL statements or function references. A SUBROUTINE or FUNCTION segment may, in turn, transfer control to other segments, but may not return control to the MASTER segment, except by a RETURN statement (see below).

SUBROUTINE SEGMENT

A SUBROUTINE segment must take the form of a complete self-contained program, in the sense that any REAL variables, classes, sets, arrays, distributions, or histograms eappearing, must be defined in the segment by the appropriate definition statements. Values associated with any of these items may be transmitted to the segment from other segments by placing the items in a COMMON storage block, or specifying them as *arguments* to the SUBROUTINE statement.

A SUBROUTINE segment may contain any statements except MASTER, SUBROUTINE, FUNCTION or BLOCK DATA. A SUBROUTINE segment must finish with the statement

END

SUBROUTINE

The first statement of a SUBROUTINE segment must be a SUBROUTINE statement, of the form

SUBROUTINE £*name* (a_1, a_2, a_3,...)

where *name* is any valid FORTRAN name, and must be preceded by the character £. Any string of numeric characters appearing in the name must also be followed by the £ character.

a_1, a_2, a_3,... are dummy arguments; when the segment is executed each of these will be replaced, wherever they appear in the segment, by corresponding actual arguments specified in the CALL statement. A SUBROUTINE statement with no arguments is permitted.

Examples

```
1    SUBROUTINE £SUB2£ (A, B, X, Y)
2    SUBROUTINE £PRINTOUT (SETA)
```

CALL

Control is transferred to a SUBROUTINE segment by a CALL statement, of the form

CALL £*name* (a_1, a_2, a_3,...)

where *name* is the name of a SUBROUTINE segment, and a_1, a_2, a_3 are actual arguments which will replace the corresponding dummy arguments throughout the SUBROUTINE segment. Actual arguments must correspond in number, order and type, to the dummy arguments which they replace.

Examples

```
1       CALL £SUB2£ (VARA, VARB, MATRIXA, MATRIXB)
2       CALL £PRINTOUT (ATSEA)
```

RETURN

The statement

```
RETURN
```

will normally appear at least once in any SUBROUTINE segment. This statement transfers control to the statement immediately following the CALL statement in the calling segment.

EXAMPLE OF SUBROUTINE

```
      SUBROUTINE £PRINTOUT (SETA)
      CLASS X.20 SET SETA
      FOR I = SETA
          WRITE (3 ,10) I
10    FORMAT (1H, I4)
      RETURN
      END
```

This subroutine prints the class indices of the members of a specified set. A class X has been invented merely to dimension the dummy set SETA; any set may appear as the actual argument to the subroutine. The subroutine would be called by a statement such as

```
CALL £PRINTOUT (ATSEA)
```

and would then be executed, operating on the members of the set ATSEA.

FUNCTION SEGMENTS

A FUNCTION segment is similar to a SUBROUTINE segment, except that it provides only a single numerical result. Control is transferred to a FUNCTION segment each time the function name appears; the function is then evaluated and the numerical result substituted for the function name at the point where it appears in the calling segment. The function name may appear in an expression wherever a variable name is permitted.

Thus if a statement appeared as follows

```
A = X + £FUNCT (Y, Z)
```

control would pass to the FUNCTION segment with the name £FUNCT. The segment £FUNCT would operate on the current values of Y and Z to provide a value; this value would then be substituted for £FUNCT (Y, Z) in the above expression, and the expression would then be evaluated in the normal way.

FUNCTION

A FUNCTION segment must start with the statement

```
FUNCTION £name (a1, a2, a3,... )
```

where *name* is any valid FORTRAN name. Any string of numeric characters appearing in the name must be followed by another £. The type of the function, that is, whether the result is to be REAL or INTEGER, is indicated by the first letter of the name. The first character must be I, J, K, L, M, or N if the result is to be INTEGER; any other initial letter indicates a REAL result.

a_1, a_2, a_3, ... are dummy arguments which are replaced by actual arguments when the function is evaluated. At least one argument must appear.

