nical Documentary
t No. ESD-TDR-64-320

MILITRAN
PROGRAMMING MANUAL

Prepared for the

OFFICE OF NAVAL RESEARCH
NAVY DEPARTMENT
WASHINGTON, D. C.

and the

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
L. G. HANSCOM FIELD, BEDFORD, MASS.

U. S. Navy Contract No. Nonr 2936(00)

by

SYSTEMS RESEARCH GROUP, INC.
1501 Franklin Avenue
Mineola, L. I., New York

JUNE 1964

When US Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

DDC AVAILABILITY NOTICE

Qualified requesters may obtain copies of this report from the Defense Documentation Center (DDC), Cameron Station, Alexandria, Va. 22314. Orders will be expedited if placed through the librarian or other person designated to request documents from DDC.

Technical Documentary
Report No. ESD-TDR-64-320

MILITRAN
PROGRAMMING MANUAL

Prepared for the

OFFICE OF NAVAL RESEARCH
NAVY DEPARTMENT
WASHINGTON, D. C.

and the

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
L. G. HANSCOM FIELD, BEDFORD, MASS.

U. S. Navy Contract No. Nonr 2936(00)

by

SYSTEMS RESEARCH GROUP, INC.
1501 Franklin Avenue
Mineola, L. I., New York

JUNE 1964

DDC AVAILABILITY NOTICE

Qualified requesters may obtain copies of this report from the Defense Documentation Center (DDC), Cameron Station, Alexandria, Va. 22314. Orders will be expedited if placed through the librarian or other person designated to request documents from DDC.

FOREWORD

This is one of three technical reports being published simultaneously. The others are the MILITRAN Operations Manual for IBM 7090-7094 (Technical Documentary Report No. ESD-TDR-64-389) and the MILITRAN Reference Manual (Technical Documentary Report No. ESD-TDR-64-390). The three reports constitute a complete description and instructions for using the MILITRAN language in computer programming of simulation problems.

The MILITRAN 7090-7094 Processor, which is used to compile a problem written in MILITRAN source language into a machine language program, will be available to prospective users. Pending final arrangements, requests for information about the MILITRAN Processor should be sent to the Office of Naval Research (Code 491).

This report was prepared by the Systems Research Group, Inc., under Contract Nonr-2936(00), which was initiated by the Naval Analysis Group, Office of Naval Research, and has been jointly supported by the Office of Naval Research and the Electronic Systems Division, Air Force Systems Command.

ABSTRACT

MILITRAN is an algorithmic computer language specifically oriented to the problems encountered in simulation programming. In addition to providing overall flexibility in expressing complex procedures, the language contains features which greatly simplify the maintainence of status lists, handling of numeric and non-numeric data, and sequencing of events in simulated time.

This report is an introduction to the MILITRAN language for prospective users.

REVIEW AND APPROVAL

This Technical Documentary Report has been reviewed by the Electronic Systems Division, U. S. Air Force Systems Command, and is approved for general distribution.

John B. Curtis

J. B. CURTIS
2nd Lt., USAF
PROJECT OFFICER

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

MILITRAN is a general purpose, problem-oriented language developed by Systems Research Group, Inc. under the sponsorship of the Office of Naval Research and the Air Force Systems Command (ESD). It enables the programmer to think in terms of the problem to be solved and the method of solution, rather than in terms of the computer which is used to solve the problem. It is a symbolic language, using familiar notations from algebra and logic which are expressed by self-explanatory English words. The class of problems towards which this language is oriented is that encountered in military simulations.

The primary stimulus for MILITRAN has been twofold: (1) the increasing importance of the simulation approach in the analysis of military systems, and (2) the long periods of time and high levels of effort required to produce an operating simulation program.

MILITRAN achieves improved coding efficiency by providing a programming language - the "MILITRAN language" - which is oriented to the special problems and procedures of military simulation. Because of this spe-

cial orientation, simulation codes can usually be written in the MILITRAN language more easily, more quickly, and with less likelihood of error than by conventional programming techniques. The ultimate communication with the computer is effected through the MILITRAN compiler, a special program that translates source codes written in the MILITRAN language into a form directly understandable by the computer.

The MILITRAN process commences once the model of the situation has been developed and communicated to the MILITRAN programmer. The programmer then writes a source code for the simulation in the MILITRAN language. This source code is fed into a computer which has been loaded with the MILITRAN compiler program. The process of compilation is then carried out, with the result that the computer itself generates an object program and accessory documentation for the particular simulation. This object program is written in machine assembly language and constitutes the actual code for the simulation. Hence, it represents an independent entity and the simulation proper may then be executed without any further reference to the MILITRAN System.

The MILITRAN language is a complete, integrated programming language providing a full repertoire of con-

trol, logical, arithmetic and I/O instructions, together with a range of special features oriented towards meeting the internal structural demands of military simulation programs. These special features consist of such items as object modes, list processing statements, event processing procedures, special retrieval arrays, etc. which have been designed to increase the power of the language while preserving full flexibility to cope with any specific simulation model.

Most programming languages are designed for the development of programs to handle various types of computational problems. Such problems have numbers or variables taking on numerical values as their basic elements. The basic relations between these elements are arithmetic relations, and the processing is largely composed of arithmetic operations. A military simulation, on the other hand, has as its basic elements the particular objects which act and are acted upon in the course of the simulation. The relationships and interactions among these objects form, in one sense, the key relations dealt with in the simulation. The fundamental process of a simulation is the step by step progress through simulated time coupled with a determination of the events that have taken place, an assessment (often stochastic) of the impact of these events upon the participating objects, and

an updating procedure to insure that future events and occurrences are consistent with these continual changes in the state of the simulated system.

The MILITRAN language has the ability to deal directly and explicitly with these underlying features of a simulation situation.

To provide some insight as to how this is achieved, the role of certain MILITRAN language components will be described briefly in this chapter. This is of course in the way of a preview, and a complete discussion of each component is deferred until the appropriate subsequent chapter.

The basic elements in a simulation program are individual objects and object types. To create such entities, MILITRAN provides the system mode declaration OBJECT. The collection of all individual objects so declared may be viewed as a universe of "object values" available to the particular source program. In the same manner that numerical variables are needed in a computational problem, so in a simulation one needs variables which can take on individual objects as values. In MILITRAN such variables are declared by the PROGRAM OBJECT statement. Multi-dimensional program object arrays are

declared in a similar manner. The requirement often exists to form various groupings of objects for purposes of effecting a common processing or data assignment. In MILITRAN, this requirement is met by the CLASS declaration.

Insofar as the above discussion bears on declarations employed in MILITRAN for the creation of various program entities, it should be pointed out that MILITRAN also provides for introducing the entities required in the normal computational aspects of a simulation program. Thus, one can introduce constants, variables, multi-dimensional arrays, multi-modal vectors and vector arrays which can take on either real (i.e., floating point), integer, or logical (i.e., Boolean) as well as object values. To make the process of mode declaration as painless and reliable as possible, a "NORMAL MODE" statement is available. This statement enables the MILITRAN programmer to establish his own automatic mode assignment conventions for each portion of the source code.

As discussed previously, a major segment of any simulation program is concerned with maintaining an up-to-date account of the status of the participating objects and their interactions. In one sense, the progress of

the simulation may be viewed as essentially equivalent to the dynamic changes in object and system status. In virtually all simulations, this role of maintaining status accounts is accomplished through the device of lists. Consequently, it is exceedingly helpful to have facile procedures for constructing lists, inserting entry values, manipulating these list entries, and retrieving any desired item of information from a list. In MILITRAN this facility is provided in the list processing portion of the language.

List entries may be created by means of PLACE or PLACE ENTRY statements; modified by REPLACE or REPLACE ENTRY statements; and located by the system functions MINIMUM INDEX and RANDOM INDEX. Processing which would normally require complex iterative coding can be achieved in single concise MILITRAN statements. For instance, the REMOVE and REPLACE statements can be used to search and process entire lists. Updated values produced through the use of REPLACE or REPLACE ENTRY statements may be functions of the values being replaced.

It should be noted that in providing a list-processing language MILITRAN <u>does not prohibit</u> the use of normal data processing statements upon list elements.

In connection with this freedom, a RESET LENGTH statement permits the programmer to override automatic updating features at will.

The central dynamic feature of military simulation is the processing of simulated events occurring either at regular intervals or at critical juncture points in time. A simulated event is characterized by its time of occurrence, the participating objects, more detailed information concerning the particular circumstances of the event, together with a certain procedure for evaluating the impact of the event's occurrence and implementing this effect. Thus, a simulated potential event consists of characterizing data plus an associated processing to be carried out when the event takes place. In MILITRAN, this association of data and contingent processing is accomplished (as well as many other benefits) with the CONTINGENT EVENT and PERMANENT EVENT statements. The CONTINGENT EVENT statement defines an event type and associates with it a list whose entries have an indicated component structure. The structure of event lists may be freely arranged. Each entry in such a list corresponds to a potential event occurrence and the first component of any such entry plays the special role of representing the critical time at which this potential event may take

place. In addition to defining this list, the CONTINGENT EVENT statement delineates, for a special purpose, that segment of program steps which is enclosed by the CONTINGENT EVENT statement and the first END statement to follow it. This program segment constitutes the processing to be carried out upon the "realization" of the particular contingent event.

Each entry on a CONTINGENT EVENT list represents a potential event. The general nature of this potential event corresponds to the particular CONTINGENT EVENT with which it is associated, while its detailed character is described by the values of the components of the list entry. In military simulations, one often encounters events characterized by time of occurrence, attacking object, target object, together with other descriptive information. The MILITRAN language takes advantage of this common form by providing the system variables TIME, ATTACKER, TARGET, and INDEX. When a particular event is "realized", the list entry associated with that event automatically has the values of its first three components loaded into TIME, ATTACKER, TARGET, respectively. Similarly, the variable INDEX automatically takes on the number giving the position of the "realized" entry in its list. (These conventions exist for conven-

ience and need to be used only as long as they are convenient for the problem at hand. They are in no way restrictive and the MILITRAN programmer, if he so wishes, can ignore them with impunity.)

Having established the correspondence between potential events and vector entries in CONTINGENT EVENT lists, the meaning of event "realization" can now be described more precisely. A simulation may be viewed as the state of a system progressing through time in a predictable fashion until the occurrence of a critical event. This critical event affects the dynamic state of the system and so determines the course of things until the next critical event takes place. The system thus proceeds predictably in the intervals between the critical events which determine the conditions of the systems predictability. In particular, it is clear that the occurrence of one potential event will affect the set of future potential events. Thus, the simulation can be carried out by finding the earliest potential event, performing the appropriate revision of the dynamic state of the system and the set of future potential events, then finding the next earliest potential event, again performing the appropriate revision, etc. In terms of the simulation program, the set of future potential events is comprised of all entries on CONTINGENT EVENT lists whose time component is greater

than or equal to the simulated current time. The earliest future potential event will then be represented by that list entry in this set which has the smallest time value. This event is the one which occurs next, i.e., it is actually "realized". The revision of the system state induced by this event is then accomplished by performing the processing associated with that particular CONTINGENT EVENT. Having processed this revision, the program then searches the revised set of entries on all CONTINGENT EVENT lists, finds that entry having the smallest time exceeding current time, and then proceeds to "realize" that list entry as an event. In this manner, the simulation program determines the "course of the battle."

There is a special MILITRAN instruction which triggers this procedure - NEXT EVENT. Whenever this command is encountered, the MILITRAN program will find the list entry to be realized, and will transfer control to the associated CONTINGENT EVENT processing to effect that realization.

Although much of the processing involved in simulation programs appears similar to that used in strictly computational codes, significant differences of a general nature occur. These differences must be considered

in designing a simulation language, and have been an integral factor in the development of MILITRAN.

Simulation data must often be referred to several arguments. The range of an aircraft, for instance, is a function of aircraft type, cruising altitude, and cruising speed as well as fuel load. To accomodate for the handling of such data, MILITRAN provides for retrieval from arrays and vectors having any desired number of arguments.

Real-world situations involve real-world names, contractions of which to five or six characters are often grotesque. MILITRAN permits identifiers of up to sixty characters.

The association of data with various objects in a simulation imposes a vast bookkeeping task upon the programmer. The MILITRAN processor absorbs this effort almost entirely by allowing a wide range of object-mode subscripting.

The iterative processes in a simulation program often involve incrementation and termination criteria which cannot be expressed in the usual algorithmic languages. The MILITRAN DO-loop form allows modification of termination and incrementation criteria and even the index

itself within the iteration. Further, exit from the loop may be made at any point without loss of current values, and indices are defined even after normal exits.

Allocation of storage at running-time permits the use of a given program to evaluate many different cases without recompiling.

The processor allows the use of mixed-mode expressions whenever contextual meaning is clear. This has several implications, among them the elimination of compilation failures due to omission of unnecessary decimal points.

MILITRAN ignores blanks (except in Hollerith fields) and permits comments at any point in a statement. Although the language cannot force the programmer to document his program, it does provide facile tools for self-documentation.

From this brief sampling of the MILITRAN language, it is hoped that one can glean some of the basic ideas underlying its structure as well as some of the techniques appropriate to its intended utilization. The remainder of this document is concerned with detailed explanation and illustration of the MILITRAN vocabulary, syntax, and grammar.

CHAPTER II

GENERAL LANGUAGE CHARACTERISTICS

A MILITRAN source program is a series of MILITRAN statements which specify a sequence of operations to be performed by a digital computer. Each MILITRAN statement may be viewed as a set of elements arranged in a prescribed order which specifies one or more of the following characteristics of the program:

1. STRUCTURE of the program or its components;
2. PROCESSING to be performed within the computer;
3. INPUT/OUTPUT, or exchange of data between the computer and its external storage devices;
4. CONTROL of the sequence in which various operations are to be performed; and
5. COMPILER instructions, or specification of the manner in which the translation from MILITRAN to machine language is to be performed.

The elements which combine to form MILITRAN statements are names, constants, punctuation marks, statement type identifiers, operators, and mnemonic delimiters. These elements are in turn made up of characters, which are the basic units of any language.

Characters

The MILITRAN Basic Language is expressed in terms of the following character set:

ABCDEFGHIJKLMNØPQRSTUVW
XYZ 0123456789.(),=+-*/

The character "blank" is normally not significant in the language. Except where specifically noted in this manual, blanks may be used in any part of a statement without any effect on the statement.

"Alphabetic characters" include the letters A through Z; "numeric characters" include the digits 0 through 9;"alphameric characters" include both alphabetic and numeric characters. All others are "special characters."

Punctuation Marks

The only punctuation marks used in MILITRAN are the following:

.	Period
(	Open Parentheses
)	Close Parentheses
,	Comma
...	Ellipsis (Delimits comments)

Operators

The operators used in MILITRAN are the following:

=	Substitution
+	Addition; plus
-	Subtraction; minus
*	Multiplication
/	Division
.P.	Exponentiation
.E.	Comparison: Equal to
.G.	Comparison: Greater than
.L.	Comparison: Less than
.NE.	Comparison: Not equal to
.GE.	Comparison: Greater than or equal to
.LE.	Comparison: Less than or equal to
.IS.	Object identity
.IN.	Object inclusion
.ØR.	Logical disjunction
.NØT.	Logical negation
.AND.	Logical conjunction

.EQV. Logical equivalence

.EXØR. Logical exclusive disjunction

Names

A name is a string of one to sixty alphameric characters, the first of which is alphabetic.

Certain names have a pre-defined meaning in MILITRAN and may be used only in reference to that meaning. These names are:

ABS	GST	PRINTER
ATAN	INDEX	RANDØM
ATTACKER	INTEGER	RANDØM INDEX
CARDS	LENGTH	REAL
CØS	LØG	SIGN
EACH	LST	SIN
END CØMPILATIØN	MAX	SQRT
EPSILØN	MIN	TAN
EXP	MINIMUM INDEX	TARGET
FALSE	MØD	TIME
FØRMAT	NEXT EVENT	TRUE

The words BY, BY ENTRY, CØNTAINS, FØR, FRØM, IN, TØ, and UNTIL are used within certain statements to define the limits of various parts of those statements. Used in this context, these words are not names, but mnemonic

delimiters. Use of these alphameric combinations as names is permitted, as the distinction between name and delimiter is always contextually clear.

All names used in a MILITRAN source program are either explicitly or implicitly assigned a type. Some types of names are assigned a mode. The type of a name indicates the nature of its use in the program. The mode of a name indicates the form of data referred to by the name.

Statement Types

The basic statement in MILITRAN involves substitution of one data item for another within the computer. The substitution statement has the form

a = b

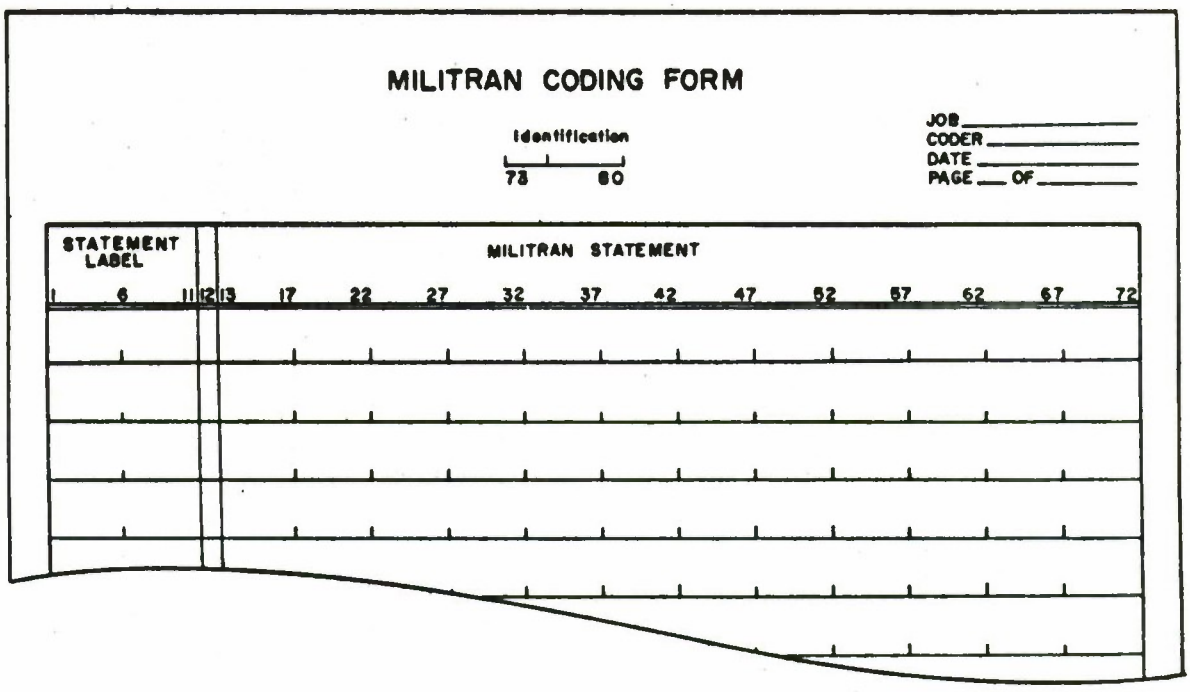
where "a" is a subscripted or unsubscripted variable name and "b" is any expression whose value is suitable for storage in "a".

