UNIVAC®

1108

MULTI-PROCESSOR SYSTEM

EXTENDED ALGOL

PROGRAMMERS REFERENCE

UP-7636

This manual is published by the Univac Division of Sperry Rand Corporation in loose leaf format. This format provides a rapid and complete means of keeping recipients apprised of UNIVAC® Systems developments. The information presented herein may not reflect the current status of the programming effort. For the current status of the programming, contact your local Univac Representative.

The Univac Division will issue updating packages, utilizing primarily a page-for-page or unit replacement technique. Such issuance will provide notification of software changes and refinements. The Univac Division reserves the right to make such additions, corrections, and/or deletions as, in the judgment of the Univac Division, are required by the development of its Systems.

PRINTED IN U.S.A

PREFACE

This is a detailed programmer's reference manual for Univac 1108 Extended ALGOL.

It is divided into three main logical areas: structure and use of the language, operating environment and systems control, and appendices covering special topics.

In the text of this manual Univac 1108 Extended ALGOL reserved words, when used as reserved words, appear in bold face type; metalinguistic variables are italicized. If a reserved word is part of a metalinguistic variable and is being used as the metalinguistic variable, then it will be italicized and not boldfaced. If the reserved word appears in text but has no relation to the ALGOL reserved word, it will not be boldfaced.

The reader is assumed to have programming experience and a familiarity with the 1108 Operating System. For those unfamiliar with ALGOL 60 and/or the 1108 Multiprocessing Operating System, the following publications are suggested as references:

1. Univac 1108 Operating System Programmers Reference Manual.
2. McCracken, Daniel D., *An Introduction to ALGOL Programming* (New York: John Wiley and Sons, 1962).
3. Dijkstra, E.W., *A Primer of ALGOL 60 Programming* (London and New York: Academic Press, 1962).
4. Naur, P., et al., *Revised Report on the Algorithmic Language ALGOL 60* (Communications of the Association for Computing Machinery, Vol. 6, No. 1, Jan., 1963).

Where it is possible, to assure adherence, the language of the Revised Report (ref. 4) has been used verbatim.

CONTENTS

1. INTRODUCTION TO 1108 EXTENDED ALGOL

A syntactic, semantic, and pragmatic description of the UNIVAC 1108 Extended ALGOL Language is delineated in this manual.

The basis for this language is the "Revised Report on the Algorithmic Language, ALGOL 60". The language, ALGOL 60, can be considered to be a subset of UNIVAC 1108 Extended ALGOL. The subset language has been expanded to include extensive input/output device communication under Exec 8, provision for formatting of data (Section 11.2.4), and the capability of sorting and merging data by use of source language statements that interface with the 1108 SORT/MERGE package. (Section 13) In addition, source language statements are available to allow the programmer to utilize the powerful multiprogramming capability of the 1108 system. Specifically, these statements permit a synchronous processing of procedures, the capability to test the status of independent events, and the facility to resume synchronous processing (Section 12). Program debugging is enhanced by the inclusion of statements to display a variable and its contents when its value is changed or to conditionally display a variable at any point in the program (Section 9.9). An option is available to generate an automatic flow-chart of the source language.

A partial word designation may be indicated for both bit and character fields of the value of a variable or the result of an expression. Also, any field of a variable word may be modified (Section 4.5).

Segmentation or paging of arrays has been implemented (Section 9.6) so that multi-dimensioned arrays of any size may be declared.

UNIVAC 1108 Extended ALGOL deviates from ALGOL 60 in the following areas:

A set of reserved identifiers has been defined (see Appendix B) to enable a more efficient and faster translation.

The compiler for UNIVAC 1108 Extended ALGOL is essentially a single pass translator; it requires, therefore, that all labels be declared in the top of the block in which they appear and that all forward references to procedures or switches be declared at the top of the block in which their declaration appears. These requirements significantly enhance translation time.

The syntax of the ALGOL language is described in terms of metalinguistic symbols and variables combined to form metalinguistic formulae.

The metalinguistic symbols as used here have the following interpretation:

< > brackets are used to contain the sequences of characters representing a metalinguistic variable

: : = a metalinguistic connective meaning "is defined as" and separates the metalinguistic variable on the left from its respective metalinguistic formula on the right.

| means "or" and separates the multiple definitions of a metalinguistic formula from one another.

Metalinguistic variables are sequences of characters enclosed in brackets (< >) which define a set of things.

Metalinguistic formulae are the statements of the rules of syntax (the grammar) for the ALGOL language. Consider as an example the simple language called "add, expression". The alphabet of "add expression" consists of the objects A, B, C, D, +.

The grammar for the language follows:

<variable>::= A|B|C|D

<expression> :: = <variable> | <expression> + <variable>

Note the recursive definition implied in the metalinguistic formula called *expression*.

If one were to parse the good expression A + B + C + D in this language, it would be as follows:

A + B + C + D
exp 1
exp 2
exp 3
exp 4

A set of appendices include a complete presentation of error diagnostics, reserved words, and metalinguistic variables.

2. STRUCTURE OF THE LANGUAGE

2.1. BASIC COMPONENTS

2.1.1. Letters

2.1.1.1. Syntax

<letter> : : = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

2.1.1.2. Semantics

All letters of the alphabet are included in the valid character set for UNIVAC 1108 Extended ALGOL. *Letters* do not have individual meanings; they are used singly or in combination for forming *identifiers* and *strings*.

2.1.1.3. Examples

- Single characters as *identifiers*

```
A : = B + C  DIV 2
```

- Multiple character *identifiers*

```
RESULT
COS
PHIL
```

- *String*

```
'THIS IS AN ALPHA STRING'
```

2.1.2. Digits

2.1.2.1. Syntax

<digit> : : = 0|1|2|3|4|5|6|7|8|9

<octal digit> : : = 0|1|2|3|4|5|6|7

2.1.2.2. Semantics

All numbers are included in the valid character set for UNIVAC 1108 Extended ALGOL. *Digits* are used for forming *numbers*, *identifiers*, and *strings*.

2.1.2.3. Restrictions

Only zero thru seven are valid for octal representations.

2.1.2.4. Examples

- integer numbers — 1, 129, 100000
- real numbers — 31.7, 7.0E2
- identifiers — TOTAL, A2, RESULT10, SWT200
- octal numbers — 6, 137, 643

2.1.3. Delimiters

2.1.3.1. Syntax

<delimiter> : : = <operator> | <separator> | <bracket> | <declarator> | <specificator>

<operator> : : = <arithmetic operator> | <relational operator>

| <logical operator> | <sequential operator>

<arithmetic operator> : : = + | − | * | / | **DIV** | ** | **MOD**

<relational operator> : : = < | = | > | **LSS** | **LEQ** | **EQL** | **GEQ** | **GTR** | **NEQ**

<logical operator> : : = **EQV** | **IMP** | **OR** | **XOR** | **AND** | **NOT**

<sequential operator> : : = **GO TO** | **IF** | **THEN** | **ELSE** | **FOR** | **DO** | **READ** | **WRITE** | **SPACE** | **REWIND** | **LOCK** | **CLOSE** | **FILL** | **ZIP** | **ON** | **EXECUTE** | **EVENT** | **WAIT** | **DELETE** | **SORT** | **MERGE** | **RELEASE** | **SAVE** | **PAGE** | **DBL** | **NO** | **PURGE** | **UNTIL** | **WHILE** | **STEP** | **REVERSE**

<separator> : : = , | ; | : | : = | & | **COMMENT** | % | <single space>

<bracket> : : = (|) | [|] | **BEGIN** | **END** | # | .[| ,[

<declarator> : : = **OWN** | **INTEGER** | **REAL** | **COMPLEX** | **BOOLEAN** | **ALPHA** | **DOUBLE** | **TASK** | **EVENT** | **ARRAY** | **SWITCH** | **LABEL** | **FORWARD** | **PROCEDURE** | **FILE** | **SWITCH FILE** | **FORMAT** | **SWITCH FORMAT** | **LIST** | **SWITCH LIST** | **MONITOR** | **DUMP** | **DEFINE** | **ABSORB** | **NAMELIST** | **LINE** | **IN** | **OUT** | **CORE**

<specificator> : : = **VALUE**

2.1.3.2. Semantics

Delimiters have a fixed meaning and their basic function, as implied by their name, is obvious.

2.1.4. Spaces

Spaces can be used between language elements for readability, but in general, spaces may be used or omitted. Exceptions are where the space becomes the *delimiter* when separating basic components such as reserved words, *identifiers*, logical values, *unsigned numbers*, and multi-character *delimiters*.

2.1.5. Comments

2.1.5.1. Semantics

In order to include explanatory text at various points in a program, the **COMMENT** convention is included.

The sequence of basic symbols:	is equivalent with
; **COMMENT** <any sequence not containing ;> ;	;
BEGIN COMMENT <any sequence not containing ;> ;	**BEGIN**
END <any sequence not containing **END** or **WHILE** or **UNTIL** or ; or **ELSE**>	**END**

2.1.5.2. Examples

```
1         10        20        30        40        50        60
BEGIN COMMENT THE FOLLOWING PROGRAM CONTAINS THE PROPER USE OF F
     COMPATABLE MARICES;

;COMMENT TABLE SEARCH;

IF A GTR B THEN BEGIN C := C + D;
X := Y END THIS IS AN EXAMPLE OF A COMMENT ELSE GO TO LAB5;
```

2.1.6. Identifiers

2.1.6.1. Syntax

<identifier>: : = <letter> | <identifier> <letter> | <identifier> <digit>

2.1.6.2. Semantics

Identifiers are used to name *labels*, *variables*, switches, *formats*, *lists*, arrays, procedures, files, and programs. The same *identifier* can be used to denote different quantities if declared in different *blocks*.

2.1.6.3. Restrictions

- *Identifiers* must begin with a letter and contain only letters and digits. Any combination, beyond the first letter, of letters or digits or both is permissable.
- Reserved words of UNIVAC 1108 Extended ALGOL may not be used as *identifiers*.
- No space may appear within an *identifier*.
- *Identifiers* may be any number of characters in length. However, only the first 12 characters are unique.

2.1.6.4. Examples

A

AC

MATRIX

DC220

AC110TO220

A1B2C3

EXAMINATIONANNUAL

EXAMINATIONMEDICAL

EXAMINATIONMATERNITY

NOTE: The last two identifiers are not unique because their first twelve characters are identical. The identifier immediately preceding the last two is unique; the twelfth character differs.

2.1.7. NUMBERS

2.1.7.1. Syntax

<number> : : = <unsigned number> | + <unsigned number> | − <unsigned number>

<unsigned number> : : = <decimal number> | <special exponent part>| <decimal number> <exponent part>

<decimal number> : : = <unsigned integer> | <decimal fraction> | <unsigned integer> <decimal fraction>

<decimal fraction> : : = . <unsigned integer>

<special exponent part> : := @ <integer>

<exponent part> : : = @ <integer> | **E** <integer> | **D** <integer>

<integer> : : = <unsigned integer> | + <unsigned integer> | − <unsigned integer>

<unsigned integer> : : = <digit> | <unsigned integer> <digit>

2.1.7.2. Semantics

Numbers may be of four basic types, **INTEGER, REAL, DOUBLE,** or **COMPLEX.**

Unsigned numbers are composed of digits and the following basic symbols:

+, −, **E**, ., @, **D.**

Integer numbers are positive or negative whole numbers or zero.

The limits of an integer number are $\pm\ 2^{35} - 1$ (34, 359, 738, 367).

A *number* is considered as **REAL** (single precision floating point) if it contains a decimal point and/or **E** or @ in the exponent. Real numbers have one to eight significant digits. The limits of a real number are approximately $\pm\ 10^{38}$.

A real number may be expressed in several ways. It need not contain a decimal point if it contains an *exponent part.* If a decimal point is used, it can appear at the beginning of the number or embedded between two digits. An exponent may follow a real number if it is preceded by **E** or @. The following are acceptable. ways of representing the real number 256:

25.6**E**1, 25.6**E**+1, 2560.0**E**−1, 256.0, .256**E**3, 25.6@1, .00256@5.

A *number* is considered as **DOUBLE** (double precision floating point) if it is **REAL** with more than 9 significant digits or contains **D** in the exponent. Double precision numbers may have up to 18 significant digits. The limits of double precision numbers are approximately $\pm 10^{\pm 308}$.

In the 1108 hardware representation the important difference between integer and real numbers is that real quantities are stored in the computer in floating point form. If arithmetic is done on a combination of **REAL** and **INTEGER** values the result is always **REAL** in UNIVAC 1108 Extended ALGOL.

NOTE: At compile time, numbers which exceed the allowable number of digits are flagged – CONSTANT TOO LARGE – and an unreliable result is returned.

2.1.7.3. Restrictions

Only the *special exponent part*, @<*integer*>, is allowed to stand alone as a real number.

2.1.7.4. Examples

- *Unsigned numbers*

  1234.567
  @68
  1234.56@78
  1234.56E78

- *Decimal numbers*

  1356
  .666
  1234.567

- *Exponent parts*

  @66
  @–36
  @+63
  E–7
  E06
  D6
  D36

- *Numbers*

number	*type*
123	integer
123.5	real
123E+2	real
123@–3	real
12345678910.5	double precision
1234.5D6	double precision
@12	real
E+5	invalid

- *Unsigned integers*

  6
  69
  253647
  34359738367

- *Decimal fractions*

  .6
  .69

- *Special Exponent part*

  @63

- *Integer number*

  6
  +63
  –636

2.1.8. Strings

2.1.8.1. Syntax

<string> : : = '<open string>'

<string bracket character> : : ='

<open string> : : = <proper string>|'<open string>'|<open string> <open string>

<proper string> : : = <any sequence of four thousand ninety six or less basic symbols>|<empty>

2.1.8.2. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string bracket character is used. *String*s are used in *move statement*s (see 6.10), *file declaration*s, *label part* (see 11.2.2), and *format declaration*s (see 11.2.4.).

*Proper string*s are used following the **COMMENT** *separator* and the **END** *bracket* with the special restrictions that are noted.

2.1.8.3. Restrictions

- Within a *proper string* a single quote must be represented as two quotes to yield one, i.e., 'This is a "string" '.
- Within the *label part* of the *file declaration*, the *string* is limited to twelve alphanumeric characters. This is a restriction of the 1108 file labelling system.
- A *string* used within an *arithmetic expression* as an *alpha constant* is limited to any combination of six alphanumeric characters and/or special symbols of the UNIVAC 1108 Extended ALGOL.
- *Proper string*s following the **END** *bracket* can be any sequence of basic symbols not containing **END**, ;, **WHILE**, **UNTIL**, or **ELSE**. **END**, **ELSE**, **WHILE**, **UNTIL**, and ; stop the scan legally; any other *operator* appearing in the *string* will stop the scan and be marked in error.
- *Proper string*s following the **COMMENT** *separator* can be any sequence of basic symbols not containing a ; .

2.1.8.4. Examples

- String

'ANY COMBINATION OF LETTERS, DIGITS, AND/OR SPECIAL CHARACTERS NOT TO EXCEED 4096'

'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789*/.@□□ #) − + <=> & $(%:,'

2.1.9. Constants

2.1.9.1. Syntax

<constant> : : = <Boolean constant>| <alpha constant>|<octal constant>|
<complex constant>|<number>

<Boolean constant> : : = **TRUE**|**FALSE**

<alpha constant> : : = '<any sequence of six or less characters >'

<octal constant> : : = %<octal number>|%**D**<octal number>

<octal number> : : = <octal digit>|<octal number> <octal digit>

<complex constant> : : = **COMPLEX** (<real part>, <imaginary part>)

<real part> : : = <real number>

<imaginary part> : : = <real number>

2.1.9.2. Semantics

Constants are quantities, characters, or character strings whose values do not change, or are regarded as fixed, during a certain sequence of program steps or mathematical operations.

The *Boolean constants* are **TRUE** and **FALSE.**

The % signifies an octal constant. Octal constants are treated as *integers.*

The % followed by a **D** signifies a double precision constant, which will be treated as double precision floating point in all arithmetic operations.

The *string bracket character* (') is used as the beginning and ending delimiter for strings.

For *number* see section 2.1.7.

2.1.9.3. Examples

alpha constants

```
'KEY''S'

'DATE'
''''
```

octal constants

```
%3742          %D3742

%432           %D2007124000000001277001423
```

complex constant

```
COMPLEX (1.0,0.1)

COMPLEX (9.9, 5.2@12)
```

Boolean constant

```
TRUE

FALSE
```

3. GENERAL COMPONENTS OF EXPRESSIONS

3.1. GENERAL

The algorithmic language, ALGOL 60, is composed primarily of *arithmetic expressions*, *Boolean expressions*, and *designational expressions*. Basic components of these expressions are *numbers*, *constants*, *variables*, *logical values*, *function designators*, and elementary arithmetic, relational, logical, and sequential *operators*.

3.2. BASIC COMPONENTS OF EXPRESSIONS

3.2.1. Variables

3.2.1.1. Syntax

<variable> : : = <simple variable>|<subscripted variable>

<simple variable> : : = <variable identifier>

<variable identifier> : : = <identifier>

<subscripted variable> : : = <array identifier> [<subscript list>]

<array identifier> : : = <identifier>

<subscript list> : : = <subscript expression>|<subscript list>,
<subscript expression>

<subscript expression> : : = <arithmetic expression>

3.2.1.2. Semantics

A *variable* is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of *assignment statements*. The type of the value of a particular *variable* is defined in the declaration for the *variable* itself or for the corresponding *array identifier*. Subscripted variables designate values which are components of multidimensional arrays. Each arithmetic expression of the *subscript list* occupies one subscript position of the *subscripted variable*, and is called a subscript. The complete list is enclosed in the subscript brackets [ ]. The array component referred to by a *subscripted variable* is specified by the actual numerical value of its subscripts.

Each subscript position acts like a *variable* of type **INTEGER** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious *variable*. The value of the *subscripted variable* is defined only if the value of the *subscript expression* is within subscript bounds of the array.

The number of subscripts in a *subscript list* must agree with the declared dimension of the array referenced. Subscript evaluation is from left to right within the *subscript list* and the result will be transferred to type **INTEGER** if the result is not of this type.

3.2.1.3. Examples

Simple variables

VALUE3

TANGENTVALUE

DELTA

A1

B

C999

Subscripted variable

A [1,14]

MATRIX [X MOD Y, PI/2]

VARPQ [**IF** P<Q **THEN** X **ELSE** Y, Z**2]

3.2.2. Function Designators

3.2.2.1. Syntax

<function designator> : : = <procecure identifier> <actual parameter part>

<procedure identifier> :: = <identifier>

<actual parameter part> : : = <empty>| (<actual parameter list>)

<actual parameter list> : : = <actual parameter >|<actual parameter list>
<parameter delimiter> <actual parameter>

<actual parameter> : : = <expression>|<array row>|<array identifier>|
<procedure identifier> | <format identifier> |
<list identifier>|<file identifier>|
<switch identifier>|<switch file identifier>|
<switch format identifier>|<switch list identifier>|
<switch file designator>|<switch format designator>|
<switch list designator>

<parameter delimiter> : : = , |) '<proper string>' (

3.2.2.2. Semantics

A function procedure is indicated by assigning a type to a *procedure declaration.*

Standard library functions are available that may be referenced without being declared. Since the library function identifiers are not reserved, the programmer may declare functions using the standard library names, thereby over-riding the system routines within the scope of the function declaration.

The standard library functions supplied for UNIVAC 1108 Extended ALGOL are described in Table 3-1.

3.2.3. Time Function

3.2.3.1. Syntax

<time function> : : = TIME (<arithmetic expression>)

3.2.3.2. Semantics

The *time function* makes available time and date information relative to the program requesting it. The various functions provided for by particular parameter codes yield the time required by the system, or certain components of it, to execute a program, or parts of a program. The *time function* can be any valid *arithmetic expression* which must yield an integer value of 0 through 4. For the valid codes, the following information will be provided:

Code	Result
0	Current Julian calendar date in FD code, e.g.,

Year	Day
68	001
67	304

1	Total elapsed time from run start time through to time of request in seconds.
2	Actual elapsed processor time in milliseconds.
3	Binary zero.
4	Time of day in milliseconds from midnight obtained from instantaneous clock reading.

If the value of the *arithmetic expression* is not one of the valid integers, a run time error is generated.

3.2.3.3. Examples

```
TIME (0)

TIME (4)

TIME ( (A+B) DIV 2)
```

FUNCTION DESIGNATOR	NUMBER OF PARAMETERS	TYPES OF PARAMETERS	TYPE OF RESULT	RESULT	DEFINITION
ABS	1	INTEGER REAL DOUBLE COMPLEX	INTEGER REAL DOUBLE REAL	|x| |x| |x| |x|	Produces the absolute value of the argument x.
ARCCOS	1	REAL DOUBLE	REAL DOUBLE	arccos (x) arccos (x)	Produces the principle value of the arccosine of the argument x.
ARCSIN	1	REAL DOUBLE	REAL DOUBLE	arcsin (x) arcsin (x)	Produces the principle value of the arcsine of the argument x.
ARCTAN	1	REAL DOUBLE	REAL DOUBLE	arctan (x) arctan (x)	Produces the principle value of the arctangent of the argument x.
COS	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	cos (x) cos (x) cos (x)	Produces the cosine of the argument x.
COSH	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	cosh (x) cosh (x) cosh (x)	Produces the hyperbolic cosine of the argument x.
DOUBLE	1	INTEGER REAL	DOUBLE DOUBLE	double precision representation of x.	Produces the double precision floating point representation of the argument x.
ETEST	1	EVENT	BOOLEAN		Returns a **TRUE** value if *event variable* set, **FALSE** if *event variable* clear.
EXP	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	exp (x) exp (x) exp (x)	Produces the exponential function of the argument x.
IMAGINARY	1	COMPLEX	REAL		Imaginary part of x.
LN	1	REAL DOUBLE	REAL DOUBLE	ln (x) ln (x)	Produces the natural logarithm of the argument x.

Table 3–1. Standard Library Functions
(Part 1 of 2)

FUNCTION DESIGNATOR	NUMBER OF PARAMETERS	TYPES OF PARAMETERS	TYPE OF RESULT	RESULT	DEFINITION
MAX	list of expressions	REAL INTEGER	REAL REAL		Returns the algebraically largest element of the argument list.
MIN	list of expressions	REAL INTEGER	REAL REAL		Returns the algebraically least element of the argument list.
RANDOM	1	REAL	REAL		Random number generator.
SIGN	1	INTEGER REAL DOUBLE COMPLEX	INTEGER	$x > 0 := 1$ $x = 0 := 0$ $x < 0 := 1$	Produces one of the three values listed depending upon the value of the argument x.
SIN	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	sin (x) sin (x) sin (x)	Produces the sine of the argument x.
SINH	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	sinh (x) sinh (x) sinh (x)	Produces the hyperbolic sine of the argument x.
SQRT	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	sqrt (x) sqrt (x) sqrt (x)	Produces the square root of the argument x.
TAN	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	tan (x) tan (x) tan (x)	Produces the tangent of the argument x.
TANH	1	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX	tanh (x) tanh (x) tanh (x)	Produces the hyperbolic tangent of the argument x.

Table 3–1. Standard Library Functions
(Part 2 of 2)

3.2.4. Transfer Functions

Transfer functions between any pair of quantities or expressions may be defined. The result of a transfer function being performed on an expression is that the result of an expression of one type is changed to another type. The standard transfer functions provided in UNIVAC 1108 Extended ALGOL are described in Table 3-2.

TRANSFER FUNCTION USED	TYPE OF INPUT ARGUMENT	OUTPUT TYPE AFTER TRANSFER	LIMITATION AND COMMENTS
ENTIER (x)	REAL or DOUBLE	INTEGER	The value produced is the largest integer not greater than the value of x.
FIXED (x)	REAL or DOUBLE	INTEGER	The value produced is the rounded result of the value of x.
REAL (x)	INTEGER DOUBLE COMPLEX	REAL	
DOUBLE (x)	INTEGER or REAL	DOUBLE	
COMPLEX (x, 0)	REAL	COMPLEX	
COMPLEX (REAL (x), 0)	INTEGER or DOUBLE	COMPLEX	

Table 3–2. Standard Transfer Functions

3.2.5. Pseudo-Transfer Functions

The pseudo-transfer functions, **BOOLEAN** and **INTEGER**, are provided so that *Boolean operators* may be applied to non-Boolean *variables* and *arithmetic operators* may be applied to *Boolean variables*. The value of the argument is not altered with the exception that when type **BOOLEAN** is applied to a type **DOUBLE** argument, the **DOUBLE** argument is truncated to type **INTEGER.** The format is as follows:

BOOLEAN (x) – for x equal to type **INTEGER, REAL, DOUBLE,** or **COMPLEX.**

INTEGER (x) – for x equal to type **BOOLEAN.**

4. EXPRESSIONS

4.1. EXPRESSIONS GENERAL

4.1.1. Syntax

<expression> : : = <arithmetic expression> | <Boolean expression> |
<designational expression>

4.1.2. Semantics

The primary entities of programs describing algorithmic processes are *arithmetic expressions*, *Boolean expressions*, and *designational expressions*.

4.2. ARITHMETIC EXPRESSIONS

4.2.1. Syntax

<arithmetic expression> : : = <simple arithmetic expression> | <if clause>
<simple arithmetic expression> **ELSE** <arithmetic expression>

<simple arithmetic expression> : : = <term> | <adding operator> <term> |
<simple arithmetic expression> <adding operator> <term>

<term> : : = <factor> | <term> <multiplying operator> <factor>

<adding operator> : : = + | −

<factor> : : = <primary> | <factor> ** <primary>

<multiplying operator> : : = * | / | **DIV** | **MOD**

<primary> : : = <unsigned number> | <variable> | <function designator> |
(<arithmetic expression>) | <constant> | <partial word
designator> | <concatenate expression> | (<assignment
statement>)

<if clause> : : = **IF** <Boolean expression> **THEN**

4.2.2. Semantics

An *arithmetic expression* is a rule for computing a numerical value. For simple *arithmetic expressions*, the value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the *expression*. In the case of *numbers*, the value is obvious. For *variables* it is the current value, and for *function designators* it is the value that is assigned upon execution of the procedure named. An *alpha constant* must not be greater than one word and is treated as an **INTEGER** *constant*. The value of *partial word designator* primaries is the current value of the field as specified by *left bit of field* arithmetic expression and the number of *bits in field* arithmetic expression. The new value stored into the *left base* as a result of the application of the concatenate operator is the value of the *concatenate expression* primary. Finally, in *arithmetic expressions* enclosed in parenthesis the value must, through a recursive analysis, be expressed in terms of the values of the other kinds of primaries.

The syntax for the general *arithmetic expression* includes an *if clause* which enables one of several *arithmetic expressions* to be selected for execution. The selection is as follows: The *Boolean expressions* of the *if clauses* are evaluated one by one in sequence from left to right until the value **TRUE** is found. The *arithmetic expression* following the **THEN** delimiter is then executed. If the final *Boolean expression* value is **FALSE**, the *expression* following **ELSE** is executed.

It is important to note that the syntax for UNIVAC 1108 Extended ALGOL permits the *expression* following **THEN** as well as the *expression* following **ELSE** to be the general *arithmetic expression*.

For example:

```
1         10        20        30        40
|         |         |         |         |
IF BE1 THEN IF BE2 THEN E1 ELSE E2 ELSE E3;
```

Apart from the *Boolean expressions* of *if clauses*, the constituents of *simple arithmetic expressions* must be of types **COMPLEX, DOUBLE, REAL,** or **INTEGER** (An **ALPHA** *variable* is treated as **INTEGER**). Mixed mode operations are allowed. The type of the result of a mixed mode operation will be according to the intersection of operand 1 and operand 2 in the following chart:

OPERAND 1	OPERAND 2			
	COMPLEX	DOUBLE	REAL	INTEGER
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX
DOUBLE	COMPLEX	DOUBLE	DOUBLE	DOUBLE
REAL	COMPLEX	DOUBLE	REAL	REAL
INTEGER	COMPLEX	DOUBLE	REAL	INTEGER

Note that the result of a **COMPLEX, DOUBLE** operand pair yields a **COMPLEX** result with a zero imaginary part.

The floating point divide operator, / , yields a **REAL** result, even if both operands are **INTEGER.**

The **MOD** operator yields the remainder of an integer division. If one or both operands associated with the **MOD** operator are of arithmetic type other than **INTEGER,** conversion to **INTEGER** will be performed previous to the execution of the **MOD** operation.

DIV is the integer divide operator. Both operands associated with the integer divide operator must be **INTEGER.**

The operation, *factor* ** *primary*, denotes exponentiation, where the *factor* is the base and the *primary* is the exponent.

Writing *i* for a number of **INTEGER** type, *r* for a number of **REAL** type, and *e* for a number of either **INTEGER** or **REAL** type, the result is given by the following rules:

*a****i* If $i > 0$, *a***a***a*...**a* (*i* times), result of the same type as *a*.

If $i = 0$, if $a \neq 0$, result of 1 of the same type as *a*
if $a = 0$, undefined

If $i < 0$, if $a \neq 0$, 1/(*a***a***a*...**a*) (the denominator has -*i* factors), result of type **REAL**

if $a = 0$, undefined.

*a****r* If $a > 0$, EXP (*r***ln* (*a*)), result of type **REAL**
If $a = 0$, if $r > 0$, 0.0, result of type **REAL**
if $r < 0$, undefined
If $a < 0$, always undefined

It should be noted that both the base and the exponent may be of type **DOUBLE** or **COMPLEX,** with the limitations as indicated by the following chart that lists the type of the result at the intersection of the base and exponent pair.

BASE	EXPONENT			
	INTEGER	REAL	DOUBLE	COMPLEX
INTEGER	INTEGER	REAL	DOUBLE	COMPLEX
REAL	REAL	REAL	DOUBLE	COMPLEX
DOUBLE	DOUBLE	DOUBLE	DOUBLE	*
COMPLEX	COMPLEX	COMPLEX	*	COMPLEX

*Illegal combination

The sequence of operations within one *expression* is generally from left to right utilizing the following rules of *arithmetic operator* precedence:

first: **

second: * / DIV MOD

third: + −

The *expression* between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an *expression* can always be arranged by appropriate positioning of parenthesis.

The maximum value of an **INTEGER** result is $\pm(2^{35}-1)$. If the result should exceed this value, the 1108 overflow indicator will be set as a result of a carry into the sign position. A hardware interrupt is not affected by fixed point overflow but the sign of the result will be changed. The maximum value of a **REAL** result is from 10^{37} to 10^{-38} and of a **DOUBLE** result is from 10^{307} to 10^{-308}. Characteristic overflow or underflow for all arithmetic operations yielding **REAL** or **DOUBLE** results can occur, causing an 1108 hardware interrupt. (The ALGOL programmer can capture this interrupt by use of the *on statement* (See Section 6.9).)

4.2.3. Restrictions

TASK and **EVENT** *variable*s may not be referenced in *arithmetic expressions*.