The value provided by the FUNCTION segment is actually substituted for the function reference, where it appears in the calling segment. In order to effect this substitution the function name is treated as a variable name in the FUNCTION segment, and must be assigned a value in an assignment statement. Thus a FUNCTION segment might have the form

```
FUNCTION £FUNCT (Y, Z)
.....
.....
£FUNCT = 3 * Y + 4 * Z
RETURN
END
```

The word RETURN must appear, in order to transfer control back to the calling segment, and the word END must be the last statement of the segment.

COMMON

This statement has the form

COMMON $name_1$, $name_2$,...

where $name_1$, $name_2$,... are unsubscripted C.S.L. names. All the items listed are placed in a special area of storage which is available to all segments of the program, provided they contain a COMMON statement. Items appearing in COMMON statements in separate segments are assigned, in the order in which they are listed, to the same storage locations. Thus values obtained in one segment may be used in another segment. Only one COMMON statement may appear in any one segment.

Example

If the statement

```
COMMON A, B, MATRIX, SETA
```

appeared in one segment, and the statement

```
COMMON X, Y, TABLE, SETB
```

appeared in another, than A and X would refer to the same storage location, and therefore the same numerical value. Likewise B and Y would refer to the same value, and so on.

A COMMON statement may contain arrays, classes, sets, and T-arrays, provided they are defined in the appropriate definition statements *before* the COMMON statement, in the segment in which they appear. The same area of COMMON storage may be represented by a series of variables in one COMMON statement and by an array in another COMMON statement.

Example

If the statement

```
COMMON A, B, C, D, E, F, G
```

appeared in one segment, and the statements

```
ARRAY A (5)
COMMON A, X, Y
```

appeared in another, then the array element A(1) and the variable A would refer to the same value, A(2) and B would refer to the same value, and so on.

Chapter 10

COMPILATION AND EXECUTION

A C.S.L. program may consist of segments written in C.S.L. or FORTRAN. All segments, however, are presented to the C.S.L. translator, which translates C.S.L. statements into FORTRAN, and copies FORTRAN statements unchanged. The translator outputs a complete FORTRAN program, which can then be compiled by the FORTRAN compiler in the normal way.

Two sets of control statements are required: a series of statements to control the operation of the C.S.L. translator and a series of statements to control the operation of the FORTRAN compiler.

C.S.L. Translator Control Statements

The statements below must appear before the first segment of a program, and may also appear between segments.

SOURCE

This statement describes the nature of the source program being presented to the C.S.L. translator. It applies to all subsequent segments until a further SOURCE statement appears.

This statement has the form

SOURCE (*peripheral, type, file.subfile*)

The parameter *peripheral* indicates the type of peripheral unit from which the source program is to be input. The unit is indicated by a two-letter code, as follows:

CR	card reader
TR	tape reader
MT	magnetic tape unit

The parameter *type* indicates the source language of the segments to be input, by means of a two -character code, as follows:

C.S.L.	S1
FORTRAN	F4

The remainder of the statement is applicable only if the source program is on magnetic tape; the third parameter gives the names of the file and subfile in which the program is stored on tape. File names and subfile names consist of up to twelve characters, and may include alphabetic and numeric characters, and the symbol -. The first character of each name must be alphabetic.

LABEL

This statement is associated with the OBJECT statement described below, and is only required if the translator is to output the translated program to magnetic tape.

The LABEL statement has the form

LABEL ($name_1$, $name_2$, n)

The parameter $name_1$ is the name of the tape to be opened; if $name_1$ is omitted a scratch tape is opened. The first comma must appear even if $name_1$ is omitted.

The parameter $name_2$ is the name with which the tape is to be re-labelled; n is the new retention period of the tape. If $name_2$ and n are omitted, the tape opened will keep its old name and retention period, and new information will be written after any information already on the tape. If the first two parameters are present and n is omitted, a retention period of 500 is assumed.