All statements which are not substitution statements are designated by system words and/or symbols called statement type identifiers. The following table lists all statement types and their primary uses. The form and characteristics of each statement is described in later sections.

Statement Type	Primary Function
BACKSPACE	INPUT/OUTPUT
BACKSPACE FILE	INPUT/OUTPUT
BINARY READ	INPUT/OUTPUT
BINARY WRITE	INPUT/OUTPUT
CLASS	STRUCTURE
CØMMØN	STRUCTURE
CØNTINGENT EVENT	STRUCTURE
CØNTINUE	CONTROL
DØ	CONTROL
END	CONTROL
END CØMPILATION	COMPILER
END CØNTINGENT EVENTS	CONTROL
END FILE	INPUT/OUTPUT
END FILE RETURN	CONTROL
END RECØRD RETURN	CONTROL
EXECUTE	CONTROL
FØRMAT	INPUT/OUTPUT
GØ TØ	CONTROL
IF	CONTROL
INTEGER	STRUCTURE
LIST	STRUCTURE
LØGICAL	STRUCTURE
NEXT EVENT	CONTROL
NEXT EVENT EXCEPT	CONTROL

Statement Type	Primary Function
NØRMAL MØDE	STRUCTURE
ØBJECT	STRUCTURE
PAUSE	CONTROL
PERMANENT EVENT	STRUCTURE
PLACE	PROCESSING
PLACE ENTRY	PROCESSING
PRØCEDURE	STRUCTURE
PRØGRAM ØBJECT	STRUCTURE
READ	INPUT/OUTPUT
READWRITE	INPUT/OUTPUT
REAL	STRUCTURE
REMØVE	PROCESSING
REMØVE ENTRY	PROCESSING
REPLACE	PROCESSING
REPLACE ENTRY	PROCESSING
RESET LENGTH	PROCESSING
RETURN	CONTROL
REWIND	INPUT/OUTPUT
STØP	CONTROL
SUSPEND FAP LISTING	COMPILER
UNLESS	CONTROL
UNLØAD	INPUT/OUTPUT
VECTØR	STRUCTURE
WRITE	INPUT/OUTPUT

The standard MILITRAN coding sheet is shown below.



MILITRAN statements are written one to a line in columns 13-72. If a statement is too long for a line, it may be continued on one or more successive lines by placing a numeric character in column 12. For the initial line of a statement, column 12 must be blank.

Statement labels may be written in columns 1 through 11. A label is a group of alphameric characters, not exceeding 11 in number, the first of which is alphabetic. These labels permit the programmer to refer to statements within the program. For example, the statement GO TO AB 12 would result in a transfer of control to the MILITRAN statement AB 12.

Comments to explain the program may be included in a MILITRAN source program. These comments are not processed by MILITRAN but are printed on the listings produced when the source program is translated into the object program.

Comments may be stated in either of the following two ways.

1. If it is desired to insert a complete separate line of comments, any non-blank, non-numeric character should be written in the continuation column (column 12).

2. If it is desired to intersperse comments within a line of coding, the comments must be enclosed in two groups of 3 periods as in:

...THESE ARE COMMENTS...

If a comment occurs at the end of a statement, the terminating periods may be omitted.

NOTE: 1. Comments cannot appear in a FORMAT statement.

2. Care must be exercised to insure the inclusion of the correct number of periods when a comment either directly precedes or follows an arithmetic, relational, or logical operator which itself has periods as part of its notation.

Columns 73-80 may be used for any desired identifying information. Information contained in these columns is not utilized by the MILITRAN processor except as identification.

No MILITRAN statement may occupy more than ten cards, exclusive of cards designated as comments by column 12.

Some examples of MILITRAN statements and their effects are:

C = A * B	The asterisk indicates multiplication. Thus, the statement means multiply A by B and set C equal to the result.
D = C/A	The slash indicates division. This causes the computer to divide the value of C by A. Using the value of C obtained from the previous example D = B.

CHAPTER III

BASIC MILITRAN ENVIRONMENT

The environment of a MILITRAN program is made up of those elements of information that will be manipulated by the program. These elements are classified as either numeric or non-numeric. MILITRAN provides several modes of expression for these elements in order to facilitate the creation and processing of various types of data. This chapter will discuss the necessary conventions a programmer must follow in order to establish a program environment. The first section of the chapter will focus on elements which are purely numeric in form while the second section will deal with non-numeric elements and show how they may be used when appropriate in conjunction with numeric elements.

As data elements may be arranged in several ways for the convenience of the programmer, the concepts of constants, variables, arrays, vectors and lists will be introduced as the discussion progresses. These terms specify the manner in which data elements are organized in the computer and apply to both numeric and non-numeric elements.

1. Numeric Elements

It is possible to write two types of numbers in the MILITRAN language: integer (or fixed point) numbers and real (or floating point) numbers. An integer number is an ordinary whole number. A real number is a fraction between 0.1 and 1.0 multiplied by a power of 10.

Integer Calculations

Calculations with integer numbers are carried out with whole numbers only; no decimal remainders are retained or used in computations. For example:

Arithmetic Statement	Result of Calculation
I = 5/2	I = 2 (instead of 2.5, since the .5 is truncated)
I = 5/2 + 7/2	I = 5 (intermediate truncation causes this to be computed 2+3 rather than 12/2.)
J = 5*2	J = 10
K = - 4 + 1	K = -3

Real Calculations

Real calculations are carried out between two decimal numbers to an accuracy of 8 decimal digits. For example:

Arithmetic Statement	Result of Calculation
I = 5./2.	I = 2.5 (2.5000000)
I = 5./2. + 7./2.	I = 6.
J = 1.6 * .7	J = 1.12
K = -2.7 + 1.2	D = -1.5

Arithmetic Constants

A constant is an element of information whose predetermined value remains fixed from one execution of the program to the next. Any quantity which appears in a MILITRAN statement in the form of a number is called an arithmetic constant. Arithmetic constants may be expressed in either integer mode or real mode.

Any number written without a decimal point, using the decimal digits 0 through 9 is an integer constant. It may consist of from 1 to 11 digits but its

magnitude must not exceed $2^{35}-1$. A preceding + or - sign is optional and an unsigned integer constant is assumed to be positive.

Examples:

The following are valid integer constants:

0

+9

186

-327

On the other hand, the following are invalid integer constants:

-3.2 (contains a decimal point)

27. (contains a decimal point)

34359738368 (exceeds the magnitude permitted)

9,738,368 (commas not permitted)

Any number written with a decimal point, using the decimal digits 0 through 9,is a real constant. It may consist of from 1 to 9 significant digits. A preceding + or - sign is optional and an unsigned real constant is assumed to be positive.

An integer exponent preceded by an E may follow a real or integer constant. All constants with such an exponent are considered real. The decimal exponent may have a preceding + or - sign. An unsigned exponent is assumed to be positive. The field following the letter E must not be blank; it may be zero.

The magnitude of a real constant must be between 10^{38} and 10^{-38} or be zero.

Examples:

The following are valid real constants and are exactly equivalent.

10300.

103E2

103E+02

10.3E3

+.103E05

1030000.E-2

The following are not valid real constants:

10300E (field following E is blank)

10300 (decimal point omitted)

103E+2.0 (exponent is not an integer constant)

Arithmetic Variables

A variable is a symbolic representation of an element of information that may assume more than one value - either each time the program is executed or at different stages within the program. As with constants, an arithmetic variable may be real or integer, depending on whether the value it will assume is to be real or integer.

Since each variable is associated with a name, a discussion of the rules for naming variables is appropriate at this point. A variable name consists of from 1 to 60 nonblank alphameric characters, except the special characters, + - .)(*,/=, the first of which must be alphabetic. The rules for naming variables allow for extensive selectivity. In general it is easier to follow the flow of a program if meaningful symbols are used wherever possible. For example, to compute distance it would be possible to use the statement

```
X = Y * Z
```

but it would be more meaningful to write

```
DIST = RATE * TIME
```

Examples:

The following are valid variable names:

MAXIMUM

ALPHA

PLANE

JET

A601

Z

The following are not valid variable names:

6ABC (first character is not alphabetic)

.BOMB (period not permitted - it is a special character)

An integer variable may assume any value expressible as an integer constant in the MILITRAN processor. An integer variable is declared as follows:

INTEGER SPEED

where SPEED may assume the value of an integer constant such as 1 or 20 at one point in the program execution and 622 at another point.

A real variable may assume any value expressible as a real constant in the MILITRAN processor. A real variable

is declared as follows:

REAL SPEED

where SPEED may assume the value of a real constant such as 1.2 at one point in the program execution and 622.0 at another point.

NOTE: Both integer and real variables may also be declared by means of the NORMAL MODE statement. See section on NORMAL MODE Declaration for description.

Arithmetic Arrays

To this point, we have discussed single elements of information; however, it is often advantageous to be able to group elements to refer to the group by one name and to refer to each individual quantity in the group in term of its place in the group. A group of elements of information is known as a table or an array and each element is distinguished from one another by subscription.

For example, assume the following is an array named TYPE:

21

37

43

51

Suppose it is desired to refer to the second quantity in the group; in ordinary mathematical notation this would be $TYPE_2$. In MILITRAN this would be :

TYPE(2)

The quantity "2" is called a subscript. Thus:

TYPE(2) has the value 37

TYPE(3) has the value 43

Similarly, ordinary mathematical notation might use TYPE to represent any element of the set TYPE. In MILITRAN, this might be written as TYPE(I) where I equals 1,2,3, or 4.

The array could be two dimensional; for example, the array QUAN:

	Column 1	Column 2	Column 3
Row 1	16	6	34
Row 2	22	15	46
Row 3	13	27	51
Row 4	64	96	88

Suppose it is desired to refer to the number in Row 2, column 3; this would be:

QUAN(2,3)

"2" and "3" are subscripts. Thus:

QUAN(4,2) has the value 96

Similarly, ordinary mathematical notations might use $QUAN_{1,J}$ to represent any element of the set QUAN. In MILITRAN this might be written as QUAN(I,J) where I equals 1,2,3, or 4 and J equals 1,2, or 3.

MILITRAN allows a great deal of latitude in the designation of arithmetic subscripts. The only form a subscript may not have is that of a logical variable or logical constant. (See section on Logical Elements.) Some examples of subscripts are:

DIST

21

2.4

TIME

A + B

A/B

C*D

The names shown above may be real or integer variables and variables in a subscript may themselves be subscripted. However, if a subscript of real mode is used it will be truncated to an integer value.

An arithmetic subscripted variable is an integer or real variable, followed by parentheses which enclose the subscripts. These subscripts are separated from each other by commas and a comma is not allowed after the last subscript. Arithmetic subscripted variables may have any number of subscripts.

Examples:

```
FIELD(2)
DAMAGE ACTION(5, SHIPTYPE)
MAX(J,K,L)
RANGE(TYPE,FUEL,ALT,SPEED,PAYLOAD)
```

The arithmetic array declaration statement declares both the mode and dimensions of the array. This statement does not generate any instructions in the object program; rather, it provides the MILITRAN processor with the information necessary to reserve locations in the computer memory for storage of the various elements of the array and also indicates if these elements will be REAL or INTEGER variables.

The following statements declare both the mode and dimensions of arrays:

REAL $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

INTEGER $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

where $n_1,n_2,\ldots,n_m$ are names and $i_1,i_2,\ldots,i_j$ are arithmetic dimensions whose value is not less than 1. There is no limit to the number of dimensions permitted in an array.

A single array declaration statement may specify any number of arrays; but if more than one array is named, the name of each subsequent array must be preceded by a comma. Since the array declaration statement lists the maximum dimensions of arrays, references to these arrays at running time must never exceed the specified dimensions.

When dimensions are the names of arithmetic variables, the MILITRAN compiler will provide instructions for entering the numeric values of the dimensions at the time of initial loading of the program. In this way, a program may perform calculations on arrays whose size is not determined until the program is executed. Although these "symbolic dimensions" may be changed during the running of the program, storage allotment will be determined by the values originally loaded.

NOTE:

1. Symbolic dimensions may not be subscripted.

2. If "symbolic dimensions" are used in procedures, they must be assigned to the COMMON area. (See section on COMMON Statement .)

3. Symbolic dimensions may be declared only in the array declaration statement. They must not be declared as variables in REAL or INTEGER declarations.

MILITRAN arrays are stored sequentially in decreasing memory addresses, the first subscript varying the most rapidly.

Examples:

```
REAL FIELD(4,4,5)
```

This example is a three dimensional array named FIELD, for which the subscripts never exceed 4,4, and 5 and for which 80 storage locations will be reserved.

```
INTEGER SPEED(PLANE, ALT, WEIGHT, DAMAGE)
```

The above is a four dimensional array named SPEED. The size of the array will be determined at program execution time when the values for the symbolic dimensions - PLANE, ALT, WEIGHT, DAMAGE - are entered.

2. Non-numeric Elements

This section will discuss the non-numeric elements which may also be part of the environment of a MILITRAN program.

Hollerith Constants

As defined previously, a constant is an element of information which does not change from one execution of the program to the next. The Hollerith constant provides a means of representing alphameric information in the form of a constant. It is written using the form "nH", where n is an unsigned decimal integer whose value is less than 7, followed by n Hollerith characters. The Hollerith characters are:

A through Z	space
0 through 9	$
+	*
-	,
.	/
)	=
(	'

Any characters permitted by the computer configuration may be used, even though not part of the character set required by MILITRAN.

Examples:

The following are valid Hollerith constants:

```
4HTIME
6HL = 2.
```

Note that blanks are considered alphameric characters and must be included as part of the count.

The following are invalid Hollerith constants:

```
11H SIMULATION (exceeds the number of
                permissible characters)
4H SHIP (number of characters specified
         by the integer does not corre-
         spond to the number of charac-
         ters in the constant)
-5H TIME (sign is not permitted)
```

Logical Elements

Logical constants, logical variables and logical arrays are elements which have or will assume a value of TRUE or FALSE.

A value which appears in a MILITRAN statement in the form of TRUE or FALSE is called a logical constant.

A logical variable is specified by the declaration LOGICAL which precedes the variable name. For example,

LOGICAL LIVE

declares LIVE to be a logical variable but it does not assume a value until it is assigned the value TRUE or FALSE as in:

LIVE = TRUE

or

LIVE = FALSE

NOTE: As mentioned above, logical constants and logical variables may not be used as subscripts.

Logical elements may also be grouped to form an array. The array declaration statement

LOGICAL $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

declares $n_1,n_2,\ldots,n_m$ to be names of arrays, whose elements will assume a logical value, and $i_1,i_2,\ldots,i_j$ are arithmetic dimensions whose value is not less than 1. For example:

LOGICAL CHARACTERISTICS(4,3)

defines a two-dimensional logical array called CHARACTERISTICS. The elements in this array may assume a value of TRUE or FALSE.

Object Elements

The basic elements in a simulation program are individual objects, object types, and object classes. MILITRAN provides a means of specifying this type of non-numeric data by the use of symbolic names. The declarations PROGRAM OBJECT, OBJECT, and CLASS enable the programmer to handle these non-numeric elements either as single elements or as arrays.

The rules governing the naming of these object identifiers are similar to those enumerated above for arithmetic elements. A name may consist of from 1 to 60 alphameric characters (except special characters), the first of which must be alphabetic. The name is preceded by a declaration of OBJECT, PROGRAM OBJECT, or CLASS. As with arithmetic variables, the following discussion will assume that object elements may only have arithmetic subscripts. However, we will subsequently show how individual objects themselves may be used to subscript REAL, INTEGER, LOGICAL, and PROGRAM OBJECT elements.

OBJECT is a declaration used to define names as those of objects. The general form of this declaration is:

OBJECT $n_1(i_1),n_2(i_2),\ldots,n_m(i_m)$

where $n_1,n_2,\ldots,n_m$ are names and $i_1,i_2,\ldots,i_m$ must be arithmetic expressions whose value is not less than 1. Each name must have a dimension because the dimension specifies the cardinality of the object element, i.e., how many objects are in the group. If the dimensions are symbolic, the values would be supplied at load time.

Examples:

OBJECT PLANE (10)

This declaration generates one element named PLANE and also specifies that PLANE is a set or group of 10 planes. The programmer could subsequently refer to any member of the group of 10 planes - i.e. PLANE(3) PLANE(6) etc., because use of the declaration OBJECT indicates to the MILITRAN processor that PLANE(3) and PLANE(6) are members of the group named PLANE. The subscripted name PLANE(5) is an identifier which specifies the fifth PLANE. A reference to any member of the group named PLANE causes the processor to produce the coding required to generate and save the name of the particular

member of the set.

```
OBJECT CARBINE(400), RIFLE(600)
```

In this example, the OBJECT declaration generates two elements - the first is named CARBINE and has a cardinality of 400, the second is named RIFLE and has a cardinality of 600. As we can see, the declaration OBJECT is especially useful because it enables the programmer to specify a group of objects without reserving a location in the computer memory for each object within the group. In other words, the above example specifies a total of 1000 objects - yet these objects occupy only a few words of computer memory.

```
OBJECT PLANE(NUMBER)
```

This is an OBJECT declaration which has a symbolic dimension. Therefore, the processor would provide the necessary instructions to load the value of NUMBER which indicates the cardinality of the object PLANE at program execution time.

PROGRAM OBJECT is a declaration which may be used to specify a single variable or group of variables which will assume the value of names. It is declared as follows:

PROGRAM OBJECT n_1,n_2,n_3

where n_1,n_2,n_3 are names. It also may be used to declare the mode and dimensions of an array or group of arrays which will assume name values if the declaration is stated as follows:

PROGRAM OBJECT $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_k)$

where $n_1,n_2,\ldots,n_m$ are names and $i_1,i_2,\ldots,i_k$ are arithmetic expressions.

Examples:

PROGRAM OBJECT COMBAT WEAPON

This example specifies one variable which may assume the value of any object identifier in the object program.

The statement

COMBAT WEAPON = PLANE(5)

would assign to COMBAT WEAPON the value of the identifier PLANE(5).

PROGRAM OBJECT TANK(10)

This declaration means that TANK is a one dimensional array for which the value of the subscript never exceeds 10. This declaration therefore causes 10 storage locations to be reserved for the elements of the array named TANK. These

elements will be object identifiers.

PROGRAM OBJECT OFFENSIVE WEAPON(5,4),DEFENSIVE WEAPON(6,5,4)

Two arrays are declared above; the first is a two dimensional array named OFFENSIVE WEAPON for which the subscripts never exceed 5 and 4 and for which 20 storage locations will be reserved, the second is a three dimensional array called DEFENSIVE WEAPON for which the subscripts never exceed 6,5 and 4 and for which 120 storage locations will be set aside. The elements in both arrays will be object identifiers.

NOTE: PROGRAM OBJECT differs from OBJECT in that it merely reserves space in the computer memory for subsequent storage of a variable or for the elements of an array; OBJECT reserves space for storage of the name and its dimension.

The CLASS declaration enables the programmer to specify that certain properties of designated objects or the objects themselves have common characteristics, and these common characteristics form a set or group.

CLASS is declared as follows:

CLASS(C) CONTAINS $a_1,a_2,\ldots,a_m$

where C is a name and $a_1,a_2,\ldots,a_m$ are either names of objects or classes, or names of objects or classes preceded by EACH*. The names $a_1,a_2,\ldots,a_m$ must be declared <u>before</u> they are used to form a class. The implications of the presence or absence of EACH* are discussed separately below.