4.2.4. Examples

Arithmetic Expressions:

```
BETA+GAMMA**2 MOD DELTA DIV EPSILON

IF BOOL1 THEN X-Y ELSE T**2

TOTAL10+ (IF BA[10] THEN SIGMA   ELSE IF BA[9] THEN SIGMA 10
          ELSE ZERO)

IF BOOL2 THEN A/B ELSE C
```

Simple Arithmetic Expression:

```
PI*R**2**H

AR1[B,3]- AR2[B*2,B+C].[X MOD Y:R+Z]

AR2[PW[X,Y,Z].[E1:E2] ] /CAT[1]&(R*S),[E3:MAX(F)-1:E3]
```

Terms:

```
A**2*(B+C);

SIN(B.[X:Y]+Z)/AR1[ROW,COL]*OMEGA*(BETA-GAMMA)
```

Factors:

```
A

A**F1 (X)

Y & ( (R+S) **X) [E1:E2:E3]
```

Primaries:

A

50.001

AR [ROW, COL, L]

PW. [E1:E2]

CON & CAT [E1:E2:E3]

COS(E1/E2)

(-A+B*C/D**2)

INTEGER(BOOL1)+BETA

4.3. BOOLEAN EXPRESSIONS

4.3.1. Syntax

<Boolean expression> : : = <simple Boolean> | <if clause> <Boolean expression> **ELSE** <Boolean expression>

<simple Boolean> : : = <implication> | <simple Boolean> **EQV** <implication>

<implication> : : = <Boolean term> | <implication> **IMP** <Boolean term>

<Boolean term> : : = <Boolean factor> | <Boolean term> **OR** <Boolean factor> | <Boolean term> **XOR** <Boolean factor>

<Boolean factor> : : = <Boolean secondary> | <Boolean factor> **AND** <Boolean secondary>

<Boolean secondary> : : = <Boolean primary> | **NOT** <Boolean primary >

<Boolean primary> : : = <logical value> | <variable> | <function designator> | <relation> | (<Boolean expression>) | <partial word designator> | <concatenate expression> | (<assignment statement>)

<relation> : : = <simple arithmetic expression> <relational operator> <simple arithmetic expression>

<relational operator> : : = < | = | > | **LSS** | **LEQ** | **EQL** | **GEQ** | **GTR** | **NEQ**

4.3.2. Semantics

A *Boolean expression* is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for *arithmetic expressions*.

It should be noted that since the *if clause* of a *Boolean expression* is defined to be **IF** followed by *Boolean expression*, a nesting of *if clauses* is perfectly valid (see examples).

Variables, *function designators*, *partial word designators*, and the *left base* and *right base* of a *concatenate expression* entered as Boolean primaries must be declared **BOOLEAN.**

An exception to the above statement is if the pseudo-transfer function **BOOLEAN** is applied to the result of an *arithmetic expression*. The least significant bit of the arithmetic argument will be tested for a **TRUE** or **FALSE** logical value.

The *logical operators*, listed in the order (highest to lowest) of precedence of use are:

NOT (negation). The negation of a **TRUE** Boolean *variable* is **FALSE** and the negation of a **FALSE** Boolean *variable* is **TRUE.**

AND (conjunction). The conjunction of two *variables* is **TRUE** if and only if both Boolean *variables are* **TRUE.**

OR (disjunction). The disjunction of two Boolean *variables* is **TRUE** if one of the *variables is* **TRUE,** or both of the Boolean *variables* are **TRUE.**

IMP (implication). An implication of two Boolean *variables* is **FALSE** if the first Boolean *variable,* or antecedent, is **TRUE** and the second Boolean *variable,* or consequent, is **FALSE;** otherwise it is **TRUE.**

EQV (equivalence). The equivalance operation on two Boolean *variables* is **TRUE** if and only if both are **TRUE** or both are **FALSE.**

In addition to the above, the set of logical operators for UNIVAC 1108 Extended ALGOL includes the exclusive or function, **XOR,** which in Boolean expressions assumes the same priority level as the inclusive or, **OR:**

XOR The result of the application of the exclusive or operation on two Boolean *variables* is **TRUE** if and only if one or the other, but not both, is **TRUE.**

The Boolean primary, *relations*, yields a **TRUE** or **FALSE** result depending on the truth value of the indicated comparison between the named *arithmetic expressions*.

The *relational operators* are:

LSS or < Less Than

LEQ Less Than or Equal To

EQL or = Equal To

GEQ Greater Than or Equal To

GTR or > Greater Than

NEQ Not Equal To

All *relational operators* have the same order of precedence.

The order of precedence (from highest to lowest) for all operators in both *arithmetic expressions* and *Boolean expressions* is as follows:

**

* / **DIV** **MOD**

\+ –

LSS **LEQ** **EQL** **GEQ** **GTR** **NEQ**

NOT

AND

OR **XOR**

IMP

EQV

It should be noted that all *logical operators* operate upon the whole 1108 word. The **NOT, AND, OR,** and **XOR** utilize the equivalent 1108 hardware instructions, while **IMP** and **EQV** can be defined in terms of **AND, OR,** and **NOT** operations.

The truth value of a Boolean quantity is determined by testing the least significant bit of the word to be 1 (**TRUE**) or 0 (**FALSE**).

4.3.3. Examples

Boolean Expressions:

BOOL1

TRUE

NOT SW1

A* B>X**2

IF B1 **THEN** X **ELSE** Y **GEQ** 50

IF X **LSS** Y **THEN** B1 **ELSE** B2

IF B1 **THEN** B2 **ELSE** B3

IF X = Y **THEN** B1 **ELSE** B2 **THEN** B3 **ELSE** B4 **THEN** A **GTR** B **ELSE**

IF B5 **THEN NOT** B6 **ELSE** B7

B1 **AND** (**IF** B2 **THEN** B3 **ELSE** A **GTR** B) **OR** X+Y **LSS** T**S**R

B1 **OR** B2 **IMP** B3 **AND** B4 **OR** B5

B1 **EQV** X **GEQ** Y

4.4 DESIGNATIONAL EXPRESSIONS

4.4.1. Syntax

<designational expression> : : = <simple designational expression> | <if clause>
<designational expression> **ELSE** <designational expression>

<simple designational expression> : : = <label> | <switch designator> |
(<designational expression>)

<switch designator> : : = <switch identifier> [<subscript expression>]

<switch identifier> : : = <identifier>

<label> : : = <identifier>

4.4.2. Semantics

A *designational expression* is a rule for obtaining a *label* of a *statement*. Again, the principles of evaluation are entirely analogous to that of *arithmetic expressions*.

In the general case the *Boolean expressions* of the *if clauses* will select a *designational expression*. If this is a *label*, the desired result is already found. A *switch designator* refers to the corresponding *switch declaration* and, by the actual numerical value of its *subscript expression*, selects one of the *designational expressions* listed in the *switch declaration* by counting from left to right beginning with zero. Since the *designational expression* thus selected may again be a switch designator this evaluation is obviously a recursive process. The *subscript expression* is defined to be an *arithmetic expression* and, so, evaluation is analogous. The result, however, if not of type **INTEGER**, will be converted to **INTEGER** so that a one to one mapping can exist between the set of positive integers and zero and the result of the *subscript expression*.

If the integer value of the *subscript expression* is not a member of the set associated with the *switch identifier* named, the *switch designator* is undefined and program control continues in sequence.

4.4.3. Examples

Designational Expressions:

```
LABEL1

ENTRYPATH[N(M+1)]

IF BOOL1 THEN LABEL1 ELSE ENTRYPATH[N(+1)]

IF BOOL1 THEN IF BOOL2 THEN SWITCHTO[X] ELSE IF BOOL3 THEN
LAST ELSE START ELSE COMPUTE

SWITCH [IF A>B THEN 0 ELSE IF A=B THEN X ELSE Y]
```

Switch Designators:

```
ENTRYPATH[X+Y]

SWITCHTO [IF B1 THEN X ELSE Y]

SWITCHTO [10]
```

4.5. PARTIAL WORD DESIGNATOR

4.5.1. Syntax

<partial word designator> : : = <partial word operand> .[<bit field description>] |
<partial word operand>, [<character field description>]

<partial word operand> : : = <variable> | (<arithmetic expression>)

<bit field description> : : = <left bit of field>:<bits in field> | <left bit of field>

<character field description> : : = <left character of field>:<characters in field> |
<left character of field>

<left bit of field> : : = <arithmetic expression>

<bits in field> : : = <arithmetic expression>

<left character of field> : : = <arithmetic expression>

<characters in field> : : = <arithmetic expression>

4.5.2. Semantics

The function of the UNIVAC 1108 Extended ALGOL *partial word designator* is to allow operations on specified bit or character fields of a single or double precision word, rather than upon the entire word.

The *partial word operand* may be of type **INTEGER, REAL, DOUBLE,** or **COMPLEX.** The following chart delineates the field variability for partial word references:

TYPE OF OPERAND	Left Bit of Field	RANGE OF FIELD		
		No. Bits in Field	Left Char. of Field	No. Char. in Field
INTEGER	0→35	1→36	0→5	1→6
REAL	0→35	1→36	0→5	1→6
DOUBLE	0→71	1→36	0→11	1→6
COMPLEX	0→71	1→36	0→11	1→6

Note that when the *partial word operand* is double precision (**DOUBLE** or **COMPLEX**), any bit from 0 to 71 may be specified as the left most bit, but the number of bits in the field may not exceed an 1108 single precision word of 36 bits. Accordingly, the left character of a double precision word may be indicated by an integer value from 0 to 11, but the number of characters may not exceed 6.

The bit field and character field descriptions are *arithmetic expressions*. Therefore, like the *subscript expression*, the fields must be checked at run time. The compiler will convert the result of the *arithmetic expressions* designating the fields to type **INTEGER** if the result is other than **INTEGER**. The run time partial word routine will check to see that the sum of the **INTEGER** values for the two fields does not exceed 36 for single precision bit *partial word operands* or 71 for double precision bit *partial word operands*. The corresponding character limits are 6 and 11. The specified field of a *partial word operand* appearing as an expression will be extracted and right justified in the word.

For a *partial word designator* appearing in the left part of an *assignment statement*, the indicated value will be stored into the specified partial word field.

4.5.3. Restrictions

A *partial word designator* in a *left part list* must be the left-most part.

If a *partial word designator* is specified as an *actual parameter* in a procedure call statement, the corresponding *formal parameter* may not appear in a *left part list*.

4.5.4. Examples

MATRIX[5,4].[LB:NB]

GAMMA, [LC:NC]

BIGWD.[34:20]

(SIN(X)),[5]

(X+Y*Z), [L+2: N**2]

A, [2:3] := B := C := D, [1 : 3];

4.6. CONCATENATE EXPRESSION

4.6.1. Syntax

<concatenate expression> : : = <left base> <link part>

<left base> : : = <variable>

<link part> : : = & <right base> <link description> | <link part> & <right base> <link description>

<right base> : : = <general primary>

<general primary> : : = <primary> | <Boolean primary>

<link description> : : = [<left bit of left base>:<left bit of right base>:<number of bits in link>]

<left bit of left base> : : = <arithmetic expression>

<left bit of right base> : : = <arithmetic expression>

<number of bits in link> : : = <arithmetic expression>

4.6.2. Semantics

The word concatenate means a chaining together of objects. The UNIVAC 1108 Extended ALGOL *concatenate expression* allows such an operation up to word length for a single precision or double precision *left base*. Moreover, any bit field of the *right base* primary may be extracted and stored into any bit field of the *left base* variable, provided the fields are of the same length. Note the recursive syntax definition for the *link part*, signifying concatenation of any number of *right base* elements provided word left for the *left base* is not exceeded.

The *link description* fields are all defined to be *arithmetic expressions* and so must be checked to be within limits at run time.

The limits of variability for the *link description* fields and limit of the sum of the *left bit of field* and number of *bits in field* is as follows:

CONCATENATE OPERAND	LEFT MOST BIT IN FIELD	NUMBER OF BITS IN FIELD	SUM OF LEFT BIT AND NUMBER OF BITS
INTEGER	0→35	1→36	36
REAL	0→35	1→36	36
DOUBLE	0→71	1→72	72
COMPLEX	0→71	1→72	72

4.6.3. Restrictions

When combining single and double precision bases, a meaningful result will be obtained only if the number of *bits in field* value does not exceed 36.

4.6.4. Examples

X&Y [LBX:LBY:NB]

X&Y [LBX:LBY:NB] & Z [LBX+B:LBY+Q:NB]

R & ((X+Y) * Z [0:20:16]

DA&DB [36:0:36]

5. STATEMENTS, COMPOUND STATEMENTS, BLOCKS, AND PROGRAMS

5.1. SYNTAX

<statement> : : = <unconditional statement> | <conditional statement> | <iterative statement>

<unconditional statement> : : = <basic statement> | <compound statement> | <block>

<basic statement> : : = <unlabelled basic statement> | <label> : <basic statement>

<unlabelled basic statement> : : = <assignment statement> | <go to statement> | <dummy statement> | <fill statement> | <procedure statement> | <I/O statement> | <zip statement> | <sort statement> | <merge statement> | <move statement> | <compare statement> | <activity statement> | <on statement>

<compound statement> : : = <unlabelled compound statement> | <label> : <compound statement>

<unlabelled compound statement> : : = **BEGIN** <compound tail>

<compound tail> : : = <statement> **END** | <statement> ; <compound tail>

<block> : : = <unlabelled block> | <label> : <block>

<unlabelled block> : : = <block head> ; <compound tail>

<block head> : : = **BEGIN** <declaration> | <blockhead> ; <declaration>

<conditional statement> : : = <if statement> | <if statement> **ELSE** <statement> | <label> : <conditional statement>

<if statement> : : = <if clause> <statement>

<if clause> : : = **IF** <Boolean expression> **THEN**

<iterative statement> : : = <for statement> | <do statement>

5.2. SEMANTICS

The ALGOL *statement* is the fundamental unit of operation within the UNIVAC 1108 Extended ALGOL language. *Statements* are grouped into three major categories: unconditional, conditional, and iterative. *Unconditional statements* directly specify a particular action to be performed. *Conditional statements* will select one of several different courses of action depending on a calculated Boolean value. The *iterative statements* describe a repetitive process and provide methods of forming loops in a program.

Statements are normally executed consecutively as written. This sequence of sequential operation may be broken by *unconditional statements* which explicitly define a labelled successor, i.e., *go to statements* or by *conditional statements* which provide a means whereby the execution of a *statement,* or series of *statements,* is dependent upon the logical value produced by evaluation of a *Boolean expression;* i.e., *if statements.*

In order for one *statement* to refer to another and make it possible to effect a transfer of control, it is necessary to be able to identify a *statement.* A *statement* is identified by a *label.* A statement *label* can be any valid *identifier* and is separated from the *statement* which it names by a colon (:); i.e., *label: statement.*

Compound statements or *blocks* are sets of *statements* which are treated as a unit. Groups of *statements* forming a compound statement must be bracketed by the reserved words **BEGIN** and **END.**

A series of *statements* or *compound statements* common to each other by virtue of the defining *declarations* are enclosed within the statement parentheses, **BEGIN** and **END,** and constitute an ALGOL *block.*

Every program must at least be one *block* and may be composed of many *blocks*; that is, it must start with a **BEGIN,** be followed by any necessary *declarations,* then be followed by the *statements* of the program, and terminate with an **END.**

Every *block* automatically introduces a new level of nomenclature.

The whole program is a *block* and *declarations* at the beginning of the outermost *block* refer to the whole of that and any nested *blocks*; i.e., global *variables. Declarations* at the beginning of inner *blocks* are only known within the inner *block* in which they are declared; i.e., local *variables.* An *identifier* declared in one *block* is undefined in any *block* that is not internal to it. Should an *identifier* declared in an inner *block* already have a meaning established in the encompassing outer *block,* then the outer *block* meaning is temporarily ignored. At the end of the inner *block,* the outer *block* meaning is restored. Hence, the range of a *declaration* extends over the whole of the *block* in which it is declared, including all inner *blocks,* unless the *identifier* is redeclared in such an inner *block.* If redeclared, the new *declaration* holds for the redeclaring *block* and all subsequent inner *blocks* until the **END** bracket of the declaring *block* is encountered.

The fundamental concept of local and global *identifiers* is demonstrated in the schematic diagram of *block* structure shown in figure 5-1. In Figure 5-1 the *variables* A, B, C, and BI are local to *block* B1 and global to *blocks* B2, B3 and B4. The variable AI is local to B1, global for B4 and redefined with the value declared in B2 as local for B2 and global for B3. The *variables* D and E are local to *block* B2, global to B3 and undefined in B1 and B4. *Variables* F and G are local to B3 and undefined for all other *blocks*, and *variables* H, J, K, L, and M are local to B4 and undefined for all other *blocks*.

```
B1: BEGIN

    REAL A, B, C;

    INTEGER AI, BI;

    B2: BEGIN

        ALPHA D, E, AI;

        B3: BEGIN

            REAL F, G;

            END B3;

        END B2;

    B4: BEGIN

        ALPHA H, J;

        REAL K, L, M;

        END B4;

END B1;
```

Figure 5–1. Schematic Representation of Block Structure in ALGOL.

6. UNCONDITIONAL STATEMENTS

6.1. GENERAL

6.1.1. Syntax

<unconditional statement> : : = <basic statement> | <compound statement> | <block>

<basic statement> : : = <unlabelled basic statement> | <label> : <basic statement>

<unlabelled basic statement> : : = <assignment statement> | <go to statement> | <dummy statement> | <fill statement> | <procedure statement> | <I/O statement> | <zip statement> | <on statement> | <move statement> | <compare statement> | <sort statement> | <merge statement> | <activity statement>

<compound statement> : : = <unlabelled compound statement> | <label> : <compound statement>

<unlabelled compound statement> : : = **BEGIN** <compound tail>

<compound tail> : : = <statement> **END** | <statement> ; <compound tail>

<block> : : = <unlabelled block> | <label> :<block>

<unlabelled block> : : = <block head> ; <compound tail>

<block head> : : = **BEGIN**<declaration> | <block head> ; <declaration>

6.1.2. Semantics

Unconditional statements directly specify a particular action to be performed. This group of *statements* includes the basic constructs implied by the *assignment statement, go to statement, procedure statement* and *I/O statements*. Each *basic statement* listed will be discussed separately.

6.2. ASSIGNMENT STATEMENT

6.2.1. Syntax

<assignment statement> : : = <left part list> <arithmetic expression> | <left part list> <Boolean expression>

<left part list> : : = <left part> | <left part list> <left part> | <partial word designator> : =

<left part> : : = <variable> : = | <procedure identifier> : =

6.2.2. Semantics

Assignment statements serve for assigning a value to one or more *variables* or *procedure identifiers*. The simplest form of an *assignment statement* is : *variable*, replacement operator (: =), *expression;* i.e., V : = E.

Evaluation of the *assignment statement* proceeds in the following order:

(1) Any *subscript expression* occurring in the *variables* of the *left part* are evaluated in sequence from left to right.

(2) The *expression* following the final (right-most) replacement operator is evaluated.

(3) The value of the *expression* is assigned to all *left part* variables, with *subscript expressions*, if any, having the values derived in the first step. If the value from step 2 yields a type different from that of a left part variable, an attempt is automatically made to convert the value to the type of the *left part* variable.

Throughout UNIVAC 1108 Extended ALGOL, **ALPHA** *variables* are compatible with type **INTEGER.** Therefore, if the result of the *arithmetic expression* is of type **ALPHA** or **INTEGER** and an **ALPHA** *variable* appears in the *left part list,* the result will be stored unchanged. If the result is any other arithmetic type, it will be converted to **INTEGER** and stored into the **ALPHA** *variable.*

6.2.3. Restrictions

1. If the type of the *expression* is **BOOLEAN,** the *left part list* elements must be **BOOLEAN.**

2. Neither **TASK** nor **EVENT** *variables* may be used in *assignment statements.*

6.2.4. Examples

```
1         10        20        30        40
N := N + 1;
A := B > C;
A := B := C := D := F + G / D / V / J;
X := Y := Z := 0;
```

6.3. GO TO STATEMENT

6.3.1. Syntax

<go to statement> : : = **GO TO** <designational expression>

6.3.2. Semantics

A *go to statement* interrupts the normal sequence of operations and provides a method of unconditional transfer to the point in the program defined by the value of the *designational expression.* When the *designational expression* is a *label,* the *statement* causes a transfer to the point in the program indicated by the *label.* In the case of a more complex *designational expression,* the next *statement* executed will be the one having the *label* indicated by the value of the *designational expression.*

6.3.3. Restrictions

*Label*s must be declared in the *block* in which they appear as a statement label. A *go to statement* cannot lead from outside a *block* to a point within that *block;* each *block* must be entered at the head of the block so that the associated *declaration*s can be invoked.

In the event of an undefined *switch designator,* the *go to statement* is equivalent to a *dummy statement* and transfer of control does not occur; normal consecutive sequence of *statement* execution continues.

6.3.4. Examples

```
1         10        20        30        40
GOTO LOOP;
GOTO SWTCHA[3];
GOTO IF A NEQ 0 THEN SWTCHA[3] ELSE LOOP;
```

6.4. DUMMY STATEMENT

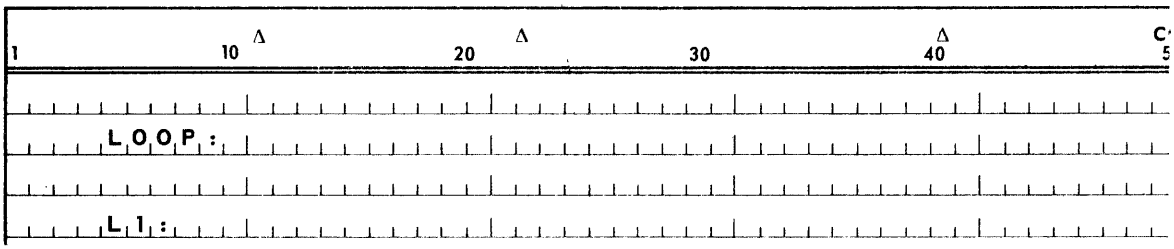
6.4.1. Syntax

<dummy statement> : : = <empty>

6.4.2. Semantics

A *dummy statement* does not cause the execution of any operation. It may be used to place a *label.*

6.4.3. Examples



6.5. FILL STATEMENT

6.5.1. Syntax

<fill statement> : : = **FILL** <array identifier> [<row designator>] **WITH** <value list>

<array identifier> : : = <identifier>

<row designator> : : = * | <row>,*

<row> : : = <arithmetic expression> | <row> ,<arithmetic expression>

<value list> : : = <constant> | <value list> , <constant> | <string> | <value list>, <string>

6.5.2. Semantics

The *fill statement* causes one *row* of an array to be filled with a list of specified values.

The *row designator* identifies which *row* is to be filled. The right most subscript position must contain an asterisk (*). In a single dimensional array only the asterisk is given as the *row designator.*

If the *arithmetic expressions* used as *row designators* yield other than integer values, they are converted to **INTEGER.**

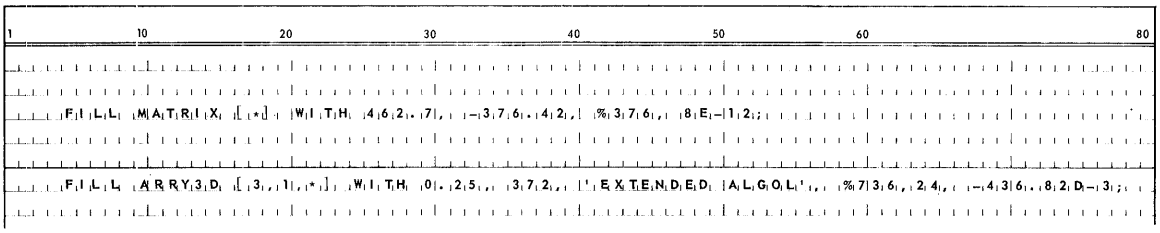
The values specified in the *value list* will replace the current values of the *row* being filled. The number of elements in the *row* indicated may differ from the number of values in the *value list* specified. If the number is less than the number of elements, the elements with the largest subscript values will remain the same.

If the number of values is greater than the number of elements, the right-most values in the *value list* are ignored and only as many values will be used as exist corresponding element positions.

RESTRICTIONS:

Complex constants may not be used in a *fill statement.*

6.5.3. Examples



6.6 PROCEDURE STATEMENT

6.6.1. Syntax

<procedure statement> : : = <procedure identifier> <actual parameter part>

<procedure identifier> : : = <identifier>

<actual parameter part> : : = <empty> | (<actual parameter list>)

<actual parameter list> : : = <actual parameter> | <actual parameter list> <parameter delimiter> <actual parameter>

<actual parameter> : : = <expression> | <array row> | <array identifier> | <procedure identifier> | <file identifier> | <format identifier> | <list identifier> | <switch identifier> | <switch file identifier> | <switch format identifier> | <switch list identifier> | <switch file designator> | <switch format designator> | <switch list designator>

<parameter delimiter> : : = , |)'<proper string>'(

<array row> : : = <array identifier> [<row designator>]

6.6.2. Semantics

A *procedure statement* calls for the sequential execution of a previously defined *procedure body.* The *procedure identifier* designates the particular *procedure body* to be executed and the *actual parameter part* supplies the arguments to be passed to the procedure.

Formal and *actual parameters* must correspond in number, type and kind. This correspondence must also be positional, i.e., correspondence is checked by comparing, in the order given, the *formal parameter list* of the procedure declaration with the *actual parameter list* of the procedure call.

If the *formal* and *actual parameters* do not agree in kind or type, or if they differ in number, an error message will be generated at run time. Correspondence is checked at run time so that the *actual parameters* of a referenced external procedure may also be verified.

Basically the *procedure statement* operates as follows:

All *formal parameters* which were value declared in the heading of the procedure *declaration* are assigned the values of the corresponding *actual parameters*; assignments are considered as being performed explicitly before entering the *procedure body*. These *formal parameters* subsequently are treated as local to the *procedure body*.

All *formal parameters* which were not value declared are replaced by the corresponding *actual parameters* throughout the *procedure body*. There is no conflict between *identifiers* inserted through this process and other *identifiers* local to the *procedure body* which are identical to the *formal parameters*.

The *procedure body*, modified by value assignments and name replacements, is then executed.

There are two means of calling for parameters within a procedure: call by value and call by name. For a parameter to be called by value it must be specified in the *value part* of the *procedure declaration;* all other parameters are considered called by name.

6.6.2.1. Value Assignment (Call by Value)

Procedure parameters called by value are evaluated once and a quantity, identified by the *formal parameter*, is created local to the procedure and the result of this evaluation is assigned to it. Thereafter, the corresponding *actual parameter* is inaccessible to the procedure unless the procedure is called again.

Evaluation of *actual parameters* called by value and the subsequent value assignment to their corresponding *formal parameters* is performed in the order given by the *actual parameter list* of the calling *statement*. All value assignments take place before entry is made into the *procedure body*.

Actual parameters which may be called by value are arithmetic, Boolean, and *designational expressions* and *array identifiers*.

If the *actual parameter* is an *expression* it is evaluated according to the rules previously defined for *expressions* and the value is stored in the local cell set aside for this parameter.

If the *actual parameter* is an *array identifier*, the corresponding *formal parameter* must be used as an *array identifier*. An identical array local to the procedure is created and all current values of the actual array are then assigned to the corresponding elements of the newly created array. Only declared **CORE** arrays may be used in this manner.

6.6.2.2. Call By Name

Parameters called by name are re-evaluated each time they are referenced within the procedure body. An actual parameter called by name is treated as a non-local quantity and is accessible throughout the procedure. If the *actual parameter* is a *simple variable* called by name wherever the corresponding *formal parameter* appears in the *procedure body* it is replaced by the *identifier* of the *simple variable*. If the *actual parameter* is a *subscripted variable,* wherever the corresponding *formal parameter* appears in the *procecure body,* it is replaced by the *subscripted variable;* the *subscript expression* remains intact, and is evaluated each time it is referenced during execution.

If the *actual parameter* is a *partial word designator,* it replaces the corresponding *formal parameter* throughout the *procedure body.* The *partial word designator* is referenced each time it is encountered during execution of the procedure. The corresponding *formal parameter* must not appear in the *left part* of an *assignment statement.* If the *actual parameter* is a *function designator,* the corresponding *formal parameter* is replaced by the *function designator* wherever it appears, and it is evaluated each time it is encountered during execution of the *procedure body.*

When an *expression* is called by name, the corresponding *formal parameter* is replaced by the *expression.* This *expression,* then, is evaluated each time it is encountered during execution of the *procedure body.*

If the *actual parameter* called by name is an *array identifier,* the corresponding *formal parameter* is replaced by the *array identifier* wherever it appears in the *procedure body.* No local array is generated and any use of this *formal parameter* in the *procedure body* references the actual array designated by the *array identifier.*

If the *actual parameter* is a *switch identifier, switch file identifier, switch format identifier, or switch list identifier,* the corresponding *formal parameter* is replaced by the respective *identifier* wherever the *formal parameter* occurs in the *procedure body.* Thus a switch, switch file, switch format, or switch list declaration which has been declared outside the *procedure body* can be accessed during the execution of the *procedure body.*

If the *actual parameter* passed is a *procedure identifier,* the corresponding *formal parameter* is replaced by the *procedure identifier* wherever the *formal parameter* appears in the *procedure body.* Access can thus be made to another procedure which has been declared outside the *procedure body.*

If the *actual parameter* passed is a *file identifier, format identifier,* or *list identifier,* the corresponding *formal parameter* is replaced by the *identifier* of the *actual parameter* wherever the *formal parameter* appears in the *procedure body. I/O statements* in a *procedure body* can thus utilize *files, formats,* and *lists* which have been declared outside the *procedure body.*

For value called parameters, the replacement of *formal paramters* by *actual parameters* occurs prior to execution of the *procedure body.*

For name replaced parameters, each time the replaced parameter is encountered, the mechanism goes outside of the procedure, re-evaluates, and then uses the value.

Finally, the *procedure body* as modified by value and/or name replacement of parameters is executed.

6.6.3. Restrictions

- A *formal parameter* which occurs as a *left part* variable in an *assignment statement* within the *procedure body* and is not called by value, can only correspond to an *actual parameter* which is a *variable*.
- A *partial word designator* must not be passed as an *actual parameter* if the *formal parameter* is to be used on the left-hand side of an *assignment statement*.
- An array used as an *actual parameter* must have the same number of dimensions as its corresponding *formal parameter*. Arrays used as *formal parameters* must have **CORE** specified if the *actual parameter* to be passed is a declared **CORE** array, or if it is a one dimensional array.
- A non-**CORE** declared array may not be passed to a *formal parameter* which is **VALUE** declared.

6.6.4. Examples

```
1         10        20        30        40
PROC1 (NAME A);
PROC2 (ARR[2, 3], CONST1);
PROC3 (SIN(A+3), CONST3, A+B/2+C*D;
```

A more extensive set of examples is provided in the section on *procedure declarations*.

6.7. I/O STATEMENTS

All *I/O statements* will be discussed in section 11.3, Input/Output.

6.8. ZIP STATEMENT

6.8.1. Syntax

<zip statement> : : = **ZIP WITH** <array row>

6.8.2. Semantics

The *zip statement* allows one object program to initiate a completely separate run stream, which may include other compilations or executions. The *zip statement* causes information in the designated *array row* to be recognized as control card information. The *array row* must contain a '▼START' control card followed by parameter information and a period.

For a more detailed explanation of the ▼START control card see UNIVAC 1108 EXECUTIVE Programmers Reference Manual, Section 5.4.7.

The ▼START control card has as its parameter the name of a catalogued file. This file may be assigned by the user externally of via an ALGOL *file statement*. The file must contain an actual run stream, which was created by the ▼DATA or ▼FILE processor or by an ALGOL program.

6.8.3. Example

```
1         10        20        30        40
FILL MATRIX [*] WITH '▼START','FILE1','.';
ZIP WITH MATRIX [*];
```

In the example given FILE1 could have been created in the following manner:

```
1         10        20        30        40        50
▼ASG,C FILE1,F///50
▼DATA,1L FILE1
▼RUN RUN1D,39602,PROJECT,5,50
▼ALGOLSE ALG1,RALG1
    BEGIN REAL X;
    .
    .
    .
    END;
▼END. THIS CONTROL CARD TERMINATES THE DATA PROCESSOR
```

When the *zip statement* is encountered the run in FILE1 will be given to the Executive and the program ALG1 will be compiled and executed.

6.9. ON STATEMENT

6.9.1. Syntax

<on statement> : : = **ON** <condition> <statement> | <**ON** <condition>

<condition> : : = **OVERFLOW** | **UNDERFLOW** | **ZERODIVIDE**

6.9.2. Semantics

The *on statement* allows the programmer to specify an alternate statement sequence to be executed if an unusual *condition* occurs during program execution. When these *statements* are present, it will override the normal sequence of the operating system. After execution of these *statements*, control is transferred back to the point in the program following the statement at which the *condition* occurred.