OBJECT

The OBJECT statement indicates how the translated program, in FORTRAN, is to be output. The statement has the form

OBJECT (*peripheral*, *file.subfile*)

where *peripheral* is a two-letter code indicating the type of peripheral unit to be used, as follows:

CP	card punch
TP	tape punch
MT	magnetic tape unit

The remainder of the statement, which is only applicable if the program is to be output to magnetic tape, gives the names of the file and subfile in which the program is to be stored. The name of the file is the same as the name allocated to the tape in the LABEL statement. If magnetic tape output is specified, the OBJECT statement must be preceded by a LABEL statement.

LISTING

The statement

LISTING (n)

indicates what listing is to take place during compilation, as follows:

LISTING (1)	C.S.L. segments are listed, together with control statements and errors.
LISTING (2)	The FORTRAN object program is listed, together with FORTRAN source segments, control statements and errors.
LISTING (3)	The complete source and object programs are listed, together with control statements and errors.

If the statement is omitted, only control statements and errors are listed. The LISTING statement may appear between segments, to give different forms of listing for different segments.

SWITCH

The statement

SWITCH

is required when the segment or segments following are to be input from a different peripheral unit to the one previously specified. A SOURCE statement specifying the new peripheral must precede the SWITCH statement, but reading from the new unit will not commence until the SWITCH statement is encountered.

ENDSUBFILE

The statement

ENDSUBFILE

may appear between segments, or at the end of a complete program. This statement closes the subfile opened by the previous OBJECT statement. If any further compilation is to take place another OBJECT statement must appear after the ENDSUBFILE statement.

FILE AND SUBFILE NAMES

Wherever file and subfile names are specified in a statement, the following rules must be observed.

1 Each name may contain up to 12 characters. The letters A to Z, the numbers 0 to 9 and - (hyphen, or 'minus' sign) are the only characters permitted.

2 The first character of each name must be alphabetic.

3 Spaces may not appear preceding either name. Spaces may appear within a name, but are significant, and must appear whenever the subfile is subsequently referred to. The compiler inserts spaces after each name, if necessary, to make up 12 characters, but these spaces need not appear when the subfile is referred to.

4 Except in the LABEL statement, both names must appear.

IBM - READING MODE

The 1900 C.S.L. translator will accept programs written in the IBM language C.S.L. 1, provided the necessary control statements are added. An IBM-Reading Mode is provided, in which the translator will accept programs punched on cards in the IBM character code, so that C.S.L. programs prepared for IBM computers may be run on a 1900 series computer with a minimum of alteration. IBM-Reading Mode is specified by the statement

ZIBM

preceding the program and control statements.

END OF COMPILATION

The end of the complete C.S.L. source program is marked by

****

in columns 1 to 4. This indicates to the translator that no further segments are to be input.

EXAMPLE OF CONTROL STATEMENTS

The complete input to the C.S.L. translator might be as follows.

```
SOURCE (CR, S1)
LABEL (,SIMULATION, 50)
OBJECT (MT,SIMULATION.STEELWORKS1)
```

```
        LISTING(3)
        Source segments
        .....
        .....
        ENDSUBFILE
****
```

FORTRAN Compiler Control Statements

The FORTRAN program output by the C.S.L. translator is re-input for compilation by the FORTRAN compiler in the normal way. The user, however, must provide the necessary control statements to control the mode of compilation and execution of this program. These statements may be input separately at the second stage of compilation, or may appear in the original C.S.L. program, in which case they must be preceded by * in column 1. They are then copied directly into the translated program. A brief description of the necessary statements is given here; a full description of the 1900 FORTRAN compiling system can be found in the I.C.T. 1900 series FORTRAN manual, Part 3.

The input to the 1900 FORTRAN compiler, during one compiling run, consists of the following items, in the order shown.

Listing statement (optional)

SEND TO statement (optional for some compilers)

Program Description segment

Source Language segments

FINISH statement

LISTING STATEMENT

During compilation of a FORTRAN program the compiler outputs a partial or complete list of source program statements, errors found in the program, and some additional information.