The following declaration specifies the object elements which will be used to form classes:

OBJECT FORD(10), MERC(5), LINC(3), CHEV(10), OLDS(5),

CADDY(3), PLY(10), DODGE(5), CHRYS(3)

The following declarations show how classes may be formed from previously defined objects or classes, which are <u>not</u> preceded by EACH.

CLASS(ECONOMY) CONTAINS FORD, CHEV, PLY
CLASS(MODERATE) CONTAINS MERC, OLDS, DODGE
CLASS(PRESTIGE) CONTAINS LINC, CADDY, CHRYS
CLASS(PRICE) CONTAINS ECONOMY, MODERATE, PRESTIGE

All the object elements within the class and the class name itself may be referenced. The following table

indicates the subscripts which may be used with the object identifiers comprising the above class declarations.

CLASS NAME	REPRESENTATIVE	MEMBERS OF CLASS
ECONOMY(1)	FORD(1)	FORD(1,2,...,10)
ECONOMY(2)	CHEV(1)	CHEV(1,2,...,10)
ECONOMY(3)	PLY(1)	PLY(1,2,...,10)
MODERATE(1)	MERC(1)	MERC(1,2,...,5)
MODERATE(2)	OLDS(1)	OLDS(1,2,...,5)
MODERATE(3)	DODGE(1)	DODGE(1,2,...,5)
PRESTIGE(1)	LINC(1)	LINC(1,2,3)
PRESTIGE(2)	CADDY(1)	CADDY(1,2,3)
PRESTIGE(3)	CHRYS(1)	CHRYS(1,2,3)
PRICE(1)	FORD(1)	FORD(1,2,...,10); CHEV(1,2,...,10); PLY(1,2,...,10)
PRICE(2)	MERC(1)	MERC(1,2,...,5); OLDS(1,2,...,5); DODGE(1,2,...,5)
PRICE(3)	LINC(1)	LINC(1,2,3); CADDY(1,2,3); CHRYS(1,2,3)

All Fords, Chevs, Plys are members of the class ECONOMY. This class has 30 members because the OBJECT declaration specified FORD(10), CHEV(10), PLY(10). Therefore, the maximum subscript which may be used with each of the object elements specified as members of a class is equal to the dimension specified in the OBJECT declaration. Class MODERATE has 15 members; class PRESTIGE has nine members.

Since class PRICE is composed of ECONOMY, MODERATE, and PRESTIGE all the cars which are members of these three classes are members of the class PRICE. Therefore, class PRICE has 54 members.

The class name may also be subscripted. The maximum subscript for a class name is determined by the format of the CLASS declaration. In the above example, the members of the class were not preceded by EACH*. This signifies to the MILITRAN compiler that the separate members of the class are identical in some sense to the other members which have the same name...i.e.: FORD(1) is identical to FORD(9); DODGE(2) is identical to DODGE(4). Because they are identical, a single representative from each name may be used to represent the entire group. MILITRAN selects the first one. Therefore, the maximum subscript which may be used with a class name is a number equal to the number of representatives in the class. Since class Economy has 3 representatives -

FORD(1), CHEV(1), and PLY(1) the maximum subscript that may be used with ECONOMY is 3. For example:

ECONOMY(1) is equivalent to FORD(1) which in turn also represents FORD(2,3,...,10).

ECONOMY(2) is equivalent to CHEV(1) which in turn also represents CHEV(2,3,...,10).

ECONOMY(3) is equivalent to PLY(1) which in turn also represents PLY(2,3,...,10).

The subscripts and the meaning of the subscripted class name for the classes MODERATE and PRESTIGE are determined in the same manner as for the ECONOMY.

The declaration for class PRICE stated that it contains ECONOMY, MODERATE, and PRESTIGE. This declaration means that all members of the CLASS ECONOMY are identical in some sense and the same is true for all members of the CLASS MODERATE and all members of the CLASS PRESTIGE. MILITRAN selects 3 representatives for this class. For example:

PRICE(1) is equivalent to FORD(1) which in turn represents FORD(2,3,...,10), CHEV(1,2,...,10), PLY(1,2,...,10).

PRICE(2) is equivalent to MERC(1) which in turn represents MERC(2,3,...,5), OLDS(1,2,...,5), PLY(1,2,...,5).

PRICE(3) is equivalent to LINC(1) which in turn represents LINC(2,3), CADDY(1,2,3), CHRYS(1,2,3).

The name of a member of a class may be preceded by the functional modifier EACH*. It is used in a CLASS declaration to specify that the members of a set of object elements are not identical and therefore cannot be represented by a single member of the group.

Examples:

```
OBJECT FORD(10), MERC(5), LINC(3), CHEV(10), OLDS(5),
       CADDY(3), PLY(10), DODGE(5), CHRYS(3)

CLASS(ECONOMY) CONTAINS EACH*FORD,EACH*CHEV,EACH*PLY
CLASS(MODERATE) CONTAINS EACH*MERC,EACH*OLDS,EACH*DODGE
CLASS(PRESTIGE) CONTAINS EACH*LINC,EACH*CADDY,EACH*CHRYS
CLASS(PRICE)CONTAINS EACH*ECONOMY,EACH*MODERATE,EACH*PRESTIGE
```

The following table indicates the subscripts which may be used with the object identifiers in the above class declarations.

CLASS NAME	REPRESENTATIVE	MEMBERS OF CLASS
ECONOMY(1)	FORD(1)	FORD(1,2,...,10); CHEV(1,2,...,10);
ECONOMY(2)	FORD(2)	PLY(1,2,...,10)
⋮	⋮	
ECONOMY(10)	FORD(10)	
ECONOMY(11)	CHEV(1)	
ECONOMY(12)	CHEV(2)	
⋮	⋮	
ECONOMY(20)	CHEV(10)	
ECONOMY(21)	PLY(1)	
⋮	⋮	
EXONOMY(30)	PLY(10)	
MODERATE(1)	MERC(1)	MERC(1,2,...,5); OLDS(1,2,...,5);
⋮	⋮	
MODERATE(5)	MERC(5)	DODGE(1,2,...,5)
MODERATE(6)	OLDS(1)	
⋮	⋮	
MODERATE(11)	DODGE(1)	
⋮	⋮	
MODERATE(15)	DODGE(5)	
PRESTIGE(1)	LINC(1)	LINC(1,2,3); CADDY(1,2,3);
⋮	⋮	
PRESTIGE(3)	LINC(3)	CHRYS(1,2,3)
PRESTIGE(4)	CADDY(1)	
⋮	⋮	
PRESTIGE(7)	CHRYS(1)	
⋮	⋮	
PRESTIGE(9)	CHRYS(3)	

CLASS NAME	REPRESENTATIVE	MEMBERS OF CLASS
PRICE(1)	FORD(1)	FORD(1,2,...,10); CHEV(1,2,...,10);
PRICE(2)	FORD(2)	PLY(1,2,...,10)
⋮	⋮	MERC(1,2,...,5); OLDS(1,2,...,5);
PRICE(11)	CHEV(1)	DODGE(1,2,...,5)
⋮	⋮	LINC(1,2,3); CADDY(1,2,3);
PRICE(20)	CHEV(10)	CHRYS(1,2,3)
PRICE(21)	PLY(1)	
⋮	⋮	
PRICE(30)	PLY(10)	
PRICE(31)	MERC(1)	
⋮	⋮	
PRICE(35)	MERC(5)	
PRICE(36)	OLDS(1)	
⋮	⋮	
PRICE(40)	OLDS(5)	
PRICE(41)	DODGE(1)	
⋮	⋮	
PRICE(45)	DODGE(5)	
PRICE(46)	LINC(1)	
⋮	⋮	
PRICE(49)	CADDY(1)	
⋮	⋮	
PRICE(52)	CHRYS(1)	
⋮	⋮	
PRICE(54)	CHRYS(3)	

The rules governing the subscripting of the members of the class are the same as stated for the previous example.

However, since the members of the class are preceded by EACH* this signifies to the compiler that the separate members of the class are not identical to other members which have the same name...i.e., FORD(1) is not identical to FORD(6); CADDY(1) is not identical to CADDY(3). Since they are not identical, a single representative from each name cannot represent the entire group. Therefore, when the name of class members are preceded by EACH*, the maximum subscript that may be used with the class name is a number equal to the sum of the maximum subscripts of the members of the class. For example: the maximum subscript for the class ECONOMY is 30 because the maximum subscript for FORD is 10, for CHEV 10, for PLY 10.

The subscripts and the meaning of the subscripted class name for the classes MODERATE, PRESTIGE and PRICE are determined in the same manner as for the class ECONOMY.

The next two examples show how a class may be composed of object elements, which may or may not be preceded by EACH*.

```
OBJECT FORD(10), MERC(5), LINC(3), CHEV(10), OLDS(5),
       CADDY(3), PLY(10), DODGE(5), CHRYS(3)

CLASS(CAR) CONTAINS EACH*FORD, MERC, LINC, EACH*CHEV
                    OLDS, CADDY, EACH*PLY, DODGE, CHRYS
```

CLASS NAME	REPRESENTATIVE	MEMBERS OF CLASS
CAR(1)	FORD(1)	FORD(1,2,...,10);MERC(1,2,...5); LINC(1,2,3)
CAR(2)	FORD(2)	CHEV(1,2,...,10);OLDS(1,2,...,5); CADDY(1,2,3)
⋮	⋮	PLY(1,2,...,10);DODGE(1,2,...,5); CHRYS(1,2,3)
CAR(10)	FORD(10)	
CAR(11)	MERC(1)	
CAR(12)	LINC(1)	
CAR(13)	CHEV(1)	
CAR(14)	CHEV(2)	
⋮	⋮	
CAR(22)	CHEV(10)	
CAR(23)	OLDS(1)	
CAR(24)	CADDY(1)	
CAR(25)	PLY(1)	
CAR(26)	PLY(2)	
⋮	⋮	
CAR(34)	PLY(10)	
CAR(35)	DODGE(1)	
CAR(36)	CHRYS(1)	

```
OBJECT MERC(5), CADDY(3), DODGE(5), CHRYS(3)
CLASS(MODERATE) CONTAINS MERC, DODGE
CLASS(PRESTIGE) CONTAINS EACH*CADDY, EACH*CHRYS
CLASS(CAR) CONTAINS EACH*MODERATE, PRESTIGE
```

CLASS NAME	REPRESENTATIVE	MEMBERS OF CLASS
MODERATE(1)	MERC(1)	MERC(1,2,...,5);DODGE(1,2,...,5)
MODERATE(2)	DODGE(1)	
PRESTIGE(1)	CADDY(1)	CADDY(1,2,3);CHRYS(1,2,3)
PRESTIGE(2)	CADDY(2)	
PRESTIGE(3)	CADDY(3)	
PRESTIGE(4)	CHRYS(1)	
PRESTIGE(5)	CHRYS(2)	
PRESTIGE(6)	CHRYS(3)	
CAR(1)	MERC(1)	MERC(1,2,...,5);DODGE(1,2,...,5)
CAR(2)	DODGE(1)	CADDY(1,2,3);CHRYS(1,2,3)
CAR(3)	CADDY(1)	

3. NORMAL MODE Declaration

In the discussion of the different elements which may comprise a program environment, we have seen that these elements may be described as having modes: real, integer, logical or program object. If the element is a variable, the mode may be specified in a REAL, INTEGER, LOGICAL or PROGRAM OBJECT statement. Another method of declaring the mode of variables is through the use of the NORMAL MODE declaration. This declaration allows the programmer to specify a convention by which names not explicitly declared may be assigned modes. Although the NORMAL MODE declaration is used to set the mode of variables instead of the regular declarations of REAL, INTEGER etc., a declaration of NORMAL MODE does not override a specific declaration of mode.

NORMAL MODE is declared as follows:

NORMAL MODE $m_1(a_1,a_2,\ldots,a_k),m_2(b_1,b_2,\ldots,b_r)$

where m_1 and m_2 are mode names (REAL, INTEGER etc.) and $a_1,a_2,\ldots,a_k$ and $b_1,b_2,\ldots,b_r$ are single alphabetic characters. The statement causes all names beginning with the letters $a_1,a_2,\ldots,a_k$ to be assigned the mode specified by m_1 and all names beginning with the letters $b_1,b_2,\ldots,b_r$ to be assigned

the mode specified by m_2. Any letters which have not been specifically associated with a mode name will be assigned to REAL MODE.

Examples:

```
NORMAL MODE REAL(A,B,C,D,E,F,G,H,I,J,K),
            INTEGER(L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z)
```

Names of variables beginning with the letters A through K are assigned REAL MODE: names beginning with L through Z are assigned INTEGER MODE.

```
NORMAL MODE REAL(A,B,C,D,E), INTEGER(F,G,H,I,J)
            LOGICAL(L,M,N,O,P),PROGRAM OBJECT(Q,R,S,T,U,V,W,X,Y,Z)
```

Names beginning with A through E are assigned REAL MODE: names beginning with F through J are assigned INTEGER MODE; names beginning with L through P are assigned LOGICAL MODE, and those beginning with Q through Z are assigned PROGRAM OBJECT MODE.

```
NORMAL MODE INTEGER(A,B,C,D), LOGICAL(E,F,G,H)
```

Names beginning with A through D are assigned INTEGER MODE and those beginning with E through H are

assigned LOGICAL MODE. Since I through Z are not specifically associated with a mode, all names beginning with these letters would be assigned to REAL MODE.

If the NORMAL MODE statement is written as:

NORMAL MODE $m_1(a_1,a_2,\ldots,a_k),m_2(b_1,b_2,\ldots,b_r),m_3$

all the names beginning with letters not specifically associated with m_1, or m_2 would be assigned to the mode specified by m_3.

Examples:

NORMAL MODE REAL(A,B,C), INTEGER(D,E,F,G), PROGRAM OBJECT

Names beginning with A, B, and C are assigned REAL MODE and those beginning with D through G are assigned INTEGER MODE. Since PROGRAM OBJECT does not have any letters specifically associated with it, all unspecified letters (H through Z) are assigned PROGRAM OBJECT MODE.

If the statement is written as:

NORMAL MODE INTEGER

or

NORMAL MODE PROGRAM OBJECT

every letter would be assigned to the mode specified in the NORMAL MODE statement.

Each NORMAL MODE statement sets the mode for names beginning with the specified letters until overriden by a subsequent NORMAL MODE statement. If an unspecified variable is declared (one which is not preceded by a mode description) before a NORMAL MODE statement is given, the MILITRAN compiler will assign the mode REAL to the unspecified element.

4. Vectors

The MILITRAN program environment may also be comprised of elements of information in the form of vectors, a vector being a group of arrays.

The vector declaration is written as:

VECTOR $N((a_1,a_2,\ldots,a_i), d_1,d_2,\ldots,d_i)$

where N is the name of a vector; $a_1,a_2,\ldots,a_i$ are the vector components, each of which is a name and all of which are associated with name "N"; and $d_1,d_2,\ldots,d_i$ are the dimensions associated with each component.

Retrieval of any member of the vector may be accomplished in the following manner:

1. by subscripting the name of the vector component (array), in which case the number of subscripts is equal to the number of arguments in the dimension.

2. by subscripting the name of the vector, in which case the number of subscripts is equal to the number of arguments in the dimension + 1. The terminating subscript denotes the vector component. For example:

```
VECTOR Q((QR,QS,QT),4,5)
```

defines a vector "Q" comprised of three components, QR, QS, QT, each of which is a two dimensional array for which 20 storage locations will be reserved, resulting in a total storage allocation of 60 for the vector. The assignment of storage is such that all of QR is followed by all of QS, which is followed by all of QT.

Suppose it is desired to retrieve the twentieth element in the vector. If the name of the vector component is subscripted, it would be written as:

```
QR(4,5)
```

If the name of the vector itself is subscripted, it would be written as:

```
Q(4,5,1)
```

(The third subscript "1" indicates the first component "QR".) Similarly, Q(4,2,3) refers to the same element as QT(4,2).

Modes of Vectors

Both the vector name and the component names always have a mode but there is no requirement that these modes be the same or that they be compatible. The mode or modes of a vector and its components may be declared in REAL, INTEGER, PROGRAM OBJECT and LOGICAL declarations, or they may be assigned the NORMAL MODE. However, if the mode of the vector is not compatible with the mode or modes of the components, the compiler will always assume that the data involved is already as of the same mode of the name that is used. The following rules determine the mode that will be assigned to a vector and its component arrays.

1. If both the name of the vector and its components are each specifically declared in REAL, INTEGER, PROGRAM OBJECT, or LOGICAL declarations, the vector and the components of the vector are each assigned the mode specified by the declaration.

Examples:

```
REAL A
INTEGER B,C
PROGRAM OBJECT D
LOGICAL E
VECTOR A((B,C,D,E),2,3)
```

```
INTEGER A
REAL B,C,D,E
VECTOR A(B,C,D,E),2,3)
```

```
REAL A,B
INTEGER C,D
PROGRAM OBJECT E
VECTOR A((B,C,D,E),2,3)
```

In the above examples, the modes of the vector "A" and its components "B,C,D,E" are those so designated by the mode declarations.

2. If the name of the vector is <u>not</u> specifically declared but the component names are each specifically declared in REAL, INTEGER, PROGRAM OBJECT, or LOGICAL declarations, the vector name is assigned the NORMAL MODE in effect at the appearance of the VECTOR statement, and the component names are assigned the mode specifically declared for each component.

Example:

```
INTEGER B,C
PROGRAM OBJECT D
LOGICAL E
VECTOR A((B,C,D,E),2,3)
```

The mode of vector "A" is the NORMAL MODE in effect at the time the vector statement is encountered by the compiler. The mode could be REAL, INTEGER, PROGRAM OBJECT, or LOGICAL.

3. If the name of the vector is specifically declared and the names of the components are not specifically declared, the components are assigned the same mode as declared for the vector name itself.

Example:

```
REAL A
VECTOR A((B,C,D,E),2,3)
```

Components "B,C,D,E" are assigned the REAL mode

```
INTEGER A
PROGRAM OBJECT D,E
VECTOR A((B,C,D,E),2,3)
```

Components B,C are assigned INTEGER mode. D,E are assigned PROGRAM OBJECT mode.

4. If both the name of the vector and the names of its components are not specifically declared, the vector and its components are assigned the NORMAL mode in effect at the appearance of the vector statement.

Example:

VECTOR A((B,C,D,E),2,3)

"A" and "B,C,D,E" are all assigned the NORMAL mode.

NOTE: In all of the above examples, the mode of the vector and its components could have been declared after the appearance of the VECTOR statement. A specific declaration anywhere in the program overrides NORMAL MODE.

5. COMMON Statement

The COMMON statement provides the programmer with the option of controlling the assignment of the locations which will be occupied by variable data. The form of the COMMON statement is:

COMMON $n_1,n_2,\ldots,n_1$

where each n is the name of a variable, nonsubscripted array name, nonsubscripted vector name, or nonsubscripted list name.