There are two ways in which the *on condition* is made inactive. Termination of the block in which the *on condition* statement occurred will inactivate the special action. If the same *on condition* is specified and no *statement* given, the *condition* reverts to normal system action.

Valid conditions are:

Condition Name	Cause	Normal System Action
OVERFLOW	Floating Point Overflow	Comment and terminate
UNDERFLOW	Floating Point Underflow	Comment, set result to zero and continue
ZERODIVIDE	Division by zero; floating or integer	Comment and terminate

These *conditions* will result in the normal system action unless an associated *on statement* is given. The **ON** *condition* holds for the scope of the block in which it is given until another **ON** *condition* is specified, which will override the previous one.

6.9.3. Examples

Initiate special action

```
1         10        20        30        40        50        60        80
BEGIN
REAL A, B, C;
PROCEDURE RECOVER(I);
END OF PROCEDURE;
ON OVERFLOW BEGIN I := 1; RECOVER(I); END;
A := 10**35;
B := A*A; COMMENT GENERATES OVERFLOW - MAXIMUM IS EXCEEDED;
BEGIN COMMENT NESTED BLOCK;
ON UNDERFLOW WRITE (FILE1, FUNDR);
B := 1/A/A; COMMENT GENERATES UNDERFLOW;
ON OVERFLOW; COMMENT UPON OVERFLOW CONDITION SYSTEM ACTION IS NOW
PERFORMED;
END NESTED BLOCK;
COMMENT THE ON STATEMENTS GIVEN IN THE NESTED BLOCK ARE NO LONGER
EFFECTIVE, SYSTEM ACTION WILL NOW BE PERFORMED FOR UNDERFLOW.
OVERFLOW WILL CAUSE THE PROCEDURE RECOVER TO BE CALLED;
END PROGRAM;
```

6.10. SOURCE TO DESTINATION STATEMENTS

6.10.1. Syntax

<source to destination statement> : : = <move statement> | <compare statement> | <transfer in statement> | <transfer out statement>

<move statement> : : = **MOVE** (<source part>,<destination>)

<source part> : : = <source> <slink> | <source part>,<source> <slink>

<source> : : = <subscripted variable> | <string>

<slink> : : =,[<left char>:<number of char>] | <empty>

<left char> : : = <arithmetic expression>

<number of char> : : = <arithmetic expression>

<destination> : : = <subscripted variable><dlink>

<dlink> : : = , [<left char>]

<compare statement> : : = **COMPARE** (<source part>,<destination>)

<transfer in statement> : : = **MOVE OCT** (<source><slink>,<variable>)

<transfer out statement> : : = **MOVE DEC** (<variable>,<out destination>)

<out destination> : : = <subscripted variable><slink>

6.10.2. Semantics

- The *source to destination statements* provide for the transfer, comparison, or conversion of characters from one or more *sources* to a single *destination.*

- The *move statement* transfers, left to right, characters from any number of *sources* to a single *destination.*

  The only limitation on the number of characters is that the destination array provided must be large enough to contain them. If this limit is exceeded, a run time error message will be given.

- The *compare statement* tests characters from any number of *sources* for equality with the characters of a single *destination.* The result of the comparison is indicated by the variable **COMP.** If all characters are equal, **COMP** is assigned the Boolean value **TRUE**, otherwise, **COMP** is **FALSE. COMP** will be declared by the *compare statement* if it has not been previously declared in the local *block.*

- The *transfer in statement* converts characters (presumably numeric fieldata) from a single *source*, to its octal equivalent, and stores this result in the *destination* variable specified; i.e. decimal to binary conversion.

The *destination* variable is assumed **INTEGER.** Since the number of characters specified in the *source* may be of any positive integer magnitude, care must be taken not to exceed the storage capacity of the *destination* variable, otherwise an unrecoverable error condition will result.

In converting the *source* character string leading blanks are ignored. If the first non-blank character is a fieldata plus or minus it is used to determine the sign of the octal equivalent, otherwise the first non-blank and succeeding characters in the specified field are treated as positive numeric fieldata and when converted the result is considered positive.

- The *transfer out statement* moves octal data from a single *source* variable which is assumed type **INTEGER**, converts it to decimal in the field data form, and moves this result into the specified *destination* character locations; i.e., binary to decimal conversion. Note that *out destination* is unique to this statement. Care must be taken to see that the number of characters specified in *slink* is sufficient to handle the number being converted including a sign character, otherwise an unrecoverable error will result. If the number of characters plus the sign character is less than the number specified in *slink,* the string is right justified with leading blanks.

- In all *source to destination statements* the *left char* field must yield an integer value ranging from 0 through 5; characters in a word are numbered left to right; values outside of this range will generate an unrecoverable error. Evaluation of the arithmetic expression of the *number of char* variable must yield an integer number with a value of one or more; a value of zero or less will generate an unrecoverable error.

6.10.3. Restrictions

- Bit specification is not permitted in the *source to destination statements*. Bit specification is provided for in partial word operations.

- *Slink* may only be empty if the source is a *string*.

6.10.4. Examples

In the following examples, consider A, D, E, and H as declared **ALPHA** *arrays* and F, G, and I as **INTEGER** *variables*.

```
1        10        20        30        40        50        60        80
MOVE (A[1,2], I:6, 'ABC', D[1], [0]);
COMPARE (A[1,3], [3:1], 'JANUARY 1, 1968', D[1], [3]);
MOVE OCT (E[1,2,3], [1:6], F);
MOVE DEC (D[J], H[J], [1:2]);
MOVE DEC (G, H[1:2]);
```

NOTE: The argument list of **MOVE** can be identical with that of **COMPARE.**

6.11. SORT/MERGE STATEMENTS

Because of the extensive parameter requirements of the *sort statement* and *merge statement,* they will not be covered here but will be handled separately in Section 13 in this manual.

6.12. ACTIVITY STATEMENTS

Activity statements provide multiprocessing capabilities; they are covered separately in Section 12 of this manual.

7. CONDITIONAL STATEMENTS

7.1. SYNTAX

<conditional statement> : : = <if statement> | <if statement>**ELSE**<statement> |
<label>:<conditional statement>

<if statement> : : = <if clause> <statement>

<if clause> : : = **IF** <Boolean expression> **THEN**

7.2. SEMANTICS

The *conditional statement* provides a way to change the sequence of execution of *statements* at run time.

The execution of a *conditional statement* may be described as follows:

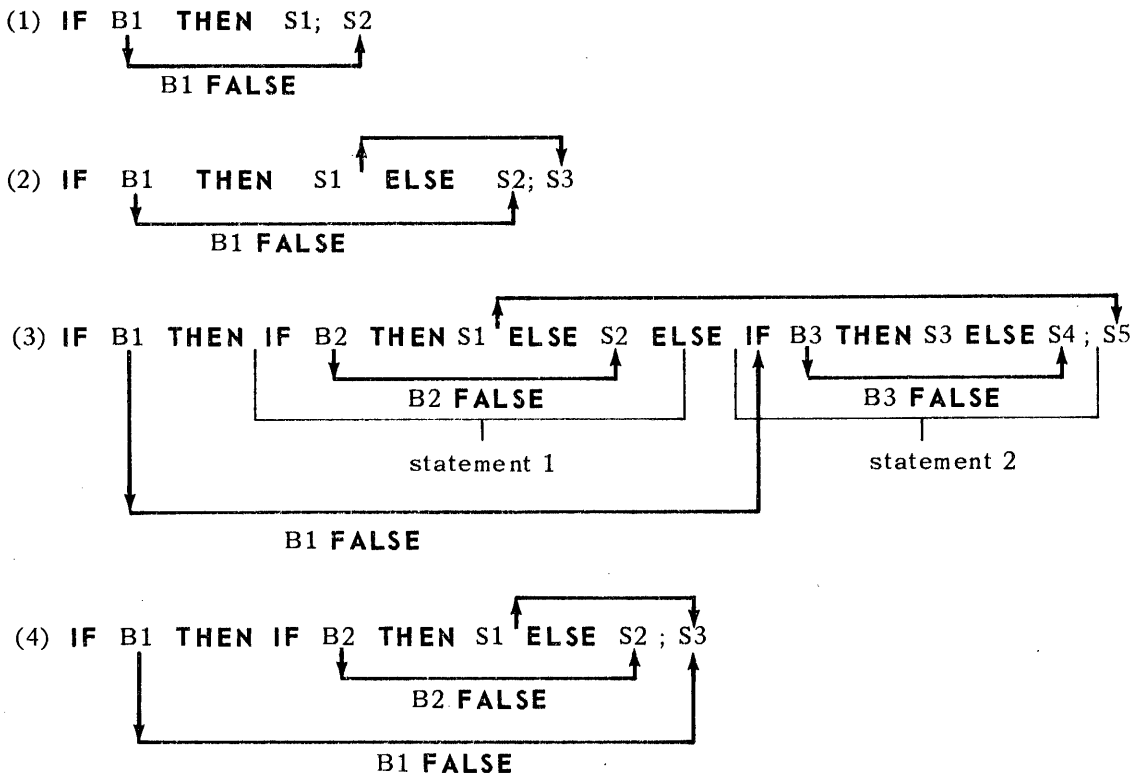
The *Boolean expressions* of the *if clauses* are evaluated one after the other in sequence from left to right.

If a **TRUE** value is found, the *statement* following **THEN** will be executed. Unless this statement defines its successor explicitly, the next *statement* to be executed will be the *statement* that follows the complete *conditional statement.*

If the *conditional statement* consists only of the *if statement* and the result of the *Boolean expression* is **FALSE** control will pass to the *statement* following the complete *conditional statement.*

If the delimiter **ELSE** is specified and the result of the *Boolean expression* is **FALSE,** the statement following **THEN** is by-passed and the *statement* following **ELSE** will be executed.

For further explanation note the following figure:



NOTE: (3) illustrates that both the *statements* following **THEN** and **ELSE** may be *conditional statements* (ALGOL 60 only allows the *statement* following **ELSE** to be conditional)

(3) and (4) denote that **ELSE** is always associated with the nearest **THEN.**

Figure 7–1. Schematics of Conditional Statements

A *go to statement* may reference a labelled *statement* within a conditional statement. Execution begins with the labelled *statement* and succeeding order from this point in the program is then determined in the same manner as if entrance had been made at the beginning of the *conditional statement.*

7.3. EXAMPLES

IF CLAUSES

```
IF A - B NEQ 0 THEN
IF BOOL1 AND BOOL2 THEN
```

IF STATEMENTS

```
1         10        20        30        40        50        60        80
IF (A+B**2) > LIMIT THEN PROC(P1,P2);
IF BOOL1 OR NOT (BOOL2 AND BOOL3) THEN GO TO LAB X;
```

CONDITIONAL STATEMENTS

```
IF BOOL1 THEN BEGIN X:=SIN(X); GO TO START2 END ELSE X:=X+1;
IF A<B AND C=D THEN FOR I:=1 STEP 1 UNTIL 10 DO MAT[I,I]:=MAT[I,I]
   ELSE IF A=B THEN FOR I:=1 STEP 1 UNTIL D DO MAT[I,I]:=0;
```

8. ITERATIVE STATEMENTS

8.1. GENERAL

8.1.1. Syntax

<iterative statement> : : = <for statement> | <do statement>

8.1.2. Semantics

Iterative statements are used as a convenient method of forming loops in a program; they permit the repetitive execution of a *statement* zero or more times.

8.2. FOR STATEMENT

8.2.1. Syntax

<for statement> : : = <for clause> <statement> | <label>: <for statement>

<for clause> : : = **FOR** <variable> :=<for list>**DO**

<for list> : : = <for list element> | <for list>,<for list element>

<for list element> : : = <arithmetic expression> | <arithmetic expression>**STEP** <arithmetic expression> **UNTIL** <arithmetic expression> | <arithmetic expression> **WHILE** <Boolean expression> | <arithmetic expression> **STEP**<arithmetic expression> **WHILE** <Boolean expression>

8.2.2. Semantics

A *for statement* provides a method of forming loops in a program; it allows for the repetitive execution of the *statement* following the *for clause* zero or more times.

There are three distinct steps involved in the execution of a *for statement:*

(1) Value assignment to the controlled *variable.*

(2) Test of the limiting condition.

(3) Execution of the *statement* following the **DO.**

In a *for list* the initial value assigned to the controlled *variable* is that of the left-most arithmetic expression in the *for list element.*

The *for list* may contain more than one *for list element.* The process described by more than one *for list element* in a *for list* is exactly like that described by writing a series of *for statements*, each with one of the *for list elements*, with identical controlled *variables*, and the same *statement* following each **DO.**

The *for list element* determines what values are to be assigned to the controlled *variable* and what test to make of the controlled *variable* to decide whether or not to execute the *statement* following **DO.** When a *for list element* has been exhausted, the next element in the *for list* is considered, progressing from left to right. When all the elements in a *for list* have been utilized, the *for list* is considered exhausted and control is continued in sequence.

All *for list elements* must be of a type compatible with the controlled *variable* which may be any type of a simple or *subscripted variable.*

There are four kinds of *for list elements*:

(1) The *for list element* as a simple *arithmetic expression.*

FOR V : =<arithmetic expression> **DO** S;

A *for list element* may be an *arithmetic expression,* in which case only one value is assigned to the controlled *variable,* V. There is no limiting condition, therefore no test is made. After assigning the initial value to the controlled *variable* the *statement,* S, following **DO** is executed.

FOR V : = E_1, E_2, E_3 ...E_n **DO** S;

A *for list* may contain a list of *for list elements* which may be *arithmetic expressions.* In this case the controlled *variable,* V, is successively given the values of the *arithmetic expressions* E_1, E_2, E_3, ...E_n. The *statement,* S, is executed once for each value of V.

(2) The *For list element* as a **STEP-UNTIL** element.

FOR V : =<arithmetic expression> **STEP** <arithmetic expression>
UNTIL <arithmetic expression> **DO** S;

or for ease of description

FOR V : = E_1 **STEP** E_2 **UNTIL** E_3 **DO** S;

where E_1 is the starting or initial value of V
E_2 is the increment by which V is increased algebraically
E_3 is the limiting or terminal value of V

If the *for list element* is of the **STEP-UNTIL** element form, a new value is assigned to the controlled *variable* each time the *statement* following **DO** is executed. First, an initial value, that of E_1, is assigned to the controlled *variable*. All subsequent assignments are equivalent to: V : = V + E_2, and are made immediately after the *statement* following the **DO** is executed. The limiting condition on the value of V is given by E_3, which is evaluated each time through the loop. A test is made immediately after each assignment to V to determine whether or not the value of V has passed E_3. If V has not passed E_3, the *statement* following **DO** is executed. If V has passed E_3, the element has been exhausted and the *statement* following **DO** is not executed. If the value of E_2 is zero, the program may easily be caught in a closed loop.

(3) The *for list element* as a **WHILE** element.

FOR V : = <arithmetic expression>**WHILE** <Boolean expression>**DO** S;

or for ease of description

FOR V : = E **WHILE** B **DO** S ;

where E is an *arithmetic expression*
B is a *Boolean expression*
S is an ALGOL *statement*

The value of E is assigned to the controlled *variable* V as long as the logical value of the *Boolean expression*, B, is **TRUE.** First the value of E is assigned to the controlled *variable*. A test is made on the logical value produced by B; if the value is **TRUE** the *statement*, S, following **DO** is executed. This process is continued until the value of B is **FALSE**, at which time the *list element* has been exhausted.

(4) The *for list element* as a **STEP-WHILE** element.

FOR V : = <arithmetic expression> **STEP** <arithmetic expression>
WHILE <Boolean expression> **DO** S ;

or for ease of description

FOR V : = E_1 **STEP** E_2 **WHILE** B **DO** S ;

where E_1 is the starting or initial value of V
E_2 is the increment by which V is increased algebraically
B is a *Boolean expression*

If the *for list element* is a **STEP-WHILE** element, it calls for a new value to be assigned to the controlled *variable* V if the value of B is **TRUE** each time the *statement* following **DO** is executed. First, the initial value, E_1, is assigned to the controlled *variable*. All subsequent assignments are: V : = V + E_2, and are made immediately after the *statement* following **DO** is executed. A test is made after each assignment to V to determine if the logical value of B is **TRUE.** If the value of B is **TRUE,** the statement following **DO** is executed; otherwise, the list element has been exhausted.

Value of the controlled *variable* upon exit from the *for statement* varies. If the *statement* S has a *go to statement* leading out of the *for statement*, the value of the controlled *variable* is the same as it was before the *go to statement* was executed.

If the exit is made from the *for statement* because of the exhaustion of the *for list*, then the value of the controlled *variable* is undefined and is not accessible after exit.

8.2.3. Restrictions

A transfer to a labelled *statement* within the scope of a *for statement* through the use of a *go to statement* from outside of the *for statement* is not allowed; the *label* is undefined.

8.2.4. Examples

- *for list elements*

  A

  I + 2

  10 **STEP** -1 **UNTIL** 0

  1 **STEP** 1 **WHILE** I = J

  A + B **WHILE** C>A

- *for lists*

  A, I + 2

  1, 2, 3, 4 **STEP** 2 **UNTIL** 100, 200 **WHILE** I>J, 400 **WHILE** I = J + K

- *for clause:*

  FOR I : = 1 **STEP** 1 **UNTIL** 10 **DO**

  FOR J : = 1, 2, 10, 20 **STEP** 2 **WHILE** J < I **DO**

■ *for statements*

```
1         10        20        30        40
FOR A:=B DO S1;
FOR A:=B, (C+3)/(B-4), D DO S;
FOR A:=B, C, D, E STEP 2 UNTIL 100 DO S;
FOR A:=B, C+D STEP -E UNTIL 0 DO S;
FOR A:=B, C, D WHILE E<F DO S;
FOR A:=B, (C+2)/J STEP 1 WHILE A>D DO S;
```

8.3. DO STATEMENT

8.3.1. Syntax

<do statement> : : = **DO** <statement> **UNTIL** <Boolean expression> | **DO** <statement> **WHILE** <Boolean expression>

8.3.2. Semantics

The *do statement* provides another method of controlling an iterative loop in a program. It allows for the repetitive execution of the *statement* following the DO zero or more times until the limiting condition is met or while the limiting condition holds.

The **DO-UNTIL** construction executes the *statement* following the **DO** repetitively until the value of the *Boolean expression* becomes **TRUE.**

The **DO-WHILE** construction executes the *statement* following the **DO** repetitively until the value of the *Boolean expression* becomes **FALSE.**

In both cases, program execution continues with the next *statement* in sequence when the condition is met.

8.3.3. Examples

In the following examples S1, S2,...S5 represent any valid ALGOL *statements.*

```
1         10        20        30        40
DO S1 UNTIL A=100; S2;

DO BEGIN S1; S2; S4 END UNTIL B>A+C; S5;

DO S1 WHILE A>B; S2;

DO BEGIN S1; S2; S3 END WHILE A GEQ C; S4;
```

9. DECLARATIONS

9.1. GENERAL

9.1.1. Syntax

<declaration> : : = <type declaration> | <label declaration> | <switch declaration> | <forward reference declaration> | <array declaration> | <absorb declaration> | <define declaration> | <monitor declaration> | <dump declaration> | <procedure declaration> | <I/O declarations>

9.1.2. Semantics

Declarations serve to define certain properties of a quantity and assign an *identifier* to the quantity so that it may be referenced in a program. A *declaration* for an *identifier* is valid only for the *block* in which it is declared. Outside of the *block* in which it is declared, a particular *identifier* has no meaning and can be used for other purposes.

Dynamically this implies that at the time of entry into a *block* (through the **BEGIN** since *labels* inside are local and not accessible from outside) all *identifiers* declared in the block heading assume the significance implied by the nature of their *declarations*. If these *identifiers* have already been defined by other *declarations* outside of the current *block*, they are temporarily given a new significance, and the encompassing outer meaning is forgotten. *Identifiers* not redeclared for the current *block* retain their old meaning. The following diagram demonstrates this principle:

```
1         10        20        30        40
BEGIN
  COMMENT BLOCK1;
  REAL A, B, C;
  INTEGER D, E;
  ALPHA ARRAY MAT[1:20];
    :
    :
    :
  BEGIN COMMENT BLOCK2-NESTED;
    REAL A, D;
    INTEGER F;
    REAL ARRAY B[1:10];
      :
      :
    A:=C;
  END BLOCK2;
    :
    :
  END BLOCK1;
```

C, E, and MAT declared in block 1 are global to block 2. The *variable*s A, B, &D are redefined in block 2 and therefore are local to block 2 and any reference to them in block 2 will reference the new meaning and not the meaning declared in block 1. In block 1 references can only be made to *variable*s defined in block 1; no variables declared in block 2 are accessible to block 1.

Upon exit from a *block* (through **END**, or by a *go to statement)* all *identifier*s declared for that *block* become inaccessible.

Aside from *identifier*s associated with the standard functions, all *identifier*s used within a program must be declared. No *identifier* may be declared more than once in any one block head.

9.1.3. Examples

```
1        10        20        30        40
LABEL L1,L2;
BOOLEAN B1,B2;
INTEGER A,B,C,D;
REAL TRACK,ELEVATION,ANGLE;
ALPHA DATE,NAME;
OWN INTEGER E,CTR,BLKENTRY;
```

9.2. TYPE DECLARATIONS

9.2.1. Syntax

<type declaration>: : = <local or own type> <type list>

<local or own type>: : = <type> | **OWN** <type>

<type> : : = **INTEGER** | **REAL** | **COMPLEX** | **BOOLEAN** | **ALPHA** | **DOUBLE** | **TASK** | **EVENT**

<type list> : : = <simple variable> | <type list> , <simple variable>

9.2.2. Semantics

Type declarations serve to declare certain *identifiers* to represent *simple variables* of a given type.

Upon first entry into a *block*, all *variables* are initially set to zero.

An additional *declarator*, **OWN**, may mark a declaration. *Variables* declared **OWN** retain their value upon exit from the block; at reentry into the particular block, *variables* maintain the value they had upon last exit. Values of *variables* not declared **OWN** are cleared to zero upon reentry into the *block* and must be re-initialized by the program.

Type declared *variables* are represented as follows:

INTEGER – Integral values represented internally by 36 bits. The range of an integer (in magnitude) is zero through $2^{35}-1$ inclusive.

REAL – Floating point numbers internally represented by 9 bits for sign and exponent of the number and the exponent and 27 bits for the fraction. The range of a real number (in magnitude) is 10^{-38} to 10^{38} with approximately 8 digits of precision. Any real quantity which is less than 10^{-38} is represented by zero.

DOUBLE – Floating point numbers internally represented by 12 bits for sign and exponent of the number and 60 bits for fraction. The range of a **DOUBLE** number (in magnitude) is approximately 10^{-308} to 10^{308} with approximately 18 digits of precision.

COMPLEX – Complex values of the form A + i*B where A and B are real numbers.

BOOLEAN – Truth values, **TRUE** or **FALSE**.

ALPHA – Any set of 6 or less characters.

TASK and **EVENT** types are used in connection with the *activity statements* for multiprocessing of ALGOL procedures and will be discussed in Section 12, ACTIVITY CONTROL.

9.2.3. Examples

```
1         10        20        30        40
INTEGER A, B, C, DOG;
REAL R1, R2;
DOUBLE DR1;
COMPLEX CX1, CX2;
BOOLEAN B1, B2, B3;
ALPHA NAME, DATE, IDENT;
```

9.3. LABEL DECLARATION

9.3.1. Syntax

<label declaration> : : = **LABEL** <label list>

<label list> : : = <label>|<label list>,<label>

<label> : : = <identifier>

9.3.2. Semantics

*Label*s are used to name certain points within a program so that these named points may be referenced by *go to statement*s.

The label declaration in a block heading defines those labels that will be used in that block to mark statements.

9.3.3. Restrictions

- All *label*s must be declared and have meaning only within the block in which they are declared.
- In nested blocks, a *label* must be declared in the head of the innermost block in which the associated labelled statement appears.
- A block can be entered only through the block head.
- If any statement in a *procedure body* is labelled, this *label* must be declared in the *procedure body*.
- A *procedure body* itself must not be labelled.

9.3.4. Examples

```
1         10        20        30        40
BEGIN COMMENT BLOCK1;
INTEGER I;
LABEL L1, L2, L3, L4;
L1: ---
L2: BEGIN COMMENT BLOCK2;
LABEL L4, L5;
SWITCH SW1:=L1, L4, L5;
GO TO L4;
L4: ---
GO TO L3;
GO TO SWT1[I];
L5: ---
END BLOCK2;
L3: ---
L4: END BLOCK1;
```

Note that in the previous example L4 is redeclared in Block 2. The **GO TO** L4 statement of Block 2 will result in a jump to L4 in Block 2. The **GO TO** L3 statement of Block 2 will result in a jump to the outer block, thereby causing an end to Block 2. The reference to SW1 in Block 2 could result in a jump within Block 2 or to Block 1, depending on the value of I.

9.4. SWITCH DECLARATION

9.4.1. Syntax

<switch declaration> : : = **SWITCH** <switch identifier> : = <switch list>

<switch identifier> : : = <identifier>

<switch list> : : = <designational expression> | <switch list>, <designational expression>

9.4.2. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 0, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

A designational expression that appears as an element of a *switch list* will be evaluated at the level of declaration of the switch, not at the level of the switch designator if the levels are different. Note the following example:

```
1        10        20        30        40        50        60        80
BEGIN
  INTEGER A,B;
  LABEL L1,L2,L3,L4;
  SWITCH T:=L2,L3,L4;
  THEN L2 ELSE L3,L4;
  IF A=B
BEGIN
  INTEGER I;
  REAL A;
  LABEL L1,L2;
  GO TO T[I];
END;
END;
```

The designational expression TESTPATH [I], is at a different block level from the *switch declaration,* TESTPATH . Furthermore, A, L1, and L2 have been re-declared in the inner block. If the value of I (The *subscript expression* of the switch designator) should result in a reference to the first or second element of the switch declaration, then A, L1, or L2 of the *outer* block would be referenced; i.e. the reference would be at the level of the *switch declaration.*

9.4.3. Example

```
1         10        20        30        40
SWITCH  SW:=LAB1,LAB2;
SWITCH  SW1:=L1,L2,IF B THEN SW[I] ELSE L3;
SWITCH  ERRORSWITCH:=START,PARITYER,EXIT;
```

9.5. FORWARD REFERENCE DECLARATION

9.5.1. Syntax

<forward reference declaration> : : = <forward procedure declaration>|<forward switch declaration>

<forward procedure declaration> : : = **FORWARD** <procedure type> **PROCEDURE** <procedure list>

<procedure type> : : = <empty> | <type>

<procedure list> : : = <procedure identifier> | <procedure list>,<procedure identifier>

<forward switch declaration> : : = **FORWARD SWITCH** <switch identifier list>

<switch identifier list> : : = <switch identifier> | <switch identifier list>, <switch identifier>

9.5.2. Semantics

The *forward reference declaration* establishes the existence of a forward referenced *procedure* or *switch* which cannot be declared in the normal manner prior to its use. A *procedure* or *switch* must be defined before it may be referenced in the program.

However, in the special cases such as (1) a *procedure* calls another *procedure* which in turn references the first *procedure;* (2) a *switch* references another *switch* which in turn references the first *switch;* it is impossible to declare both *procedures* or *switches* before they reference each other. In order to use such recursive references, the *forward reference declaration* is used. The *procedure identifiers* in the *procedure list* or the *switch identifiers* in the *switch identifier list* will be marked temporarily as defined. The normal *procedure declaration* or *switch declaration* must appear later in the program.

9.5.3. Examples

```
1         10        20        30        40        50
BEGIN FORWARD PROCEDURE P;
  PROCEDURE Q;
  BEGIN
    :
    :
  P;
    :
    :
  END;
  PROCEDURE P;
  BEGIN
    :
    :
  Q;
    :
    :
  END;
  INTEGER I, J, K;
  LABEL L1, L2, L3;
  FORWARD SWITCH SW2, SW3;
  SWITCH SW1 := L1, L2, IF A THEN SW2[I] ELSE SW3[I];
  SWITCH SW2 := L3, SW1[J], SW3[J];
  SWITCH SW3 := L1, L3, SW1[K];
    :
    :
END;
```

9.6. ARRAY DECLARATIONS

9.6.1. Syntax

<array declaration> : : = <array type> **ARRAY** <array list>|<array type> **CORE ARRAY** <array list>

<array type> : : = <empty> | <type> | **OWN**<type>

<array list> : : = <array segment>|<array list> <array segment>

<array segment> : : = <array identifier> [<bound pair list>]|<array identifier>, <array segment>

<bound pair list> : : = <bound pair> | <bound pair list>,<bound pair>

<bound pair> : : = <lower bound> : <upper bound>

<lower bound> : : = <arithmetic expression>

<upper bound> : : = <arithmetic expression>

9.6.2. Semantics

The concept of 'array' considerably enhances the power of ALGOL 60. An array can be considered to be a sequence of variables, referenced by the same identifier, but with each variable in the sequence being uniquely addressable by its fixed location in the sequence.

The *array declaration*, then, defines one of several *identifiers* to represent multi-dimensional arrays of subscripted variables, and gives the dimension of the arrays, the bounds of the subscripts, and the type of the variables.

Arrays, like *simple variables*, may be declared to be **OWN** or by default are local to the block of declaration. Initially all arrays are set to binary zero. If declared **OWN**, the value assigned to each element of the array must be preserved at the end of the *block of declaration* for the array and restored upon re-entry. This concept implies non-stack storage; UNIVAC 1108 Extended ALGOL, however, utilizes the dynamic stack to contain an **OWN** array while the block in which it was declared is active, but maintains a current copy of the **OWN** array on drum to reinitialize the array upon re-entry of its block of declaration.

Arrays must be declared to be of type **REAL, INTEGER, DOUBLE, COMPLEX, BOOLEAN,** or **ALPHA;** with all elements of any array being of the same type. If type is not specified, **REAL** is implied.

The dimension of an array is fixed at declaration time by the number of *bound pairs* specified; each reference to an array element, therefore, must list an entry for each dimension.

The *bound pairs* of the *array declaration* define the *lower bound* and *upper bound* for each dimension specified. The syntax permits the bounds to be defined by *arithmetic expressions*, the bounds, then, may be fixed at compilation time if an *integer constant* is given for the bounds, or the bounds may vary if an expression of variable primaries is used to define the bounds.

A dynamic, or variably bound, array is assigned storage in the stack according to the current value of the variables defining the bounds. In order for such variables to have a value, they must be declared in a block that is global to the block of the *array declaration*. For this reason, dynamic arrays may not be declared in the outermost block and dynamic **OWN** arrays are not permitted.

An array is defined only when the values of all *upper bounds* are not smaller than those of the corresponding *lower bounds*. The result of a *subscript expression* must be of type **INTEGER;** if the result is not of this type, the compiler will generate a transfer to type **INTEGER.**

The limit for the value of the size of any dimension is 4096. If **CORE** array is declared, storage will be allocated in the stack for the entire array at declaration time. In the absence of the word **CORE,** only the area specified by the rightmost *bound pair* will be reserved in the stack; the entire non-core, or segmented array, will be resident on the drum within the scope of the non-core *array declaration.*

The programmer need not be concerned with the segmenting of arrays since the compiler generates code to execute this function.

It should be emphasized that run time will be increased in direct proportion to the frequency of referencing different sets† of array data.

All one dimension arrays are **CORE** arrays, even if the word **CORE** is omitted in the declaration. At every array reference, the values of the *subscripts* are checked to see that the address designated is contained within the array. If it is not, a non-recoverable run time error will result.

9.6.3. Examples

```
1        10        20        30        40        50        60        80
ARRAY A[1:10,0:M:5,0:N,-5:12];
OWN INTEGER ARRAY KEEP[1:X[50];
REAL CORE ARRAY BETA[X:Y,X+10:Y+10,X**2:Y**2];
```

†Set is defined as the stack-resident area of a segmented array defined by the rightmost bound pair, for a two dimensional array, it is one row.