The mode of listing is specified by one of the statements

LIST (p) or

SHORTLIST (p)

where p is a two-letter code indicating the type of peripheral on which this list is to be output.

The LIST statement gives a full list of source program statements, compiler control statements, and errors.

The SHORTLIST statement gives only a list of errors, compiler control statements, and segment headings.

Listing is normally output on a line printer (code LP). If the listing statement is omitted SHORTLIST (LP) is assumed. A tape punch (TP) may be specified, but only with the SHORTLIST statement.

SEND TO

The SEND TO statement indicates the type of unit on which the semi-compiled program is to be output. This statement has the form

SEND TO (p, a)

where *p* is a two-letter code (CP, TP, MT) indicating the peripheral type, and *a*, which is only applicable if magnetic tape is specified, gives the names of the file and subfile where the program is to be stored. *a* has the form

file.subfile

where *file* is a file name, and *subfile* is a subfile name.

The statement

SEND TO (MT)

outputs the program to a scratch tape, which is given the file name

PROGRAM *name*

where *name* is the name assigned to the program in the Program Description segment.

PROGRAM DESCRIPTION SEGMENT

PROGRAM

The first statement of the Program Description is the PROGRAM statement, having the form

PROGRAM (*name c*)

where *name* is a series of exactly four characters, by which the program is to be known to Executive, and by which it can be identified at run-time. The parameter *c* is an accounting code, of up to eight characters, which is optional, but is required by some installations.

INPUT, OUTPUT, USE AND CREATE

These statements define the number and type of peripheral units to be used by the program for the input and output of data. They also assign to each unit of each type one or more numbers which are used to refer to that unit in READ or WRITE statements.

The statements have the form

INPUT $m_1, m_2, \ldots = P$

OUTPUT $m_1, m_2, \ldots = P$

USE $m_1, m_2, \ldots = P$

CREATE $m_1, m_2, \ldots = P$

Each *m* is an integer in the range 0 to 4095, which is used to refer to the peripheral in a READ or WRITE statement. *P* consists of a two-letter code indicating the type of peripheral, followed by an integer in the range 0 to 63 which distinguishes different peripherals of the same type. When magnetic tape is specified, *P* may also specify the file name of the tape, when it may take the following forms:

MT*n*(*f*)

MT*n*/FORMATTED (*f*)

where the two statements refer to unformatted and formatted files respectively; *f* is a file name.

When magnetic tape is specified, the four statements have the following meanings:

INPUT	Obtain the named magnetic tape and allow it to be used for input only.
OUTPUT	Obtain the named magnetic tape and allow it to be used for output only.
USE	Obtain the named magnetic tape (or a scratch tape, if no name is specified) and allow it to be used for input and output.
CREATE	Obtain a scratch tape, name it, and allow it to be used for output, followed by input if required.

USE and CREATE may only specify magnetic tape input and output.

Example

```
INPUT1=CR0
OUTPUT2=LP0
USE3=MT0
USE4=MT1
CREATE5=MT2 (OUTPUTFILE)
```

This series of statements specifies that the program is to use a card reader for input, a line printer and magnetic tape for output, and two 'work' tapes, which will be scratch tapes before and after use.

END

The Program Description segment must be terminated by the statement

```
END
```

READ FROM

This statement enables the input to the compiler to be switched from one input unit to another. It has the form

```
READ FROM (p, a)
```

where p is a two-letter code (CR, TR, MT) and a gives a file name and subfile name, when magnetic tape is specified. The parameter a has the form

$$file.subfile$$

where $file$ is a file name and $subfile$ is a subfile name.

FINISH

The final segment of a program is followed by the statement

```
FINISH
```

which indicates to the compiler that no further program is to be compiled.