Variable names which appear in a COMMON statement are assigned to a separate portion of memory, enabling a program and its subprograms to share storage locations. For example, during execution of a MILITRAN program, while different variable data may be required at different times by

separately compiled portions of the program, it may not be necessary for all such data to occupy distinct storage locations. The COMMON statement enables such variable data to share storage locations, resulting in a large saving of storage space.

The locations assigned to the variable names appearing in COMMON statements are assigned in the sequence in which the names appear in the statements, starting with the first COMMON statement of the program.

Use of the COMMON statement also permits the data stored in the COMMON area to be accessed by programs which have been compiled separately. In this way, arguments which are required for functions or subroutines may be transmitted from one program to another. This may be accomplished by having the corresponding variables occupy the same location in the COMMON area, which in turn is accomplished by having them occupy corresponding positions in the COMMON statements of the two programs.

For example, if a program has the following COMMON statements:

```
COMMON A,B,C,D
COMMON SPEED, DIST, RATE
```

the variables will appear in the COMMON area in the following sequence:

A

B

C

D

SPEED

DIST

RATE

If another program required only the use of variables B,C, SPEED, and RATE, dummy variables could be named in the COMMON statements of the second program in order to force reservation of the necessary locations to cause the same locations to be assigned to the corresponding variables.

NOTE: If an array, vector, or list is assigned to the COMMON area, the name as it appears in the COMMON statement is nonsubscripted. The amount of space to be reserved is determined by the dimension that has been stated in the array, vector, or list declaration.

6. Object Elements used as Dimensions and Subscripts

The discussion of the use of nonnumeric elements as dimensions and subscripts has been deferred in order to permit a sequential explanation of the various types of declarations. However, each of the array and vector declarations

may be dimensioned or subscripted by an object element.

Object Elements used to Specify Dimensions

In order to specify the dimension of an array or vector by using an object element, the object must be declared in:

1. An OBJECT declaration, or
2. A CLASS declaration

The value of the dimension will be the cardinality of the object.

For example, if object elements are declared as follows:

```
OBJECT FORD(10), MERC(5), LINC(3)
```

the dimensions of an array could be specified as:

```
REAL SPEED (FORD, MERC, LINC)
```

SPEED is a three dimensional array for which 150 locations will be reserved because the value of each dimension is specified by the cardinality of the designated object.

The next three examples use a CLASS name to specify a dimension.

```
OBJECT FORD(10), MERC(5), LINC(3), CHEV(10), OLDS(5),
       CADDY(3), PLY(10), DODGE(5), CHRYS(3)

CLASS ECONOMY CONTAINS FORD, CHEV, PLY

CLASS MODERATE CONTAINS EACH*MERC, EACH*OLDS, EACH*DODGE

CLASS PRESTIGE CONTAINS EACH*LINC, EACH*CADDY, CHRYS

REAL SPEED (ECONOMY)
```

Speed is now a one dimensional array for which 3 locations will be reserved.

```
REAL SPEED (MODERATE)
```

Speed is a one dimensional array for which 15 locations will be reserved.

```
REAL SPEED (PRESTIGE)
```

Speed is a one dimensional array for which 7 locations will be reserved.

Object Elements used as Subscripts

Subscripted OBJECT names, subscripted CLASS names, and PROGRAM OBJECT names which may or may not be subscripted may be used to subscript REAL, INTEGER, LOGICAL, and PROGRAM OBJECT elements. For example, if certain elements are declared as follows:

```
OBJECT FORD(10), MERC(5), LINC(3), CHEV(10), OLDS(5),
       CADDY(3), PLY(10), DODGE(5), CHRYS(3)

CLASS ECONOMY CONTAINS FORD, CHEV, PLY

CLASS MODERATE CONTAINS EACH*MERC, EACH*OLDS, EACH*DODGE

CLASS PRESTIGE CONTAINS EACH*LINC, EACH*CADDY, CHRYS

PROGRAM OBJECT THIS CAR, POOL CAR(3)

CLASS CAR CONTAINS EACH*ECONOMY, EACH*MODERATE, EACH*PRESTIGE

REAL COST (CAR)

INTEGER AGE (CAR)
```

the following are some of the object elements which might be used as subscripts:

COST (FORD(3))	which is COST (1)
COST (MERC(4))	which is COST (7)
COST (CAR(10))	which is COST (10)
COST (THIS CAR)	whose value depends upon that of "THIS CAR"
COST (POOL CAR(2))	whose value depends upon that of POOL CAR(2)

CHAPTER IV

PROGRAMMING IN MILITRAN:

ARITHMETIC AND LOGICAL PROCESSING

The preceding chapters discussed the elements of information which comprise a program environment —— the data which are referenced by the program in order to solve a problem and the instructions to the compiler which provide information about the source program. This chapter will explain some of the statements which cause calculations to occur and decisions to be made.

1. Expressions

A MILITRAN expression is a sequence of constants, subscripted and non-subscripted variables separated by operation symbols, commas, and parentheses. The MILITRAN language contains two kinds of expressions: arithmetic and logical.

The simplest form of an arithmetic expression is a single quantity. This quantity may be an arithmetic constant or an arithmetic variable - subscripted or non-subscripted.

Examples:

3

7.4

MAX

TIME(2)

Compound arithmetic expressions may be formed by combining simple arithmetic expressions through the use of arithmetic operators. These operators are:

ABS(x)	absolute value of x
.P.	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

If A and B are any arithmetic expression, then these operators are defined in the following manner:

ABS(A)	means the positive magnitude of A
A.P.B	means the value of A raised to the power B.(A^B)
A*B	means the value of A multiplied by the value of B
A/B	means the value of A divided by the value of B
A+B	means the value of A plus the value of B
A-B	means the value of A minus the value of B

For Example:

```
ABS(MINIMUM)
3.6 .P.I
RATE + TIME
DIST(I)/TIME(J)

ALPHA + BETA
ALPHA - BETA
```

In expressions with three or more variables, some means for describing the exact order in which the operations are to be performed is necessary. For example, the expression:

```
X + Y * Z
```

may be computed in two ways. One way is to add X and Y and multiply the sum by Z. The other is to multiply Y by Z and add X to the product. Therefore in order to clarify the order of operations, MILITRAN prescribes a firm set of rules. The order of precedence is as follows:

```
ABS(x)
.P.
/
*
-
+
```

In the example, X + Y * Z, MILITRAN would multiply Y by Z and add X to the product.

The following examples illustrate the ordering of expressions:

2.P.3-4	gives 4	exponentiation is performed before addition or subtraction.
2.P.3/2	gives 4	exponentiation is performed before multiplication or addition.
3 + 7 * 2	gives 17	multiplication is performed before addition or subtraction.

In the next example, if A has assumed a value of - 2 the use of the operator ABS would cause MILITRAN to evaluate A and place its new value in the expression which would then be evaluated according to the rules defined above.

4 + 6 * ABS(A)	gives 16	the absolute value of A is 2 which is then multiplied by 6. When 4 is added to the resulting product, a value of 16 is obtained.

When two or more operations are to be computed, the ordering rules may considerably alter the result. For example:

20/4*5 gives 25

If it were intended that the multiplication occur first, the expression should be written as:

20/(4*5) gives 1

The above example illustrates the method used to override the order of operations. Parentheses are used to specify the order of operations in an expression.

Examples:

2 + 4 * 3 - 6/2 gives 11 as a result, whereas

((2 + 4) * 3 - 6) /2 gives 6.

The expression

A + B * C/D + E.P.F - G

will be taken to mean

$$A + \frac{C}{D} * B + E^{F} - G$$

Using parentheses, the expression could be written,

$$(A + B) * C/D + E.P.F - G$$

which would be taken to mean

$$(A + B) * \frac{C}{D} + E^{F} - G$$

Expressions with repeated exponentiation must have clarifying parentheses. For example:

(2.P.2).P.3	gives 64
2.P.(2.P.3)	gives 256

Arithmetic expressions may be of a single mode or a combination of real and integer modes. However, if the elements comprising an expression are of mixed modes, the appearance of a real variable or a real constant with a fractional part causes the entire expression to be evaluated in the real mode. For example:

3.2/2 gives 1.6

because the divisor 2 is first evaluated as a real number.

If a real constant or real variable is used as a subscript or an argument of a dimension, the real quantity will be truncated to an integer value before use. For example, if I assumes the value 7.4 and is later used

as a subscript in an expression as in

A + B + B(I)

I would be truncated to 7 before the evaluation of the expression.

Logical Expressions

A logical expression consists of certain sequences of logical constants, logical variables, arithmetic expressions, and object elements separated by logical operators or relational operators. A logical expression always has the value TRUE or FALSE. The simplest form of a logical expression is a single quantity - a logical constant or logical variable (subscripted or non-subscripted).

Examples:

TRUE

FALSE

PLANE(2) (where PLANE has been defined as a LOGICAL array)

Compound logical expressions may be formed by combining simple logical expressions through the use of logical operators. Logical operators may operate only on logical expressions. They are listed below in decreasing order of precedence:

.NOT.	Negation
.AND.	Conjunction
.EXOR.	Exclusive disjunction
.OR.	Disjunction
.EQV.	Equivalence

The periods are part of the logical operator notation and must be present.

If A and B are any logical expressions, then the logical operators are defined in the following manner:

.NOT.A	has the value TRUE only if A is FALSE; it has the value FALSE only if A is TRUE.
A.AND.B	has the value TRUE only if both A and B have the value TRUE; otherwise it has the value FALSE.
A.OR.B	has the value TRUE if either A or B is TRUE; it has the value FALSE only if both A and B are FALSE.
A.EXOR.B	has the value TRUE if either A is TRUE and B is FALSE or A is FALSE and B is TRUE; it has the value FALSE only if both A and B are TRUE or if both A and B are FALSE.

A.EQV.B	has the value TRUE either if both A and B have the value TRUE or if both A and B have the value FALSE; otherwise it has the value FALSE.

The logical operator .NOT. must be immediately followed by a logical expression. The other logical operators must be preceded and followed by logical expressions to form compound logical expressions.

Examples:

.NOT.TRUE	always has the value FALSE
.NOT.FALSE	always has the value TRUE
TRUE.AND.FALSE	always has the value FALSE
TRUE.OR.FALSE	always has the value TRUE
PLANE1.OR.PLANE2	may have the value TRUE or FALSE depending on the values of the logical variables PLANE1 and PLANE2.
FALSE.EXOR.TRUE	always has the value TRUE
TRUE.EXOR.TRUE	always has the value FALSE
SWITCH1.EXOR.SWITCH2	may have the value TRUE or FALSE depending on the values of the logical variables SWITCH1 and SWITCH2

SWITCH1.EQV.SWITCH2	the logical variables SWITCH1 and SWITCH2 must have the same value for the statement to be TRUE

The above examples are logical expressions which have only one logical operator. The following examples illustrate the use of several logical operators in compound expressions and the use of parentheses for the purpose of overriding the hierarchy of operations.

The logical expression:

.NOT . TRUE . AND . FALSE

has the value FALSE because .NOT. operates only on the logical constant, variable, or expression immediately to the right. However, if the expression is written as

.NOT. (TRUE.AND.FALSE)

it would have a value of TRUE, because the expression (TRUE.AND.FALSE) would be evaluated first.

In the next example:

(X.AND..NOT.Y) .OR. (.NOT.X.AND.Y)

the expression would have a value of TRUE if X is TRUE and Y is FALSE or if X is FALSE and Y is TRUE. It could only have a value of FALSE if both X and Y are TRUE if both X and Y are FALSE.

Relational Expressions

MILITRAN provides a further extension to the set of logical operators in order to permit the formation of compound logical expressions. This second group of operators is called relational operators. One set of relational operators acts on arithmetic elements, the other set acts on object elements.

An arithmetic relational expression consists of two arithmetic expressions, separated by an arithmetic relational operator. An arithmetic relational expression always has the value TRUE or FALSE.

The arithmetic relational operators are:

.E.	equal to
.NE.	not equal to
.G.	greater than
.GE.	greater than or equal to
.L.	less than
.LE.	less than or equal to

The periods are part of the arithmetic relational operator notation and must be present.

If A and B are any two arithmetic expressions, then the arithmetic relational operators are defined as follows:

A.E.B has the value TRUE only if the value of A is equal to the value of B; otherwise it has the value FALSE.

A.NE.B has the value TRUE only if the value of A is not equal to the value of B; otherwise it has the value FALSE.

A.G.B has the value TRUE only if the value of A is greater than the value of B; otherwise it has the value FALSE.

A.GE.B has the value TRUE only if the value of A is greater than or equal to the value of B; otherwise it has the value FALSE.

A.L.B has the value TRUE only if the value of A is less than the value of B; otherwise it has the value FALSE.

A.LE.B has the value TRUE only if the value of A is less than or equal to the value of B; otherwise it has the value FALSE.

The two arithmetic expressions in a relational expression may be of the same mode or one may be real and

the other integer. In the latter case, the integer expression will be evaluated and the result converted to a real number before it is compared to the second expression. For example, if A and B are integer variables and C and D are real variables in the relational expression

(A + B) .LE. (C/D)

A would be added to B, C would be divided by D and the sum of A + B would be converted to a floating point number before it is compared to the result of the division.

NOTE: Integer numbers greater than $2^{27}-1$ cannot be accurately converted to floating point numbers. Therefore, care should be exercised in the construction of relational expressions of mixed modes.

Examples:

In the following examples, A and B are integer variables and C and D are real variables.

A.E.2	this expression has the value TRUE only if the integer variable A is equal to 2.
D.NE.6.9	this expression has the value TRUE only if the real variable D is not equal to 6.9
B.GE.(C+6.2)*(D/3)	this expression has the value

TRUE only if the variable B is greater or equal to the value of the expression (C+6.2)*(D/3.). In accordance with the rule that the appearance of any floating point element causes the entire expression to be evaluated in the floating point mode, the following sequence of events would occur before the comparison is effected.

1. the real variable C is added to the real constant 6.2.
2. the integer constant 3 is floated before it is used as the divisor in (D/3).
3. the two floating point results are multiplied.
4. since the final product is a floating point number, the integer variable B is converted to a real number before the comparison is effected.

An object relational expression consists of two single object elements separated by an object relational operator. An object relational expression always has the value TRUE or FALSE.

Object relational operators may operate only on object elements. The object relational operators are:

.IN.	inclusion
.IS.	equivalence

The periods are part of the object relational operator notation and must be present.

The general form of an object relational expression which uses the object relational operator .IN. is

A.IN.B

where A is a single object element and is declared as follows:

1. in a PROGRAM OBJECT declaration. If the PROGRAM OBJECT declaration refers to a single variable, A is not subscripted; but if the declaration specifies an array, then A must be subscripted.

2. in an OBJECT declaration A must be subcripted because an OBJECT declaration always specifies a group of elements.
3. in a CLASS declaration where A is a member or the name of a class. A must be subscripted because it represents one object within the class.

and B is the name of a group of object elements, and therefore is never subscripted. B is declared as follows:

1. in an OBJECT declaration
2. in a CLASS declaration where B is the name of a class.

If A and B are any object elements and have been declared in accordance with the rules stated above, then A.IN.B is TRUE:

1. if A is a subscripted object element (e.g.MERC(4)) or a PROGRAM OBJECT that takes on the value of the object element, and B is the name of the object set of which A is a member.

Example:

```
OBJECT MERC(10)
MERC(4).IN.MERC
```

has the value TRUE because MERC(4) is a subscripted object element that is a member of the object set MERC.

2. if A is a subscripted object element (e.g. MERC(4)) or a PROGRAM OBJECT that takes on the value of the object element and B is a class which contains the object set (preceded or not preceded by EACH*) of which A is a member.

Example:

OBJECT MERC(10)

CLASS (MODERATE) CONTAINS MERC

or

OBJECT MERC(10)

CLASS (MODERATE) CONTAINS EACH* MERC

MERC(4).IN.MODERATE

has the value TRUE because MERC(4) is a subscripted object element that is a member of the class MODERATE.

3. if A is a subscripted object element (e.g. MERC(4)) or a PROGRAM OBJECT that takes on the value of the object element and B is a class which contains a class (preceded or not preceded by EACH*) which in turn contains the object set of which A (preceded or not preceded by EACH*) is a member.

Example:

OBJECT MERC(10)

CLASS (MODERATE) CONTAINS EACH*MERC

```
CLASS (PRICE) CONTAINS EACH* MODERATE
        or
OBJECT MERC(10)

CLASS (MODERATE) CONTAINS EACH*MERC

CLASS (PRICE) CONTAINS MODERATE
        or
OBJECT MERC(10)

CLASS (MODERATE) CONTAINS MERC

CLASS (PRICE) CONTAINS EACH* MODERATE
        or
OBJECT MERC(10)

CLASS (MODERATE) CONTAINS MERC

CLASS (PRICE) CONTAINS MODERATE

MERC (4) .IN.PRICE
```

has the value TRUE because MERC(4) is a subscripted object element that is a member of the class MODERATE which in turn is a member of the class PRICE.

4. if A is a subscripted class element or a PROGRAM OBJECT that takes on its value, A must be traced back to a subscripted object element in order to determine whether A.IN.B is TRUE as shown in the examples given above.

Examples:

The OBJECT and CLASS declarations which are used to illustrate the evaluation of .IN. expressions

in the following examples are identical to the ones used in the section on classes, in order to permit reference to the table of subscripts.

```
OBJECT FORD(10),MERC(5),LINC(3),CHEV(10),OLDS(5),
       CADDY(3),PLY(10),DODGE(5),CHRYS(3)
CLASS(ECONOMY)CONTAINS FORD,CHEV,PLY
CLASS(MODERATE)CONTAINS MERC,OLDS,DODGE
CLASS(PRESTIGE)CONTAINS LINC,CADDY,CHRYS
CLASS(PRICE)CONTAINS ECONOMY,MODERATE,PRESTIGE
CHEV(2) .IN. CHEV                has the value TRUE
CHEV(10) .IN. FORD               has the value FALSE
DODGE(2) .IN. FORD               has the value FALSE
PLY(10) .IN. ECONOMY             has the value TRUE
ECONOMY(3) .IN. PLY              has the value TRUE
OLDS(4) .IN. MODERATE            has the value TRUE
DODGE(5) .IN. PRESTIGE           has the value FALSE
CADDY(3) .IN. PRICE              has the value TRUE
PRICE(2) .IN. MODERATE           has the value TRUE
PRICE(3) .IN. MODERATE           has the value FALSE
OLDS(4) .IN. PRICE               has the value TRUE
PRICE(1) .IN. PRESTIGE           has the value FALSE
PRESTIGE(1) .IN. PRICE           has the value TRUE
                                 (because PRESTIGE(1) is
                                  equivalent to LINC(1))
```

ECONOMY(3) .IN. CHEV has the value FALSE

Tracing ECONOMY(3) back, its representative is a subscripted object element, PLY(1). According to the rules stated above, PLY(1). IN. CHEV is FALSE.

PRICE(2) .IN. MERC has the value TRUE

Tracing PRICE(2) back it is represented by MERC(1), and MERC(1) .IN. MERC is TRUE.

MERC(3) .IN. MERC has the value TRUE

MERC(3) is a subscripted object element which is a member of the object set MERC.