9.7. ABSORB DECLARATION

9.7.1. Syntax

<absorb declaration> : : = **ABSORB** <subscripted variable>,<absorb array declaration>

<absorb array declaration> : : = <sarray type> **ARRAY** <array segment>

<sarray type> : : = <type> | <empty>

<array segment> : : = <array identifier>[<bound pair list>] | <array identifier>, <array segment>

<bound pair list> : : = <bound pair> | <bound pair list>, <bound pair>

<bound pair> : : = <lower bound>:<upper bound>

<lower bound> : : = <arithmetic expression>

<upper bound> : : = <arithmetic expression>

9.7.2. Semantics

The *absorb declaration* allows two or more arrays to share common storage.

The *subscripted variable* must refer to a previously declared **CORE** array; to be henceforth identified as the absorbing array. The absorbing array may absorb any number of arrays with the same *bound pair* specification in the same *absorb declaration.* Arrays with different *bound pair* specifications may be absorbed by the same absorbing array in individual *absorb declarations*.

The number of words in the absorbing array from the point of absorption must be equal to or greater than the number of words in the absorbed array. The mapping of an absorbed array into an absorbing array is accomplished at run time; therefore, both absorbing and absorbed arrays may have dynamic bounds.

The type fields of the absorbed and absorbing array need not be the same.

9.7.3. Restrictions

- Absorbing arrays must be **CORE** arrays.
- Absorbed arrays may not be declared with **OWN** or **CORE** specifications, these attributes are inherited from the absorbing array.
- An array may be absorbed only once, since it is absorbed by a declaration.
- An absorbed array may not itself absorb arrays.

9.7.4. Examples

```
1         10        20        30        40        50
REAL ARRAY MATRIX [1:100];
INTEGER CORE ARRAY SUM[I:K**2, J:(K+1)*2];
:
:
ABSORB MATRIX[10], INTEGER ARRAY I[1:20];
:
ABSORB MATRIX [80], DOUBLE ARRAY D [0:9];
:
ABSORB MATRIX [1], INTEGER ARRAY I, J, K[1:20];
:
ABSORB MATRIX [1], ALPHA ARRAY LETTER[L1:U1, L2:U2];
:
ABSORB SUM [X, Y], REAL ARRAY RES[1:50];
:
ABSORB SUM [X+2, 5], INTEGER ARRAY PART [L1:U1, L2:U2];
```

9.8. DEFINE DECLARATION

9.8.1. Syntax

<define declaration>: : = **DEFINE** <definition list>

<definition list> : : = <definition part> | <definition list>,<definition part>

<definition part>: : = <defined identifer> = <definition> #

<definition> : : = <symbol> | <definition> <symbol>

<symbol>: : = <delimiter> | <identifier> | <constant>

9.8.2. Semantics

The *define declaration* defines an *identifier* as a set of basic ALGOL components. The appearance of a *defined identifier* results in it being replaced by its associated *definition*. Such replacement must result in a legitimate ALGOL construct. When *identifiers* are used in a *definition* they will, at the time the *defined identifier* is used, refer to the meaning applicable at the level of the *define declaration*. This point is important only when the *defined identifier* is used in a nested block which contains a declaration affecting an identifier in the defining construct. A *define declaration* may contain another *defined identifier*.

9.8.3. Restrictions

- Any *basic component* used in a *definition* must be completely contained therein; for instance, a part of an *identifier* cannot be used in a *definition* with the other part being supplied later in the program.

  This rule also applies to *delimiters, unsigned numbers* and *strings*.

- All *identifiers* in the *definition* must be previously declared.
- The meaning of an *identifier* in the *definition* given is always employed whenever the defined *identifier* is used.
- The well-formed construct of a *definition* must not contain a *declarator* or a *specificator*.
- A *defined identifier* must not be used in a *fill statement*, a *format declaration* nor as a *formal parameter*.

9.8.4. Example

```
1         10        20        30        40        50        60
DEFINE RKUNGEKUTTA#,ROOT=(-B+SQRT(B**2-4*A*C))/2A#;
DEFINE INTEGRAL#=INTEGRAL(X,Y,Z)#;
DEFINE FIMDO=FOR I:=1 STEP 1 UNTIL M DO#;
DEFINE ASG=A:=INT#;
  or equally valid
DEFINE ASG=A:=INT;#;
```

9.9. MONITOR DECLARATION

9.9.1. Syntax

<monitor declaration> : : = **MONITOR** <mfile part> (<monitor list>)

<mfile part> : : = <empty> | <file identifier>

<monitor list> : : = <monitor list element> | <monitor list>,<monitor list element>

<monitor list element> : : = <simple variable> | <subscripted variable> | <array identifier> | <switch identifier> <procedure identifier> | <label>

9.9.2. Semantics

The *monitor declaration* allows the programmer to display *variables* whenever their values are changed and to trace the path of a program by displaying *labels* and *switches* when they are encountered.

The *file part* of a *monitor declaration* is optional except as noted in restrictions. When given it does *not* specify that this particular *monitor list* is placed in the given file. Instead, all monitored elements are placed on the most recently given file.

When a monitored *variable* or *array* or *procedure identifier* is used in the left hand part of an assignment statement; the following information is written on the designated file:

identifier : = value assigned or

array identifier [S_1, S_2, ..., S_n] : = value assigned

When a monitored *switch* is encountered, the following information is written:

switch identifier [V] , where V is the current value of the subscript expression.

When a monitored *label* is encountered the following information is written:

label identifier

When an *array identifier* is given, all *variable*s in the array are monitored. If a *subscripted variable* is given only that element of the array will be monitored.

9.9.3. Restrictions

- In each block before a reference to a monitored identifier occurs at least one MONITOR declaration with a non-empty file part must have appeared.
- All *monitor list element*s must have been previously declared and labels may only be monitored in the *block* in which they are declared.

9.9.4. Examples

```
1        10        20        30        40        50        60        80
BEGIN    REAL      A, B, C, D;
         INTEGER   S;
         LABEL     L1;
         FILE      OUT1      FILE 1 (1, 2, 2);
         FILE      OUT2      FILE 2 1 (1, 2, 2);
         MONITOR   FILE 1 (A, B, L1);
         MONITOR   FILE 2 (C, D);
         COMMENT   A, B, C, D, L1 WILL BE MONITORED ONLY IF THE STATEMENT
                   WAS THE LAST STATEMENT GIVEN;
BEGIN    REAL      A;
         INTEGER ARRAY Y 1 (1:10);
         FILE      OUT3      FILE 3 1 (1, 2, 2);
         MONITOR   FILE 3 (A, I [S]);
         COMMENT   ALL VARIABLES IN ARRAY NOW MONITORED FILE 3;
         :
         END;
         :
COMMENT  UPON LEAVING THE NEST BLOCK MONITORED VARIABLES
         AGAIN GO ON FILE 2;
         END;
```

9.10. DUMP DECLARATION

9.10.1 Syntax

<dump declaration> : : = **DUMP** <dump part>

<dump part> : : = <file identifier> (<dump list>)<label>:<dump condition>

<dump condition> : : = **WHEN** <Boolean expression> | **WHEN** <Boolean expression>
EVERY <arithmetic expression> | **WHEN** <Boolean expression>
EVERY <arithmetic expression> **WHILE** <Boolean expression>|
WHEN <arithmetic expression>| **WHEN** <arithmetic expression>
EVERY <arithmetic expression>| **WHEN** <arithmetic expression>
EVERY <arithmetic expression> **WHILE** <Boolean expression>|<empty>

<dump list> : : = <dump list element>|<dump list>,<dump list element>

<dump list element> : : = <simple variable> | <subscripted variable> | <array identifier> | <label>

9.10.2. Semantics

The *dump declaration* allows a programmer to display *variable*s at any point during the execution of his program.

The *dump condition* is evaluated when the *label* is encountered. If there is a *dump condition* and the **WHEN** and **WHILE** portions of the condition have been satisfied, the *dump declaration* becomes active. The **DUMP** will then be executed each time its label is encountered provided the **WHILE** and **EVERY** portions of the *dump condition* are satisfied (if present). Once the *dump declaration* has become active and the **WHILE** condition is not satisfied the **DUMP** becomes inactive and remains so until the block in which it is declared is re-entered. If a *dump condition* is not present, the **DUMP** will be executed each time its *label* is encountered.

In the case of **WHEN** *dump condition* it may be represented either by an arithmetic or *Boolean expression*. If the condition is an *arithmetic expression,* it will refer to the number of times the label named in the *dump declaration* has been encountered. If it is a *Boolean expression,* a true value means the condition is satisfied.

All *dump list element*s are placed on the file given immediately preceding the *dump list*.

The *dump list elements* are displayed in the same format as in the *monitor declaration*.

9.10.3. Restrictions

The colon following the *label* is given only when a *dump condition* is present.

9.10.4. Examples

```
1         10        20        30        40        50
BEGIN
  REAL A, B, C, D; INTEGER J, I; BOOLEAN BOOL;
  ARRAY X, Y, Z[1:10];
  LABEL L1, L2, L3, L4;
  FILE OUT F1 1((1,22));
  FILE OUT F2 1((1,22));
  DUMP F1 (L1, A, B, C) L1: WHEN 5 EVERY 1;
  DUMP F2 (L1, X, Y[J]) L1;
  DUMP F1 (Z) L2: WHEN X[1]=A EVERY 2*I WHILE BOOL;
    :
    :
    :
  END;
```

9.11. I/O DECLARATIONS

I/O declarations will be covered in Section 11.2.

10. PROCEDURE DECLARATIONS

10.1. GENERAL PROCEDURE DECLARATIONS

10.1.1. Syntax

<general procedure declaration>: : = <procedure declaration> | <external procedure declaration>

10.1.2. Semantics

The following pages describe the syntax and semantics for both the *procedure declaration* and *external procedure declaration*. The *external procedure declaration* is a special feature of UNIVAC 1108 Extended ALGOL and enables communication between an ALGOL program and programs compiled at a previous time written in ALGOL or FORTRAN.

10.2. PROCEDURE DECLARATION

10.2.1. Syntax

<procedure declaration>: : = **PROCEDURE** <procedure heading> <procedure body> |
<type> **PROCEDURE** <procedure heading><procedure body>

<procedure heading> : : = <procedure identifier> <formal parameter part> ;
<value part><specification part>

<procedure identifier> : : = <identifier>

<formal parameter part> : : = <empty> |(<formal parameter list>)

<formal parameter list> : : = <formal parameter> | <formal parameter list>
<parameter delimiter> <formal parameter>

<formal parameter> : : = <identifier>

<value part> : : = **VALUE** <identifier list> ; | <empty>

<identifier list> : : = <identifier> | <identifier list>,<identifier>

<specification part> : : = <specifier> <identifier list>; |<specification part>
<specifier><identifier list>; |<empty>

<specifier> : : = **LABEL** | <type> | **SWITCH** | **PROCEDURE** | <type> **PROCEDURE** |
ARRAY | <type> **ARRAY** | **FILE** | **LIST** | **FORMAT** | **SWITCH**
FORMAT | **SWITCH LIST** | **SWITCH FILE** |

<procedure body> : : = <statement>

10.2.2. Semantics

The ALGOL procedure provides a convenient means of defining an algorithm and giving it a name so that it may be referenced or called anywhere within the scope of the declaration of the *procedure identifier.* Furthermore, different *actual parameters* or arguments may be passed to the procedure at each call.

The *procedure declaration* consists of the *procedure heading* and the *procedure body.* The identifier of the procedure appears in the *procedure heading* followed by a list of names which designate *formal parameters.* The *formal parameter list* may be empty, but if it is not, each *formal parameter* name must be further defined by the *specification part.* Since the dimension of an array is not indicated by the specification, all arrays, including those intended to be of single dimension, † must be specified to be **CORE** arrays if that is the programmers intent. If the word **CORE** does not precede **ARRAY** in the *specification part,* the code generated by the compiler at the array reference will be that for a segmented array.

Formal parameters are really dummy variables to which an *actual parameter* value of identical type and kind will be passed when the procedure is activated by a procedure call statement. *Formal parameters* may be called by value or called by name. If the *formal parameter* identifier appears in the value part then when the procedure is entered the value of the corresponding *actual parameter* will be calculated and in effect will be substituted for the *formal parameter* at each occurrence within the *procedure body.* If a *formal parameter* does not appear in the *value part,* it is assumed to be called by name. This means that at every reference to the *formal parameter* within the *procedure body,* the value of the *actual parameter* will be calculated (or recalculated). It is possible, then, for the value of a *formal parameter* called by name to vary from reference to reference within the *procedure body.*

The attributes of *actual parameters* and *formal parameters* are checked for equivalence at run time; if they are not identical in kind and type, an unrecoverable run time error results. (See Appendix A Error Diagnostics).

An exception to the rule of complete agreement between *actual parameters* and *formal parameters* is that an *actual parameter* that is defined to be an *array row* may be passed to a *formal parameter* that is specified to be an **ARRAY** and is referenced as a single dimension array within the *procedure body.*

If a *type declaration* appears before the word **PROCEDURE** in the heading, a function procedure is declared. A function procedure is an *arithmetic* (or Boolean) *primary* and may only be referenced in *expressions.* The name of the function procedure must appear in a *left part list* of an *assignment statement* in the *procedure body.* This is, of course, necessary so that a function procedure may exit with a value. A *go to statement* leading out of the *procedure body* is invalid. A *label,* however, may be passed as a parameter; thereby enabling an exit from the procedure other than the terminating **END** of the *procedure body.*

Declarations may be made in the *procedure body,* defining *identifiers* that are local to the *procedure body.*

† *All arrays used as single dimension must be declared CORE.*

10.2.3. Example

```
1        10        20        30        40        50        60        80
PROCEDURE MATEMULT ((RMAX, CMAX, RVMAX, M, V, P, ERR));  VALUE RMAX, CMAX,
RVMAX;  INTEGER RMAX, CMAX, RVMAX;  REAL ARRAY M, V, P;  LABEL ERR;
COMMENT MATEMULT MULTIPLIES A MATRIX M OF RMAX ROWS AND CMAX
COLUMNS BY A VECTOR V OF RVMAX ROWS. THE PRODUCT IS P OF RMAX ROWS;
BEGIN
   INTEGER R, C;
   IF CMAX NEQ RVMAX THEN GO TO ERR;
   FOR R:=1 STEP 1 UNTIL RMAX DO
   BEGIN
      SUM:=0;
      FOR C:=1 STEP 1 UNTIL CMAX DO
         SUM:=SUM + M[R,C] * V[C];
      P[R]:=SUM;
   END;
END;
```

10.3. EXTERNAL PROCEDURE DECLARATION

10.3.1. Syntax

<external procedure declaration> : : = **EXTERNAL** <kind> **PROCEDURE** <proc list> | **EXTERNAL** <kind> <type> **PROCEDURE** <proc list>

<kind> : : = FORTRAN | <empty>

<proc list> : : = <proc part> | <proc list>,<proc part>

<proc part> : : = <procedure identifier> |
<procedure identifier> (<efile part><element part>)

<efile part>: : = <file name>. | <empty>

<element part> : : = <element name> | <element name>/<version name> | <empty>

10.3.2. Semantics

External procedures are procedures whose bodies do not appear in the main program. They are compiled separately and linked to the main program at its execution. The *external procedure declaration* serves the purpose of informing the compiler of the existence of these procedures, their types (if any), and the proper manner to construct the necessary linkages.

The file and element information is given in standard UNIVAC 1108 Executive System format. This information is used by the ALGOL compiler when it generates maps for run-time execution.

If no file or element is given, the external procedure is assumed to be in the file TPF$ and will be automatically collected by the system. If only a <*file name*> is given, the compiler adds a LIB *file name* statement to its map of the object code. This file is assumed to have been VPREP'ed and it will be searched for any external procedure names. Thus, if a user has many external procedures in a file, he need only give that *file name* once. If the user wishes to name specific elements, he should give *file name . element name/version name*.

Examples:

```
1        10        20        30        40        50        60        80
EXTERNAL PROCEDURE X(FILE1),Y,Z;
COMMENT THE PROCEDURES X AND Y ARE IN FILE1. PROCEDURE Z IS
        Z MAY BE IN THE BEFORE FILE$;
EXTERNAL FORTRAN PROCEDURE A,B(FILE2.MATE);
COMMENT THE PROCEDURES A AND B ARE IN FILE2 AND ARE
        OF FILE2.MATE KIND;
```

NOTE: The effect of *external procedure declarations* in cumulative, i.e., after 6 files have been specified, an external procedure may reside in any of them.

The word 'FORTRAN' has special significance only in this context. Procedures of kind <empty> are ALGOL procedures and are treated exactly like an ordinary procedure declared within the program. However, they need not be written in ALGOL language. Procedures of kind 'FORTRAN' are FORTRAN subroutines or functions.

10.3.2.1. ALGOL External Procedures

An ALGOL program which consists entirely of a procedure is nonexecutable because it contains only a procedure declaration (section 10.1). When such a program is compiled, the name of the procedure is marked as an entry point when the program is entered into the program file. The first six characters of the procedure name must define it. Such a procedure may be referenced from another ALGOL program as an external procedure.

```
1        10        20        30        40        50        60        80
ALGOL;STREAM(FILE2,STREAM(FILE2,ALGOL));
PROCEDURE MULT(I,J,RESULT);
VALUE I,J; RESULT I,J;
BEGIN INTEGER I,J; REAL RESULT;
COMMENT THIS PROCEDURE MULTIPLIES TWO INTEGERS AND
        PRODUCES A REAL RESULT;
RESULT:=I*J;
END PROGRAM;
```

In order to call this procedure from another program the following statements could be used:

```
1         10        20        30        40
BEGIN INTEGER A, B;
REAL PRODUCT;
EXTERNAL PROCEDURE MULT(FILE2.ALGI);
A:=B:=2;
MULT(A, B, PRODUCT);
END;
```

10.3.2.2. FORTRAN Subprograms

A FORTRAN subroutine or a FORTRAN function may be made available to an ALGOL program by the *external procedure declaration.*

*Actual parameter*s in calls on such procedures may be either *expression*s or arrays. The FORTRAN subprogram is a subroutine or function depending on the absence or presence of *type* in the *external procedure declaration.* A FORTRAN function is used like an ALGOL functional procedure i.e., as an expression. For example, if MULT (above) were a FORTRAN subroutine:

```
1         10        20        30        40        50        60        80
@FORI, SI   TPF$. PRO, TPF$. RO
    SUBROUTINE MULT(I, J, RESULT)
    RESULT = I*J
    RETURN
    END
```

In order to call this subroutine from an ALGOL program the following statements are used:

```
BEGIN INTEGER A, B; REAL PROCEDURE MULT; COMMENT NO FILE IS GIVEN BECAUSE
    MULT IS IN TPF$;
A:=B:=2;
MULT(A, B, PRODUCT);
END;
```

11. INPUT/OUTPUT

11.1. GENERAL DESCRIPTION

The purpose of this section is to acquaint the user with the interfaces of the ALGOL I/O Library and the 1108 Executive System and with the processes initiated during program execution by compiler generated linkages derived from *I/O declarations* and *I/O statements*.

This section is not intended to pre-empt, supplant, or modify any part of the 1108 Executive Programmers Reference Manual. Therefore, the 1108 PRM should be referenced for clarification of any statement regarding the Executive system.

Descriptions of Executive requirements are intended only as an aid to understanding the methods employed in the internal processes of conversion from ALGOL *I/O declarations* and *I/O statements* to Executive interfaces, and consequently to assist the programmer in the construction of efficient Algol *I/O statements* and *I/O declarations*.

11.1.1. File Assignments

The Executive requires that files to be read or written must be assigned to the requesting run by use of Executive Control Statements. The control statements may be submitted to the Executive by external or internal methods. Externally submitted control statements are those accessed by the Executive from run streams either introduced to the System as card input or as pre-stored drum files or as elements within a file. Internally, control statements are submitted by an executing program to the Executive via an Executive Request for the Control Statement Formatter (ER CSF$).

The ALGOL Compiler generates File Assignment Control Statements as the products of *file declarations*. The File Assignment Statements are submitted to the Executive either internally or externally according to the device type of the file.

11.1.1.1. Tape Files

Tape File Assignment Statements are externally submitted as they are inserted into the RUN stream by the ALGOL initializing element prior to execution of the program. The Executive can, therefore, satisfy the tape unit requirements before initiating execution of the program. If the tape files or tape units are not available the run will be delayed until the facility requirements are satisfied.

Tape files are never considered as permanent files by the ALGOL I/O System, i.e., tape files are never catalogued. The T option (temporary) is always specified in a tape File Assignment Statement. Other possible options are H- high density, L - low density, and E - even parity (BCD).

11.1.1.2. Drum Files

Drum File Assignment Statements are submitted internally during execution via ER CSF$. Drum files are always assigned to FASTRAND or Simulated FASTRAND.

(1) SERIAL and RANDOM Files

The C option (catalogue if RUN completes normally) is always specified for files declared with **SAVE** in the *file lock part* and/or **OUT** or empty *input/output part* and with *drum file description part* present.

If **SAVE** is specified cataloguing is accomplished at the end of the block in which the file was declared, or at occurrence of a *close statement* or *lock statement.*

If **SAVE** is not specified and a *lock statement* is not encountered, cataloguing is inhibited when the file is closed.

For Files declared **IN,** the A option, implying reference to a previously catalogued file, is always specified.

Absence of the *drum file description part* in a *file declaration* always causes assignment with the A option.

(2) UPDATE Files

UPDATE may not be used for temporary files. The A option, implying reference to a previously catalogued file, is always specified.

UPDATE may not be used to create a file. However, a file declared **SERIAL** may be written and catalogued and released with the *lock statement* and then in another block, nested or disjoint, declared as an **UPDATE** file and processed.

11.1.1.3. Punch and Print Files

Punch and print File Assignment Statements are internally submitted to the Executive during program execution via ER CSF$. FASTRAND or Simulated FASTRAND is always specified as the associated device.

Punching and printing are processed in the Alternate File Mode causing output to the assigned file rather than a punch or printer. At RUN completion or programmed file release punching and printing is initiated under Symbiont controls. The file is then no longer available to the executing program.

Printer back-up tape files may be declared, however, actual printing of the tape file can only be accomplished by an Executive feature totally unrelated to the ALGOL I/O system, and therefore, becomes completely a user responsibility.

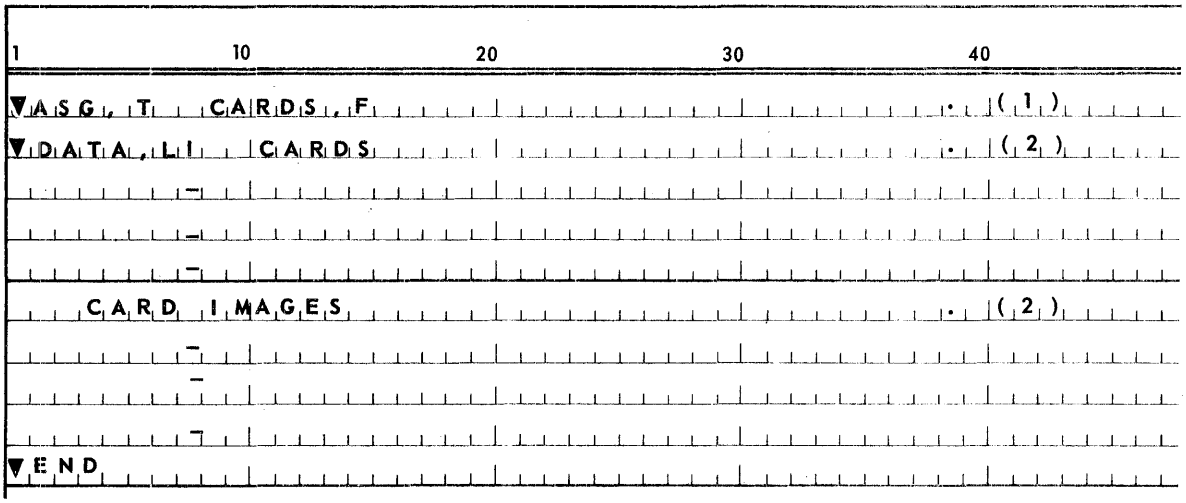
11.1.1.4. Card Files

Card files are introduced to the ALGOL system only through the use of the DATA processor. The DATA processor is invoked by the processor call statement ▼DATA. Card images following the ▼DATA Statement and preceding the related ▼END statement are written to the file specified in the ▼DATA statement and previously externally assigned by the user.

The file name specified in both the ▼ASG and ▼DATA statements must be identical to the file name specified in the corresponding ALGOL *file declaration.*

11.1.1.4.1. Card Files on Fastrand

Card files may be placed on Fastrand in the following simple manner:

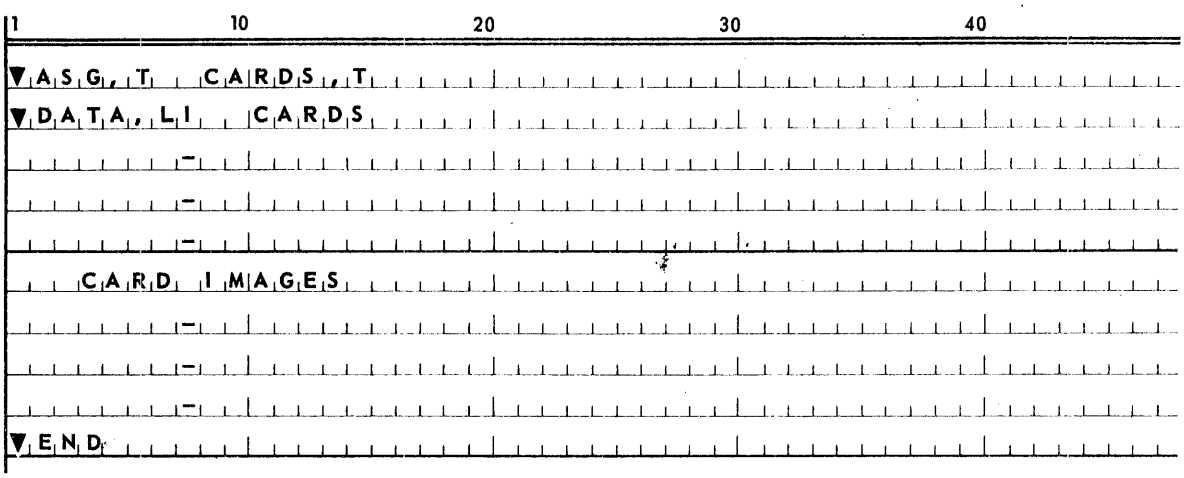


(1) The ▼ASG statement causes assignment of a FASTRAND file of the name CARDS. The "T" option specifies the file is to be temporary, i.e., the file will be released at RUN completion or at the occurrence of the Executive Control Statement, ▼FREE CARDS. The "F" specifies that FASTRAND is the device to be assigned.

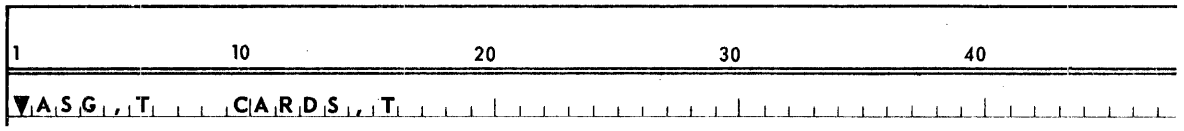
(2) The ▼DATA and ▼END statements are as stated earlier. The "I" option is required to specify "Initial" entry into the system of the following images. The "L" option causing the images following to be listed as they are written to the file specified.

11.1.1.4.2. Card Files on Tape

A card file to be used as input frequently to an ALGOL program may be placed on tape by use of the DATA statement in the following simple manner:



The external control statement required in an ALGOL program RUN stream to assign the card file, now on tape, would be as follows:



where the device specification "T" following the file name indicates tape input.

Provided that the Assignment Statement precedes program execution in the RUN stream, the ALGOL I/O system will accept the specified DATA tape file as card input and process it as such. The input of card images from tape is restricted to tape files created by the ▼DATA statement and tape files outputted by the ALGOL I/O system as blocked items. The creation of the DATA tape file is unrelated to the ALGOL program. The process is an Executive feature.

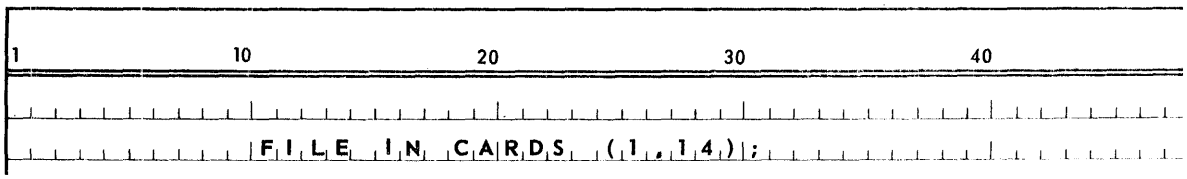
11.1.1.5. Compiler Generated Assignment Table

At the completion of an ALGOL compilation a table of File Assignment Statements is listed. The table reflects, with exceptions, the actual control statements to be submitted to the Executive by the ALGOL initializing element and/or by the ALGOL system at execution.

The exceptions noted above are as follows:

For files declared as **DRUM** specifying a *drum file description part* the *size of areas* and *number of areas* may be specified as arithmetic expressions and therefore are unknown values at time of compilation. The FASTRAND Assignment Statement requires specification of the FASTRAND area size to be reserved for the file. The specification field for the reserve is filled with asterisks during compilation. The asterisks are replaced with the actual value during execution when the value is known.

(2) Ambiguity in device association can result from *file declaration*s intended to specify labelled input tapes or card files. Example:

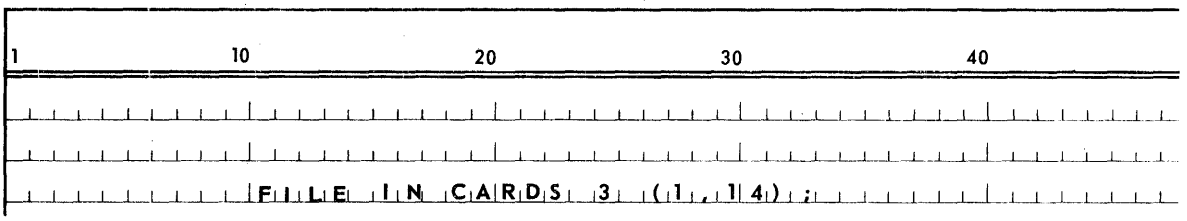


The compiler cannot determine the intended device and therefore assumes tape and produces a tape File Assignment. The statement is listed in the table of assignments which is printed following compilation under the heading CARDS or TAPE.

At the time of program execution the ALGOL initializing element will resolve the conflict. If a DATA file of the name in question has been previously assigned to the RUN, the generated tape Assignment Statement will not be submitted to the Executive. If the file name is unknown by the Executive, i.e., in the absence of an external assignment, the tape Assignment Statement will be utilized.

Inefficiency in the file assignment due to ambiguity in device association can be easily avoided by the use of the *output media part* value "3" in *file declaration*s specifying card files.

Example:



The compiler will not assume a tape file if "3" is used.

(3) All card, printer, and punch File Assignment Statements are listed under the heading PERIPHERALS. The card File Assignment Statements are never submitted to the Executive. They are listed to acknowledge the file and its associated device.

(4) The file names of files declared as designated devices are listed under that heading. Device association will be attempted by the ALGOL initializing element as described in the following section on user external assignments.

11.1.1.6. User External Assignments

11.1.1.6.1. External Assignment Statements

The ALGOL programmer may override any or all compiler generated File Assignment Statements. The presence of an external File Assignment Statement in the run stream will inhibit the submission of the ALGOL compiler generated statement.