Diagnostic Output

SYNTACTICAL ERRORS

During compilation both the C.S.L. and FORTRAN compilers output lists of errors in the source program. A list of error messages for the C.S.L. compiler is given in Appendix 2. A list of error messages for each FORTRAN compiler may be found in the appropriate compiler specifications.

LOGICAL ERRORS

A source program may be syntactically correct, yet not carry out correctly the operations intended by the programmer. Errors of this type cannot be picked out by the compiler, but a variety of information may be output as the program is running, to assist the programmer in finding the point in the run at which an error occurred.

Diagnostic Control

The diagnostic facilities of C.S.L. are mostly optional, and controlled by three variables, K1, K2 and K3. The values of these variables are the first items read by the object program. In order to assign values to these variables the C.S.L. compiler generates the statements

```
        READ (1, 25002) K1, K2, K3
25002 FORMAT (3I10)
```

The user must supply values in the form of a single record in the format 3I10, on the input unit which has been assigned the number 1 in the program description.

The significance of the values assigned to K1, K2, and K3 is described below.

In addition, certain diagnostic facilities are made available by compiling sections of the program in Index Checking Mode. Compilation in Index Checking Mode is started by the instruction

```
START
```

and terminated by the instruction

```
FINISH
```

in the C.S.L. source program.

The whole program, or any number of sections, of any size, may be compiled in Index Checking Mode, but the statement START and FINISH must not appear in indented sections of the program.

In sections of the program compiled in this mode, each time a subscript appears, referencing an array element or attribute, statements are inserted to check whether, during execution, the value of the subscript is within the permissible range.

Example

If the statement

```
Y = MATRIX(N)
```

appears, where MATRIX is an array of 20 elements, an error will be detected and signalled if, at the time of execution of the statement, the value of N is outside the range 1 to 20.

CHECK OUTPUT

If the variable K1 is assigned a *positive* value, any CHECK statements in the program output the values of the specified variables. If K1 is zero or negative, CHECK statements are ignored.

TIME-ADVANCE AND ACTIVITIES OUTPUT

If K2 is *positive,* messages are output at each time-advance, and at each execution of a BEGIN or RECYCLE statement. If K2 is zero or negative no messages will be output.

ERROR MESSAGES

If K3 is *positive* certain kinds of errors are dealt with in such a way that the run can continue. The error procedures are as follows.

1 In the complex forms of the ADD and SAMPLE statements, if the index parameter has a value outside the permitted range the following action is taken. The value of the parameter is replaced by 1 or by the maximum permitted value, according to whether the erroneous value was too low or too high. The statement is then carried out, and the program continues. Error messages are printed, in the form

INDEX VALUE ... ON LINE ... IS OUTSIDE PERMITTED RANGE 1 TO ...

ERROR IN ROUTINE ...

VALUE CORRECTED TO ... AND EXECUTION CONTINUES

2 When, in Index Checking Mode, an error is found in a subscript value, the error is corrected and error messages are output as in 1, and the program continues.

3 If a set overflows, the overflowing entity or entities are ignored and the program continues. Error messages are output in the form

SET OVERFLOW

ERROR IN ROUTINE ...

OVERFLOW ENTITIES IGNORED AND EXECUTION CONTINUES

If K3 is zero or negative, any of the above errors causes execution to halt.

Error messages are printed, as above, but the message indicating that execution is continuing is replaced by the message

ERROR EXIT

Further diagnostic facilities are provided in FORTRAN, and may be used in the execution of C.S.L. programs. These facilities are too extensive to be described here. Details can be found in the I.C.T. 1900 series FORTRAN manual, Part 3.

Overlays

Programs may be written which are too large to be held in the core store of the computer. The 1900 FORTRAN compiling system permits such a program to be run with parts of the program overlaid from a backing store medium such as magnetic tape. In other words, only part of the program is in the core store at a time, the remainder being held on the backing store.

To use the overlay system the programmer divides his program into units. One unit is not overlaid; this is called the permanent unit, and is held in the *permanent area* of store.