MODERATE(4) .IN. PRICE has the value FALSE

Since class MODERATE has three representatives, MERC(1), OLDS(1), and DODGE(1), the maximum subscript that may be used with MODERATE is (3).

```
OBJECT FORD(10),MERC(5),LINC(3),CHEV(10),OLDS(5),
       CADDY(3),PLY(10),DODGE(5),CHRYS(3)
CLASS(ECONOMY) CONTAINS EACH*FORD,EACH*CHEV,EACH*PLY
CLASS(MODERATE) CONTAINS EACH*MERC,EACH*OLDS,EACH*DODGE
CLASS(PRESTIGE) CONTAINS EACH*LINC,EACH*CADDY,EACH*CHRYS
CLASS(PRICE) CONTAINS EACH*ECONOMY,EACH*MODERATE,EACH*PRESTIGE
CHEV(7) .IN. ECONOMY                    has the value TRUE
PLY(2) .IN. PRICE                       has the value TRUE
OLDS(3) .IN. PRESTIGE                   has the value FALSE
MODERATE(15).IN. DODGE                  has the value TRUE
PRESTIGE(4) .IN. LINC                   has the value FALSE
PRICE(36) .IN. OLDS                     has the value TRUE
PRICE(49) .IN. FORD                     has the value FALSE
PRICE(50) .IN. CHRYS                    has the value FALSE
PRESTIGE(1) .IN. PRICE                  has the value TRUE
                                        because PRESTIGE(1)
                                        is equivalent to LINC(1))

OBJECT FORD(10),MERC(5),LINC(3),CHEV(10),OLDS(5),
       CADDY(3),PLY(10),DODGE(5),CHRYS(3)

CLASS (CAR) CONTAINS EACH*FORD,MERC,LINC,EACH*CHEV,
                     OLDS,CADDY,EACH*PLY,DODGE,CHRYS
CADDY(3) .IN. CAR                       has the value TRUE
OLDS(5) .IN. CAR                        has the value TRUE
CAR(16) .IN. CHEV                       has the value TRUE
CAR(25) .IN. OLDS                       has the value FALSE
CAR(34) .IN. DODGE                      has the value FALSE
```

```
OBJECT MERC(5),CADDY(3),DODGE(5),CHRYS(3)
CLASS (MODERATE) CONTAINS MERC,DODGE
CLASS (PRESTIGE) CONTAINS EACH*CADDY,EACH*CHRYS
CLASS (CAR) CONTAINS EACH*MODERATE,PRESTIGE
MERC(4) .IN. MODERATE                    has the value TRUE
CHRYS(3) .IN. CAR                        has the value TRUE
PRESTIGE(2).IN. CHRYS                    has the value FALSE
PRESTIGE(6) .IN. CHRYS                   has the value TRUE
MODERATE(1) .IN. DODGE                   has the value FALSE
CAR(2) .IN. DODGE                        has the value TRUE
CAR(3) .IN. CHRYS                        has the value FALSE
```

The general form of an object relational expression which uses the object relational operator .IS. is:

A.IS.B

where A and B are individual object elements and are declared as follows:

1. in a PROGRAM OBJECT declaration. If the PROGRAM OBJECT declaration refers to a single variable, the single variable (A or B) is not subscripted; but if the declaration specifies an array, A and B must be subscripted.

2. in an OBJECT declaration A or B must be subscripted.

3. in a CLASS declaration where A or B are members of a class and therefore must be subscripted.

If A and B are any object elements and have been declared in accordance with the rules stated above, .IS. is defined as follows:

A.IS.B	has the value TRUE only if object element A is <u>identical</u> to object element B; otherwise it has the value FALSE

Examples:

```
OBJECT FORD(10),MERC(5),LINC(3),CHEV(10),OLDS(5),
       CADDY(3),PLY(10),DODGE(5),CHRYS(3)

CLASS(ECONOMY)CONTAINS FORD,CHEV,PLY
CLASS(MODERATE) CONTAINS MERC,OLDS,DODGE
CLASS(PRESTIGE) CONTAINS LINC,CADDY,CHRYS
CLASS(PRICE) CONTAINS ECONOMY,MODERATE,PRESTIGE
```

FORD(1).IS. ECONOMY(1)	has the value TRUE

```
PRICE(1) .IS. FORD(1)                    has the value TRUE
DODGE(4) .IS. MODERATE(3)                has the value FALSE
DODGE(1) .IS. MODERATE(3)                has the value TRUE
LINC(1) .IS. PRICE(3)                    has the value TRUE
CHEV(1) .IS. PRICE(1)                    has the value FALSE
OBJECT MERC(5),CADDY(3),DODGE(5),CHRYS(3)
CLASS(MODERATE) CONTAINS MERC,DODGE
CLASS(PRESTIGE)CONTAINS EACH*CADDY,EACH*CHRYS
CLASS(CAR) CONTAINS EACH*MODERATE,PRESTIGE
DODGE(1) .IS. MODERATE(2)                has the value TRUE
CHRYS(3) .IS. PRESTIGE(6)                has the value TRUE
MERC(1) .IS. CAR(1)                      has the value TRUE
CADDY(3) .IS. CAR(3)                     has the value FALSE
CAR(1) .IS. MERC(2)                      has the value FALSE
CAR(1) .IS. MERC(1)                      has the value TRUE
```

2. Arithmetic Statements

The arithmetic statement defines a numerical calculation. Its general form is

A = B

where A is a real or integer variable, subscripted or not subscripted, and B is an arithmetic expression. The arithmetic statement has two functions. First, it causes the computation of the expression to the right of the equals symbol and second, it causes the value of the variable to the left of the equals symbol to be replaced by

the result of the calculation.

In MILITRAN, the expression to the right of the equals symbol is converted, after it has been evaluated, to the mode of the variable to the left of the equals symbol with the following exception:

When the expression to the right of the equals symbol is a single subscripted variable which denotes a member of a vector whose retrieval form is that of the vector rather than the component name, the expression is not converted to the mode of the variable to the left of the equals symbol. The value is stored without being converted.

Examples:

In the following examples, A and B are integer variables, C and D are real variables, E is the name of a vector whose mode is real and whose component arrays are also all of real mode.

A = B replace A with the current value of B.

A = C truncate C to an integer, convert it to an integer constant and replace the value of A with the value of C.

D = B	convert B to a real number and replace D with the value of B.
B = A*(C/2)	the appearance of the real variable C causes the A and 2 to be converted to real numbers before the value of the expression is computed. The result is then converted to integer mode and replaces the previous value of B.
B = 3.4/6.4	after this arithmetic statement has been executed, B will have a value of 0.
A = E(5,B)	replace A with the current value of E(5,B). Do <u>not</u> convert the value which is a real number to an integer.

3. Logical Statements

The logical statement defines a logical calculation. Its form is:

A = B

were A is a logical variable, subscripted or not subcripted and B is a logical expression. The logical statement computes the value of the logical expression (either TRUE or FALSE) to the right of the equal symbol

and replaces the previous value of the logical variable to the left of the equal symbol.

Examples:

A = TRUE	store the logical constant TRUE in A
B = .NOT.C	if C is TRUE store the value FALSE in B; if C is FALSE store the value TRUE in B.
L = X.AND.Y	if both X and Y are TRUE, L will assume the value TRUE; if either or both X and Y are FALSE, then L will assume the value FALSE.
L = .NOT.(X.AND.Y)	both X and Y must be TRUE in order for L to assume the value FALSE.
P = (X.G.Y).OR.B	if the numerical value of X is greater than the numerical value of Y or B is TRUE, then P assumes the value TRUE. P assumes a value of FALSE only if the numerical value of X is less than the value of Y, and B is FALSE.

H = (A.E.D).EXOR.(P.IN.G)	if the numerical value of A is equal to the numerical value of D and P is not a member of set G, then H will assume a value of TRUE; if the numerical value of A is not equal to the numerical value of D and P is a member of set G, then H will also assume a value of TRUE. Otherwise, H assumes a value of FALSE.

CHAPTER V

CONTROL STATEMENTS

During the execution of a MILITRAN program, instructions are normally taken from sequentially ascending locations. However, the execution of instructions does not have to occur sequentially. It is possible through the use of sequential operators or control statements to alter the process of sequential execution and to cause the computer to repeat, skip, or interrupt a sequence of MILITRAN statements. In this way it is possible to modify the sequence in which any statement or block of statements is executed. By providing a program with the ability to control its own course of execution, these statements greatly increase the scope of the system.

1. GO TO

GO TO is used to unconditionally alter the normal sequential execution of statements. It indicates the statement that is to be executed next.

The form of a GO TO statement is:

GO TO s

where s is the name of the next statement to be executed.

Examples:

GO TO ACT 1100

GO TO B702

GO TO NT7

Use of the GO TO statement is shown in the following coding example.

Statement Label	MILITRAN Statement
ACT 100	RATE = 7.0
ACT 101	TIME = 2.0
ACT 102	GO TO ACT 104
ACT 103	RATE = 5.0
ACT 104	DIST = RATE*TIME

When control reaches statement ACT 100, RATE will be given the value 7.0. Then TIME will be given the value 2.0. The next statement, GO TO ACT 104 will cause statement ACT 103 to be skipped and statement ACT 104 will be executed next, giving DIST a value of 14.0.

2. PAUSE

The PAUSE statement causes the computer to come

to a temporary halt. If the start key is pressed, the object program will resume execution with the next MILITRAN statement. PAUSE is written as:

```
PAUSE j
```

where j is any unsigned octal integer of 1 to 4 digits, and may be omitted. This number will be shown on the computer console when the computer stops.

Examples:

```
PAUSE 1
PAUSE
```

3. STOP

The STOP statement causes the immediate termination of the object program. After STOP a restart cannot occur. The STOP statement should occur at the logical end of the program rather than the physical end. More than one STOP statement may be used in a program. The form of the STOP statement is:

```
STOP
```

4. IF Statement

The IF statement determines the statement to be

executed next dependent on the value of a logical expression. It is written as follows:

$$IF(b),\ S_T,S_F$$

where b is a logical expression and S_T and S_F are statement labels. If the value of the logical expression (b) is TRUE, control will be transferred to the statement labelled S_T; if the value is FALSE, control will be transferred to the statement labelled S_F. If the second label is omitted, the next program statement will be executed.

Examples:

```
IF(TIME.GE.HORIZON),ENDCYCLE,DC100
```

If the value of TIME is greater or equal to the value of HORIZON, control will be transferred to the statement labelled ENDCYCLE. If the logical expression is false, control will be transferred to statement DC100.

```
IF(TIME.GE.HORIZON),ENDCYCLE

T = A/3.4
```

If the value of the logical expression is TRUE, control will be transferred to ENDCYCLE. Otherwise, the next statement will be executed.

5. UNLESS Statement

The UNLESS Statement also determines the statement to be executed next, dependent on the value of a logical expression. It is written as follows:

$$\text{UNLESS(b), } S_F, S_T$$

where b is a logical expression and S_F and S_T are statement labels. If the value of the logical expression (b) is FALSE, control will be transferred to the statement labelled S_F; if the value is TRUE, control will be transferred to the statement labelled S_T. If the second label is omitted, the next program statement will be executed.

Examples:

```
UNLESS(PAST REPORT+REPORT INTERVAL.LE.TIME),A100,A250
```

If the value of the logical expression is FALSE, control will be transferred to statement A100; if the expression is TRUE, control will be transferred to statement A250.

```
IF(TIME.GE.HORIZON),ENDCYCLE
UNLESS(REPORT INTERVAL.LE.TIME),RPER2000
ENDCYCLE
```

Control will be transferred to ENDCYCLE if TIME is greater than or equal to HORIZON. If the first statement is FALSE, the succeeding statement will be evaluated and if it TRUE, ENDCYCLE will be executed. If both statements are FALSE, control will be transferred to the statement labelled RPER2000.

6. DO Statement

The technique of repeating a section of a program, with some type of modification between repetitions, is called looping. The DO statement is a powerful tool which permits a significant reduction in the number of instructions required to perform a given procedure and also greatly simplifies the programming of loops. MILITRAN provides two forms of the DO statement.

Form 1

DO(s) UNTIL b, n = e_1,e_2

where s is a statement label, b is a logical expression, n is an arithmetic variable, either subscripted or non-subscripted, e_1,e_2 are arithmetic expressions. When e_2 is omitted it is assumed = 1. When e_1 is omitted, both e_2 and the equals sign must be omitted and e_1 and e_2 are assumed = 1. When n is omitted, the statement ends with the Boolean condition and looping continues until b.EQV.TRUE. The

logical expression b is evaluated before each iteration.

Form 1 of the DO statement is a command to iterate through the statement labelled s until the logical expression b is TRUE. The first time, the statements are executed with n equal to e_1. For each succeeding iteration, n is increased by e_2. When the logical expression b assumes the value TRUE, control passes to the statement following the last statement in the range of the DO - the statement immediately following the statement labelled s.

This form of the DO statement has three functions:

1. It establishes an index (n) which takes on a new value for each iteration. This index may be used as a subscript or in computations.

2. It causes looping through any desired series of statements, as many times as required.

3. It increases the index by any specified increment for each separate iteration through the series of statements in the loop.

As an example, consider the following program:

Statement Label	Statement
	REAL A(10),B(10)
ADD40	DO(ADD50)UNTIL N1.G.10,N1=1,1
ADD50	A(N1) = B(N1)*2

The first statement reserves space for two one-dimensional arrays, A and B. The DO statement, statement ADD40, is a command to execute the following statements up to and including statement ADD50. The DO loop is equivalent to execution of the statements

```
A(1) = B(1)*2
A(2) = B(2)*2
A(3) = B(3)*2
     .
     .
     .
A(10) = B(10)*2
```

When the DO loop is entered (statement ADD40) N1 is set to 1, the logical expression N1.G.10 is evaluated and since it is FALSE statement ADD50 is executed. N1 is increased by 1, the logical expression is evaluated again and the loop will continue until N1 assumes a value of 11. The program

will then continue with the statement following statement ADD50.

The following is a comparison of statement ADD40 with the general form of the DO, and an introduction of some of the terms used in discussing DO statements.

DO	(s)	UNTIL	b,	n =	e_1,	e_2
DO	(ADD50)	UNTIL	N1.G.10,	N1 =	1,	1
	Range		Terminating Condition	Index	Initial Value	Increment

Range: The range is the series of statements to be executed repeatedly. It starts with the DO and includes all the statements following the DO up to and including statement (s). In the example the range consists of statements ADD40 and ADD50.

Terminating Condition: The terminating condition is the logical expression which controls the number of iterations to be performed. When the logical expression assumes a value of TRUE, the DO is satisfied. In this case, as soon as N1 assumes a value of 11, execution of the range ceases.

Index: The index is any arithmetic variable. The

index will change for each execution of the range. In the example, the index N1 is also used as a subscript, in another problem it might be used in computations, or might not be used in the range at all.

Initial Value: The initial value may be any arithmetic expression, and is the value assigned the index for the first execution of the range. In the example, the initial value is 1 - an integer constant. In another problem it might be a real constant or a subscripted arithmetic variable.

Increment: The increment (any arithmetic expression) is the amount by which the value of the index will be increased after each execution of the range.

Examples:

In the next example, the range of the DO loop consists of one statement, the DO statement itself.

```
      INTEGER     A(100)
LBL   DO(LBL)UNTIL(A(I).E.6).OR.(I.G.100),I=1,1
```

The DO statement (LBL) is a command to cycle

through the array named "A" and obtain the first entry equal to 6. The logical expression (I.G.100) is necessary in case array "A" does not contain an entry equal to 6.

In the following example, the terminating condition is a logical expression which includes an object relational expression.

```
          PROGRAM OBJECT B(100)
LBL       DO(LBL)UNTIL(B(I).IN.CAR).OR.(I.G.100),I=1,1
```

This DO statement will cycle through the array names "B" and will obtain the first entry which is in the class CAR.

Form 2

The second form of the DO statement operates on object elements exclusively. It is written as

DO (s) FOR a.IN.b

where s is a statement label, a is a single variable which has been declared in a PROGRAM OBJECT declaration and b is either the name of a class as specified in a CLASS declaration or the name of an object as specified in an OBJECT declaration. Names a and b are never subscripted.

This form of the DO statement is a command to iterate through the statement labelled s. The first time,

variable "a" is set to the identity of the first member of "b". For succeeding iterations, "a" assumes in turn the identity of all members of "b". Iteration ends when all members of "b" have been covered sequentially.

As an example, consider the following program:

Statement Label	Statement
	INTEGER AGE(CAR),I,J
	PROGRAM OBJECT A
LX2	J = 0
LX3	I = 0
LX4	DO(LX7) FOR A.IN.CAR
LX5	UNLESS(AGE(A).GE.5),LX7
LX6	I = I + 1
LX7	J = J + 1

This program counts both the total number of cars and those cars which are 5 years or older.

1. The first statement reserves space for one dimensional array "AGE", whose dimension is equal to the number of the members in class CAR; and also declares I and J to be integer variables.

2. The second statement declares "A" to be a single variable which will assume the identity of an object.

3. LX2 sets the value of J to zero; LX3 sets the value of I to zero.

4. LX4 is a command to execute the following statements up to and including the statement named LX7. When the DO loop is entered A is set to the identity of the first member of class CAR.

5. LX5 retrieves the age of the first car and compares it to 5. If it is less than 5, control passes to the statement LX7; if the age is equal to or greater than 5, control passes to LX6.

6. LX6 increments the counter "I" by one;
LX7 increments counter "J" by one.

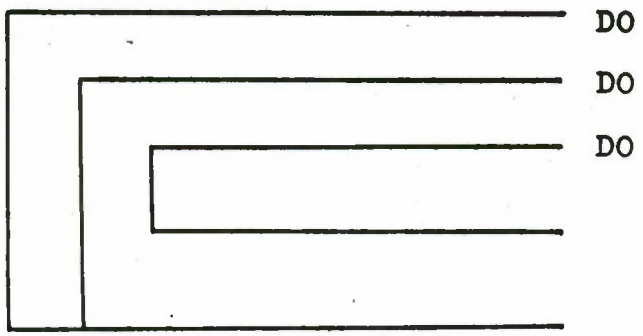
7. After statement LX7 has been executed, control is transferred to statement LX4. The second time through the loop, A is set to the identity of the second member of class CAR and its age is retrieved and evaluated. The looping continues until all members of the class have been evaluated.

Restrictions on the use of DO statements

1. a DO loop may be contained within the range of another DO loop. When this situation occurs, all of the

statements in the range of the inner DO must be within the range of outer DO.

For example:



is a permitted configuration (the brackets indicate the range of the DO's), but:

is not a permitted configuration. Transfers of control are permitted both from inside the range of a DO loop to outside its range and from outside the range to inside.

2. the last statement in the range of a DO loop cannot be a GO TO statement, an IF statement which has two labels or an UNLESS statement which has two labels. If it is necessary to end a DO loop with any of these statements, the CONTINUE statement must be used to terminate the range of the DO loop.

7. CONTINUE

CONTINUE is a dummy statement which does not generate any instructions in the object program. It must be used to terminate the range of a DO loop which would otherwise end with a GO TO statement or the forms IF or UNLESS statements which have two labels. CONTINUE is also used as the last statement of a DO loop when it is desired conditionally to skip the several statements in the range and proceed with the next iteration of the loop.