The ALGOL initializing element is given control prior to execution of the ALGOL object program. The initializing element queries the Executive as to the status of each file specified in a *file declaration*. If the file is externally assigned the initializing element bypasses assignment action for the file in question. However, compatibility checks are made to insure, for example, that a file declared as output has not been externally assigned to a device restricted to input, i.e., a card file. The following table reflects allowable changes in the device declared in the file declaration via external assignments.

Device Declared In ALGOL Program As	May be Externally Assigned as
CARDS	CARDS, TAPE, FASTRAND
TAPE IN	TAPE, FASTRAND, CARDS
TAPE OUT	TAPE, FASTRAND
DRUM IN SERIAL	FASTRAND, TAPE, CARDS
DRUM OUT SERIAL	FASTRAND, TAPE
PRINTER	FASTRAND, TAPE
PUNCH	FASTRAND, TAPE
DESIGNATED DEVICE	any except CARDS

Note: Files declared RANDOM or UPDATE may not be changed by external assignment.

TABLE 11-1. Internal/External Device Assignment

Buffer and item sizes must be compatable for changed devices. Card files, as described previously, are expected to be inputted in the System Data Format as created by the DATA statement. Therefore, an External File Assignment changing a card file to a tape or FASTRAND file must specify a file created by either a DATA statement or a file outputted by the ALGOL I/O system with buffer specifications of 224 word block size, 14 or less word item size. Tape or drum serial files changed to card files must have been declared with buffer specifications of blocked items with a block size of 224 words.

A punch file changed to a tape file by an External File Assignment will produce as output a tape file in the format acceptable to the ALGOL I/O system as card input.

11.1.1.6.2. Designated Devices

If a file declared as a designated device is not externally assigned by the user the initializing element will query the console operator with the message, "file name. DEVICE".

Acceptable responses are any of the *file declaration output media types* except "3" (CARDS). Drum files must be indicated by S, R, or U responses to indicate accessing technique. Care should be taken by the programmer to avoid the console communication method of device association. It is time consuming and requires operator knowledge of the response or on the spot attendance of the programmer. It is offered by the ALGOL I/O system as a last resort effort prior to program termination action.

11.1.2. Blocking Specifications

11.1.2.1. Unblocked Records

The buffer size specified for unblocked records is accepted as the actual data word size of the block to be read or written. Three control words are added to the buffer size when creating a block.

Example:

The *blocking specifications* of

1,253

will create a block of two hundred fifty-six words. Three control words and two hundred fifty-three data words are written per block.

The control words are invisible to the user, i.e., they are never passed to the calling program when read.

FORTRAN V unformatted data blocks on tape are compatable with the above specification. FASTRAND files created and/or read by FORTRAN V yield two hundred forty-nine data words. To write or read FORTRAN V unformatted data blocks on FASTRAND the blocking specification of (1,249) is compatable.

11.1.2.2. Blocked Records

The buffer size specified for blocked records is accepted as the actual block size to be written. A single control word is affixed to each item of twenty-two or less words. An additional control word is inserted for each additional twenty-two words of an item or portion thereof.

Items or portions of items are never spanned across blocks. A bypass record is written to fill any unused words at the end of a block.

Example:

The blocking specification of

1,224,22

will create a block of two hundred twenty-four words. Nine items of twenty-two words and one control word, and one bypass record of one control word and sixteen void words will be written per block. The bypass record will never be passed to the calling program when the records are read.

FORTRAN V formatted data blocks on tape or FASTRAND are compatable with the blocked specification example presented previously. The bypass records will be ignored.

To eliminate bypass records and unused space in blocked records the user should compute buffer sizes in terms of item sizes and control words.

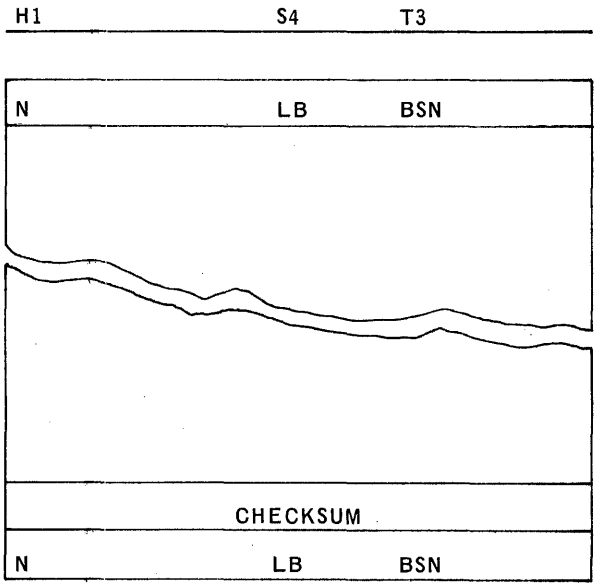
In the determination of the most efficient block size for blocked variable length records the maximum record size sould be considered as the item size to establish an acceptable minimum number of records per block.

As indicated previously, a blocked record file is produced in modified Systems Data Format.

SDFF block size is two hundred twenty-four words while ALGOL I/O permits block size to be specified by the programmer. The ALGOL I/O bypass control word would be considered as a data control word if read by the Executive System. An SDFF, acceptable to the Executive system, can be output on tape or FASTRAND through the ALGOL I/O system as an unlabelled file if the standard SDFF block size is specified and care is taken to assure that each entire block is filled with items thereby eliminating bypass records. An SDFF can be read in the forward direction only by the ALGOL I/O system provided the standard buffer size and maximum item length are properly specified. SDFF label, bypass, and end of file control words are acknowledged by the ALGOL I/O system.

11.1.2.3. ALGOL I/O Formats

11.1.2.3.1. Unblocked Records



N = number of significant words (buffer size specified)

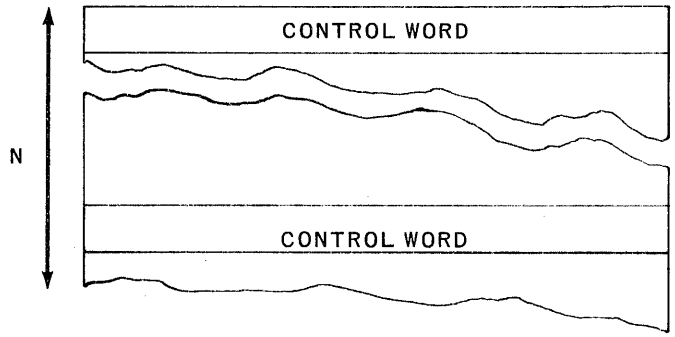
LB = last block flag

0 = not last block

1 = last block

BSN = always 0

11.1.2.3.2. Blocked Records



$N \leq 22$

Control Word

S1	S2	S3	S4	T3
it	IL	PL	ti	SEQ NO.

S1 = it = 00 = data or bypass image
= 077 = EOF image

S2 = IL = image length n

S3 = PL = length of previous image

S4 = ti = 010 = data word image
= 040 = bypass image
= 030 = EOF image

T3 = SEQ. NO. = image sequence number within logical record

11.1.3. Internal Buffering

Punch and print files are not buffered by ALGOL I/O. The PNCHA$ and PRNTA$ interfaces with the Executive provide the required buffering action.

Card files being of Systems Data Format are always read with a look ahead factor of one buffer, i.e., two blocks of two hundred twenty-four words are initially read. When the second block is accessed the first is refilled (if end of file has not been encountered). The two block look-ahead provides immediate sequential access to a minimum of twenty-eight card images. The Systems Data Format of DATA files truncates unused (blank) words at the end of the image. Therefore, depending on the content of each card the number of images per each two hundred twenty-four word block may vary from fourteen to one hundred twelve. The truncated blank words of a card image are restored upon access of the image. Card file item size is always fourteen words.

Tape and drum **SERIAL** or drum **UPDATE** files are read or written with a look ahead factor of the number of buffers specified plus one. Drum **RANDOM** files are never buffered.

Direction changes in reading unblocked serial files cause look ahead buffers to be drained and refilled with blocks read in the opposite direction.

The time wasted in this process should be carefully considered when programming frequent direction reversals. Direction changes in reading blocked records will not cause buffers to be drained and refilled unless continued reads require access of the next or previous buffer. Frequent direction changes if confined to records within the current buffer are not penalized by the system overhead required for draining and refilling buffers.

11.1.4. File Labelling

11.1.4.1. Tape File Labels

11.1.4.1.1. Output

An output tape file is initially read to test for the presence of a label. If a label exists, expiration of the purge date is checked. If no label exists or the purge date has expired, the tape is positioned at load point for a subsequent write operation. If purge date has not expired, the tape is rewound to load point and a console message is outputted to inform the operator of this status. A response of G to the console message will permit the file to be overwritten despite the purge date. A response of A is interpreted as an indication that a new tape had been mounted and the label check will be reinitiated.

A standard one hundred twenty word label is written as the first block of a labelled output tape.

11.1.4.1.2. Input

The standard one hundred twenty word label is expected as the first block of a labelled input tape. The file name within the label is compared to the *file identifier* specified in the *file declaration*. If the file name is correct, the program continues. If the file name is incorrect or the label is not present, a console message is outputted and the tape is repositioned to load point. An operator response of G will permit the file to be read despite the file name discrepancy. A response of A indicates that a new tape has been mounted and the label check should be reinitiated.

11.1.4.2. Drum File Labels

All files declared as **DRUM** are written with an initial label block (one sector). Within blocked record files the first word of the label is a SDFF label control word which when read by the Executive System or by FORTRAN V causes the entire sector to be bypassed. In unblocked files the label block is a void block, i.e., zero significant words. The ALGOL I/O system will check purge date within the standard label prior to performing a write operation. If the purge date has not expired a console message permits the response, G, to continue processing. the only alternate response acceptable to the ALGOL I/O system is an A, instructing program termination.

Buffer and/or item sizes will be extracted from the standard labels and (1) will be compared for compatibility with the specifications of the *file declaration* or (2) will be utilized in file processing in the absence of *file declaration* specification.

11.2 DECLARATIONS

11.2.1. General

Each particular *I/O declaration* will be discussed separately in succeeding sections.

11.2.1.1. Syntax

<I/O declaration> : : = <file declaration> | <switch file declaration> | <format declaration> | <switch format declaration> | <list declaration> | <switch list declaration> | <namelist declaration> | <line declaration>

11.2.1.2. Semantics

I/O declarations define the files, formats and lists to be used in *I/O statements*.

11.2.2. File Declaration

11.2.2.1. Syntax

<file declaration> : : = <reel save part> <file lock part> <mode part> <density part> FILE <in-out part> <file identifier> <label equation part> <station part> (<buffer part> <save factor>)

<reel save part> : : = <empty> | **SAVE**

<file lock part> : : = <empty> | **SAVE**

<mode part> : : = <empty> | **ALPHA**

<density part> : : = <empty> | **HIGH** | **LOW**

<in-out part> : : = <empty> | **IN** | **OUT**

<file identifier> : : = <identifier>

<label equation part> : : = <output media part> <drum file description part> <label part>

<output media part> : : = 0|1|2|3|4|5|6|7|8|9| **DRUM** <drum access technique> | <empty>

<drum access technique> : : = **SERIAL** | **RANDOM** | **UPDATE** | <empty>

<drum file description part> : : = [<number of areas> : <size of areas>] | <empty>

<number of areas> : : = <arithmetic expression>

<size of areas> : : = <arithmetic expression>

<label part> : : = <file identification part> | <multi-file identification part> <file identification part> | <file identification prefix> <file identification part> | <empty>

<file identification part> : : = "12 or less string characters"

<multi-file identification part> : : = "12 or less string characters"

<file identification prefix> : : = "12 or less string characters"

<station part> : : = <empty> | <station identification> | <station part> <station identification>

<buffer part> : : = <number of buffers> , <buffer length> , <maximum record length> | <number of buffers> , <buffer length> | <number of buffers>, <record specifications> | <number of buffers>

<number of buffers> : : = <arithmetic expression>

<buffer length> : : = <arithmetic expression>

<maximum record length> : : = <arithmetic expression>

<record specifications> : : = <unblocked specification> | <blocking specifications> | <empty>

<unblocked specification> : : = <fixed physical record size>

<blocking specification> : : = <fixed logical record size>,<fixed physical record size> | <fixed physical record size>,<fixed logical record size>

<fixed physical record size> : : = <arithmetic expression>

<fixed logical record size> : : = <arithmetic expression>

<save factor> : : = **SAVE** <arithmetic expression> | <empty>

11.2.1.2. Semantics

(1) Reel Save Part

The *real save part* is applicable to magnetic tape files only. When **SAVE** is used the operator will be instructed to remove the reel when the block in which the file was declared is exited or when a *close statement* is encountered.

(2) File Lock Part

The *file lock part* is applicable to DRUM files only and relevant only when a file is initially created. When **SAVE** is used the file is catalogued (made permanent) when the block in which the file is declared is exited or when a *close statement* or *lock statement* specifying the file is encountered.

(3) Mode Part

The *mode part* is applicable only to magnetic tape files. When **ALPHA** is used BCD format is assumed.

(4) Density Part

The *density part* is applicable to magnetic tape files only. **HIGH** specifies 800ppi., **LOW**, 200ppi. In the absense of *density part*, high density, 800 ppi., is assumed.

(5) In/Out Part

IN or **OUT** must be specified for files having only one mode, i.e. card files may only be read, therefore **IN** is required; for printer files **OUT** is required. **IN** or **OUT** may be specified for drum or tape files to protect against erroneous efforts to write onto input or read from output. The absense of **IN** or **OUT** permits reading and/or writing of drum or tape files. The *in-out part* must be empty for **UPDATE** drum files and tapes to be written, rewound and read.

(6) File Identifier

For card files the identifier must be exactly that specified externally in the ASG and DATA control statements.

The *file identifier* must be specified in all *I/O statements* referencing the declared file. At the completion of any *read statement* the *file identifier* may be referenced as an *integer variable* to yield number of words read.

(7) Output Media Part

The types of the *output media part* identify the associated device.

TYPE	DEVICE
0	Card Punch
1	Printer
2	Labelled Tape
3	Card Input File
4	Printer
5	Labelled Designated Device
6	Printer
7	Unlabelled Designated Device
8	Paper Tape Punch
9	Unlabelled Tape
DRUM	FASTRAND or Simulated FASTRAND
empty	Labelled Tape

Types 1, 4, 6 are identical.

The value "3" is obviously an exception in that it alone specifies an input file. It is not mandatory, however, more efficient file assignment will result from the use of "3" in *file declarations* of card input files.

A "Designated Device" is accepted as a file without device association at compile time. At execution of the resulting object program the file must be either externally assigned or the type keyed-in via console in reply to the query "file identifier. DEVICE". Acceptable key-ins are any of the numeric types or **R** (random), **S** (serial), **U** (update) for drum files. Card files may not be keyed-in.

(8) Drum Access Technique

The accessing technique must be specified for files declared as **DRUM . SERIAL** indicates sequential access of records/blocks. Buffering of higher numbered records/blocks is performed on files read and buffering of records/blocks is performed for files written.

Files declared **RANDOM** will not be buffered for reads or writes, however a write of blocked records will be preceded by a read to protect adjacent records. A *record address* is always required for **RANDOM** reads or writes.

UPDATE processing is possible only with permanent files previously created by the **SERIAL** or **RANDOM** process. Reads and writes are buffered. Writing of blocked records will be preceded by a read if the block is not currently in the core buffer.

(9) Drum File Description Part

The *drum file description part* is relevant only when the file is initially created. It should not be used with files declared **IN.** Absence of the *drum file description part* implies reference to a catalogued file for files declared **OUT.**

For **SERIAL** and **RANDOM** files if the *drum file description part* is present and the file is declared **OUT** or the *in-out part* is empty a temporary file is created and made permanent only if **SAVE** is specified in the *file lock part* and/or a *lock statement* is encountered.

Drum file description part is never specified for files declared **UPDATE.**

The creation of a **RANDOM** file will fill the entire area specified preceding the current record with void (zero) records if not previously written. Subsequent write operations will overlay the void records. Subsequent read operations will return either the specified record if present or zeros.

The creation of a **SERIAL** file will assign the specified size as the reserve size. The unused area following a close of the file will be released to the Executive. However, if catalogued, the file may be extended to the system specified maximum. If temporary a close results in release of all areas assigned.

The *size of areas* times the *number of areas* yields the total number of words to be assigned on drum. Internal computation converts the number of words in terms of sectors (28 words), tracks (64 sectors), and positions (64 tracks). Track and position are acceptable granules to the Executive system. Assignment of drum (FASTRAND or Simulated FASTRAND) is made in terms of tracks or positions as reserve sizes of the file. Unused areas are available to the Executive for other assignment. However, extension of the file up to the system maximum is available to the user upon request. Blocks written to drum always begin at the first word of a sector and may end at any word within a sector.

(10) The *file identification part* will be utilized as file name in all internal library Executive references. In its absence, *file identifier* will be used as the file name. The Executive references are file assignments, cataloguing, releasing, reading, writing, etc.

(11) File Identification Prefix

The *file identification prefix* is applicable to files declared **DRUM** only. The *file identification prefix* will be utilized as the file qualifier. In its absence the Project ID specified in the RUN control image will be used by the Executive as the file qualifier.

The file qualifier is required in all references to catalogued files. If *file identification prefix* is absent in a declaration referencing a catalogued file the qualifier is automatically the Project ID.

(12) Multi-File Identification Part

The *multi-file identification part* is applicable to magnetic tape files only. When present it is used as the *file identification part* as described above. The *file identification part* is then used as the tape label name and written in or searched for in the labels of multi-file tape reels.

(13) Station Part

Station part will be utilized to cause print files to be output to the remote station initiating the RUN. Station numbers have yet to be determined.

(14) Buffer Part

Buffer sizes for card input or output files are 14 words. Printer files require 22 word buffers. Buffer specifications for card or printer files if other than the above will be ignored and the standard sizes assumed.

If a file is to be blocked, the logical and physical record sizes specified will be compared. The lesser value will always be assumed to be the logical record size without regard to the sequence in which they occur.

The block size written to drum or tape for unblocked items is three words greater than the specified buffer size. Block size of blocked items is equal to the specified buffer size, however, the number of items per block is equal to block size divided by item size plus the number of control words per item. For item size of twenty two words or less, the number of control words per item is one; for item size greater than twenty-two words an additional control word is required for each additional twenty-two words or portion thereof. *Blocking specifications* should be computed to compensate for control word requirements. Items or portions of items less than twenty-words are never spanned across blocks. Bypass records are used to fill any unused words at the end of a block.

(15) Save Factor

The *save factor* is applicable to labelled tape and drum files only. The value specified will be added to the current date and inserted in the file label. The file cannot be written again until the purge date in the label, if any, has expired.

11.2.2.3. Restrictions

(1) File names must be unique throughout an ALGOL program. Files may not be declared twice in the same block.

(2) Files declared **UPDATE** always reference catalogued files previously created and declared as **SERIAL** or **RANDOM.**

(3) Temporary files are released at the end of the block in which they are declared or if **CLOSE** is encountered. Subsequent references to a released file will result in transfer of control to EOF or EOR labels when specified or the next statement if EOF or EOR are absent.

A *file declaration* in a nested or disjoint block specifying the same name as a previously closed temporary file will cause assignment of a new temporary file of that name, however the content of the released file is unavailable.

(4) A second *file declaration* of a previously declared file, if in a nested block, will cause an automatic close of the previously declared file.

(5) Files may not be declared within a procedure.

11.2.2.4. Examples

```
1         10        20        30        40        50        60
FILE IN CARDS 3(1,,14);
FILE OUT PRINTER 1 (1,,22);
FILE IN TAPE (12,250);
FILE OUT TAPE (1,530,,50);
FILE TAPE 9 (1,,440,,10);
FILE OUT TAPE 'MULTIFILE' 'LABELNAME' (1,,252,,20 SAVE 1);
ALPHA FILE IN TAPE (1,,50);
SAVE FILE OUT TAPE 9 (1,,400);
FILE ONE DRUM SERIAL [10:500] (1,,500);
FILE OUT FILE1 DRUM SERIAL [A+B:SIZE] 'DRUM' 'FILE1' (1,,600);
SAVE FILE OUT TWO RANDOM [10:6000] 'RANDOM' 'FILE2' (1,,420,,100);
```

11.2.3. Switch File Declaration

11.2.3.1. Syntax

<switch file declaration> : : = **SWITCH FILE** <switch file identifier> : =
<switch file list>

<switch file identifier> : : = <identifier>

<switch file list> : : = <file identifier> | <switch file list>,<file identifier>

<file identifier> : : = <identifier>

11.2.3.2. Semantics

The *switch file declaration* associates a *switch file identifier* with a number of files as designated by the *file identifiers* in the *switch file list.*

Associated with each of the *file identifiers* in the *switch file list* is an integer reference. The references are 0,1,2, ... n−1, obtained by counting the identifiers from left to right. This integer indicates the position of the *file identifier* in the list. The *file identifiers* are referenced, according to position, by *switch file designators.*

If the *switch file designator* yields a value which is outside the range of the *switch file list,* the file so referenced is undefined. Each *file identifier* used in a *switch file list* must have appeared previously in a *file declaration,* and each file is governed according to the *file declaration* in which it was declared.

11.2.3.3. Examples

```
1        10        20        30        40        50
SWITCH FILE SWHTAPE:=TAPE1,TAPE2,TAPE3;
SWITCH FILE SWHUNIT:=CARDOUT,TAPEOUT,PRINT;
```

11.2.4. Format Declaration

11.2.4.1. Syntax

<format declaration> : : = **FORMAT** <input or output> <format edit part>

<input or output> : : = **IN**| **OUT** | <empty>

<format edit part> : : = <format identifier>(<editing specifications>)|
<format edit part>,<format identifier>(<editing specifications>)

<format identifier> : : = <identifier>

<editing specifications> : : = <editing segment> | <editing specifications>/|
/<editing specifications>|<editing specifications>/
<editing segment>

<editing segment> : : = <editing phrase>|<repeat part>(<editing specifications>)|
<editing segment>,<editing phrase>|<editing segment>,<repeat
part>(<editing specifications>)

<editing phrase> : : = <repeat part> <editing phrase type> <field part> | <string>

<repeat part> : : = <empty> | <unsigned integer>

<editing phrase type> : : = A|D|E|F|I|L|O|X|P|R|S

<field part> : : = <empty> | <field width>.<decimal places>

<field width> : : = <unsigned integer>

<decimal places> : : = <unsigned integer>

11.2.4.2. Semantics

The *format declaration* associates a set of *editing specifications* with a *format identifier*. The following discussion of *format declarations* is divided into two parts: those used for input and those used for output.

11.2.4.2.1. Input Editing Specifications

Input data can be introduced to the system by various media such as punched cards, magnetic tapes, or paper tapes. Once the information is in the system, however, it may be considered a string of 6-bit characters regardless of the input equipment used.

For editing purposes this string can be processed as a set of six-bit characters. The input *editing specifications*, through the *editing phrases*, designate where and in what form the initial values of variables are to be found in this string.

11.2.4.2.1.1. Input Editing Phrases

The *editing phrases* designate six-bit character processing. They describe a portion of the input data in which the initial value of one variable is to be found.

A phrase such as rAw has the same effect as Aw, Aw..., Aw(r times), where r is the *repeat part* and w the *field width*. The *field width* may specify from one to 63 characters. If the *repeat part* of an *editing phrase* is empty, it is given a value of 1.

Characteristics of the input *editing phrase types* are summarized in the following table:

Editing Phrase Type	Editing Phrase Example	Type of Variable Being Initialized	Example of Field Contents
A	**A**6	Any	TOTALS
D	**D**	None	Any Operand
E	**E**9.2	Real or Complex	+1.18 **E**-03
F	**F**7.1	Real or Complex	-3892.5
I	**I**6	Integer	-12345
L	**L**4	Boolean	**TRUE**
O	**O**	Any	777771244121
R	**R**11.2	Integer, Real or Complex	+2143567**E**+4
S	**S**+2	Real or Complex	None
P	**P**15.7	Double	+2.1234567**D**-212
X	**X**6	None	Any 6 Characters

Table 11–2. Characteristics of Types of Input Edit Phrases

The definition of each input *editing phrase type* is given below:

a. **A** – initializes a variable to the characters found in the field described by the *field width*. If the *field width* is greater than N (where N is either six for a single precision variable or twelve for a double precision or a complex variable), the left most N characters are taken as the value to be assigned to the variable. If the *field width* is less than N, spaces are filled to the right of the characters in the field to make a total of six characters. The type of the variable can be any, however, the data transferred from the data string is treated as alpha information and the variable is initialized with alpha characters.

b. **D** – causes six characters in the input data string to be ignored. The *field part* should be empty. The use of *editing phrase type* D is equivalent to the use of editing *phrase* X6 (See type X).

c. **E** – initializes a variable to the number found in the field described by the *field width*. The *field width* must be at least 7 greater than the number of *decimal places* specified since the input data is required to be of either of the following forms:

±n.dd---d@±ee

±n.dd---dE±ee

The sign of the number must appear first. A digit and a decimal point must follow the sign. One or more digits may follow the decimal point. The number of digits following the decimal point must equal the number of *decimal places* indicated by the *editing phrase*. Following the digits must be the symbol **E** or @, the sign of the exponent, and a two digit exponent. The sign of the number may be indicated by +, –, or a single space which is interpreted as positive.

d. **F** – initializes a variable to the number found in the field described by the *field width*. The input data must be found in one of the following forms:

±nn---n.dd–d

±.dd---d

The sign of the number is optional. If there is a sign, it must appear first; if there is no sign, the number is assumed to be positive. A decimal point must be present; zero or more digits may precede it. There must be as many digits after the decimal point as specified by the *decimal places* in the *editing phrase*. The number must be right-justified in the designated field.

e. **I** – initializes a variable to the integer found in the field described by the *field width*. The sign of the number is optional; the applicable rules are the same as in the case of editing phrase **F**. The number itself may consist of one or more digits which must be right-justified in the designated field.

f. **L** – initializes a variable to the logical value found in the field described in the *field width*. There are two possible values, **TRUE** and **FALSE**. The programmer may truncate these input words as follows:

T or F must appear in the field but need not be the leftmost character. The T or F can be preceded or followed by any number of spaces to fill the field. If T or F is not the leftmost character of the field, it must be preceded only by spaces. Whenever the T or F appears it can be followed by any characters in the field and need not be RUE or ALSE, respectively.

g. **O** – initializes a variable to the contents of N octal digits taken from the input string, where N is equal to 12 for a single precision variable or equal to 24 for a double precision variable or a complex variable. The *field part* is ignored and should be left empty. However, the *field width* is always set to 12 or 24 and 12 or 24 characters are taken from the input data string. If either 12 or 24 characters are not available in the input data string, invalid data will be transferred.

h. **P** – is used with double precision numbers. **P** initializes a variable to the number found in the field described by the *field width*. The *field width* must be at least 8 greater than the number of *decimal places* specified since the input data is required to be of the following form:

±n.ddd---d**D**±eee

The sign of the number must appear first. The sign may be indicated by +, –, or a single space which is interpreted as positive. A digit and a decimal point must follow the sign. One or more digits may follow the decimal point. The number of digits must be equal to the number of *decimal places* in the *editing phrase*. Following the digits must be the symbol **D**, the sign of the exponent, and a 3 digit exponent.

i. **R** – initializes a variable to the contents of an input field which may be written according to the specifications of the **I, F, E,** or **P** *editing phrase*. The **R** field has the following syntax:

<R field> : : = <signed R field> | <unsigned R field>

<signed R field> : : = <sign> <unsigned R field>

<sign> : : = + | –

<space>: : = <single space> | <space><single space>

<unsigned R field> : : = <I field>|<F field> | <E field> |<P field> | <space> | <space> <unsigned R field> | <unsigned R field> < space>

<I field> : : = <unsigned integer>

<F field> : : = <I field> . <unsigned integer> | . <unsigned integer> |
<unsigned integer> .

<E field> : : = <F field> <E part> | <E part> | <F field> <space> <E part>

<E part> : : = **E** <exponent> | @<exponent>

<P field> : : = <F field> <P part> | <P part> | <F field><space><P part>

<P part> : : = **D**<exponent>

<exponent> : : = <unsigned exponent> | <sign> <unsigned exponent>

<unsigned exponent> : : = <digit> | <digit> <digit>

The action of the **R** format (RW) is essentially a free field scan within the fixed field **W**. Four cases are possible:

(1) Entire field is blank. For this condition a minus zero is generated. This condition can be detected programmatically.

(2) Visible decimal point appears. For this case the field would contain an F field with or without an E part or P part. That which is allowed or restricted is as follows:

(a) If an E part or P part appears, it must be to the right of the F field.

(b) Leading blanks, trailing blanks and blanks between the F field and E part or P part are allowed but ignored.

(c) Imbedded blanks within F field or within the E part or the P part constitute an error condition.

(d) The *decimal places* (d) of the RW.d phrase has no meaning and is ignored.

(3) Implied Decimal Point

The field contains an I field with or without an E part or P part. For this case, d specifies the location of an implied decimal point between the dth and (d+1)th positions (counting from right to left). That which is allowed or restricted is as follows:

(a) Digits to the left of the implied decimal location are considered integer; to the right, fractional.

(b) If the I field is completely to the left of the assumed decimal point such that blanks appear to the right of the I field but left of the assumed decimal point location, an error condition is assumed.

Example:

Note: ♭ denotes blanks.

123♭♭♭. = error

(c) If the I field is completely to the right of the assumed decimal point location such that the blanks appear to the right of the assumed point and to the left of the I field, those blanks are considered to be zeros.

Example:

Note: ♭ denotes blanks.

.♭♭♭123=.000123

(d) Blanks found embedded within an I field constitute an error condition.

Example:

Note: ♭ denotes blanks.

123.♭♭45 =error

(e) Blanks to the left of the I field and to the right of the I field and blanks between the I field and the E part or P part (if any) are allowed and ignored.

.123♭♭E −2=.123E-2

(f) An error condition is assumed if the implied decimal point falls within the E part or P part.

(4) The field contains an E part or P part only. An E part or P part with or without a leading sign can be used. For this case, a ±1.0@ee, or a ±1.0Deee is generated. Leading and trailing blanks are allowed but ignored. The *decimal places* (d) of the RW.d phrase is meaningless and is ignored.

(5) Miscellaneous notes which apply to all are as follows:

(a) Overpunching of numeric data is not allowed.

(b) In any of the above cases if the data are not right justified and other than trailing blanks are found, an error condition is assumed.

(c) An error condition is assumed if anything other than an I field, F field, E field, P field, or blank field is found.

(d) A single digit exponent is allowed.

j. **S** – provides the means of scaling data. The **S** *editing phrase* is applied to **R** phrases only. The **S** phrase has the following form:

S<exponent>

When an **S** phrase is encountered in a format, all subsequent values associated with an **R** format phrase are multiplied by the designated power of 10, the exponent. More than one **S** phrase may appear in a format, each taking precedence over the one before.

Example of **S** editing:

S–2
S+4
S3
S+20

k. **X** – causes the number of characters indicated by the *field width* to be ignored. The *repeat part* of the **X** *editing phrase* has no meaning. It is ignored and should be left empty.

l. Strings – if the input *editing phrase* is a *string*, the string in the *format declaration* is replaced by the corresponding input string. The number of characters transferred from the input string is equal to the number of characters in the *format declaration* which are enclosed between the *string bracket characters*. For *strings* used in *format declarations*, a maximum of 132 characters is allowed.

11.2.4.2.1.2. Error Conditions

When an error condition is encountered during input editing processing, the following action takes place:

1. If no *parity action label* is specified in the *read statement*, the program will be terminated.

2. If a *parity action label* is specified in the *read statement*, control is transferred to this label. The buffer remains unchanged, that is, the erroneous data remains in the buffer and will be accessed by the next reference to the file.

11.2.4.2.2. Output Editing Specifications

Output can be performed by the system through various media such as magnetic tape, line printer, and drum. The information in the system ready for output (in the buffer area) but not yet transferred to the output equipment may be considered as a string of six-bit characters, regardless of the output media to be used.