The remaining units are called *overlay units,* and each is assigned to an *overlay area* of store. There may be several overlay areas and each one may have several overlay units assigned to it, but an overlay area may contain only one overlay unit at any one time.

The structure of the overlaid program is defined in OVERLAY statements in the FORTRAN Program Description. These statements are described, together with full details of the use of the overlay system, in the I.C.T. 1900 series FORTRAN manual, Part 3, Chapter 6.

A feature of the 1900 FORTRAN system is that it does not require the user of the overlay system to call any special subroutines. The source program is written in the normal way and segments are called by CALL statements or function references exactly as if the whole program were to be in core store. Any segment may call any other segment, whether it is in the same area or not. If the segment called is in an overlay unit the system will check to see if that unit is already in the core store. If it is not, it will be copied in from the backing store, and the required segment will then be entered in the normal way. The RETURN statement will have a similar effect, checking to see if the unit containing the calling segment is in the core store, and copying it in if it is not. Thus the program can be divided into segments and overlay units in any way the programmer requires, with the exception that the MASTER segment must be in the permanent unit.

A convenient use of overlays in C.S.L. is to overlay activities. In a typical simulation some activities may be repeated at frequent intervals, while others are only occasionally executed, so the programmer may obtain a considerable saving in storage, in return for a slight increase in running time, by holding the less frequently used activities on tape.

The technique described above requires that the program be written as a series of subroutines. The initial test statements of each activity may be contained in the MASTER segment and held permanently in core store, while the remainder of each activity is contained in a subroutine segment and held in an overlay unit. The program would then have the following structure:

```
MASTER name
.....
ACTIVITIES
BEGIN activity1
test1
test2
.....
CALL segment1(.....)
BEGIN activity2
test3
test4
.....
CALL segment2(.....)
.....
END
SUBROUTINE segment1(.....)
.....
.....
END
SUBROUTINE segment2(.....)
.....
etc.
```

Chapter 11

EXAMPLE PROGRAM

Example: Clinic with Four Doctors

THE MODEL

Four doctors are in attendance at a clinic. Patients arrive one at a time, and are treated as soon as a doctor is available. If more than one doctor is free, the choice of which doctor shall treat a patient is made according to a simple priority system. The four doctors form a class, and so are referred to as DOCTOR.1, DOCTOR.2, DOCTOR.3, DOCTOR.4. If DOCTOR.1 is busy, DOCTOR.2 is chosen, when DOCTOR.1 and DOCTOR.2 are busy DOCTOR.3 is chosen and so on.

Though the patients would probably form a queue in reality, the program merely records the number of patients waiting at any time. This quantity is the variable PATIENTS. Patients arrive at intervals of time which are normally distributed, with a mean of 7 (minutes) and a standard deviation of 1. After 180 minutes the clinic 'closes', and no more patients arrive, though the patients already waiting are treated.

The times taken by the doctors to treat patients are sampled from a distribution provided as input data, and called DOCTIME. The set DOCFREE contains the names of the doctors who are free at any time.

PROGRAM AND RESULTS

The following pages show the listing of the C.S.L. source program, and one page (the last) of the printout obtained when the program was run.