The form of the CONTINUE statement is:

```
CONTINUE
```

Example:

```
Statement Label     Statement
                    REAL A(10),B(10)
    ADD40           DO(ADD70)UNTIL N1.G.10,N1=1,1
    ADD50           A(N1)=B(N1)*2
    ADD60           GO TO ADD100
    ADD70           CONTINUE
```

In the above example, if ADD60 were the last statement in the range the sequence at ADD100 would have no way to re-enter the loop. The CONTINUE statement at ADD70

provides such a re-entry point.

The next example shows how one statement in the range may be conditionally skipped using a CONTINUE statement.

Statement Label	Statement
	INTEGER AGE(CAR),I,J
	PROGRAM OBJECT A
LX2	J = 0
LX3	I = 0
LX4	DO(LX8) FOR A.IN.CAR
LX5	J = J + 1
LX6	UNLESS(AGE(A).GE.5),LX8
LX7	I = I + 1
LX8	CONTINUE

Use of the CONTINUE statement is by no means limited to defining the ranges of DO loops. Any point in a program may be defined by means of a CONTINUE statement.

CHAPTER VI

LISTS AND LIST PROCESSING STATEMENTS

The LIST declaration and its associated list processing statements have been provided to facilitate the creation, maintenance, and processing of the various elements in a simulation. Concise MILITRAN statements eliminate the need for the complex iterative coding which is normally required to maintain the current status of the elements in a simulation.

1. LIST

A MILITRAN LIST is a one dimensional vector whose components may be processed by a special group of system routines - the list processing statements. The LIST statement is written as

$$\text{LIST } n((c_1,c_2,\ldots,c_i),d)$$

where n is the name of the LIST; $c_1,c_2,\ldots,c_i$ are the names of the LIST components, all of which are associated with the name "n", and d is the dimension associated with each component.

The LIST statement only declares the dimension of the LIST; the rules for declaring the modes of a list and its components are the same as those for declaring the modes of a vector and its components.

Example :

```
LIST SHIPS((DESTROYER,CRUISER,CARRIER),10)
```

This statement defines a LIST named SHIPS, which has 3 components - DESTROYER, CRUISER, and CARRIER. Each component has a dimension of 10 -- resulting in 30 locations reserved for the LIST SHIPS.

2. LIST PROCESSING STATEMENTS

The list processing statements are used to maintain the current status of the various elements in a simulation.

LIST entries may be created by means of PLACE or PLACE ENTRY statements; modified by REPLACE or REPLACE ENTRY statements; destroyed by REMOVE or REMOVE ENTRY statements and located by the system functions MINIMUM INDEX and RANDOM INDEX. In addition, the elements in a LIST may also be oper-

ated on by the regular MILITRAN data processing statements.

Before discussing the list processing statements in detail, it is necessary to clarify the meaning of some of the terms used in connection with these statements.

When a list processing statement or its definition refers to an entry, it is referring to corresponding elements in each component of the LIST. For example, if a LIST is declared as:

```
LIST A((B,C,D,E),4)
```

there are four components - B,C,D, and E, each of which contains four elements. An entry in LIST A is a group of four corresponding elements such as:

B(1),C(1),D(1),E(1)	-	1st entry of LIST A
B(2),C(2),D(2),E(2)	-	2nd entry of LIST A
B(3),C(3),D(3),E(3)	-	3rd entry of LIST A
B(4),C(4),D(4),E(4)	-	4th entry of LIST A

The list processing statements <u>always</u> operate on entries. If a LIST has only one component, then entry and element are synonymous.

Every MILITRAN LIST is associated with an integer value which represents the length of the LIST. The length of the LIST refers to the current number of entries in the LIST.

It is important to distinguish between the length of the LIST and its dimension. The dimension refers to the number of locations in computer storage which are reserved by the LIST declaration for the LIST entries. The dimension remains constant throughout execution of the program. The length of the LIST changes - it is automatically updated by those list processing statements which create or destroy entries. The list processing statements which modify existing entries do not change the length of the LIST.

The compilation phase of MILITRAN sets the length of any declared LISTS to zero. If all the entries in a LIST are created by the list processing statements, the length of the LIST will be created automatically. However, if a LIST is created by reading data into it, the initial length must be set by the program as described below under RESET LENGTH.

As stated above, the list processing statements only operate on entries. If it is necessary to modify or retrieve an element within a single component of a multi-

component LIST, the element may be subscripted in exactly the same manner as are elements in a vector.

In the explanation of the list processing statements, the following notation will be used:

n - is the name of a LIST which is being interrogated or in which data is being entered, modified, or removed.

k - refers to an integer constant, integer variable, or an arithmetic expression which after evaluation will return an integer value. The value represents the index of entry (the position of entry in the LIST) for LIST n.

m - is the name of a LIST from which data is being retrieved.

j - is an integer value, as described under k, which represents the index of entry for LIST m.

e - is any expression which is going to be entered in a component of a LIST or will modify an existing element in a LIST. It must be compatible with the mode of the component as it is defined in the mode declaration.

b - is any logical expression.

p - is any integer constant, integer value, or arithmetic expression which after evaluation will return an integer value. The value represents the length of a LIST.

LENGTH

LENGTH(n)

where n is the name of a LIST, returns the current number (integer) of entries in LIST "n".

The LENGTH function provides a convenient means of determining when all the entries in a list have been eliminated from the battle. For example,

```
A101  IF (LENGTH(GREEN ARMY).E.0), B100
A102  (continue with processing)
```

When the LENGTH of GREEN ARMY assumes a value of 0, all the entries in GREEN ARMY have been eliminated and control is transferred to statement B100. If all the entries have not been eliminated, the program continues sequentially with statement A102.

RESET LENGTH

RESET LENGTH(n) TO p

when n is the name of a LIST and p is an integer constant, integer variable, or an arithmetic expression which after evaluation will return an integer value, unconditionally

sets the length of LIST "n" to the value represented by "p".

When the entries in a LIST are a function of input, i.e., the data is read directly into a LIST, the RESET LENGTH statement must be given before any processing is carried out on the LIST. For example, if a LIST is declared as follows:

```
LIST A((B,C,D),3)
```

and an input statement reads in 3 entries, the length of the LIST should be set as follows:

```
RESET LENGTH(A) TO 3
```

If the dimension of the list is a "symbolic dimension", such as:

```
LIST A((B,C,D),X)
```

the value of X would also have to appear in the input. After the value of X has been read in, the statement:

```
RESET LENGTH(A) TO X
```

would set the length of A to X.

PLACE

PLACE(e_1,e_2,...,e_i) IN n

where e_1, e_2,...,e_i are any expressions and n is the name of a LIST, evaluates the expressions represented by each e, enters the value of the corresponding e in each of the components of LIST n, and then automatically updates the length of LIST n.

e_1, e_2,...,e_i must be compatible with the components of the LIST as they are defined in the LIST and mode declarations both as to number and mode. For example, if a LIST is declared as follows:

LIST SHIPS (TYPE,SPEED,RANGE),4)

and the mode declarations state:

PROGRAM OBJECT SHIPS, TYPE
REAL SPEED, RANGE.

the modes of the expressions in the statement:

PLACE(SUBMARINE,SSPEED,SRANGE) IN SHIPS

must be declared as follows:

```
PROGRAM OBJECT SUBMARINE
REAL SSPEED, SRANGE
```

If the list contained 4 entries before the PLACE statement was encountered, the length of the list would be updated to 5 after execution of the PLACE statement.

REMOVE ENTRY

```
REMOVE ENTRY n (k)
```

where n is the name of a LIST and k is an integer constant or integer variable which designates an entry in the LIST, causes the entry represented by k to be removed from LIST n and then automatically updates the length of LIST n.

For example, if LIST Z has three components - A, B, and C -- and the current length of LIST Z is 6, the statement

```
REMOVE ENTRY Z(X)          (where X has assumed a value of 4)
```

will cause the 4th entry to be removed from LIST Z, and the length updated to 5.

NOTE: The LIST is also compressed to remove the blank entry.

PLACE ENTRY

PLACE ENTRY m (j) IN n

causes the j^{th} entry in LIST m to be entered in LIST n. The length of LIST n is automatically updated.

The modes and number of components in LIST n must be compatible with the modes and number of components in LIST m.

NOTE: The j^{th} entry in LIST m is not modified in any way by the PLACE ENTRY statement.

REPLACE ENTRY

REPLACE ENTRY n (k) BY $(e_1, e_2, \ldots, e_i)$

where $e_1, e_2, \ldots, e_i$ are any expressions and n is the name of a LIST, evaluates the expressions represented by each e, enters the value of the corresponding e into the k^{th} position of the corresponding component of LIST n.

$e_1, e_2, \ldots, e_i$ must be compatible with the components of the LIST as they are defined in the LIST and mode declarations both as to number and mode.

REPLACE ENTRY BY ENTRY

REPLACE ENTRY n (k) BY ENTRY m (j)

causes the k^{th} entry in LIST n to be replaced by the j^{th} entry in LIST m.

If the names "n" and "m" are identical, (i.e. refer to the same LIST), "m" may be omitted.

The modes and number of components in LIST n must be compatible with the modes and number of components in LIST m.

NOTE: The j^{th} entry in LIST m is not modified in any way by the REPLACE ENTRY BY ENTRY statement.

Conditional Expressions in List Processing Statements

Each of the list processing statements described below contains a logical expression or set of expressions which determines whether or not the entries in the designated LIST will be operated upon as specified by the particular list processing statement. A conditional list processing statement implies a Do-loop because the statement can operate on more than one entry.

When the form:

$$(b_1, b_2, \ldots, b_i)$$

is used, each b represents a logical expression that refers to a corresponding component of the designated LIST. Therefore, the number of expressions must not exceed the number of components in the LIST.

For example, if a LIST has four components as in:

LIST A ((B, C, D, E), 3)

the conditional list processing statement may contain four logical expressions.

The $(b_1, b_2, \ldots, b_i)$ form causes every entry in the LIST to be examined.

An additional qualification may be imposed on the index of entry. When the form:

$$(b_1, b_2, \ldots, b_i, b_x)$$

is used, b_x refers to a condition which is being placed on the entry index (the position of the entry in the LIST).

In this case, if a LIST contains n components, n+1 conditions are imposed.

The (b_1, b_2,...,b_1, b_x) form limits the entries to be examined to those specified by the expression b_x.

When a conditional list processing statement is encountered during execution of the object program, only those entries which satisfy the following conditions will be processed:

1. The evaluation of each logical expression must result in a value of TRUE in reference to its corresponding component,

2. and when a condition is placed on the entry index (b_x form), this logical expression must also have a value of TRUE.

The symbol * is used to denote the current value of a component or the current index.

For example, the logical expression:

(* . G . 3, * . G . 6)

would refer to a LIST with two or more components. Those entries for which the first element exceeds 3, and the second element exceeds 6 would be operated upon.

NOTE: Control always passes to the instruction following the

conditional list processing statement, whether or not any entries are processed.

The conditional list processing statements are:

REMOVE

REPLACE

REPLACE BY ENTRY

REMOVE

REMOVE (b_1, b_2,...,b_i) FROM n

where b_1, b_2,...,b_i are any logical expressions and n is the name of a LIST, evaluates the expressions represented by each b and removes all the entries in LIST n which cause all of the expressions b_1, b_2,...,b_i to be evaluated as TRUE.

b_1, b_2,...,b_i must be compatible with the components of the LIST as to number and mode.

The LIST is compressed to remove blank entries, and the length of the LIST is automatically updated.

For example if a LIST is declared as:

LIST SHIPS (TYPEA, SPEED, RANGE), 40)

and the mode declarations state:

```
PROGRAM OBJECT SHIPS, TYPE
INTEGER SPEED, RANGE
CLASS(SUBMARINE)CONTAINS TYPEA, TYPEB, TYPEC
```

The statement:

```
REMOVE (*.IN SUBMARINE, *.G.20, *.G.1000) FROM SHIPS
```

would remove every entry in the LIST ships that was in CLASS SUBMARINE whose speed is greater than 20 and whose range is greater than 1000.

If the form:

$$\text{REMOVE } (b_1, b_2, \ldots, b_i, b_x)$$

is used, b_x is the condition placed on the entry index.

In the example given above, if the REMOVE statement had been written as follows:

```
REMOVE (*.IN.SUBMARINE, *.G.20, *.G.1000, *.LE.20)
```

the last conditional would restrict the removal to the first 20 entries provided that the other conditions are also met.

REPLACE

REPLACE (b_1, b_2,...,b_i) BY (e_1, e_2,...,e_i) IN n

where b_1, b_2,...,b_i are any logical expressions, e_1, e_2,...,e_i are any expressions, and n is the name of a LIST, evaluates the expressions represented by each b, and replaces all the entries in LIST n, which cause all of the expressions b_1, b_2,...,b_i to be evaluated as TRUE, with the value of the corresponding e.

The form:

REPLACE(b_1, b_2,...,b_i, b_x) BY (e_1, e_2,...,e_i) IN n

may also be used.

The modes and number of components in LIST n must be compatible both with the modes and number of expressions represented by each b and each e.

REPLACE BY ENTRY

REPLACE(b_1, b_2,...,b_i) BY ENTRY m (j) IN n

evaluates the logical expressions represented by each b, and replaces all the entries in LIST n which cause all of the expressions b_1, b_2,...,b_i to be evaluated as TRUE, with the j^{th} entry in LIST m.

The form:

REPLACE (b_1, b_2,...,b_i, b_x) BY ENTRY m (j) IN n

may also be used.

The modes and number of components in LIST n must be compatible with both the number of expressions represented by each b and with the modes and number of components in LIST m.

If the names "n" and "m" are identical, "m" may be omitted.

MINIMUM INDEX

MINIMUM INDEX (n(b_1, b_2,...,b_i), s)

or

MINIMUM INDEX ($n(b_1, b_2,\ldots,b_i, b_x)$, s)

where n is the name of a LIST, each b is a logical expression, and s is a statement label, evaluates the expressions represented by each b and returns the lowest index in LIST n whose entry causes all of the logical expressions to be evaluated as TRUE. If no such entry is found, control is transferred to the statement labelled "s".

MINIMUM INDEX is a function which may be used in any expression anywhere in the program. The value returned is an integer value.

MINIMUM INDEX may be shortened to INDEX without loss of meaning.

RANDOM INDEX

RANDOM INDEX ($n(b_1, b_2,\ldots,b_i)$, s)

or

RANDOM INDEX ($n(b_1, b_2,\ldots,b_i,b_x)$, s)

where n is the name of a LIST, each b is a logical expression, and s is a statement label, evaluates the expressions represented by each b and returns an index in LIST n chosen at random from all the indices whose entries cause all of the logical expressions to be evaluated as TRUE. If no such entry is found, control is transferred to the statement labelled "s".

RANDOM INDEX is a function which may be used in any expression anywhere in the program. The value returned is an integer value.

LST

LST is used to impose an additional qualification on one of the components in an entry.

If LST precedes a logical expression in a conditional list processing statement, only the entry whose corresponding component is the least of all the components will be operated upon, provided that all logical expressions assume a value of TRUE.

LST may be used with only one expression in a conditional list processing statement.

For example:

$$(b_1, b_2, LST(b_3), \ldots, b_1)$$

is valid.

$$(b_1, LST(b_2), LST(b_3), \ldots, b_1)$$

is invalid.

If two entries are found, the entry with the least index is chosen.

If LST refers to b_x as in

$$(b_1, b_2, b_3, \ldots, b_1, LST(b_x))$$

the minimum index is sought.

GST

GST is similar to LST except that the entry whose corresponding component is the greatest of all such components will be operated on.

Additional Rules for Specifying the Formation of Logical Expressions in Conditional List Processing Statements

Certain abbreviations and conventions may be used in the logical expressions of a conditional list processing statement.

1. If the logical expression is a single quantity - the logical constant TRUE - the value of each of the elements in the corresponding component is not examined. i.e: in the expression $(b_1, TRUE, b_3)$ only

b_1 and b_3 are evaluated.

2. Omitted expressions are assumed to be TRUE - i.e: $(b_1,,b_3)$ is equivalent to $(b_1,TRUE,b_3)$.

3. An arithmetic expression which does not have an arithmetic relational operator is assumed to be *.E.e (where e is an arithmetic expression) i.e:

5

is equivalent to:

*.E.5

4. A PROGRAM OBJECT variable alone is equivalent to *.IS.e (where e is a PROGRAM OBJECT variable) i.e: if TANK is declared in a PROGRAM OBJECT declaration, then:

TANK

is equivalent to:

*.IS.TANK

5. An OBJECT or CLASS name alone is equivalent to *.IN.e (where e is an

OBJECT or CLASS name).i.e. if TANK is declared in an OBJECT declaration, then:

TANK

is equivalent to

*.IN.TANK

CHAPTER VII

EVENTS

A military simulation program usually involves the processing of simulated events which occur either at regular intervals or at critical junctures in time. The event statements facilitate the processing of these simulated occurrences by providing the programmer with the means to associate the data related to a specific event with the processing that must be effected each time the event occurs.

The event statements are:

PERMANENT EVENT

CONTINGENT EVENT

END

END CONTINGENT EVENTS

NEXT EVENT

NEXT EVENT EXCEPT

PERMANENT EVENT and CONTINGENT EVENT define the start of processing for a specific event and also declare a list. The END statement defines the end of processing for a specific event. The processing associated with the occurrence of a particular event is effected by the group

of MILITRAN processing statements enclosed between either a PERMANENT EVENT statement or a CONTINGENT EVENT statement and an END statement. Therefore, the "event" itself may be described as the "event" statement, followed by a group of processing statements, followed by an END statement. The remaining statements enumerated above are control statements.

1. PERMANENT EVENT

A "PERMANENT EVENT" is one which occurs at regular intervals in time. The start of processing and the data associated with a "PERMANENT EVENT" is declared as follows.

PERMANENT EVENT N($(a_1,a_2,\ldots,a_i)$, d)

where N is both name of the event and the name of the list associated with the event; $a_1,a_2,\ldots,a_i$ are the list components, each of which is the name of an array and all of which are also associated with the name N, and d is the dimension associated with each component.

The entries in a list associated with a "PERMANENT EVENT" may be retrieved in the same manner as are entries in a vector. The rules for declaring the mode of a PERMANENT EVENT list and its components are also the same as those for declaring the mode of a vector and its

components. A PERMANENT EVENT does not require a list, in which case N is just the name of the event and the PERMANENT EVENT statement only defines the start of processing for the event named N.

2. CONTINGENT EVENT

A "CONTINGENT EVENT" is one which occurs at a critical juncture in time. In military simulations, the potential event is always associated with the time of occurrence, and usually has an attacking object, a target object, and other descriptive information associated with it.

The start of processing and the data associated with a "CONTINGENT EVENT" is declared as follows:

CONTINGENT EVENT N $((a_1,a_2,\ldots,a_1),d)$

where N is both the name of the event and the name of the list associated with the event; $a_1,a_2,\ldots,a_1$ are the list components, each of which is the name of an array and all of which are also associated with the name N, and d is the dimension associated with each component.

A "CONTINGENT EVENT" must have a list. However, this list may have, if desired, only one component. The rules for declaring the mode of a CONTINGENT EVENT are the same as those for declaring the mode of a vector

and its components.

The first component (or the only component), a_1, always represents the critical time at which the potential event may take place. This component must be of REAL mode.