The output *editing specifications*, by means of the *editing phrases*, designate when and in what forms the values of expressions are to be placed in the output data string.

11.2.4.2.2.1. Output Editing Phrases

The *editing phrases* describe a portion of the output data string into which output information is to be placed. This information may be one of three kinds:

a. The value of an expression.

b. The characters of the *editing phrase* itself (where the *editing phrase* is a string).

c. The insertion characters 0 (zero) and spaces.

A phrase such as rAw has the same effect as Aw, Aw, ... , Aw (r times), where r is the *repeat part* and w is the *field width*. The *field width* in an *editing phrase* may specify a length of one to 63 characters. If the *repeat part* of an *editing phrase* is empty, it is given a value of one.

Characteristics of the output *editing phrase types* are summarized in Table 11–3.

Editing Phrase Type	Editing Phrase Example	Type of Evaluated Expression	Example of Field Contents
A	A6	Any	RESULT
D	D	None	6 zeros (000000)
E	E11.4	Real, Complex	−1.2500E-02
P	P13.5	Double	−1.25012D+124
F	F8.3	Real, Complex	6735.125
I	I6	Integer	ßß1416
L	L5	Boolean	ßTRUE
O	O	Any	777721412712
R	R11.4	Real, Complex, Double	ß2.1231E+09
S	S−2	Real, Complex, Double	None in field; result: 10**(−2)*R(subsequent)
X	X8	None	8 blanks

Table 11-3. Characteristics of Types of Output Editing Phrases

The definition of each output *editing phrase* is given below:

a. **A** – place the value of one expression (N characters, where N is equal to six for a single precision expression and twelve for a double precision or a complex expression) in the field described by the *field width*. The counting of the field begins at the high order character, i.e. the leftmost character. If the *field width* is greater than N, N characters are placed left justified in the field, the remaining field is filled with spaces. If the *field width* is less than N, the leftmost characters of the expression value are placed in the field. The *expression* can be of any type; however, the expression is treated as type **ALPHA** when an **A** phrase is used in the *editing specification*. If an **A** type phrase is used for an *expression* which has type other than **ALPHA**, alpha information must be set for the value of the expression before output editing has taken place.

b. **D** – places six zeros in the output data string. No expression is associated with the **D** phrase. The *field part* should be empty.

c. **E** – places the value of one expression in the field described by the *field width*. This value has the following form when placed in the output data string: ßn.dd...dE±ee

The sign of the number is represented by a single space if positive, and a minus sign if negative. If the *field width* minus seven is greater than the number of *decimal places* specified, leading spaces are used to complete the field, then the sign of the number, the first significant digit, and a decimal point are inserted. The value of the expression is rounded to the number of *decimal places* specified by the *editing phrase*. If the number of significant digits in the expression value is less than the number of *decimal places* specified, the digits are left-justified with trailing zeros. To complete the field, the symbol **E**, the sign of the exponent, and the appropriate two-digit exponent are inserted. The sign of the exponent is indicated by either + or −.

d. **F** − places the value of one expression in the field described by the *field width*. This value has the following form when placed in the output string:

␣nn−−−n.dd−−d

The sign of the number is represented by a single space if positive, and a minus sign (−) if negative. The value of the expression is rounded to the number of *decimal places* specified by the *editing phrase*. If the number of significant digits thus obtained is less than *field width* minus two, leading spaces are used to complete the field. If the number of significant digits is more than *field width* minus two, the entire field will be filled with asterisks (*).

e. **I** − places the value of one expression in the field described by *field width*. The expression is rounded to an integer and placed right-justified in the field, preceded by leading spaces, if any are required. If the number of significant digits is greater than the *field width* minus one, the entire field will be filled with asterisks (*). The sign of the number is the same as for the E *editing phrase type*.

f. **L** −places the value of one Boolean expression in the field designated by *field width*. Table 11−3 shows the effect of various values of *field width*.

Field Width	BOOLEAN VALUE	
	TRUE	FALSE
L1	T	F
L2	TR	FA
L3	TRU	FAL
L4	TRUE	FALS
L5	TRUE␣	FALSE
Ln, where n>5	Skip n-5 then same as L5	

Table 11-4. Boolean Values for Various Field Widths in Output Editing Phrase

g. **O** – place the value of one expression in the output data string. The number in the field is an octal digit representative. The *field part* is ignored and should be left empty. However, the *field width* is always set to twelve or twenty-four depending upon whether it is a single precision value or double precision value or complex value of the expression.

h. **P** – place the value of one expression in the field described by the *field width*. **P** is used for double precision numbers. This value has the following form when placed in the output data string:

$$\not{b}\, n.dd---d\mathbf{D}\pm eee$$

The sign of the number is represented by a single space if positive and a minus sign (-) if negative. If the *field width* minus eight is greater than the number of *decimal places* specified, leading spaces are used to complete the field. Then the sign of the number, the first signficant digit, and a decimal point are inserted. The value of the expression is rounded to the number of *decimal places* specified by the *editing phrase*. If the number of significant digits in the expression value is less than the number of *decimal places*, the digits are left-justified with trailing zeros. To complete the field, the symbol **D**, the sign of the exponent, and the appropriate 3 digit exponent are inserted. The sign of the exponent is indicated by either + or –.

i. **X** – places a number of single spaces, as indicated by the *field width*, in the output string.

j. **R** – places the value of one expression in the field described by the *field width*. The output will be either an **F**-type, an **E**-type or a **P**-type field, depending upon the magnitude of the expression and the type of the expression. Assuming that:

E = exponent number,
sign = 0 for +, 1 for –,
W = *field width*,
d = number of *decimal places* to the right of decimal point, and
I = number of decimal digits to the left of decimal point, then:

(1) The output will be in **F**- format

(a) if the absolute value of the number is equal to or greater than 1 but less than the maximum allowable integer, and

$$w \geq I+d+1+sign.$$

(b) or if the absolute value of the number is less than 1, and either

$$ABS(E) \leq d$$

or

$$w < d+6+sign$$

or

$$w < d+7+sign$$

(2) The output will be in **E**-format or **P**-format if the conditions for **F**-format are not met, and

for **E**-format $w \geq d+6+sign$

or

for **P**-format $w \geq d+7+sign$

(3) If none of the above conditions are met, the field will be filled with asterisks.

k. **S** – the values associated with the subsequent **R** format phrases will be multiplied by such powers of 10 as designated by the integer in the **S** format phrase itself. More than one **S** phrase may appear in a format, each taking precedence over the one before.

l. String – an output editing phrase may itself be a *string*. This *editing phrase* is defined as placing itself, except for the delimiting *string bracket characters*, in the output string. However, an apostrophe can be placed in the output data string by placing two consecutive apostrophes in the string output *editing phrase* where a single apostrophe is desired in the output area. For a *string* used in *format declarations*, a maximum of 132 characters is allowed.

11.2.4.2.3. The Meaning of Symbol /

The symbol / (slash) in the *editing specifications* indicates a termination of a record. The rightmost parenthesis of the *editing specifications* perform the function of one slash. For input *editing specifications*, n consecutive slashes cause n-1 records to be skipped (spaced) from the input file. For output *editing specifications*, n consecutive slashes cause n-1 blank records (records filled with spaces) to be written out to the designated output file.

11.2.4.2.4. Editing Specification for Complex Values

A complex value is represented by an ordered pair of real numbers. Therefore, all *editing phrases* used for real numbers can be used for complex numbers except that a pair of *editing phrases* is required for each complex number. In order to describe the format of a complex number, two *editing phrases*, one for each portion of the complex number, must be used.

11.2.4.2.5. Restrictions

1. Input *editing specifications* cannot be used as output *editing specifications*; the reverse is also true.

2. An input *editing phrase* must not be a string.

11.2.4.2.6. Examples

```
1        LABEL          10        OPERATION          20        OPERAND          30          40          COMMENTS          50          60
         FORMAT IN EDIT (X4,2I6,5E9.2,3F5.1,X4);
         FORMAT IN F1 ((A6,5(X3,2E10.2,2F6.1)),3I7),F2((A6,D,A6));
         FORMAT OUT F1O R M1 ((X55,'HEADING'),X57),F QRM2 ((X10,4A6/X7,5A6/X2,
                5A6));
         FORMAT OUT F3 ((10,23,O)†);
         FORMAT OUT F4((F5.2,X2,R3.1,S-21));
         FORMAT FM1 ((S+2,3R1,2.3,S-2,4R10.4));
         FORMAT OUT FM ((A6,X7,A8/////));
```

† *The last character before the right parenthesis is the letter O not zero.*

11.2.5. Switch Format Declaration

11.2.5.1. Syntax

<switch format declaration> : : = **SWITCH FORMAT** <switch format identifier> := <switch format list>

<switch format identifier> : : = <identifier>

<switch format list> : : = <editing specification part> | <switch format list>, <editing specification part> | <format identifier> | <switch format list>,<format identifier> | <format identifier>,<switch format list>

<editing specification part> : : = (<editing specifications>)

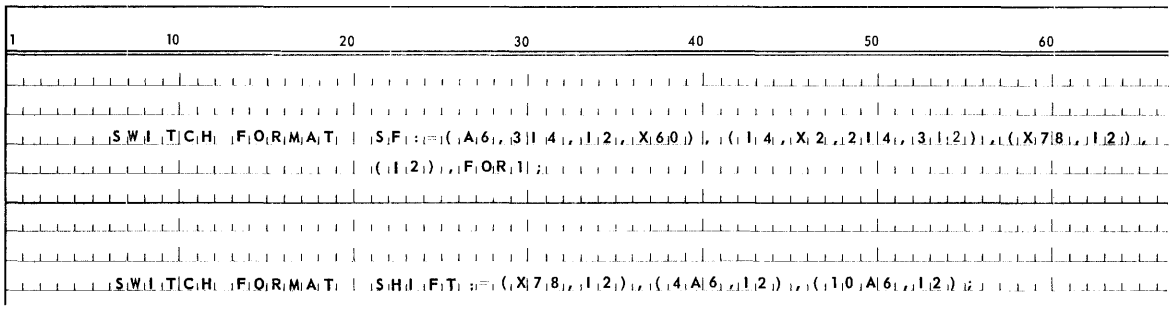
11.2.5.2. Semantics

The *switch format declaration* associates a *switch format identifier* with the *editing specification part* or *format identifier* in the *switch format list.* Associated with each *editing specification part* or *format identifier* is an integer reference. The references are 0, 1, 2 ..., obtained by counting the editing elements of the *switch format list* from left to right. The integer reference indicates the position of the *editing specification part* or *format identifier* in the list. The *editing specification parts* and *format identifiers* are referenced according to position, by *switch format designators.*

If a *switch format designator* yields a value which is outside the range of the *switch format list,* the format so referenced is undefined. If a *format identifier* is used in a *switch format list,* it must previously be defined in a *format declaration.*

Editing specifications are identical to the *editing specifications* of the *format declaration* (11.2.4.).

11.2.5.3. Examples



11.2.6. List Declaration

11.2.6.1. Syntax

<list declaration> : : = **LIST** <list specification>

<list specification> : : = <list identifier> (<list>) | <list specification>,
<list identifier> (<list>)

<list identifier> : : = <identifier>

<list> : : = <list segment> | <list>,<list segment>

<list segment> : : = <expression part> | <for clause> <list segment> |
<for clause> [<expression list>]

<expression part> : : = <arithmetic expression> | <Boolean expression>

<expression list> : : = <list segment> | <expression list> , <list segment>

11.2.6.2. Semantics

A *list declaration* associates a set of expressions (arithmetic or Boolean) with a *list identifier*. The *list identifier* may be used in a *read statement* for the variables to be initialized and the order in which the initializing is to be done. The *list identifier* may be used in a *write statement* for specifying values to be included in an output operation. These values are placed in the output string in the order of their appearance in the *list declaration*. Variables in a *list declaration* may be either local or non-local to the block in which the *list declaration* appears.

Restrictions:

1. Since any expression other than a variable is meaningless in an input operation, a *list identifier* used in a *read statement* must refer to a *list declaration* which includes variables only.

2. When used for input, the variables in a *list declaration* must have been declared as type **REAL, INTEGER, ALPHA, BOOLEAN, DOUBLE,** or **COMPLEX.**

11.2.6.3. Examples

```
LABEL          OPERATION          OPERAND                    COMMENTS
1         10        20        30        40        50        60
          LIST  L1 ((X, Y, A[J]), FOR I := P STEP 1 UNTIL 5 DO B[I]);
          LIST  ANSWERS ((P, Q, Z, SQRT(R)), RESULTS (X1, X2, X3, X4/2));
          LIST  LIST3 ((FOR I := 0 STEP 1 UNTIL 10 DO FOR J := 0 STEP 1 UNTIL
                15 DO A[I, J]));
          LIST  L4 ((B AND D, C, NOT A[B1], IF X=0 THEN R1 ELSE R2));
          LIST  RESULTS ((FOR I := 1 STEP 1 UNTIL N DO A[I], FOR J := 1 STEP
                1 UNTIL K DO B[I, J], C[J]));
```

11.2.7. Switch List Declaration

11.2.7.1. Syntax

<switch list declaration> : : = **SWITCH LIST** <switch list identifier> : =
<switch list list>

<switch list identifier> : : = <identifier>

<switch list list> : : = <list identifier> | <switch list list>,<list identifier>

11.2.7.2. Semantics

A *switch list declaration* associates a *switch list identifier* with a number of *list identifiers*. Associated with each of the *list identifiers* is an integer reference which is obtained by counting the *list identifiers* from left to right starting with zero. This integer indicates the position of the *list identifier* in the *switch list list*. These *list identifiers* are referenced by means of *switch list designators*.

If a *switch list designator* yields a value which is outside the range of the *switch list list*, the *list* so referenced is undefined. Each *list identifier* used in a *switch list list* must have appeared previously in a *list declaration*.

11.2.7.3. Example

```
1         10        20        30        40
          SWITCH LIST LX1 := L1, L2, L3;
```

11.2.8. Namelist Declaration

11.2.8.1. Syntax

<namelist declaration> : : = **NAMELIST** <namelist list part>

<namelist list part> : : = <namelist identifier> (<namelist parameter part>)|
<namelist list part> , <namelist identifier> (<namelist parameter part>)

<namelist identifier>: : = <identifier>

<namelist parameter part> : : = <namelist element> | <namelist parameter part>, <namelist element>

<namelist element> : : = <simple variable> | <array identifier> | <label identifier>

11.2.8.2. Semantics

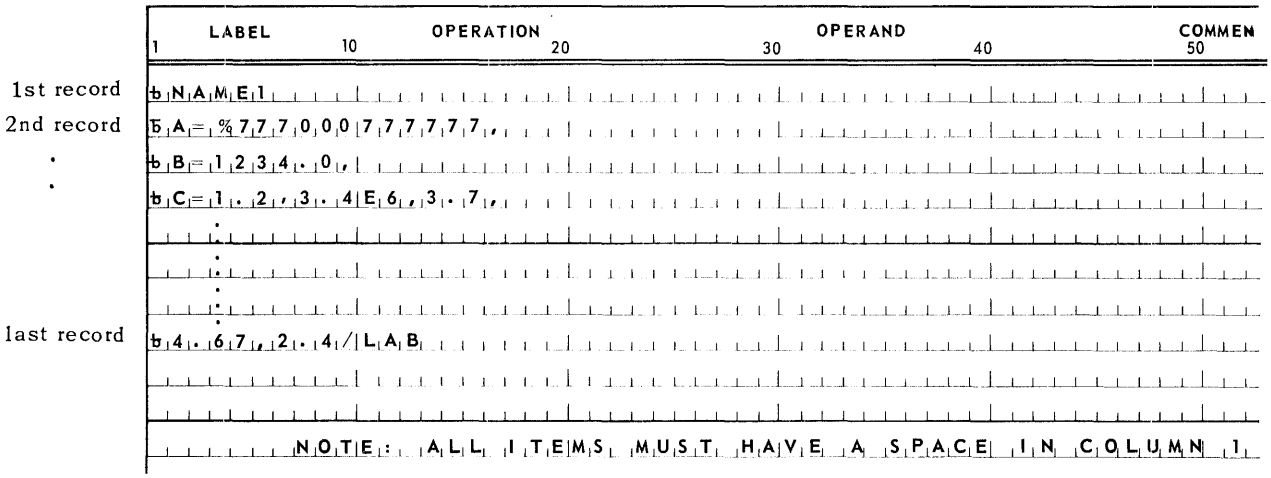
A *namelist declaration* associates a list of *namelist elements* with a *namelist identifier.* The *namelist identifier* may be used in an automatic data *read statement* for specifying that the variables to be initialized and the value to be read in are provided in the input file itself.

The formats of input data records are specified under *read statements* (Section 11.3.2). A specific requirement of input data records introduced into the system via the DATA statement for use with the *namelist declaration* is that a space must appear in column one of every item.

Assuming that NAME1 is defined in a *namelist declaration* and used in a *read statement* and A,B, C, LAB, and FILE1 are declared as follows:

```
1         10        20        30        40
          REAL  A,B;
          ARRAY C[1:5];
          LABEL LAB;
          FILE IN FILE1 3(1,14);
          NAMELIST NAME1(A,B,C,LAB);
            .
            .
          READ (FILE1,NAME1);
```

Possible input data records of FILE1 may be as follows:

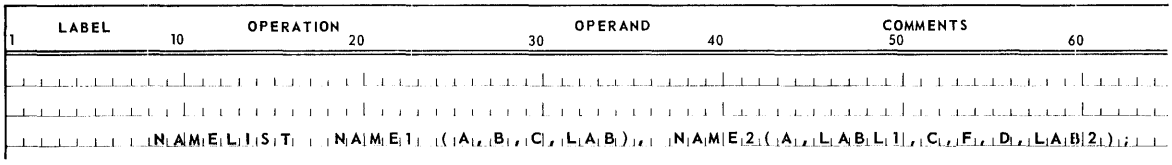


The above example would cause records from FILE1 to be read into variables A, B and array C. Upon executing the last record, indicated by the slash(/), program control will transfer to the indicated label, LAB.

11.2.8.3. Restrictions

- The first input data record must be the *namelist identifier* which is referenced by the *read statement*.
- All *variable identifiers* and *label identifiers* used in the input data records must have been defined previously in a *namelist declaration*.
- If a *label identifier* is used in the input data record, it must appear in the last record and immediately follow the symbol / (slash). A single space following the symbol / indicates that there is no *label* following.

11.2.8.4. Example



11.2.9. Line Declaration

11.2.9.1. Syntax

<line declaration>::= **LINE** <file identifier>(<paper size>, <channel specification>)<heading control>

<paper size>::=<unsigned integer>

<channel specification>::=<channel number>:<line number>|<channel specification>,<channel number>:<line number>

<channel number>::=<unsigned integer>

<line number>::=<unsigned integer>

<heading control>::=[**NO**]|[**NO**,<heading>]|[<heading>]|<empty>

<heading>::=<string>

11.2.9.2. Semantics

The *line declaration* is used in conjunction with *file declarations*. The *file identifier* so referenced in the *line declaration* must be a print file. The UNIVAC 1108 high speed printer does not use the paper loop mechanism to control carriage return and spacing of a line printer. Form control and spacing is completely variable and controlled entirely by program. In UNIVAC 1108 Extended ALGOL, the *line declaration* is used to describe print files. The form, number of lines per page, and the *channel number* associated with a specific *line number* are specified by the *line declaration*.

11.2.9.2.1 Paper Size

The length of the printer form is expressed as *paper size*. *Paper size* specifies number of lines per page. Standard *paper size* is 66 lines per page.

11.2.9.2.2 Channel Number

Channel numbers referenced the channels on printer control tapes. *Channel numbers* are used in conjunction with *line numbers* to give an associated reference number and line number which is to be spaced to when utilized in a *write statement*. All *channel numbers* must be defined in the *line declaration* to describe the *channel number* and its associated *line number* prior to being referenced by a *write statement*. A maximum of eleven *channel numbers* with an associated *line number* for each may be specified.

For example, channel 1 is the 'home paper' channel. If skip to channel 1 is used in a *write statement*, it causes a 'page eject' to the line specified in its associated *line number* specification.

11.2.9.2.3 Heading Control

Heading control indicates whether or not a heading is to be printed on each new page.

(1) If *heading control* is empty, it causes the current date and page number to be printed on each page by EXEC 8.

(2) If **NO** is used, it will not print the EXEC 8 date and page number on each page. Therefore, if a line declaration with a **NO** specified does not appear the EXEC 8 date and page number will always be printed on every page.

(3) If only *heading* is used, the *string* in the *line declaration* will be printed with the EXEC 8 date and page number appearing on each page.

(4) If **NO** and *heading* are used, only the *string* in the *line declaration* will be printed out and no EXEC 8 date or page number will be printed.

11.2.9.3. Restriction

1) The *file identifier* used in a *line declaration* must have appeared previously in a prevailing *file declaration.*

2) In the absence of a *line declaration,* standard UNIVAC 1108 print page definition will be assumed. Standard UNIVAC 1108 printer page definition is 66 lines per page, with a top margin setting of 6 lines and a bottom margin setting of 3 lines, thus giving 57 printable lines.

3) Each *channel number* declared in a *line declaration* must be associated with only one *line number.*

4) *Channel number* can only be 1 through 11.

11.2.9.4. Example

```
1         10        20        30        40
LINE F1 (66,1:6,6:11,10:60);
```

The above example declares that the form is 66 lines per page; and it will be positioned to line 6 before printing begins. Each skip to channel 1 in a write statement causes the paper to be ejected and repositioned to line 6 of the new page. Channel 6 will be associated with line 11, and channel 10 with line 60. Since no *heading control* is specified, the EXEC 8 date and page number will be printed on each page.

11.3. STATEMENTS

11.3.1. General

Each particular *I/O statement* will be discussed separately in succeeding sections.

11.3.1.1. Syntax

<I/O statement>::=<read statement>|<write statement>|<space statement>|
<rewind statement>|<close statement>|<lock statement>

11.3.1.2. Semantics

I/O statements cause values to be communicated to and from a program and provide programmatic control of files and I/O units.

11.3.2. Read Statement

11.3.2.1. Syntax

<read statement>::=**READ** <direction> (<input parameters>) <action label>

<direction>::=<empty>|**REVERSE**

<input parameters>::=<file part> <buffer release>|<file part>
<buffer release>, <format and list part>|
<file part> <buffer release>, <free field part>|
<array row>, <format and list part>|<array row>,
<free field part>

<file part>::=<file identifier>|<switch file designator>

<buffer release>::=<empty>|[**NO**]|<record address and release part>

<record address and release part>::=[<address>]|[**NO**]|<empty>

<address>::=<arithmetic expression>

<format and list part>::=<format part>|<format part>,<list>|
<format part>, <list part>|*, <list>|
*,<list part>|<alpha array identifier>,
<list>|<alpha array identifier>,<list
part>|<arithmetic expression>,<array row>|
<namelist identifier>

<free field part>::=/,<list>|/,<list identifier>

<format part>::=<format identifier>|<switch format designator>

<list part>::=<list identifier>|<switch list designator>

<action label>::=[<end of file label>:<parity label>]|[<end of file
label>]|[:<parity label>]|<empty>

<end of file label>::=<label identifier>

<parity label>::<label identifier>

11.3.2.2. Semantics

The *read statement* causes values to be assigned to program variables and/or places information in strings defined in the *format declaration.*

- *Direction*

  REVERSE is used to specify the reading of magnetic tape or drums in the reverse direction. Otherwise this field is empty.

  For drum files the use of **REVERSE** causes adjustment so that the value of the record pointer is decreased by one from its designated setting before the read is performed. If the value of the record pointer is N when a read reverse is executed, the record pointer is set to N-1 before the read is performed. At the completion of the read reverse, the record pointer remains at N-1.

- *File part*

  This field specifies which file is to be read. If *array row* is used instead of the *file part,* it indicates an Edit and Move *read statement.*

- *Buffer release*

  The *buffer release* indicates whether the input buffer is to be refilled after it has been read and edited. If [**NO**] is used the buffer will not be filled and the current buffer is the next one to be accessed.

- *Record address and release part*

  This field applies only to drum files. The *address* specifies the relative address of the record in the file to be read and is edited as specified in the *read statement.* The record pointer is set to the *address* before the read is executed. The record pointer will not be adjusted after the read is executed. An *address must be used* with files declared **RANDOM.** If [**NO**] is not used and *address* is not specified, the record read will be the current one pointed at by the record pointer. After the read, the record pointer will be adjusted to point to the next record. If [**NO**] is used, the record read will be the current one pointed to by the record pointer. After the read, the record pointer will not be adjusted, i.e., the record pointer will be the same one as before the read was executed.

- *Format and list part*

  The *format and list part* specifies the action to be taken on input data. If no *format and list part* is given, one logical record will be passed without being read. Such a statement acts as a *space statement* which only spaces one record.

A *format identifier* alone indicates that the referenced *format declaration* contains a *string* into which corresponding characters of the input data are to be placed, i.e., replace the *string* in the *format declaration* with the *string* in the data. The referenced *format declaration* must only contain one *string*.

A *format indentifier* together with a *list* or *list identifier* designates that the input data is to be edited according to the specifications of the referenced *format declaration* and assigned to the variables of the referenced *list*.

The asterisk, *, together with a *list* or *list identifier* specifies that the input data is to be processed at word length and that it is to be assigned to the variables of the referenced *list* without being edited. The number of words read is determined by the number of variables in the *list* or the buffer size, whichever is smaller.

The *arithmetic expression* with an *array row* specifies that input data is to be processed at word length and that it is to be assigned to the elements of the designated *array row* without being edited. The number of words read is determined by the number of elements in the *array row*, the buffer size, or the value of the *arithmetic expression*, whichever is smallest.

The *alpha array identifier* together with a list or *list identifier* specifies that the input data is to be edited according to the calculated *format specification* stored at the referenced alpha array at execution time and is assigned to the variables of the referenced *list*. The calculated format must be placed in a one dimensional alpha array prior to execution of the *read statement*.

The *namelist identifier* indicates an automatic data read statement. The *list* information is provided by the referenced *namelist declaration* and also is provided in the input file itself. The format information is provided in the input file. Input data records are defined as follows:

(1) The first character in each data record is always ignored. The first record of a group of data records to be read must be the *namelist identifier* which is declared in the referenced *namelist declaration*. This identifier is followed by the data items.

(2) The data items must have the following form:

variable element = DATA WORD

They are defined as follows:

<namelist element> : : = <simple variable> | <array identifier> | <label>
<DATA WORD> : : = <value list>

<value list> : : = <initial value>, | <value list>, <initial value>,

<initial value> : : = <number> | <string> | % <octal number>

(3) Any selected set of the *variables* or array *identifiers* belonging to the *namelist parameter part* of the *namelist identifier* which appear on the first record may be used in the manner specified by the above data item.

(4) The end of a group of data records is signaled by a slash (/) instead of a comma. A label appearing immediately following a slash, i.e., /LAB1, will cause a program transfer to the specified label, LAB1, which must be specified in the *namelist declaration.* A slash followed by a space indicates that no label is specified. The program will transfer to the statement immediately following the *read statement* after the read is executed. See *namelist declaration,* Section 11.2.8.2, for examples.

- *Free field part*

  When the *free field part* is used, no *format declaration* is required to provide the *editing specifications* for data. *Editing specifications* are determined by the format of the data. Data must be formatted as described under FREE FIELD INPUT, Section 11.3.3.

- *Action labels*

  The *action labels* provide a means of transferring control from a *read statement* (or *space statement*) when an end-of-file condition or parity condition occurs.

  An end-of-file condition occurs for drum files whenever an attempt is made to read a record whose address us greater than that of the EOF indicator. The EOF indicator is always set to the address of the highest addressed record written in the file.

- *Edit and Move*

  An *array row* may be used in a *read statement* instead of a *file part.* It provides the means of utilizing the editing features of a read without using I/O files and buffer areas. In effect, the *array row* designated in the *read statement* is analogous to a buffer area.

- When a *read statement* is executed without specifying an input file, data in the designated *array row* is edited and placed in the *list.* The *format part* determines what editing is to take place as the data is moved from the *array row* to the *list.*

11.3.2.3. Restrictions

- When a parity error occurs in reading from tape the redundant record is left in the buffer.

- Tape files must be closed before changing directions when the tape is positioned following an end-of-file.

- Labels and variables used in an input file for a *namelist* data read statement must be declared by a *namelist declaration* in the block in which the *read statement* appears or be global to it.

11.3.2.4. Examples

```
LABEL     OPERATION     OPERAND                                   COMMENTS
1         10            20                30                      40        50
          READ (FILEID, FMT, LISTID) [LEOF];
          READ (FILEID [NO], FMT, LISTID) [EOF: MODE];
          READ REVERSE (FILEID, FMT, A, B, C, ARA[1]) [:LPAR];
          READ (FILEID, *, LISTID) [EOF:PAR];
          READ (FILEID, X+Y, ARA[*]) [LEOF:LPAR];
          READ (FILEID, FMT);
          READ REVERSE (FILEID, 50, ARA2[1,*]) [:LPAR];
          READ (FILEID);
          READ (FILEID, /, FOR I:=0 STEP 1 UNTIL 16 DO A[I]);
          READ REVERSE (OLDFILE, FRMAT, LST);
          READ (FREEFILE, 1, FREELIST) [:PAR];
          READ (NEWFILE[NO], *, BILST) [EOF:PAR];
          READ (FILEID, NAMELISTID);
          READ (FILEID [IF X>N THEN 0 ELSE 1], 50, AES[*]);
          READ (A[*], FMT, LST);
          READ (DD[*], /, R, A);
          READ (DATA[NEXT], NOREC, ARA[I,*]);
```

11.3.3. Free-Field Input

11.3.3.1. Syntax

<free field sentence> : : = <field> <field delimiter> | <free field sentence> <field> <field delimiter>
<field> : : = <number> | <string> | % <octal number> | / | <empty>

<field delimiter> : : = , | <letter> <any proper string not containing a comma>,

11.3.3.2. Semantics

All free-field input is in the form of *free-field sentences*. Each field in a sentence is associated with the list element to which it corresponds according to position. A *free-field sentence* is not affected by the end of a record. A *field*, or *field delimiter*, may be carried over from one record to another. Continuation from record to record is automatic, until the *list* is exhausted; unused characters (if any) on the last record read are lost. All blanks in free-field sentences, except those in strings, are completely ignored. Fields are handled as follows:

- *Numbers*

  Numbers which either are represented in integer form or contain a decimal fraction and/or *exponent part* are converted to **integer, real, double** or **complex** according to the type of the variables in the list, respectively. Two real numbers are required for each complex variable.

- *String*

  *String*s may be any length. Each list element receives six characters until either the **list** or the *string* is exhausted. The string characters are stored left-justified with space fill on the right in the list elements. *String*s are enclosed in the *string bracket character*, i.e. '. In order to read an apostrophe or a quote within a *string*, two *string bracket character*s must appear in succession to represent one in the input record.

- *Empty*

  An empty field causes the corresponding list element to be ignored.

- *Slash* (/)

  The slash (/) field causes the remainder of the current record to be ignored. The record following the slash is considered the beginning of a new field; therefore the slash field does not require a field delimiter. (The slash field is unique in this regard.) A slash field has no effect on list elements.

- *Logical Values*

  For *free-field input*, the integer value 1 (one) and 0 (zero) represent the logical values **TRUE** and **FALSE** respectively.

11.3.3.3. Restriction

Free-field input may not be used with blocked tape records.