The lines

```
                ENDSUBFILE
    ****
```

should appear at the end of the program listing; they are not shown here because they appear on a new page in the printout.

```
PROGRAM      1           LISTING(3)
             2           SOURCE(CR,S1)
             3           LABEL(,FORTFILE,500)
             4           OBJECT(MT,FORTFILE    .FORTSUBFILE1)
             5     *     PROGRAM(DOCT)
             6     *     COMPRESS INTEGER AND LOGICAL
             7     *     INPUT 1=CR0
             8     *     OUTPUT 6=LP0
             9     *     END
            10           MASTER DOCTOR
            11           CLASS TIME DOCTOR.4 SET DOCFREE
            12           ARRAY SEEN(4),DOCTIME(2.11)
            13           READ(1,101)DOCTIME
            14       101 FORMAT(22I2)
            15           DIST DOCTIME
            16           STREAMA=13
            17           STREAMB=17
            18           T.PATIENT=2
            19           DOCTOR LOAD DOCFREE
            20           FOR I=1,4
            21             SEEN(I)=0
            22             T.DOCTOR.I=-1
            23           PATIENTS=0
            24           BEGIN DOOR SHUTS
            25           CLOCK GE 180
            26           T.PATIENT=-1
            27           BEGIN PATIENT ARRIVES
            28           T.PATIENT EQ 0
            29           T.PATIENT=DEVIATE(STREAMA,1,7)
            30           PATIENTS+1
            31           BEGIN DOCTORFREE
            32           FOR I=1,4
            33             T.DOCTOR.I EQ 0
            34             SEEN(I)+1
            35             DOCTOR.I TAIL DOCFREE
            36           BEGIN DOCFREEPATWAIT
            37           PATIENTS GT 0
            38           FIND I DOCFREE MIN(I)
            39           PATIENTS-1
            40           DOCTOR.I FROM DOCFREE
            41           T.DOCTOR.I=SAMPLE(1,DOCTIME,STREAMB)
            42           RECYCLE
            43           BEGIN FINISH
            44           PATIENTS EQ 0
            45           CLOCK GE 180
            46           UNIQUE(4) I DOCFREE
            47           WRITE(6,100)CLOCK,SEEN
            48       100 FORMAT(7H CLOCK=,I4/4I8)
            49           EXIT
            50           END
```

RESULTS

```
FINISH                (B)
RECYCLE
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
NEW ACTIVITY PASS WITH CLOCK =          170
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
NEW ACTIVITY PASS WITH CLOCK =          171
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
NEW ACTIVITY PASS WITH CLOCK =          175
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
                      (R)
FINISH                (B)
RECYCLE
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
NEW ACTIVITY PASS WITH CLOCK =          181
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
NEW ACTIVITY PASS WITH CLOCK =          196
DOOR SHUTS            (B)
PATIENT ARRIVES       (B)
DOCTORFREE            (B)
DOCFREEPATWAIT        (B)
FINISH                (B)
CLOCK= 196
     14       9        2        1
NORMAL EXIT
```

Appendix I

C.S.L. STRUCTURAL WORDS

The structural words of C.S.L. are listed below in alphabetic order. None of these words may be used as a name in a C.S.L. program.

ACTIVITIES
ADD
ALL
ANY
ARRAY

BEGIN

CALL
CHAIN
CHECK
CLASS
CLEAR
COMMON
CONVERSE
COUNT

DEVIATE
DIST
DUMMY

ELSE
EMPTY
END
ENDSUBFILE
EQ
EQUALS
EX
EXISTS
EXIT

FIND
FINISH
FIRST
FLOAT
FOR
FORMAT
FROM
FUNCTION

GAINS
GE
GO
GOTO
GT

HEAD
HIST

IF
IN
INPUT
INTO

LABEL
LAST
LE
LISTING
LOAD
LOSES
LT

MASTER
MAX
MIN

NE
NEGEXP
NIN
NOTIN

OBJECT
OR
OUTPUT

PRINT
PUNCH

QUALIFY

RANDOM
RANK
READ
RECYCLE
REPEAT
RETURN

SAMPLE
SET
SOURCE
SPLIT
START
SUBROUTINE
SUM
SWITCH

TAIL
TAPE
TIME
TO

UN
UNIQUE

WITHIN
WRITE

ZERO
ZIBM

Appendix 2

TRANSLATOR SPECIFICATIONS AND OPERATING INSTRUCTIONS

PROGRAM NAMES

There are two versions of the translator, XDC1 and XDC2.

CONFIGURATION REQUIRED

XDC1 requires 12K words of core store.

XDC2 requires 20K words of core store.

The following magnetic tapes are required for both translators:

The Program Library Tape,

1 scratch tape,

1 or more output tapes, if the translated program is to be held on tape.