The second component, a_2, if present, may be assigned the value of the attacking object, in which case, a_2 must be of PROGRAM OBJECT mode.

The third component, a_3, if present, may be assigned the value of the target object, in which case, a_3 must be of PROGRAM OBJECT mode.

The remaining components, if present, may be used for additional descriptive information and may be of any mode.

To facilitate the processing of a CONTINGENT EVENT, the MILITRAN processor provides four variables. These variables are automatically reserved by the processor; they do <u>not</u> have to be declared. They are:

1. INDEX which is of INTEGER mode.
2. TIME which is of REAL mode.
3. ATTACKER which is of PROGRAM OBJECT mode.
4. TARGET which is of PROGRAM OBJECT mode.

Transfer of control to a CONTINGENT EVENT statement is accomplished by a selection process described under "NEXT EVENT." This transfer of control implies that a specific entry in the contingent event list has been selected for processing. The position of the entry in the list is automatically stored in the variable INDEX. The corresponding list entry of the first component is automatically stored in the variable TIME. If a second component has been declared in the list, the corresponding value of this component is stored in ATTACKER. Likewise, if a third component has been declared, its corresponding value is stored in TARGET.

These variable names - INDEX, TIME, ATTACKER, TARGET - may be used to retrieve the entries in the list. For example, if an event is declared as:
CONTINGENT EVENT BLAST((MOMENT,LAUNCHER,RECEIVER),4)
and the second entry has been selected for processing, the value of INDEX would be 2 and the values of TIME, ATTACKER, TARGET would be automatically updated as though the following statements had been written:

```
TIME = BLAST(INDEX,1)
ATTACKER = BLAST(INDEX,2)
TARGET = BLAST(INDEX,3)
```

3. NEXT EVENT

The NEXT EVENT statement is a control statement. The various control statements, discussed previously, altered the process of sequential execution of the program either unconditionally or conditionally by first evaluating a logical expression and then transferring to a statement out of the normal sequence, dependent on the value of the logical expression. NEXT EVENT alters the sequential execution of the object program but the method used to determine the selection process for event processing is based on a "time" value.

The form of the NEXT EVENT statement is:

NEXT EVENT

The NEXT EVENT statement is used to start event processing and to proceed from one event to the next. The following rules determine the sequence for event processing:

1. When an object program contains both "Permanent Events" and "Contingent Events" -
    a. The first NEXT EVENT statement appearing outside the bounds of an "event" which is reached during the execution of the program, will transfer control to the first "Permanent Event" appearing in the source program. NEXT

EVENT statements appearing within the bounds of a "Permanent Event", cause control to be transferred to the next permanent event as it appears in the source program.

b. when the last "Permanent Event" has been executed, control will be transferred to the "Contingent Event" whose associated list contains in its "a_1" component (as described above) the smallest value for "time" equal to or greater than the variable TIME. When the "Contingent Event" has been selected, the position of the selected entry is placed in INDEX and TIME is set to a_1 (INDEX); ATTACKER to a_2(INDEX); and TARGET to a_3 (INDEX).

If two or more events are found whose time values are equal, the event selected will be the first one stated in the source program.

c. A NEXT EVENT statement appearing within the bounds of a "Contingent Event" causes control to be transferred to the first "Permanent Event", which will cause a repeat of the Event Cycle.

2. When the object program contains only "Contingent Events", selection occurs as explained under 1b above; and a NEXT EVENT statement appearing within the bounds of a "Contingent Event" causes control to be transferred to the next "Contingent Event".

Modifications of NEXT EVENT

The processing sequence for "events" may be modified by the use of the following form:

NEXT EVENT(n_1,n_2,...,n_i)

where each n is the name of an event, either "permanent" or "contingent".

If n_1,...,n_i includes both permanent and "contingent" events, control is transferred to the first n appearing in the parentheses which is a "permanent" event. If all n_1,...,n_i are "contingent" events, the event is selected by means of the "time" value as in 1b above. This form limits the selection of events to include only the events n_1,n_2,...,n_i.

NEXT EVENT EXCEPT(n_1,n_2,...,n_i) where each n is the name of an event, either "permanent" or "contingent".

This form limits the selection of events to exclude the events named n_1,...,n_i. If the event in

which the NEXT EVENT ($n_1,\ldots,n_i$) or NEXT EVENT EXCEPT statement is excluded from consideration, selection occurs as though the NEXT EVENT had occured outside the bounds of any event.

4. END

END

defines the end of processing for a specific event. When an END statement can be reached by the program logic, it is always interpreted as an unmodified NEXT EVENT.

5. END CONTINGENT EVENTS

END CONTINGENT EVENTS (S)

where S is a statement label, causes control to be transferred to S. However, an END CONTINGENT EVENTS statement does not transfer control at the time it is encountered; but acts as a modifier of all forms of the NEXT EVENT statement only if a NEXT EVENT is being processed which would normally transfer control to a "Contingent Event"; and no "Contingent Event" has a list entry whose time component is equal or greater than the current value of TIME. If these conditions exist control will be transferred to the statement labelled S.

An END CONTINGENT EVENTS statement may be placed anywhere in the source program and the location represented by S may be changed during the running of the object program by the use of several such statements.

CHAPTER VIII

PROCEDURES

Procedures are programs arranged to permit entry from other programs. The use of procedures allows the programmer to cause processing to occur at several points in his program without repeatedly specifying each statement necessary to effect that processing.

All MILITRAN procedures are compiled separately. This feature allows previously written programs to be used by a new program without recompiling.

Procedures may be "executed" in two ways: as "subroutines" or as "functions". Subroutines are entered by means of EXECUTE statements. Functions are entered by means of expressions. For example, the statement

```
EXECUTE SINE(THETA,RESULT)
```

causes the subroutine "SINE" to be entered. When execution of "SINE" is completed, control is returned to the statement following the EXECUTE statement. The statement

```
A = SIN(THETA) + COS(THETA)
```

causes functions "SIN" and "COS" to be entered during the

evaluation of the expression. Values returned from each function are used in the expression in place of the function names.

1. MILITRAN-Coded Procedures

Procedures may be coded in MILITRAN by using two special statements. They are:

PROCEDURE

RETURN

Procedures may be entered from MILITRAN programs by means of the statement

EXECUTE

The PROCEDURE statement is the entry point to a procedure and defines it to be such, as well as defining its name and arguments. EXECUTE causes control to be transferred to the specified subroutine and RETURN causes control to be transferred to the main program.

PROCEDURE

PROCEDURE n

or

PROCEDURE $n(a_1,a_2,\ldots,a_n)$

where n is the name of the procedure and each a_i, if

present, is a name denoting an argument of the procedure, causes the source program named "n" to be compiled as a relocatable subroutine.

The name of a procedure may have up to 60 characters, but the first six characters of a procedure name must be unique among all other procedure names. A subroutine introduced by the PROCEDURE statement must be a MILITRAN program and may contain any MILITRAN statements except another PROCEDURE statement.

The arguments in a PROCEDURE statement are dummy names which are replaced at the time of execution by the actual arguments supplied in the EXECUTE statement(see below). There must, therefore, be correspondence in number, mode, and order, between the two sets of arguments. Furthermore, when a dummy argument is an array, vector, or list name, the corresponding actual argument must also be an array, vector, or list name.

EXECUTE

EXECUTE $n(a_1,a_2,\ldots,a_j)$

where n is the name of a procedure and each a_1, if present, is any expression denoting an actual argument, transfers control to the procedure named "n" and stores the actual arguments in the dummy arguments of the procedure.

RETURN

RETURN a

causes control to be returned to the main program with the value of the expression "a" in the accumulator. If "a" is not present, the accumulator contents are unspecified.

There must be at least one RETURN statement in every procedure, and there may be more than one if a procedure can terminate at more than one point.

If the procedure is to be used as a "function", the form "RETURN a" must be used.

2. Library Functions

Procedures which are likely to occur in many programs have been pre-written and included in the MILITRAN library. All of these functions return values in REAL mode and require arguments of REAL mode.

The following library functions are available:

1. LOG (v)

   The natural logarithm of v is returned.

2. SIN(v), COS(v), TAN(v)

   These trigonometric functions return sin(v),

cos (v), and tan(v). The argument "v" is assumed to be in radians.

3. ATAN (v_1,v_2)

   The smallest positive angle (in radians) whose tangent is v_1/v_2 is returned. The signs of v_1 and v_2 are considered separately in order to select the proper quadrant.

4. SQRT(v)

   The value $\sqrt{|v|}$ is returned.

5. EXP(v)

   The value e^v is returned, where e is the base of natural logarithms.

6. RANDOM

   A pseudo-random number (x) is returned in REAL mode. $0 \leq x < 1$.

3. Open Functions

Up to this point we have been considering "closed" procedures, i.e., programs which may be entered repeatedly with different arguments. The MILITRAN language also provides several "open" functions. Open functions cause coding sections to be inserted in the program each time they appear.

The following open functions are provided in MILITRAN. All arguments (v) must be REAL or INTEGER in mode.

1. MOD(v_1,v_2)

   The value v_1 mod v_2 is returned in the mode of v_1. Arguments (v) must be of the same mode.

2. INTEGER(v)

   The value v is returned in INTEGER mode. Any fractional part of v is truncated.

3. REAL(v)

   The value v is returned in REAL mode. If v is of integer mode it may not exceed $2^{27} - 1$.

4. SIGN(v_1,v_2)

   The value v_1 is returned in the mode of v_1 and bearing the sign of v_2.

5. MIN(v_1,v_2,...,v_n)

   MAX(v_1,v_2,...,v_n)

   The minimum or maximum value among all values v_1,v_2,...,v_n is returned in the mode of v_1. All arguments (v) must be of the same mode.

6. EPSILON(v)

   The value $v + \epsilon$ is returned in the mode of v. The increment ϵ is the smallest increment of v which can be expressed within the computer.

7. ABS(v)

   The value $|v|$ is returned in the mode of v.

CHAPTER IX

INPUT AND OUTPUT STATEMENTS

1. Introduction

The input-output statements are used to control the flow of data into and out of the computer. The statements are used to specify the kind and amount of data to be transmitted, the tape units to be used for the reading or writing of the data, and the format of the external data representation.

The input-output statements are:

FORMAT

READ

WRITE

READWRITE

BINARY READ

BINARY WRITE

BACKSPACE

BACKSPACE FILE

END FILE

REWIND

UNLOAD

The READ, WRITE, and READWRITE statements cause the transmission of BCD data between the tape units and computer storage. The linkage between the internally stored data and its external representation is accomplished by specifying in the input or output statement the statement label of a FORMAT statement. The FORMAT statement contains a set of editing codes which enable the translation of internal representation to external representation and vice versa. BINARY READ and BINARY WRITE are used where work tapes or other internal extensions are desired. The remaining input-output statements - - BACKSPACE, BACKSPACE FILE, END FILE, REWIND, UNLOAD - - are magnetic tape control statements.

2. Input-Output Lists

Five of the input-output statements call for the transmission of data. They are READ, WRITE, READ-WRITE,BINARY READ, and BINARY WRITE, and each of these statements may have, as one of its components, a list of variable names. These variable names refer to the space in the computer memory reserved for the storage of variable data. Therefore, the function of the input-output list is to provide the data transmission statement with the information necessary to either store (for input) or retrieve (for output) the data. When the input or output

command is executed, the program references the list to determine where the data is to be stored or where it is to be obtained. The input-output list is ordered, and its order must be the same as the order in which the data fields appear on tape(for input) or will exist (for output).

For example, if an input tape contains data in the following sequence:

10 72 63 5 12 474 322

and this tape is read in by an input statement whose associated list is:

A,B,C,D, SPEED, RANGE, DIST

the data will be read in and stored in the location specified by the variable names, as though the following statements had been written:

```
A = 10
B = 72
C = 63
D =  5
SPEED = 12
RANGE = 474
DIST = 322
```

Similarly, if this list is used with an output statement, the data will appear on the output medium in the order specified by the list.

Integer and real quantities, and logical and object elements may be specified by the same input-output list, but each item in the list must be separated from the succeeding item by a comma.

Any number of quantities may appear in a single list. However, if a record contains more quantities to be transmitted than there are variable names in the list, only the number of quantities specified in the list are transmitted, and the remaining quantities are lost. Thus, if a record contains three quantities and a list contains two, the third quantity is lost. Conversely, if a list contains more quantities than the record, succeeding records will be read or written until all the items specified in the list have been transmitted.

The items specified in a list may each take one of several forms. The simplest is a single variable name, as in:

A, B, C, D, SPEED, RANGE, DIST

A list may also contain subscripted variable names, if the subscripts have been previously defined,

as in:

A,B(6), C(3,4), D(A)

In the above example, A may be used as a subscript for D because A itself appears earlier in the list and will have assumed a value by the time the data represented by D(A) is encountered.

When input-output of an array or vector is desired, an implied DO-loop either of Form 1 or Form 2 (nested to any depth) may be used for the input-output list. The vector or array is specified as a subscripted variable name. For example, the input-output list which might be associated with a one dimensional array could be written as follows:

((A(I))UNTIL I.G.10, I=1,1)

The above input-output list would cause 10 items of data to be transmitted in the following sequence:

A(1) A(2) A(3) . . . . . A(9) A(10)

When the list is encountered, the following would occur:

1. I is set to 1.
2. The logical expression I.G.10 is evaluated, and since it is FALSE, the first item of data - A(1) - is transmitted.

3. I is incremented by 1, the logical expression is evaluated again and the transmission of data - A(2) A(3) . . . . A(10) - continues until I assumes a value of 11.

The next list might be used with a two dimensional array which has 6 components.

(((A(I,J))UNTIL I.G.3,I=1,1)UNTIL J.G.2,J=1,1)

The data associated with this list would be transmitted as follows:

A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)

When the list is encountered, execution occurs as follows:

1. J is set to 1

2. I is set to 1

3. The logical expression - J.G.2 is evaluated and, since it is FALSE, the expression I.G.3 is evaluated. Since this too is FALSE, the first item of data - A(1,1) - is transmitted.

4. I is incremented by 1; I.G.3 is again evaluated, and the transmission of data continues until the terminating condition is reached when I assumes a value of 4. At this point the first column of the array - A(1,1) A(2,1) A(3,1) will have been transmitted.

5. J is incremented by 1

6. I is set to one

7. J.G.2 is evaluated and since it is still FALSE, I.G.3 is evaluated. At this point, this expression is FALSE and the second column of data is transmitted - A(1,2) A(2,2) A(3,2). When the terminating condition for I.G.3 has been met, J is again incremented and now assumes a value of 3 which is the terminating condition for the expression J.G.2. The loop is satisfied and the next sequential instruction will be executed.

As mentioned above, an implied DO-loop of Form 2 may be used in an input-output list. This form of an implied DO-loop will transmit a value or series of values which are to be associated with object elements. The general form of this type of list is:

((A) FOR B.IN.C)

where A is a subscripted or nonsubscripted variable which has been declared in a REAL, INTEGER, PROGRAM OBJECT, or LOGICAL declaration (or by means of the NORMAL MODE declaration); B is a single variable which has been declared in a PROGRAM OBJECT declaration; and C is either the name of a class as specified in a CLASS declaration or the name of a group of objects as specified in an OBJECT declaration.

This form of input-output list will cause transmission of one item of data for each member of "C". When the first item is transmitted, variable B is set to the identity of the first member of C. For succeeding transmissions, "B" assumes the identity of all members of "C". Transmission of data ceases when all members of "C" have been covered sequentially.

As an example, consider the following input-output list:

```
((RANGE(AC)) FOR AC.IN.AIRCRAFT)
```

If AIRCRAFT had been declared an object element with a cardinality of 10 as in:

```
OBJECT AIRCRAFT(10)
```

and RANGE had been declared an integer array with a dimension equal to the cardinality of AIRCRAFT as in:

```
INTEGER RANGE(AIRCRAFT)
```

the input-output list would provide the data transmission statement with the location for the storage or retrieval of the ten quantities which would be the ranges of each of the ten aircraft.

As the above DO-type input-output list indicates, the range of the implied DO list must be clearly defined by

means of parentheses. Both the data and the loop itself must be in parenthesis. The following example illustrates a more complicated list which utilizes implied DO-loops.

```
A,C(2),((J,((E(J,I))I.G.2,I=1,1))FOR U.IN.V)
```

3. FORMAT

The manner in which data is represented in the machine differs from the external representation of the same information. As a result, it is necessary to specify the type of editing to be performed on the information that is being transmitted. This is accomplished by means of a FORMAT statement which may be referenced by a READ, WRITE, or READWRITE statement. The FORMAT statement is a nonexecutable statement which specifies the type of conversion to be performed upon each member of a list. The FORMAT statement also indicates where the data is located on the input record or is to be located on the output record.

The general form of the FORMAT statement is:

FORMAT (Format Specification)

where Format Specification refers to any of the following editing codes:

I, E, F, O, J, L, A, H, X

If more than one editing code appears in a FORMAT specification, it must be separated from the preceding editing code by a comma, with the exception of X and H type codes.

Conversion of Numeric Data

There are four editing codes which control the conversion of numeric data

Internal Representation	Editing Codes	External Representation
Integer	Iw	Decimal Integer
Real Point	Ew .d	Floating Point - with Exponent
Real Point	Fw .d	Fixed Point - without Exponent
Binary Number	Ow	Octal Integer

In the table above, w and d are unsigned integer constants; w represents the width of a given field and d specifies the number of decimal places to the right of the decimal point. If w is greater than required to contain a given number, the contents of the field are right justified, and the leftmost spaces are filled in with zeros (for data being input) or blanks (for data being output).

I Conversion

Numbers under control of an I type editing code are interpreted as decimal integers. For example, typical input or output might be:

16

-97

2482

The numbers are right justified in the specified field and if the number to be converted has more digits than are specified the Format specification, the excess digits are dropped from the left.

The following examples illustrate how each of the integers on the left would be written externally if a specification of I4 had been given: (b represents a blank)

Internal	External
2462	2462
12	bb12
-137	-137
-6841	6841
86321	6321

F Conversion

Numbers under control of an F type editing code are interpreted as fixed point decimal numbers. Typical input or output might be :

16.7

-21.2

174.3

-269.4

.3

-0.06

346.

The general editing code for an F type conversion is Fw. d, where w is the width of the entire field, including a space for the decimal point and a space for the sign, and d is the fractional portion. If insufficient spaces are designated for the fractional portion, it is rounded; zeros are filled in from the right if excessive spaces are reserved. The integer portion is handled in the same manner as numbers converted by the I type conversion.

The following examples show how each of the numbers on the left would be written externally if a specification of F 6.2 had been given.

Internal	External
47.25	b47.25
123.62	123.62
-123.62	123.62
4725.634	725.63
-.1	b-0.10
-6.	b-6.00

E Conversion

Numbers under control of an E type editing code are interpreted as a decimal fraction to a power of 10. The first significant digit appears to the right of the decimal point. Typical input or output might be

0.1431E03

0.20E-02

0.671E02

The general editing code for an E type conversion is Ew. d, where d is the fractional portion and w is the width of the entire field including a space for the decimal point, a space for the sign, and four spaces for the exponent. As with the F type conversion, if insufficient spaces are designated for the fractional portion, it is rounded; zeros are filled in from the right if excessive spaces are reserved.