11.3.3.4. Examples

Consider each of the following lines as individual records,

```
1
2
3
@
29
,
+ 1 23 . 0 @ +
29
,
+ .123@29
, 0, X, A1 , 4 A 5 B, / CARD 124
15 IGNORED, ZERO,
% 177, % 30, 'THIS IS A STRING', 'DON''T',''
STRING' , , , ,
```

If the above records *(free field sentence)* were read with the statement

```
1         10        20        30        40        50
READ (FILEID, /, FOR I := 0 STEP 1 UNTIL 16 DO A[I]);
```

where A was declared as an array of type real, values would be assigned to A as follows:

A [0] = 123@29	A [6] = 4.0
A [1] = 123@29	A [7] = 15.0
A [2] = .123@29	A [8] = Unchanged
A [3] = 0	A [9] = 177 (octal)
A [4] = Unchanged	A [10] = 30 (octal)
A [5] = Unchanged	A [11] = THIS␣I

A [12] = S␣A␣ST

A [13] = RING␣␣

A [14] = DON'T␣

A [15] = STRING

A [16] = Unchanged

11.3.4. Write Statement

11.3.4.1. Syntax

<write statement> : : = **WRITE**(<output parameters>)<waction label>

<output parameters> : : = <file part> <carriage control> | <file part> <carriage control>, <wformat and list part> | <file part> | <array row>, <wformat and list part>

<file part> : : = <file identifier> | <switch file designator>

<carriage control> : : = [**PAGE**] | <skip to channel> | [**DBL**] | [**NO**] | <record address part> | <empty>

<record address part> : : = [<address>] | <empty>

<address> : : = <arithmetic expression>

<skip to channel> : : = [<arithmetic expression>]

<waction label> : : = [<end of reel label> : <parity label>] | [:<parity label>] | [end of reel label>] | <empty>

<parity label> : : = <label identifier>

<end of reel label> : : = <label identifier>

<wformat and list part> : : = <format part> | <format part>, <list> | <format part>, <list part> | *, <list> | *, <list part> | <arithmetic expression>, <array row> | <alpha array identifier>, <list> | <alpha array identifier>, <list part>

<format part> : : = <format identifier> | <switch format designator>

<list part> : : = <list identifier> | <switch list designator>

11.3.4.2. Semantics

- *File part*

  The *file part* specifies the file to be used. If an *array row* is used instead of the *file part*, it indicates that it is an Edit and Move *write statement*.

- *Carriage control*

The *carriage control* allows for paper control on the line printer. *Carriage control* is irrelevant and ignored on all other units except the *record address part* which applies only to drum files. **PAGE** causes the printer to skip to the next page after the line is printed. *Skip to channel* causes the printer to skip to the channel indicated by the value of the *arithmetic expression* after each line of print. The *arithmetic expression* should have a value from 1 through 11. If the *arithmetic expression* yields a value other than integer, it will be rounded to type integer in accordance with the rules applicable to the evaluation of subscripts. If *skip to channel* is used, the channel number and its corresponding line number must be defined in the *line declaration* which follows the *file declaration.*

DBL causes the line printer to double space after the line is printed. **NO** causes the printer to suppress spacing after the line is printed.

- *W format and list part*

The *wformat and list part* specifies the action to be taken on output data. A *format identifier* alone indicates that the referenced *format declaration* contains one or more *strings* which constitutes the entire output.

A *format identifier* together with a *list* or *list identifier* designates that variables in the *list* are to be formatted according to the specifications of the *format declaration* and written as output.

The asterisk, *, together with a *list* or *list identifier,* specifies that the variables in the *list* are to be processed at word length and are to be written as output without being edited. The number of words written is determined by the number of variables in the *list* or the maximum record length, whichever is smaller. When unblocked records are being used, the maximum record length is the buffer size.

The *arithmetic expression* with an *array row* specifies that the elements of the *array row* are to be processed at word length and are to be written as output without being edited. The number of words written is determined by the number of elements in the *array row,* the maximum record length, or the value of the *arithmetic expression,* whichever is smallest. When unblocked records are used the maximum record length is the buffer size.

The *alpha array identifier,* when used in a *write statement* followed by the *list part,* provides a means of calculating a format at execution time. The calculated format must be placed in the one dimensional alpha array prior to execution of the *write statement.*

The *record address part* applies only to drum files and specifies the relative address of the record to be edited and written. If *address* is provided, the record pointer is set to *address* before the *write statement* is executed. If the *record address part* is empty, the record pointer is used as it was left by the previous *I/O statement.* For **RANDOM** files, the *address* must be provided. After a *write statement* is executed, the record pointer will be adjusted to the record following the last record written.

If only the *file part* and *record address part* are provided in a *write statement* referencing a drum file, the contents of the current buffer are written.

- *Edit and Move*

  In a *write statement*, an *array row* may be used instead of *file part*, this indicates an Edit and Move *write statement*. When an Edit and Move write is executed, data from the *list* is edited and placed into the designated *array row*. The data is edited as specified by the *format part* as it is moved from the *list* to the *array row*. In effect, the *array row* designated is used as a buffer area for writing a record.

- *Waction labels*

  Waction labels provide a means of transferring control from a *write statement* when an end of reel condition or a parity condition occurs.

11.3.4.3. Restrictions

- Writing of mixed mode tapes is not possible.
- When a parity error occurs in writing a tape, the redundant record is left in the buffer.
- If a parity error occurs and the label is not present, the program terminates.
- If an end of reel occurs and the label is not present the program continues; the record is not written.

11.3.4.4. Examples

```
LABEL        OPERATION            OPERAND              COMMENTS
1         10         20         30         40         50         60

          WRITE (FILEID,FMT,LISTID);

          WRITE (FILEID [PAGE]);

          WRITE (FILEID,FMT);

          WRITE (FILEID,*,LISTID) [:PAR];

          WRITE (FILEID [DBL],FMT,A,B,C,ARA[6]);

          WRITE (FILEID,X+Y-Z,ARA3[1,1,*]);

          WRITE (FILEID);

          WRITE (FLE[X+2],FT,LST) [EOF];

          WRITE (FILEID,10,A[*]);

          WRITE (FILEID,FRMT,LST) [EOF:PAR];

          WRITE (FILEID,FMT,FOR I:=1 STEP 1 UNTIL 100 DO ARA[I],A,B);

          WRITE (A[*],FMT,LST);

          WRITE (FILEID,A[*],LST);
```

11.3.5. Space Statement

11.3.5.1. Syntax

<space statement> : : = **SPACE** (<file part> , <number of records>) <action label>

<number of records> : : = <arithmetic expression>

<file part> : : = <file identifier> | <switch file designator>

<action label> : : = [<end of file label> : <parity label>]
| | [<end of file label>] | | [: <parity label>] | <empty>

11.3.5.2. Semantics

The *space statement* is used to bypass input logical records without reading them.

The value of the *arithmetic expression* determines the number of records to be spaced and the direction of the spacing. If the expression is positive, the records are spaced in a forward direction; if negative, in the reverse direction.

For drum file, the *space statement* is used to adjust the value of the record pointer; i.e., the value of *number of records* is added to the current record pointer.

11.3.5.3. Examples

```
1         10        20        30        40
SPACE (FILE1D, 5) [LEONF: LPAR];

SPACE (FILE1D, -3) [LEOF: LPAR];

SPACE (FILE1D, A+B-C);
```

11.3.6. Close Statements

11.3.6.1. Syntax

<close statement> : : = **CLOSE** (<file part>, **RELEASE**) | **CLOSE** (<file part>, **SAVE**) | **CLOSE** (<file part>, *) | **CLOSE** (<file part>, **PURGE**) | **CLOSE**(<file part>)

11.3.6.2. Semantics

The *close statement* causes the referenced file to be closed.

The following actions take place:

- On a card output file, a card containing an ending label is punched.
- On a line printer file, the printer is skipped to the next page, an ending label is printed, and the printer is again skipped to the next page.
- On a tape output file, the end of file mark is written after the last block on tape.
- On a drum file, the buffer areas reserved for the file are returned and if the specified file is a temporary file, the drum space for the file is returned.

If **RELEASE** is used, the I/O unit is released to the system. If the file is a tape file, the tape is rewound.

If **SAVE** is used, the I/O unit is released to the system, but the unit requirement of the program is not reduced. If the file is a tape file, the tape is rewound.

If **PURGE** is used, the permanent file is closed, decatalogued, and released to the system.

If the asterisk symbol, *, is used, the file must be a tape file or drum file. The I/O unit remains under program control, and if a tape file, the tape is not rewound. This construct is used to create multi-file reels.

When the asterisk symbol, *, is used on multi-file input tapes, the following action takes place:

- If the last reference to the file was a forward *read statement* or a forward *space statement* and a **CLOSE** *(file part,*)* is executed, the tape is positioned forward to the label of the succeeding file, if any.
- If the last reference to the file was a reverse *read statement* or a reverse *space statement* and a **CLOSE**(file part,*) is executed, the tape is positioned to a point just in front of the beginning label for the file.
- If the **CLOSE** (file part,*) is executed after the *end-of-file label* branch has been taken, no action is performed to position the file.

When the **CLOSE** (<file part> ,*) is used on a single-file reel, the action taken is the same as for a multi-file reel. The next reference to this file must be a *read statement* in the opposite direction from that of the prior read on the file.

When the **CLOSE** (<file part>) is used, it applies only to a random access drum file.

All file buffer areas are released as a result of all *close statement*s being executed except **CLOSE** (<file part> ,*).

11.3.6.3. Examples

```
1         10        20        30        40
CLOSE (FILEID, RELEASE);
CLOSE (FILEID, SAVE);
CLOSE (FILEID, *);
CLOSE (FILEID, PURGE);
CLOSE (FILEID);
```

11.3.7. Rewind Statement

11.3.7.1. Syntax

<rewind statement> : : = **REWIND** (<file part>)

<file part> : : = <file identifier> | <switch file designator>

11.3.7.2. Semantics

The *rewind statement* is used only in reference to tape files and drum files. For tape files, it causes the referenced file to be closed and the tape to be rewound. The I/O unit remains under program control.

For drum files the *rewind statement* causes the record pointer to be set to the address of the first record in the file.

11.3.7.3. Restriction

On paper tape files, the *rewind statement* may be used only on input.

11.3.7.4. Example

```
1         10        20        30        40
REWIND (FILE1);
```

11.3.8. Lock Statement

11.3.8.1. Syntax

<lock statement> : : = **LOCK** (<file part>, **RELEASE**) | **LOCK** (<file part>, **SAVE**) | **LOCK** (<file part>)

11.3.8.2. Semantics

The *lock statement* is used only in reference to tape files or drum files. For tape files it causes the referenced file to be closed, the tape to be rewound, an end of file to be written if output, and an operator message to be printed instructing the operator to remove the reel and save it. If **RELEASE** is used, the I/O unit is released to the system after a new reel is made ready, and the unit requirement is relieved. If **SAVE** is used, the I/O unit is released to the system after a new reel is made ready, but the unit requirement is not relieved. The **RELEASE** or **SAVE** must be used for tape files to indicate whether the I/O unit requirement is relieved or not.

For drum files all *lock statements* cause similar actions. These are as follows:

(1) A temporary file declared **SERIAL** or **RANDOM** is made permanent, catalogued.

(2) The buffer areas reserved for the file are returned.

11.3.8.3. Examples

```
1         10        20        30        40
LOCK (FILEID, RELEASE);

LOCK (FILEID, SAVE);

LOCK (FILEID);
```

11.4. I/O SWITCH DESIGNATORS

11.4.1. General

Each particular I/O switch designator will be discussed separately in succeeding sections.

11.4.1.1. Syntax

<I/O switch designator > : : = <switch file designator> | <switch format designator> | <switch list designator>

11.4.1.2. Semantics

I/O switch designators are used in *I/O statements* in the same manner as *file identifiers*, *format identifiers*, and *list identifiers*.

11.4.2. Switch File Designator

11.4.2.1. Syntax

<switch file designator> : : = <switch file identifier> [<subscript expression>]

<switch file identifier> : : = <identifier>

11.4.2.2. Semantics

Switch file designators are used in *I/O statements* in the same fashion as *file identifiers*.

A *switch file designator* is used in conjunction with the *switch file declaration* specified by the *switch file identifier*. The value of the *subscript expression* determines which *file identifier* in the related *switch file list* is to be selected for use in the *I/O statement*. The value of the *subscript expression* must correspond to the position of the *file identifiers* in the *switch file list*. The values of these positions start with zero. If the value of the expression is other than integer, it will be converted to an integer in accordance with the rules applicable to *subscript expressions*.

If a *switch file identifier* is used as a parameter in a procedure, it must be indicated in the *specification part* of the procedure. The *specifier* used to indicate this is **SWITCH FILE**. The actual parameter which corresponds to such a *formal parameter* must be a *switch file identifier*.

If a *switch file designator* is used as an *actual parameter* to a procedure, the corresponding *formal parameter* must appear in the *specification part* preceded by the *specifier*, **FILE**.

11.4.2.3. Restrictions

The value of the *subscript expression* should correspond to the position of one of the *file identifiers* in the *switch file list*. If the value of the expression is outside the range of the *switch file list*, the file so referenced in the *I/O statement* is undefined.

11.4.2.4. Examples

```
SWHF [I]

SWFILE [IF X>N THEN 0 ELSE 1]

F1SW [INTEGER (X<N)]
```

11.4.3. Switch Format Designator

11.4.3.1. Syntax

<switch format designator> : : = <switch format identifier> [<subscript expression>]

<switch format identifier> : : = <identifier>

11.4.3.2. Semantics

Switch format designators are used in *I/O statements* in the same fashion as are *format identifiers*.

A *switch format designator* is used in conjunction with the *switch format*

Switch format designators are used in *I/O statements* in the same fashion as are *format identifiers*.

A *switch format designator* is used in conjunction with the *switch format declaration* specified by the *switch format identifier*. The value of the *subscript expression* determines which *editing specification part* in the related *switch format list* is to be selected for use in the *I/O statement*. The value of the *subscript expression* must correspond to the position of one of the *specification parts* in the *switch format list*. The values of these positions start with zero. If the value of the expression is other than integer, it will be converted to integer in accordance with the rules applicable to *subscript expressions*.

If a *switch format identifier* is used as a formal parameter in a procedure, the *specifier* used to indicate this is **SWITCH FORMAT**. The *actual parameter* which corresponds to such a *formal parameter* must be a *switch format identifier*.

If a *switch format designator* is used as an *actual parameter* to a procedure, the corresponding *formal parameter* must appear in the *specification part* preceded by the *specifier*, **FORMAT**.

Restrictions:

1. The value of the *subscript expression* should correspond to the position of one of the *editing specification parts* in the *switch format list*. If the value of the expression is outside of the range of the *switch format list*, the *editing specification* so designated in the *I/O statement* is undefined.

11.4.3.3. Examples

SF [I]

SFHFT [**IF** X<N **THEN** O **ELSE** N]

11.4.4. Switch List Designator

11.4.4.1. Syntax

<switch list designator> : : = <switch list identifier> [<subscript expression>]

<switch list identifier> : : = <identifier>

11.4.4.2. Semantics

Switch list designators are used in *I/O statements* in the same fashion as *list identifiers*. A *switch list designator* is used in conjunction with the *switch list declaration* specified by the *switch list identifier*.

The value of the *subscript expression* determines which *list identifier* will be used from the *switch list*. The value of the *subscript expression* must correspond to the position of one of the *list identifiers* in the *switch list*. The values of these positions start with zero. If the value of the expression is other than integer, it will be converted in accordance with the rules applicable to *subscript expressions*. If a *switch list identifier* is used as a *formal parameter* in a procedure, the *specifier* used to indicate this is **SWITCH LIST.** The *actual parameter* which corresponds to such a *formal parameter* must be a *switch list identifier*. If a *switch list designator* is used as an actual *parameter* to a procedure, the corresponding *formal parameter* must appear in the *specification part* preceded by the *specifier*, **LIST.**

Restrictions:

1. The value of the *subscript expression* should correspond to the position of one of the *list identifiers* in the *switch list list;* otherwise, the list so referenced in the *I/O statement* is undefined.

11.4.4.3. Examples

SWLIST [I]

SLST [**IF** A>B **THEN** 0 **ELSE** I+1]

12. ACTIVITY CONTROL

12.1. ACTIVITY STATEMENTS

12.1.1. Syntax

<activity statement> : : = <execute statement> | <wait statement> | <delete statement> | <event statement>

12.1.2. Semantics

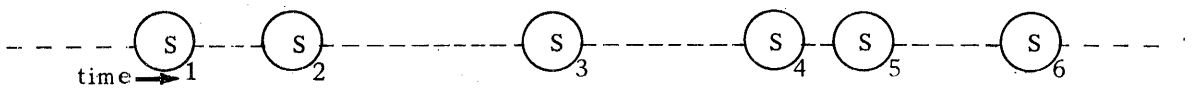
12.1.2.1. Synchronous and Asynchronous Processing

Normally, when the object code produced by the compiler is executed, it is regarded as one activity by the executive system. The program is executed serially following the logic of the source statements. This is synchronous processing.

A program can be structured so that serial execution of all parts is not necessary; it can be written as several independent parts which do not depend on each other and which can be executed in parallel. This is asynchronous processing. *Activity statements* allow programs to take advantage of this independent structure by calling procedures for processing in parallel with the main program.†

The following diagrams depict the difference between synchronous and asynchronous processing.

Figure 1 shows synchronous processing; the statements are executed serially in time. One statement is processed only when all those preceding it have been processed.

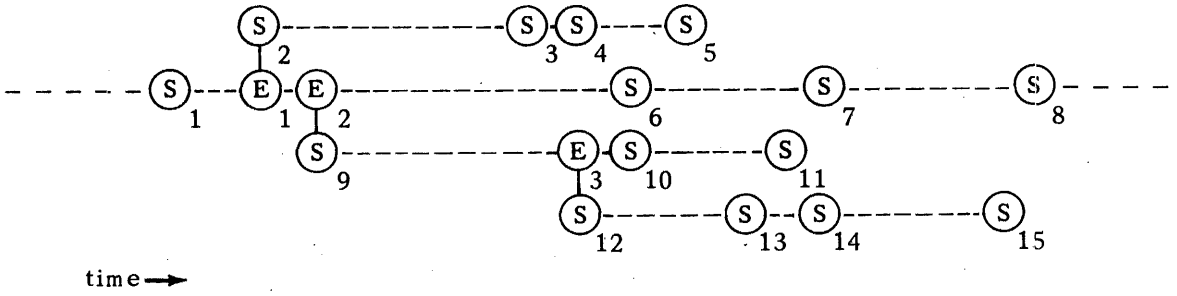


S represents an ALGOL statement.

Figure 12–1. Synchronous Processing

† For discussion purposes the main program will be treated as any other activity.

Figure 2 shows asynchronous processing. When an *execute statement* (E) is encountered an independent activity is begun. The set of *statements* on each line is then scheduled with the executive system for simultaneous execution.



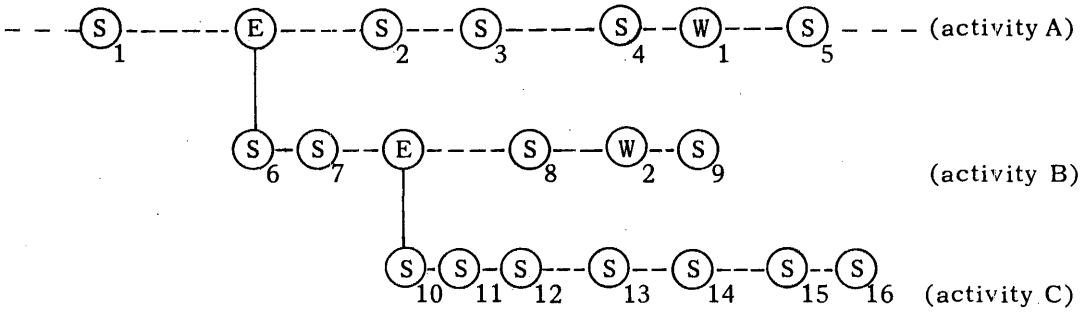
S : ALGOL statements other than *activity statements*.

E : *execute statements*

Figure 12–2. Asynchronous Processing

12.1.2.2. Synchronization of Asynchronous Activities

In order to synchronize two or more activities (e.g., when the individual results of several activities are to be printed in a single report), an activity may 'wait' until another, or several other, activities have completed.



S : ALGOL statements other than *activity statements*

E : execute *statements*

W : *wait statements*

Figure 12–3. Synchronization of Asynchronous Activities

In Figure 3, three activities are diagrammed (A, B, C). At statement W_1 activity A waits upon B. At W_2 B waits upon C.

Activity A will not proceed past W_1 until B completes. B will not complete until C completes, because at W_2 activity B waits upon C. Therefore, at the time statement S_5 is encountered, the entire process is again synchronous, and all statements following S_5 are processed serially (until another **EXECUTE** is encountered).

12.1.2.3. Task Variables

A task is composed of one or more activities, allowing the programmer to treat related activities as a unit. Tasks are formed by giving a *task variable* as a parameter to the *execute statement*. Since *task variables* carry information necessary to communicate with the executive system, they must be given when it is desired to delete† or synchronize tasks.

12.1.2.4. Event Variables

An *event variable* represents a sub-activity, and has two states, set and cleared. The state of an *event variable* is program controlled by use of the *event statements*, **SET** and **CLEAR.**

An *event variable* can be used to synchronize processing at the sub-activity level (using the *ewait statement).* The state of an *event variable* may be tested by the intrinsic Boolean function **ETEST** (Table 3.1, Section 3.2.2.2.) which yields a **TRUE** value if the variable is set. At block entry, all *event variables* are initially cleared.

12.2. EXECUTE STATEMENT

12.2.1. Syntax

<execute statement> : : = **EXECUTE** (<task specification>)

<task specification> : : = <procedure call> | <procedure call> , <task variable>

<task variable> : : = <variable>

12.2.2. Semantics

The *execute statement* allows programs to multi-process by calling procedures which will be processed simultaneously with the calling block. Each procedure is registered with the executive as an independent activity, using the Executive function, **FORK$.**

In an *execute statement,* if a *task variable* is given the procedure call is associated with that *variable.* If no *task variable* is given, the procedure call is not associated with any task.

† See special cases of the *delete statement* for exceptions.

An activity is terminated in any of the following ways:

(1) it reaches the final **END** of the procedure;

(2) it tries to complete a *goto statement* to a non-local label;

(3) it generates an unrecoverable run time error (e.g., subscript out of range; improper format, etc.),

(4) a *delete statement* is encountered naming the procedure's associated *task variable.*

In order to wait upon or delete a particular activity, a *task variable* must be given in the *execute statement,* which causes the executive to assign the activity an identity. There is a limit (36) to the number of activities with an identity which a program may have executing simultaneously. If this limit is exceeded, ALGOL will wait until an identified activity terminates and reuse that identity. The programmer need not be aware of this restriction because ALGOL will do its own run time queueing of procedures.

NOTE: There is no limit to the number of activities without identities a program may have. If it is necessary to have more than the maximum number of activities simultaneously processing, non-identified activities may be used.

12.2.3. Restrictions

It is important to consider the effect of multiprocessing upon both the system and the individual user.

The *execute statement* allows the user to take advantage of multiple processors to decrease his throughput time., e.g., in a job which has an elapsed time of 5 minutes, 10 minutes of CPU time can be taken advantage of. Extra time will be used due to overhead for initialization of activities and activity control housekeeping done by ALGOL. Multiprocessing also increases system overhead because the executive must now schedule several separate activities simultaneously.

In demand or real-time situations, a quick response can be vital. In these situations, the ability to schedule independent activities can be used to achieve this quick response.

In a batch mode, however, where rapid throughput is not necessary, caution should be used against indiscriminate use of this capability.

ALGOL library procedures (SIN, COS, MOVE, etc.) may not be used in an *execute statement.*

External ALGOL procedures, which have been separately compiled, can be used in an *execute statement.*

12.2.4. Example

LABEL | OPERATION | OPERAND | COMM

```
          TASK  TASKCONTROL;

          EXECUTE(PROC1(ARG1));

          EXECUTE(PROC2);

          EXECUTE(PROC1(INPUT,OUTPUT),TASKCONTROL);
```

12.3 WAIT STATEMENT

12.3.1. Syntax

<wait statement> : : = **WAIT** (<task list>)

<task list> : : = <task variable> | <task list>, <task variable>

12.3.2. Semantics

The *wait statement* gives to programmers the capability to synchronize processing. If all the activities associated with the *task variable*s named are completed (or if the *task variable*s were never associated with any activity) the *wait statement* has no effect on the program.

If any activity associated with the *task variable*s given is still processing or scheduled for processing, the activity issuing the wait suspends itself. It will continue processing only after all activities associated with the *task variables* in the *task list* have terminated.

12.3.3. Example

```
          TASK  TASKER,T1;
          TASK  ARRAY  CONTROL[1:10];

          WAIT(TASKER,T1);
          FOR  I:=1  STEP  2  UNTIL  5  DO  WAIT(CONTROL[I]);
```

12.4 DELETE STATEMENT

12.4.1. Syntax

<delete statement> ::= **DELETE**(<task list>) | **DELETE** (<arithmetic expression>)

<task list> ::= <task variable> | <task list>, <task variable>

12.4.2. Semantics

The *delete statement* is used when one activity wishes to terminate another activity. When the *delete statement* is encountered, it causes the immediate termination of all activities associated with the named *task variables* using the executive function ADLT$.

If an *arithmetic expression* is used, it must be equal to 0 or 1. Any other value will cause an error to occur.

If the value is zero, all activities other than the one giving the **DELETE** are terminated. If the value is one, all activities without identities (those initiated without a *task variable* in the *execute statement*) are deleted. It is possible for an activity to delete itself.

12.4.3. Example

```
1         10        20        30        40
TASK T1, T2, T3;
INTEGER I;
.
EXECUTE(PROC1(I), T1);
EXECUTE(PROC2, T1);
.
DELETE(T1, T3);
.
IF DIFF LSS 1.0E-6 THEN DELETE(I);
```

12.5. EVENT STATEMENTS

12.5.1. Syntax

<event statement> : : = <set statement> | <clear statement> | <event wait statement>

<set statement> : : = **SET** (<event list>)

<clear statement> : : = **CLEAR** (<event list>)

<event wait statement> : : = **EWAIT** (<event list>)

<event list> : : = <event variable> | <event list> , <event variable>

12.5.2. Semantics

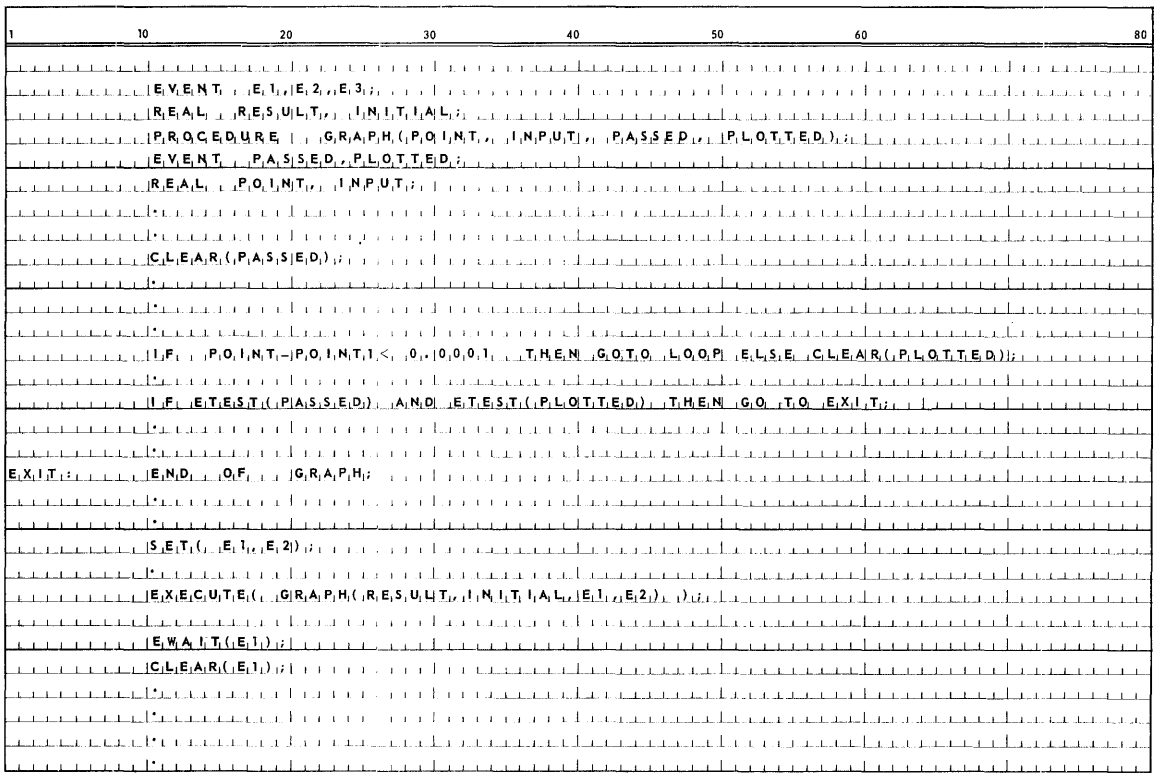
The *event statements* are used to define sub-activity states and to wait for sub-activity completion.

The *set statement* indicates that the event is active. If the intrinsic Boolean function **ETEST** is given a set *event variable* as an argument, it returns a value of **TRUE.** No waiting is done by the **ETEST** function. The *ewait statement* will suspend the activity issuing it until all *event variables* in the list are clear. By using this statement, it is possible to delay an activity until another activity has reached a certain point.

The *clear statement* indicates an event is inactive. An EWAIT issues a cleared variable causing no delay in execution. The ETEST function returns a value of **FALSE.**

NOTE: After an EWAIT on an *event variable* is passed, it leaves that *event variable* set, whether or not it was set originally.

12.5.3. Example



13. SORT/MERGE

13.1. *SORT STATEMENT*

13.1.1. Syntax

<sort statement> : : = **SORT** (<input option>, <output option>, <data reduction procedure>, <sort order>, <range inclusion>, <number of tapes>)

<input option> : : = <input file> | <input procedure>, <record length>

<input file> : : = <file identifier> | <switch file designator>

<input procedure> : : = <Boolean procedure identifier>

<record length> : : = <fixed item size> | <variable item size>

<fixed item size> : : = **FIXED RECORD** <arithmetic expression>

<variable item size> : : = **RECORD** <minimum isize>, <maximum isize>

<minimum isize> : : = <arithmetic expression>

<maximum isize> : : = <arithmetic expression>

<output option> : : = <output file> | <output procedure>

<output file> : : = <file identifier> | <switch file designator>

<output procedure> : : = <Boolean procedure identifier>

<data reduction procedure> : : = <Boolean procedure identifier> | <empty>

<sort order> : : = <own compare> | <key description>

<own compare> : : = **COMPARE** <Boolean procedure identifier>

<key description> : : = **KEY** <format>, <ordering sequence>, <word position>, <bit position>, <number of bits>

<format> : : = <alphabetic variable>

<ordering sequence> : : = <alphabetic variable>

<word position> : : = <arithmetic expression>

<bit position> : : = <arithmetic expression>

<number of bits> : : = <arithmetic expression>

<range inclusion> : : = <hivalue> | <low value> | <empty>

<hivalue> : : = **HIGH** <arithmetic expression> | **HIGH** <arithmetic expression>, **OWN KEY** <rangeown> | **HIGH** <hivalue procedure>

<rangeown> : : = <major own> | <major own>, <minor own>

<major own> : : = <arithmetic expression>

<minor own> : : = <arithmetic expression>

<hivalue procedure> : : = <Boolean procedure identifier>

<low value> : : = **LOW** <arithmetic expression> | **LOW** <arithmetic expression>, **OWN KEY** <rangeown> | **LOW** <low value procedure>

<low value procedure> : : = <Boolean procedure identifier>

<number of tapes> : : = **NTAPES** <integer number> | <empty>

13.1.2. Semantics

The *sort statement* causes data, as specified by the *input option,* to be reordered as directed by *sort order* and returned to the program in the manner specified by the *output option.* All items specified when using the *sort statement* must be specified in the order required as stated in the syntax of the *sort statement.*

■ *Input Option*

An *input option* must be given and must be the first item specified in the *sort statement.* If an *input file* is given as the *input option,* the file must be of fixed item size only. All records on the file will be taken as input. Item size and blocking information will be taken from the *file declaration* and no record specification is to be made in the *sort statement.* This file will be rewound and returned to load point after the SORT has read all the records if tape, and if drum, the pointer will be reset to point to the first item of the file.