USE OF PERIPHERALS

The present versions of the translator will not accept source program from magnetic tape; input to the translators must be from cards or paper tape. If input is from paper tape the operator is required to set switch 1 in word 30.

Both translators are overlaid, so the Program Library Tape is retained throughout the translation. If a translator is to be used frequently, the user is recommended to transfer it to his own program tape, by means of the program #XPMU.

A scratch tape is used, and remains a scratch tape after translation is completed.

An input unit is required for reading the values of K1, K2, and K3, and this unit must be designated INPUT 1. An output unit is required for diagnostic output, and this unit must be designated OUTPUT 6. Statements must therefore appear in the FORTRAN Program Description such as

```
INPUT 1 = CR0
OUTPUT 6 = LP0
```

USE OF LIBRARY SUBROUTINES

At run time a simulation program incorporates a number of library subroutines. Some of these are special C.S.L. execution routines, which carry out various standard operations, such as set-handling and time-advance; others may be FORTRAN standard subroutines or functions. Statements must therefore be input at the end of compilation of the intermediate FORTRAN program to locate the appropriate subroutine blocks on the Program Library Tape. The statements are as follows:

```
LIBRARY
READ FROM (MT,-.SRC1)
FINISH
```

The C.S.L. execution routines have been compiled in COMPRESS INTEGER AND LOGICAL mode. Consequently the user must include in his FORTRAN Program Description the line:

```
COMPRESS INTEGER AND LOGICAL
```

OPERATING INSTRUCTIONS

1 Type message FInd#XDC1

2 If input is from paper tape, type message ON #XDC1 1

3 Type message GO #XDC1 20

When translation is complete (indicated by a HALTED message) load the cards or paper tape containing the required control statements and carry out the appropriate actions to operate whichever FORTRAN compiler is to be used.

If the translator requires a peripheral unit which is not available, translation will halt and one of the following messages will be output:

```
HALTED TR
HALTED CR
HALTED TP
HALTED CP
```

where the two-letter code, TR, CR etc., indicates the peripheral unit required. When a unit becomes available translation can be restarted by the message:

```
GO #XDC1
```

When translation is complete either the message

```
HALTED OK
```

is output, if there are no errors, or the message

```
HALTED ER
```

is output, if errors have been found in the C.S.L. program.

ERROR MESSAGES

The following error codes, output in the C.S.L. listing, indicate various types of errors in the source program.

01 Error in Program Description, or control statement

02 Invalid character in column 1

Non-numeric character in columns 2 to 5

Character in column 6 on first card of statement

Illegal character

03 Misplaced period

04 Statement exceeds allowable length or statement too long for translator

05 C.S.L. variable name has more than 22 characters

06 Overflow of table of C.S.L. variable names

07 Variable used in RANK statement that has not been declared as a set

08 Illegal use of structural word

09 Time variables used both in COMMON and locally in main program

11 GAINS must be first structural word of statement or follow HEAD or TAIL

12 HEAD, TAIL, INTO must be first structural word or follow SPLIT

13 ELSE must follow SPLIT

14 INPUT, OUTPUT, TAPE can only follow READ or WRITE

15 TO can only follow GO

16 TIME, SET can only be defined after CLASS

17 ANY, MAX, MIN, FIRST, LAST can only follow FIND

18 OR must be contained within a test chain

19 Illegal variable type has been used in declaration statement

20 Syntax error in declaration statement

21 Possibly missing END statement (next statement is ignored)

22 £ or $ sign has been omitted from Subroutine or Function name

23 Illegal variable type used as a dummy argument

30 Array defined with zero dimensions

31 Time variable used incorrectly in COMMON statement

32 Mismatched parenthesis

34 First character of name non-alphabetic

35 Illegal destination clause

36 Some other character found when , or 'nothing' expected

37 Error in format of statement

40 Wrong variable type used in expression or array used without subscript

41 Set not defined on mentioned class, or both sets not defined on the same class

42 Array with wrong dimensions used for distribution