The following examples show how each of the numbers on the left would be written externally if a specification of E 11.3 had been given.

Internal	External
167.	0.167Eb03
-43.1	-0.431Eb02
.006	0.600E-02
-.00000002	-0.200E-07
-34.0062	-0.340Eb02

Ø Conversion

Numbers under control of the Ø type editing code are interpreted as octal integers. The eight digits used are 0 through 7.

The Ø type editing code causes the numbers to be right justified in the specified field and will cause excess digits to be dropped from the left if the number to be converted has more digits than are designated by the Format specification. Data to be converted by an Ø type specification may be given variable names of any mode.

Conversion of nonnumeric data

MILITRAN provides five specifications for the processing of nonnumeric data.

Editing Codes	Type of Data
Jw.d	Object Elements
Lw	Logical Elements
Aw	Alphameric Data
wH	Alphameric Data
wX	Blank Field Specification

J Specification

The J specification is used in conjunction with the input-output of object elements. Typical input or output might be

SHIP(25)
CAR(54)
TANK(100)
PLANE

There are two forms of the J editing code - Jw and Jw.d. The Jw form causes a nonsubscripted name of an object element to be right justified in the specified field. w represents the width of the entire field. If insufficient spaces are designated, the field is truncated from the left; blanks are filled in from the left if excessive spaces are reserved.

For example, if a single object element existed in the computer in the following form:

MILITARYINSTALLATION

and an output statement was written in conjunction with the format specification

FORMAT J10

the following characters would be outputted:

STALLATION

Since insufficient spaces had been allocated, the name of the object element would be truncated from the left. The statement

FORMAT J20

would permit the outputting of the entire name.

The Jw.d editing code permits the specification of a subscript. In this case, w is again the width of the entire field but it must include two spaces for parenthesis. d represents the number of digits in the subscript. The placement of data within the w and d fields is handled in the same manner as with the Jw form. Both fields are right justified and the field is truncated from the left if

insufficient spaces are specified and blanks are filled in from the left if extra spaces are reserved.

The following examples show how the object element "BOMB(1752)" would be outputted according to the specification given.

Specification	Output
J10.4	BOMB(1752)
J8.3	OMB(752)
J11.5	BOMB(b1752)
J10.3	bBOMB(752)

L Specification

The L specification is available for the input-output of the logical values, TRUE or FALSE. The format of the L specification is Lw, where w is the width of the entire field.

The effect of Lw depends on whether it is used with an input or output statement. If used with an input statement, the field represented by w must begin with a T or an F in order for the variable to assume a value of TRUE or FALSE. For example, the specification L10 could be used to input value TOMJONESJR. Since this value begins with T, the variable designated in the list of the

input statement would assume a value of TRUE. If the value FREDJJONES had been inputted, the variable would assume a value of FALSE. Since the w field is truncated from the left if insufficient spaces are specified, an incorrect value may be transmitted.

For output, one character - either T or F is transmitted. It is right justified and the remaining spaces designated by w are filled in with blanks.

A Specification

The Aw specification is one of the editing codes used to process alphameric data. This specification causes w BCD characters to be transmitted to or from a variable or an array name. Since this information can be referred to by name, it can be processed or modified by the program. The data is right justified in the specified field.

When the Aw editing code is used, six bit alphameric character codes are used for internal representation. A single variable may contain up to six alphameric characters.

For example, if a variable named Y is declared; and the format specification

FORMAT (A6)

is associated with a list in which the variable Y appears, this format statement could be used to control the transmission of alphameric data with 6 or less characters such as:

```
REPORT
TIME
HOURS
B =
```

Alphameric information with more than six characters can be contained in an array.

H Specification

The H specification is also used in conjunction with alphameric information. Alphameric data associated with the specification wH is generally used for header or fixed alphameric fields and cannot be manipulated by the program. The editing code wH must be followed by w alphameric characters. Blanks appearing in the w characters following the editing code wH are counted as characters.

For example:

```
38HbTHISbISbANbEXAMPLEbOFbALPHAMERICbDATA
```

If the H editing code is used with an input statement, w characters are extracted from the input record

and replace the w characters included with the format specification. On the other hand, if the H editing code is used with an output statement, the w characters following the specification (or the characters which replaced them as a result of input operations) are written as part of the output record.

For example, if an output statement refers to the following format specification:

```
FORMAT(17HDAY/NIGHTbROUTINE)
```

the output would appear as:

```
DAY/NIGHT ROUTINE
```

However, if an input statement whose list is "AIRCRAFT" is used to read the following information:

```
bPLANESbb25
```

and the input statement is associated with a format statement such as:

```
FORMAT(7HBOMBERSI4)
```

a subsequent output command whose list is also "AIRCRAFT" associated with the same format statement would produce the following output:

```
bPLANESbb25
```

X Specification

The X specification is used to enter blank characters in an output record or to skip characters in an input record. The editing code is wX where w is the number of blank characters to be inserted or the number of characters to be ignored.

For example, if an input record contains two 4 digit fields for integers and it is desired to read only the second quantity, the first field could be skipped by using the statement

```
FORMAT(4X4I)
```

in conjunction with the input statement. Or, the statement

FORMAT(3XF4.2) could produce the output bbb0.75.

Repetition of Editing Codes

If two or more successive elements in an input-output list are to be edited in the same manner, and these elements are contained in a single record, the editing code to be repeated may be written only once if the editing code is prefixed by an unsigned integer constant to indicate the number of repetitions desired.

For example, if a single record contained three successive integer items, each of which contained 10 characters, the format statement could be written as:

```
FORMAT(3I10)
```

instead of:

```
FORMAT(I10, I10, I10)
```

NOTE: It is not possible to repeat the wH or wX specifications by this method.

Repetition of Groups of Editing Codes

If two or more successive groups of elements in an input-output list are to be edited in the same manner, and these elements are contained in a single record, the group of editing codes to be repeated may be written only once if the group is enclosed in parentheses and then prefixed by an unsigned integer constant to indicate the number of times repetition of the group is desired.

For example, if a single record consists of seven successive fields which appear in the following sequence:

1. fixed point decimal number
2. subscripted object element
3. fixed point decimal number
4. subscripted object element

5. integer number
6. integer number
7. integer number

the format statement could be written as:

```
FORMAT(2(F6.2, J11.3), 3I4)
```

instead of:

```
FORMAT(F6.2, J11.3, F6.2, J11.3, I4, I4, I4)
```

NOTE: An additional level of parentheses is <u>not</u> permitted.

<u>The FORMAT Specification Scan</u>

When an input-output statement is executed, control switches back and forth between the input-output list and the FORMAT specification. Scanning of the FORMAT statement occurs from left to right. Successive items in the input-output list are transmitted and edited by successive corresponding specifications in the FORMAT statement until all items in the list are transmitted and edited.

If the input-output list contains more items than there are corresponding specifications in the FORMAT statement, the scan is resumed beginning with the preceding left parentheses of the FORMAT statement.

For example, if an input-output statement has the following list:

A, B, C, D, E, F, G, H

and it refers to the FORMAT statement:

FORMAT(I6, J8.3, L4)

then the variables in the list will be converted as follows:

A	I6
B	J8.3
C	L4
D	I6
E	J8.3
F	L4
G	I6
H	J8.3

Multiple Record Format Specification

It is possible for one FORMAT statement to contain editing instructions for more than one record. If a group of editing codes is followed by a slash (/), the editing codes following the slash are used to edit the next record. A slash also permits the object program to ignore n records on input or to transmit n blank records on output simply by writing n + 1 consecutive slashes.

For example, if the following FORMAT statement:

```
FORMAT(3F11.3, I7/2F10.4, I3)
```

is used with an input statement, each input command could cause the transmission of two records if the associated input-output list consists of seven variables. The first three items of the first record would be interpreted as fixed point decimal numbers and the fourth as an integer. The slash causes the next record to be read, with the first two items interpreted as fixed point decimals and the third as an integer. If there are items on the input record following those converted by codes I7 or I3, they would be ignored.

If the same FORMAT statement is used with an output statement, each output command would cause the transmission of two records. This would consequently result in two lines of print, the slash indicating the beginning of a new line.

NOTE: If this statement is used with an input-output list consisting of less than five variables, only one record would be transmitted.

The statement:

```
FORMAT(3F11.3, I7/////2F 10.4, I3)
```

used with an input statement, would cause six records to be read. However, only the first and last would be processed; the remaining four records would be ignored.

If used with an output statement, four blank records would be written between the first record which contains data and the last. If the record were printed, four lines would be skipped before the second line were printed.

Relationship between Input-Output List and Format Statement

As shown above, the input-output list specifies the location in computer storage that data is to be placed (for input) or obtained (for output). Whereas, the FORMAT statement in addition to its conversion function, specifies where the data is located on the input record or where it is to be placed on the output record.

For example, if it is desired to read the following data:

```
  12
 475
-122
   1
 67.001
  0.675
-124.012
```

under control of an input-output list such as:

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

which is associated with FORMAT statement:

FORMAT(4I4, 3F9.3)

the fields of data must be right justified in each field on the input record as follows:

bb12	b475	-122	bbb1	bbb67.001	bbb0.675	b-124.012
I4	I4	I4	I4	F9.3	F9.3	F9.3

The variables in the input-output list would be edited according to the conversion code specified, as follows:

Variable	Conversion Code	Value Assumed
A	I4	12
B	I4	475
C	I4	-122
D	I4	1
E	F9.3	67.001
F	F9.3	0.675
G	F9.3	-124.012

Scale Factors

MILITRAN allows the application of a scale factor to the E and F type conversion codes. The scale factor is indicated by prefixing the editing code with a signed integer constant followed by the letter P. The action of the scale factor is such that:

External Representation =

Internal Representation X10$^{\text{Scale Factor}}$

The use of a scale factor with an F type editing code causes the output values to be multiplied by 10^P before output.

For example:

FORMAT(1P3F11.3) might give

bbb-806.540bbbbb-0.686bbbbbb6.636

FORMAT(-1P3F11.3) would give

bbbbb-8.065bbbbb-0.007bbbbbb0.066

The use of a scale factor with an E type editing code causes the mantissas of the output values to be multiplied by 10^P and the exponents to be decreased by P. Thus:

FORMAT(1P3E11.3) would give

b-8.065Eb01b-6.860E-02bb6.636E-01

Once a value has been given for a scale factor within a FORMAT statement, it will be applied to all values edited by succeeding E and F type editing codes within the same FORMAT statement. This is true for both single-record

and multiple-record FORMAT statements. To terminate the effect of a scale factor within a specific FORMAT statement, a subsequent scale factor of 0 must be given. Scale factors have no effect on I type editing codes.

4. READ

The READ statement is written as follows:

READ (t,s) List

where t designates an I/O unit, s is the statement label of a FORMAT statement, and List is an input-output list of the quantities to be transmitted.

The READ statement causes the object program to read BCD information from I/O unit t into the locations specified by the List, under control of the FORMAT statement specified by s. If t is designated as CARDS, data will be read on-line from the card reader.

Examples:

```
READ (4,A100) A, B, C, D, E
READ (INPUT, FORMAT1) DISTANCE, SPEED
READ (CARDS, FORMAT2) TIME
```

5. WRITE

The WRITE statement is written as follows:

WRITE (t,s) List

where t designates an I/O unit, s is the statement label of a FORMAT statement, and List is an input-output list of the quantities to be transmitted.

The WRITE statement causes the object program to write BCD information on unit t, from the locations specified by the List, under control of the FORMAT statement specified by s. If t is designated as CARDS, the items in the list will be punched on the on-line card punch. If t is designated as PRINTER, the items in the list will be printed on the on-line printer.

Examples:

```
WRITE (1,EDIT) ((A(I)) UNTIL I.G. 10, I = 1,1)
WRITE (OUTPUT, CONVERT) TARGET, RANGE (3,4)
WRITE (CARDS, FORMAT1) A, B, C, D
WRITE (PRINTER, FORMAT2) A, B, C, D
```

6. READWRITE

The form of the READWRITE statement is:

READWRITE (t_1, s_1, t_2, s_2) List

where t_1 and t_2 designate I/O units, s_1 and s_2 are the statement labels of FORMAT statements, and List is an input-output list of the quantities to be transmitted.

The READWRITE statement causes the object program to read BCD information from unit t_1 into the locations specified by the List under control of the FORMAT statement specified by s_1 and then to immediately write the same information on I/O unit t_2 under the control of the FORMAT statement specified by s_2. t_1 may be designated as CARDS. t_2 may be designated as CARDS or PRINTER.

Example:

```
READWRITE (IN,FORM1,OUT,FORM2)(J,((E(I,J))UNTIL I.G.2,I=1,1))
READWRITE (CARDS,FORM1,PRINTER,FORM2)PRESSURE,D
```

7. BINARY READ

BINARY READ is written as follows:

BINARY READ (t) List

where t is an arithmetic expression designating a tape unit and List is the input-output list of the quantities to be transmitted.

The BINARY READ statement causes the object program to read binary information from the tape unit specified by t into the locations specified by List.

Examples:

```
BINARYREAD(4) VELOCITY
BINARYREAD(BIN) LONG, LAT
```

8. BINARY WRITE

The form of BINARY WRITE is:

BINARY WRITE (t) List

where t is an arithmetic expression designating a tape unit and List is the input-output list of the quantities to be transmitted.

The BINARY WRITE statement causes the object program to write binary information on the tape unit specified by t from the locations specified by List.

Examples:

```
BINARY WRITE(B) VELOCITY

BINARY WRITE(BTAPE) LONG, LAT
```

9. END FILE RETURN

When an end-of-file is encountered on an input tape, control is passed to the system monitor program. If it is desired to prevent this, the following statement may be given:

END FILE RETURN (s)

where s is the label of the statement to which control is to be transferred.

10. END RECORD RETURN

If the end of a logical record is encountered before a BINARY READ has been completed (before all the elements in a list have been transmitted), the following statement:

END RECORD RETURN (s)

will cause control to be transferred to the statement labelled s.

11. TAPE CONTROL STATEMENTS

The tape control statements are used to position magnetic tapes. A component of each of the statements is an arithmetic expression which designates the tape unit on which the specified operation is to be performed. Usually, this expression would take the form of an integer constant or an integer variable. However, if floating point quantities are specified, the expression will be truncated to an integer value before use.

BACKSPACE

BACKSPACE(t)

where t is an arithmetic expression, causes the object program to position the tape mounted on tape unit t at the

beginning of the record last read or written. For BCD tapes, the tape is moved backward one physical record; for binary tapes, it is moved one logical record.

BACKSPACE FILE

BACKSPACE FILE (t)

where t is an arithmetic expression, causes the object program to move the tape mounted on tape unit t backward until an end-of-file record or the load point is encountered.

END FILE

END FILE (t)

where t is an arithmetic expression, causes the object program to write an end-of-file mark on the tape mounted on tape unit t.

REWIND

REWIND (t)

where t is an arithmetic expression, causes the object program to rewind to the load point the tape mounted on tape unit t.

UNLOAD

UNLOAD (t)

where t is an arithmetic expression, causes the object program to rewind and put the tape mounted on tape unit t into an automatic unload status.

APPENDIX

Environment Declarations

REAL $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

INTEGER $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

LOGICAL $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

OBJECT $n_1(i_1),n_2(i_2),\ldots,n_m(i_m)$

PROGRAM OBJECT $n_1(i_1,i_2,\ldots,i_k),\ldots,n_m(i_1,i_2,\ldots,i_j)$

CLASS (c) CONTAINS $a_1,a_2,\ldots,a_m$

NORMAL MODE $m_1(a_1,a_2,\ldots,a_k),m_2(b_1,b_2,\ldots,b_r)$

VECTOR N $((a_1,a_2,\ldots,a_1),d_1,d_2,\ldots,d_1)$

COMMON $n_1,n_2,\ldots,n_1$

Arithmetic

A = B

Logical

A = B

Control Statements

GO TO s

PAUSE j

STOP

IF (b) s_t,s_f

UNLESS (b) s_f,s_t

DO (s) UNTIL b, n = e_1,e_2

DO (s) FOR a.IN.b

CONTINUE

List Processing Statements

LIST n((c_1,c_2,...,c_1),d)

LENGTH (n)

RESET LENGTH (n) to p

PLACE (e_1,e_2,...,e_1) IN n

REMOVE ENTRY n(k)

PLACE ENTRY m(j) IN n

REPLACE ENTRY n(k) BY (e_1,e_2,...,e_1)

REPLACE ENTRY n(k) BY ENTRY m(j)

REMOVE (b_1,b_2,...,b_1) FROM n

REPLACE (b_1,b_2,...,b_1) BY (e_1,e_2,...,e_1) IN n

REPLACE (b_1,b_2,...,b_1) BY ENTRY m(j) IN n

MINIMUM INDEX (n(b_1,b_2,...,b_1),s)

RANDOM INDEX (n(b_1,b_2,...,b_1),s)

RANDOM INDEX (n(b_1,b_2,...,b_1,b_x),s)

LST

GST

Event Statements

PERMANANT EVENT N($(a_1,a_2,\ldots,a_i)$,d)

CONTINGENT EVENT N($(a_1,a_2,\ldots,a_i)$,d)

NEXT EVENT

NEXT EVENT $(n_1,n_2,\ldots,n_i)$

NEXT EVENT EXCEPT $(n_1,n_2,\ldots,n_i)$

END

END CONTINGENT EVENTS (s)

Procedure Statements

PROCEDURE n

PROCEDURE n$(a_1,a_2,\ldots,a_n)$

EXECUTE n

EXECUTE n $(a_1,a_2,\ldots,a_n)$

RETURN

RETURN a

Input-Output Statements

FORMAT (Format Specification)

READ (t,s) List

WRITE (t,s) List

READWRITE (t_1,s_1,t_2,s_2) List

BINARY READ (t) List

BINARY WRITE (t) List

END FILE RETURN (s)

END RECORD RETURN (s)

BACKSPACE (t)

BACKSPACE FILE (t)

END FILE (t)

REWIND (t)

UNLOAD (t)

INDEX

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Systems Research Group, Inc.
1501 Franklin Avenue
Mineola, Long Island, N. Y.

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

MILITRAN PROGRAMMING MANUAL

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical report

5. AUTHOR(S) *(Last name, first name, initial)*

Systems Research Group, Inc.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1964	197	

8a. CONTRACT OR GRANT NO.
Nonr 2936(00)

b. PROJECT NO.
Navy NR 276-001

c. AF Proj. 2801,

d. Task 280101

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
USAF Technical Documentary Report No. ESD-TDR-64-320

10. AVAILABILITY/LIMITATION NOTICES
Qualified requesters may obtain copies of this report from the Defense Documentation Center (DDC)

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Office of Naval Research, Wash.D.C. & Electronic Systems Division, Air Force Systems Command, Bedford, Mass.

13. ABSTRACT

MILITRAN is an algorithmic computer language specifically oriented to the problems encountered in simulation programming. In addition to providing overall flexibility in expressing complex procedures, the language contains features which greatly simplify the maintainence of status lists, handling of numeric and non-numeric data, and sequencing of events in simulated time.

DD FORM 1 JAN 64 1473

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Militran Language Simulation Computers Programming Languages Data Processing Systems Information Retrieval Instruction Manuals Compiler Systems Analysis War Gaming						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.