If an *input procedure* is specified, that procedure will be invoked to furnish input to the SORT. An *input procedure* must be a Boolean procedure, with an array as its only parameter. Array size is determined by the *record length* specification. The *input procedure*, on each call, will either (1) insert the next record to be sorted into the array parameter, or (2) assign a **TRUE** value to the procedure identifier. When a **TRUE** is returned from the *input procedure* to the SORT, the SORT will not use the contents of the array, and will not call the *input procedure* again. The *record length* represents the length of the records to be sorted. *Record length* must be stated whenever an *input procedure* is specified. For fixed length records, this will be the length of each record presented to the sort. For variable length records, two values must be stated: the minimum record length and the maximum, and they must be stated in that order. Variable length record files to be sorted must have an *input procedure*, as opposed to an input *file declaration*, associated with them.

- *Output Option*

An *output option* must be supplied and must be the second item specified in the *sort statement*. If an *output file* is specified it can only reference a fixed item size file. The sort will write its output on the specified file, close the file upon completion and rewind to load point if tape or first item of the file if drum. If an *output procedure* is specified, it will be invoked once for every record that was sorted, and once to allow "end of output" action. This procedure must be untyped, and must have two parameters: the first a Boolean, and the second an array. Size of the array is determined by *record length*. The first parameter will be **TRUE** if, and only if, the last record has already been returned. If the first parameter is **FALSE** the second parameter will contain a sorted record.

- *Data reduction procedure*

The *data reduction procedure* is optional, but if indicated, must be the third item specified in the *sort statement*. If a *data reduction procedure* is specified, it will be invoked by the SORT each time records of identical keys are encountered as the data is sorted; the option to condense items is open at this point. This must be a Boolean procedure with exactly two parameters, both of which must be arrays. The result which is returned via the procedure identifier should be **TRUE** if the arrays are combined and **FALSE** if not. If the records (arrays) are combined the array given as the first parameter must contain the combined record; that given as second is ignored. The key field defined to the SORT of the two records, if not combined, or of the combined record, if combined, must not be altered. Data reduction cannot be used with files containing variable size records.

- *Sort order*

The programmer has the option of defining a key for the SORT and the SORT will generate the compare coding for determining which of two records should be used next in the sorting process, or the programmer may specify a *compare procedure* of own code.

If a *compare procedure* is specified, it is called by the SORT to determine which of two records should be used next in the sorting process. It must be a Boolean procedure with exactly two parameters, both of which must be arrays. The arrays must both be equal to the largest record contained in the file to be sorted. The result which is returned via the procedure identifier should be **TRUE** if the array given as the first parameter is to appear in the output before the array given as the second or if the order is immaterial; and **FALSE** if the array given as the second parameter is to appear in the output before the array given as the first parameter.

If a *compare procedure* is not stated, then a key must be declared. If key is specified, then the 1108 SORT will generate the compare coding. The *key description* defines the field of the record on which the file is to be ordered. Every record must contain the entire key field described. The output of the sort will be a file reordered on the specified field.

The *key description* is composed of the following five variables:

(1) *Format*

An alphabetic *variable* specifying the *format* of the key field as follows:

A alphanumeric
B signed 1108 binary
D signed decimal
M 7090 IBM* signed binary
U unsigned binary

(2) *Ordering Sequence*

An alphabetic *variable* specifying the desired *ordering sequence* of the key field as follows:

A ascending field
D descending field

(3) *Word Position*

An *arithmetic expression* which must yield an integer number specifying the number of the word within the record containing the most significant bit of the key field. The words within a record are numbered from left to right beginning with 1.

(4) *Bit Position*

An *arithmetic expression* which must yield an integer number in the range of 0 through 35 indicating the bit position within the above defined word which contains the most significant bit of the key field. The bit positions within a word are numbered from right to left beginning with 0.

(5) *Number of Bits*

An *arithmetic expression* which must yield an integer number specifying the length of the key field in bits.

*Trademark International Business Machines Corporation

■ *Range Inclusion*

Range inclusion specifications provide the user with the ability to include only values above or below a given value as input to the SORT. There are three ways in which the *range inclusion* facility can be specified or it need not appear in the *sort statement* at all.

(1) The *hivalue* or *low value* facility can be in relation to the already specified key field of the *sort statement*. In this instance, the key field must only include two words of the item and will be taken as two whole words.

(2) It can be in relation to a defined field, **OWN KEY**. This field can be other than the key field indicated in the *key description* or in the case of a *compare procedure* where no key field is defined. The **OWN KEY** specification allows for the definition to encompass not more than two words of the item. If it is only to include one word, then the word number, beginning with one and counting left to right, must be specified. If it is to encompass two words, then the word numbers of both words must be specified in sequence, first word major, second minor.

(3) It can also be a Boolean procedure. This procedure would be called by the sort in order to determine whether or not to include this item prior to inputting the item to SORT. If specifying a procedure, it must be a Boolean procedure and have an array as its only parameter. The result which is returned via the procedure identifier should be **TRUE** if the item is to be included in the sort and **FALSE** if not.

If a *range inclusion* procedure is desired then the procedure name must stand alone. If a *function designator* is specified, it must have at least one parameter.

Arithmetic expressions for **HIGH, LOW,** and/or **OWN KEY** must yield an integer value.

■ *Number of Tapes*

If scratch tapes are indicated, the number must be specified at compile time and must be an integer number within the range of 2 through 7; it must be the last item specified in the SORT statement and must be preceded by the indicator, **NTAPES**. The philosophy of ALGOL sorts will be to take a standard of core, scaled to 1108 SORT minimum requirements, and calculate a suggested drum, scaled minimum also, and avoid use of scratch tapes. Only if an exceptionally large volume input is expected should scratch tapes be specified.

13.1.3. Restriction

A general limitation is that array subscripts should not be used which exceed the maximum *record length* specified to the SORT.

13.1.4. Examples

Example #1

```
BEGIN COMMENT SORT PROGRAM DEFINING INPUT & OUTPUT PROCEDURES & KEY;
   ALPHA KIND, SEQ; INTEGER WORD, BIT; RLNTH; BOOLEAN B1, B2;
   FILE IN FIN 3 (1,,14);
   FILE OUT FOUT 1 (1,,22);
   ALPHA ARRAY ARIN,ARO[1:14];
   BOOLEAN PROCEDURE INPROC (ARI); ALPHA ARI;
   ALPHA CORE ARRAY AR1;
   BEGIN
      LABEL EOFI, P1, P2;
      READ (FIN,*,AR1); EOFI;
      INPROC:=FALSE; GO TO P1;
      EOFI: INPROC:=TRUE; GO TO P2;
      P1:
      P2: END;
   PROCEDURE OPROC (BOOL1, ARI2); ALPHA CORE ARRAY AR2; BOOLEAN BOOL1;
   BEGIN
      LABEL B2, ALPHA EOFO;
      EOFO:= 'END OF SORT';
      IF BOOL1 THEN BEGIN WRITE (FOUT,*,EOFO); GO TO B2 END ELSE
      WRITE (FOUT,*,AR2);
      B2: END;
   KIND:='A';
   SEQ:='A';
   RLNTH:=14;
   WORD:=1;
   BIT:=35;
   SORT (INPROC, FIXED RECORD RLNTH, OPROC, KEY KIND, SEQ, WORD, BIT, 1080);
END;
```

An example of a SORT program defining input and output files and a key is as follows:

```
BEGIN COMMENT SORT PROGRAM DEFINING INPUT & OUTPUT FILES &  KEYS;
  SAVE ALPHA FILE IN FIN(20,80,14);
  SAVE ALPHA FILE OUT FOUT(20,80,14);
  :
  :
  :
  SORT(FIN,FOUT,KEY,'A',1,35,108);
  :
  :
  :
END;
```

Another example of a SORT utilizing an *own compare* procedure instead of **KEY** can be generated by replacing the SORT call in the first example by the following lines:

```
EXTERNAL BOOLEAN PROCEDURE SCOMPR; COMMENT ALPHA COMPARE
  INSTEAD OF KEY;
SORT(INPROC, FIXED RECORD RLNTH, OPROC, COMPARE SCOMPR);
```

Another example of a SORT utilizing a *data reduction procedure* and an *hivalue procedure* can be generated by replacing the sort call in the first example by the following lines:

```
EXTERNAL BOOLEAN PROCEDURE DROC; EXTERNAL BOOLEAN PROCEDURE HIPROC;
COMMENT DROC IS A DATA REDUCTION PROCEDURE TO BE INVOKED
  UPON OCCURENCE OF DUPLICATE KEYS & HIPROC IS A HIGH VALUE
  PROCEDURE TO CAUSE INPUT RANGE;
SORT(FIN,FOUT,DROC,KEY,'A','A',1,35,108,HIGH,HIPROC);
```

13.2. *MERGE STATEMENT*

13.2.1. Syntax

<merge statement> : : = **MERGE** (<moutput option>, <range inclusion>, <merge order>, <merge file list>)

<moutput option> : : = <output file > | <output procedure>, <mrecord length>

<output file> : : = <file identifier> | <switch file designator>

<output procedure> : : = <Boolean procedure identifier>

<mrecord length> : : = <arithmetic expression>

<range inclusion> : : = <hivalue> | <low value> | <empty>

<hivalue> : : = **HIGH** <arithmetic expression> | **HIGH** <arithmetic expression>, **OWN KEY** <rangeown> | **HIGH** <hivalue procedure>

<rangeown> : : = <major own> | <major own>, <minor own>

<major own> : : = <arithmetic expression>

<minor own> : : = <arithmetic expression>

<hivalue procedure> : : = <Boolean procedure identifier>

<low value> : : = **LOW** <arithmetic expression> | **LOW** <arithmetic expression>, **OWN KEY** <rangeown> | **LOW** <low value procedure>

<low value procedure> : : = <Boolean procedure identifier>

<merge order> : : = <own compare> | <key description>

<own compare> : : = **COMPARE** <Boolean procedure identifier> | <empty>

<key description> : : = **KEY** <format>, <ordering sequence>, <word position>, <bit position>, <number of bits>

<format> : : = <simple alphabetic variable>

<ordering sequence> : : = <simple alphabetic variable>

<word position> : : = <arithmetic expression>

<bit position> : : = <arithmetic expression>

<number of bits> : : = <arithmetic expression>

<merge file list> : : = <merge file>, <merge file> | <merge file>, <merge file list>

<merge file> : : = <file identifier> | <switch file designator>

13.2.2. Semantics

The *merge statement* causes the data in all of the ordered files of the *merge file list* to be combined as directed by *merge order* and returned in merged sequence as specified by the *moutput option.* All items specified when using the *merge statement* must be specified in the required order as stated in the syntax of the *merge statement.*

■ *Moutput Option*

An *output option* must be supplied and must be the first item specified in the *merge statement*. If an *output file* is specified it assumes a fixed record size file. If variable length records are used in the *merge file*s, the record size of the *output file* must be equal to or greater than the largest record size of the *merge file*s. Record size and blocking information will be taken from the *file declaration* and no record specification is to be made in the *merge statement*.

End of merged items will result in an end of file sentinel being written out and the file returned to load point if tape, or first item of the file if drum (See rewind without interlock of Section 11.).

If an *output procedure* is specified, it will be invoked once for every record merged, and once to allow "end of output" action. This procedure must be untyped, and must have two parameters: the first a Boolean, and the second an array. The first parameter will be **TRUE** if, and only if, the last record has already been returned. If the first parameter is **FALSE**, the second parameter will contain a sorted record. Size of the array, specified as the second parameter, is determined by *mrecord length*. The *mrecord length* represents the maximum length of records (items) to be written out after merging, and must be specified when an *output procedure* is indicated and must not be specified when an *output file* is indicated.

■ *Range Inclusion*

Range inclusion specifications provide the user with the ability to include only values within a specified range, or only values above or below a given value as input to the MERGE. There are three ways in which the *range inclusion* facility can be specified or it need not appear in the *merge statement* at all.

(1) The *hivalue* or *low value* facility can be in relation to the already specified key field of the *merge statement*. In this instance, the key field must only include two words of the item and will be taken as two whole words.

(2) It can be in relation to a defined field, **OWN KEY**. This field can be other than the key field indicated in the *key description* or in case of a *compare procedure* where no key field is defined. The **OWN KEY** specification allows for the definition to encompass not more than two words of the item. If it is only to include one word, then the word number, beginning with one and counting left to right, must be specified. If it is to encompass two words, then the word numbers of both words must be specified in sequence, first word major, second minor.

(3) It can also be a Boolean procedure. This procedure would then be called by the merge in order to determine whether or not to include this item prior to inputting the item to MERGE. If specifying a procedure, it must be a Boolean procedure and have an array as its only parameter. The result which is returned via the procedure identifier should be **true** if the item is to be included in the merge and **FALSE** if not. If a *range inclusion* procedure is desired, then the procedure name must stand alone. If a *function designator* is specified, it must have at least one parameter.

Arithmetic expressions for **HIGH, LOW,** and/or **OWN KEY** must yield an integer value.

■ *Merge Order*

The programmer has the option of defining a key for the merge and the MERGE will generate the compare coding for determining which of two records should be used next in the merge process or the programmer may specify a *compare procedure* of own code.

If a *compare procedure* is specified, it is called by the merge to determine which of two records should be used next in the merge process. It must be a Boolean procedure with exactly two parameters, both of which must be arrays. The result which is returned via the procedure identifier should be **TRUE** if the array given as the first parameter is to appear in the output before the array given as the second parameter or if the order is immaterial, and **FALSE** if the array given as the second parameter is to appear in the output before the array given as the first parameter. The arrays must both be equal to the largest record contained in the files to be merged.

If a *compare procedure* is not stated, a KEY must be declared. If *key specification* is given, then the MERGE will generate the compare coding. The *key description* defines the field of the item on which the file is to be ordered. Every item must contain the entire key field described. The output of the merge will be a single file merged on the specified key field. The *key specification* encompasses the following five fields:

(1) *Format*

An alphabetic *variable* specifying the *format* of the key field as follows:

A alphanumeric
B signed 1108 binary
D signed decimal
M 7090 IBM signed binary
U unsigned binary

(2) *Ordering Sequence*

An alphabetic *variable* specifying the desired *ordering sequence* of the key field as follows:

A ascending field
D descending field

(3) *Word Position*

An *arithmetic expression* which must yield an integer number specifying the number of the word within the record containing the most significant bit of the key field. The words within a record are numbered from left to right beginning with 1.

(4) *Bit Position*

An *arithmetic expression* which must yield an integer number in the range of 0 through 35 indicating the bit position within the above defined word which contains the most significant bit of the key field. The bit positions within a word are numbered from right to left beginning with 0.

(5) *Number of Bits*

An *arithmetic expression* which must yield an integer number specifying the length of the key field in bits.

■ *Merge file list*

The *merge file list* names the files to be merged; it must contain at least two and may contain from two to seven ordered files to be merged. Each file named must be a fixed item size file, but the different files can have different size items. Item size and blocking information will be taken from the *file declarations* associated with the particular files named. Exhaustion of files will result in the file being returned to load point if tape, or first item of the file if drum.

13.2.3. Restrictions

A general limitation is that the subscripts of an array should not be used which exceed the maximum record length specified to the MERGE.

13.2.4. Examples

```
1        10        20        30        40        50        60        80
BEGIN COMMENT MERGE EXAMPLE USING OUTPUT FILE & KEY DESCRIPTION;
ALPHA KIND, SEQ; INTEGER WORD, BIT, A, B, RLNTH;
FILE OUTF OUTF (1,,2,2);
FILE IN M1 N1, 3(1,,1,4); COMMENT ORDERED FILES FOR MERGE: M1 N1, M1 N2, M1 N3;
FILE IN M1 N2, 3(1,,1,4);
FILE IN M1 N3, 3(1,,1,4);
KIND:='A';
SEQ:='A';
WORD:=1;
BIT:=35;
MERGE(FOUT, KEY KIND, SEQ, WORD, BIT, 108, M1 N, M1 N2, M1 N3);
END;
```

Other examples with proper procedures could be as follows:

```
1  LABEL  10  OPERATION  20  30  OPERAND  40  COMMENTS  50  60  80
MERGE(OPROC, RLNTH, COMPARE SCOMPR, M1 N1, M1 N2, M1 N3);

MERGE(OPROC, 14, HIGH UL+2, KEY KIND, SEQ, 1, 35, 36, M1 N1, M1 N2, M1 N3);
```

APPENDIX A. ERROR DIAGNOSTICS

There are two general types of errors checked for and indicated by appropriate diagnostics in the UNIVAC 1108 Extended ALGOL compiler: compilation errors and run-time errors.

COMPILATION ERRORS

For compilation errors, an asterisk (*) is printed under the symbol at which a syntax error is detected, and the error diagnostic is printed on the left-hand side of the page on the line following. The compiler attempts to recover and continue compilation following the error diagnostic. Since the compiler attempts to recover at a particular point, symbols that are invalid at that point are read, and spurious error messages are frequently generated following a legitimate error message. The spurious messages will, of course, disappear when the legitimate error is corrected.

The possible compilation time error diagnostics follow:

01 ILLEGAL CHARACTER PAIR
02 CONSTANT TOO LARGE
03 IMPROPER BLOCK STRUCTURE
04 IMPROPER DECLARATION
05 DUPLICATE DECLARATION/SPECIFICATION
06 IMPROPER DECLARATION/SPECIFICATION
07 IMPROPER SPECIFICATION
08 IMPROPER SPECIFICATION
09 MISSING SPECIFICATION
10 IMPROPER OWN DECLARATION
11 IMPROPER EXTERNAL DECLARATION
12 DUPLICATE VALUE SPECIFICATION
13 IMPROPER LABEL SPECIFICATION
14 IMPROPER VALUE SPECIFICATION
15 IMPROPER ARRAY DECLARATION
16 IMPROPER ARRAY DECLARATION
17 IMPROPER LIST DECLARATION
18 IMPROPER LOCAL DECLARATION
19 IMPROPER SWITCH DECLARATION
20 IMPROPER PROCEDURE DECLARATION
21 IMPROPER PROCEDURE PARAMETER
22 DUPLICATE PROCEDURE PARAMETER
23 IMPROPER PARAMETER DELIMITER
24 IMPROPER PROCEDURE SPECIFICATION
25 IMPROPER LABEL DEFINITION
26 DUPLICATE LABEL DEFINITION
27 IMPROPER FORMAT PHRASE
28 IMPROPER FORMAT DECLARATION
29 IMPROPER NAMELIST DECLARATION
30 IMPROPER REPEAT PHRASE
31 IMPROPER SWITCH FORMAT/FILE/LIST
32 UNDEFINED FORMAT SYMBOL
33 IMPROPER FORM DECLARATION
34 IMPROPER USE OF WITH
35 IMPROPER DEFINITION
37 IMPROPER PROCEDURE CALL
38 IMPROPER PROCEDURE ASSIGNMENT STATEMENT
39 IMPROPER IF-STATEMENT
40 IMPROPER IF-STATEMENT
41 IMPROPER USE OF THEN
42 IMPROPER USE OF ELSE
43 IMPROPER FOR STATEMENT
45 IMPROPER ACTIVITY DECLARATION
48 IMPROPER TIMING CLAUSE
52 IMPROPER GO STATEMENT
53 EXTRA RIGHT PARENTHESIS
54 EXTRA LEFT PARENTHESIS
56 MISSING OPERATOR
57 MISSING OPERAND
58 EXTRA END
59 MISSING END
60 IMPROPER USE OF DIV OPERATOR

61 IMPROPER ASSIGNMENT STATEMENT
62 UNDEFINED TRANSFER FUNCTION
63 IMPROPER USE OF A LIST IDENTIFIER
64 IMPROPER USE OF A LABEL
65 IMPROPER USE OF A RESERVED IDENTIFIER
66 IMPROPER USE OF AN ARRAY IDENTIFIER
67 UNDEFINED RELATIONAL OPERAND
69 MISPLACED SEMICOLON
70 MISPLACED COLON
71 MISPLACED COMMA
72 UNDEFINED VARIABLE
75 FILL LIST ELEMENT NOT CONSTANT
78 IMPROPER EVENT STATEMENT
79 COMPILER CAPACITY EXCEEDED
80 DICTIONARY CAPACITY EXCEEDED
84 EXTRA RIGHT BRACKET
85 EXTRA LEFT BRACKET
86 IMPROPER CONCAT SPEC
87 SORT-IMPROPER INPUT SPECIF
88 SORT-IMPROPER OUTPUT SPECIF
89 SORT-IMPROPER SPECIF BEYOND I/O
90 IMPROPER DO STATEMENT
91 MISSING FILE IDENTIFIER
92 IMPROPER IO STATEMENT
93 SORT-DROC IMPOSSIBLE
94 SORT-IMPROPER RECORD SPECIF
95 SORT-IMPROPER SPECIF SORT ORDER
96 SORT-IMPROPER FOLLOWING SORT ORDER
97 MERGE-IMPROPER SPECIFICATION
98 MERGE-ERROR OUTPUT SPECIF
99 MERGE-ERROR RECORD SPECIF
100 MERGE-ERROR MERGE ORDER
101 MERGE-ERROR MERGE FILE LIST

RUN-TIME ERRORS

An error during execution results in the printing of an error message, the name of the library procedure involved, and the line number of the ALGOL program at which execution was currently taking place. The program is then terminated. The list of run-time errors follows:

INCORRECT NUMBER OF ARGUMENTS TO
MEMORY CAPACITY EXCEEDED IN
UNDEFINED DESIGNATIONAL EXPRESSION IN
READ/WRITE ERROR REF A SEG ARRAY
SORT ERROR
RANGE INCLUSION ERROR
IMPROPER PARAMETER TO PROCEDURE
IMPROPER
UNRECOVERABLE DRUM ERROR IN
CHARACTERISTIC UNDERFLOW
INCORRECT NUMBER OF ARGUMENTS TO PROCEDURE
CHARACTERISTIC OVERFLOW
ATTEMPTED DIVISION BY ZERO
IMPROPER NUMBER OF DIMENSIONS FOR
SUBSCRIPT OUT OF RANGE FOR
RESULT UNDEFINED FOR
ARGUMENT OUT OF RANGE FOR
CALLED ROUTINE NOT IN LIBRARY
ILLEGAL OR IMPROPER SEQUENCE OF FORMAT PHRASES IN PARAMETER TO
IMPROPER PARAMETER TO
DEVICE NOT IMPLEMENTED
FILE NOT KNOWN
FILE NOT ASSIGNED
BUFFER IMPROPERLY DEFINED
ERROR IN LIBRARY CALL
FILE PREVIOUSLY RELEASED CANNOT OPEN
FILE PREVIOUSLY OPENED, NEVER CLOSED
INTERNAL FILE ERROR

FEATURE NOT IMPLEMENTED
ATTEMPT TO WRITE INPUT FILE
ATTEMPT TO READ OUTPUT FILE
FILE NOT OPENED WHEN READ ATTEMPTED
DEVICE UNKNOWN
FORMAT SPEC MISSING
UNKNOWN CONTROL WORD IN FILE
EOF, NO RETURN SPECIFIED
ATTEMPTED REWIND ON IMPROPER DEVICE
ALGOL I/O ERROR
NAMELIST INPUT NONBLANK COL 1
NAMELIST ADDRESS NOT SPECIFIED
NAMELIST NAME INCORRECT IN INPUT
VARIABLE/ARRAY/LABELNAME IN INPUT NOT FOUND
SPACE FOUND IN NAME IN NAMELIST INPUT
REVERSE READ. DEVICE ILLEGAL
ITEM SIZE GREATER THAN SPECIFIED
IMPROPER LINE CONTROL SPECIFICATION
(FORMAT) IMPROPER FORMAT DECLARATION
(FORMAT) IMPROPER EXPONENT IN INPUT DATA
(FORMAT) NUMBER IN FORMAT TOO LARGE
(FORMAT) IMPROPER FORMAT PHRASE TYPE
(FORMAT) IMPROPER FORMAT PHRASE
(FORMAT) INPUT DATA VALUE MORE THAN 1 DECIMAL POINT
(FORMAT) INPUT DATA NON-NUMERIC
(FORMAT) INPUT DATA TOO LARGE
(FORMAT) OUTPUT FORMAT USED IN READ
(FORMAT) INPUT DATA NOT LOGICAL VALUE
(FORMAT) INPUT DATA NON OCTAL
(FORMAT) IMBEDDED SPACE IN INPUT DATA
(FORMAT) INPUT VALUE HAS TRUNCATION ERROR
(FORMAT) INPUT DATA CONVERSION ERROR
(FORMAT) OUTPUT DATA TOO LARGE FOR FIELD WIDTH
(FORMAT) OUTPUT DATA VALUE EXPONENT TOO LARGE

APPENDIX B. RESERVED WORDS FOR MARINER 1108 EXTENDED ALGOL

ABSORB
ALPHA
AND
ARRAY
BEGIN
BOOLEAN
CLEAR
CLOSE
COMMENT
COMPARE
COMPLEX
CONVERSION
CORE
DBL
DEC
DEFINE
DIV
DO
DOUBLE
DRUM
DUMP
ELSE
END
EQL
EQV
ETEST
EVENT
EVERY
EWAIT
EXECUTE
EXTERNAL
FALSE
FILE
FILL
FIXED
FOR
FORMAT
FORWARD
GEQ
GO
GOTO
GTR
HIGH
IF
IMP
IN
INTEGER
KEY
LABEL
LEQ
LINE
LIST
LOCK
LOW
LSS
MERGE
MOD
MONITOR
MOVE
NAMELIST
NEQ
NO
NOT
OCT
ON
OR
OUT
OVERFLOW
OWN
PAGE
PROCEDURE
PURGE
RANDOM
REAL
RECORD
RELEASE
REVERSE
REWIND
SAVE
SERIAL
SET
SLEEP
SORT
SPACE
STEP
SWITCH
TASK
THEN
TO
TRACE
TRUE
UNDERFLOW
UNTIL
UPDATE
VALUE
WAIT
WHEN
WHILE
WITH
WRITE
XOR
ZERODIVIDE
ZIP

APPENDIX C. INDEX OF METALINGUISTIC VARIABLES

APPENDIX D. SPECIAL TOPICS

Two of the more sophisticated concepts of the ALGOL 60 language are recursive procedures and the facility to call-by-name or call-by-value procedure parameters.

D1. RECURSIVE PROCEDURE CALLS

The theoretical importance of recursive functions and the growing number of applications in simulation models and experimental mathematics justifies the need for recursive capability and the effort necessary for implementation of this capability.

The "Report on the Algorithmic Language ALGOL 60" states that, "any other (than the appearance in the left part list) occurrence of the procedure identifier within the procedure body denotes activation of the procedure".

The normal sequence of execution in an ALGOL 60 program can be altered by a procedure call statement or by naming a function procedure as a primary in an arithmetic expression. The above quoted paragraph makes it valid for any procedure to call itself, or to call a second procedure which then calls upon the first (direct recursion), or to pass to a procedure as an actual parameter a call upon itself (indirect recursion).

In either case, it is clear that a procedure may be entered several times before an exit occurs. (Note that this is not the case for a begin block; the ALGOL Report states that when a GO TO statement naming a global label to the block is executed, normal block ending must occur.)

If a procedure is involved in recursion all variables local to the procedure, including parameters called-by-value, must be uniquely identifiable for each level of recursion. The dynamic run time stack of an ALGOL implementation, which is essentially "pushed down" for each entry to a block (procedure or begin) and "popped up" at block exit, enables recursive capability.

```
REAL PROCEDURE SIGMAD (I, A, B, X);
  VALUE A; INTEGER I, A, B; REAL X;
  BEGIN
    IF B LSS A THEN SIGMAD:=0
      ELSE BEGIN
        I :=A;
        SIGMAD :=X+SIGMAD (I, A+1, B, X)
        END;
END;
```

Figure 1. Example of Direct Recursion

```
REAL PROCEDURE SIGMAI (I, A, B, X);
  VALUE B; INTEGER I, A, B; REAL X;
BEGIN
  REAL SUM;
  SUM :=O;
  FOR I :=A STEP 1 UNTIL B DO
    SUM :=X+SUM;
  SIGMAI :=SUM
END;
  :
  Y := Z + SIGMAI (C, 1, 50, SIGMAI (R, 1, 50, MAT [C, R]));
     COMMENT WITH THE PROCEDURE SIGMAI DEFINED AND CALLED
       SPECIFYING ITSELF AS A PARAMETER IN THIS MANNER INDIRECT
       RECURSION OCCURS;
```

Figure 2. Example of Indirect Recursion

D2. PARAMETERS CALLED-BY-NAME OR CALLED-BY-VALUE

When a formal parameter to a procedure appears in the value list for the procedure, the ALGOL Report specifies that the value of the actual parameter be delivered to the procedure upon entry to the procedure. If the actual parameter is a variable, the current value is delivered, not the address of the variable, so that the variable, itself, is never altered by the procedure. If the actual parameter is an expression, it is evaluated upon entry to the procedure and the result is passed to the procedure. Note that the value passed to a formal parameter called-by-value becomes local to the called procedure.

As stated in Chapter 10 of this manual, if a formal parameter does not appear in the value list of a procedure declaration, it is assumed to be called-by-name.

A formal parameter called-by-name must be evaluated at every reference to the formal parameter within the procedure body. For actual parameters that are simple variables, this possible multi-evaluation has no effect, since the address of the simple variable is delivered at each reference. Note, however, that since the address of the simple variable, not the value, is delivered to the procedure at each reference to the formal parameter, the value of the variable may be altered by the procedure.

Call-by-name implies more sophistication for actual parameters that are expressions or subscripted variables. At each reference to the formal parameter the expression is evaluated. Since operands of the expression may be known to the procedure (as global variables or as parameters), the evaluation of the expression can yield a different result upon each evaluation.

The following example program, reprinted from the "Communications of the ACM" Volume 8, Number 6, June, 1965, will enable the reader to test his understanding of call-by-name and call-by-value.

"Received February, 1965

TESTING THE UNDERSTANDING OF THE DIFFERENCE BETWEEN CALL-BY-NAME AND CALL-BY-VALUE IN ALGOL 60

The following short program incorporates most of the basic call-by-name and call-by-value problems in Algol 60 procedures. The procedures INCV (x) and INCN (x) differ only in that the formal parameter is called by value in INCV and by name in INCN. Likewise, ADDV (y) and ADDN (y) differ only in the call-by-value for ADDV and call-by-name for ADDN.

The problem is to fill in the appropriate blanks within the comment statements. This little example has been used successfully, but not extensively, both as a teaching device and (for a separate group of students) as a testing device. The answers are given in the last paragraph of this note. Test yourself before examining the answers.

```
BEGIN REAL a, b;
  REAL PROCEDURE INCV (x); VALUE x; REAL x;
    BEGIN x :=x+1; INCV :=x END;
  REAL PROCEDURE INCN (x); REAL x;
    BEGIN x :=x+1; INCN :=x END;
  REAL PROCEDURE ADDV (y); VALUE y; REAL y;
    ADDV :=y+y;
  REAL PROCEDURE ADDN (y); REAL y;
    ADDN :=y+y;
a :=1; b :=ADDV (INCV (a));
COMMENT a is now___, b is now___,
a :=1; b :=ADDV (INCN (a));
COMMENT a is now___, b is now___,
a :=1; b :=ADDN (INCV (a));
COMMENT a is now___, b is now___,
a :=1, b :=ADDN (INCN (a));
COMMENT a is now___, b is now___,
END;
```

The answers for the blanks in reverse order of their occurrence are 5, 3, 4, 1, 4, 2, 4, 1.

ROMAN L. WEIL, JR.
Graduate School of Industrial Administration
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

Received February, 1965

Volume 8/Number 6/June, 1965"

UP-7636