# Burroughs B6500

## Information Processing Systems

## ESPOL REFERENCE MANUAL

# Burroughs

# B 6500

# INFORMATION PROCESSING SYSTEM

## ESPOL

## REFERENCE MANUAL

**Burroughs Corporation**
Detroit, Michigan 48232

$3.00

Printed in U.S. America

1-70

1042744

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Sales Technical Services, Burroughs Corporation, 6071 Second Avenue, Detroit, Michigan 48232.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

## NOTE

The various elements of ESPOL are discussed in paragraphs labeled Syntax, Semantics, and Pragmatics, immediately following each pertinent subject heading. To avoid needless repetition, these subordinate headings have been omitted from the Table of Contents.

# INTRODUCTION

The B 6500 Executive System Problem Oriented Language (ESPOL) is provided primarily for the purpose of writing the B 6500 Master Control Program.

This document is intended to be a reference manual. Its purpose is to describe exactly the structure of the language. Therefore, the manual is directed toward an audience somewhat conversant in the language, rather than the uninitiated. The use of this document presupposes knowledge of B 6500 Extended ALGOL and the operational characters of the B 6500.

# SECTION 1
# SYNTAX CONVENTIONS

## GENERAL.

This section provides a formal discussion of the method used to define ESPOL. The method is rigorous so that the language might be as free from ambiguity as possible.

## METALINGUISTIC SYMBOLS.

A metalanguage is a language used to describe other languages. A metalinguistic symbol is a symbol used in a metalanguage to define the syntax of a language. The following metalinguistic symbols are used in this manual:

a. ⟨⟩ Left and right broken brackets are used to contain one or more characters representing a metalinguistic variable whose value is given by a metalinguistic formula.

b. ::= The symbol ::= means "is defined as." It separates the metalinguistic variable on the left of a metalinguistic formula from the definition of the metalinguistic variable.

c. | The symbol | means "or." It separates alternate definitions of a metalinguistic variable.

d. {} Braces are used to enclose English language definitions when it is impossible or impractical to use a metalinguistic formula.

## METALINGUISTIC FORMULAS.

Metalinguistic symbols are used in forming a metalinguistic formula. A metalinguistic formula is a rule which produces an allowable sequence of characters and/or symbols. The formulas are used to define the syntax of the ESPOL language. The entire set of such formulas developed in this manual defines the B 6500 ESPOL language.

Any mark or symbol in a metalinguistic formula, which is not one of the metalinguistic symbols, denotes itself. The juxtaposition of metalinguistic variables and/or symbols in a metalinguistic formula denotes juxtaposition of those elements in the construct indicated.

An example of a metalinguistic formula is:

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ |
⟨identifier⟩ ⟨digit⟩

This metalinguistic formula is read: an identifier is defined as a letter, or an identifier followed by a letter, or an identifier followed by a digit.

The metalinguistic formula given above defines a recursive relationship by which a construct called an identifier may be formed. That is, evaluation of the formula shows that an identifier begins with a letter. The letter may stand alone, or may be followed by any mixture of letters and digits.

METALINGUISTIC TERMS.

The following terms are used frequently in this manual:

a. Syntax - the systematic arrangement of words.

b. Semantics - the meaning of a word or arrangements of words.

c. Value - an ordered set of numbers (special case - a single number), or an ordered set of logical values (special case - a single logical value).

d. Entity - a thing that has real and individual existence (in this manual, variables, arrays, procedures, labels, etc.)

e. Quantity - an entity which assumes arithmetic values.

f. Process - an algorithm in some state of execution.

g. Scope - the scope of an entity is the block in which the entity is declared. All entities must be declared before

they are referenced in any manner with the exception of labels used under certain circumstances.

h. Recursive - circular usage.

CHARACTER SET.

The character set used in the B 6500 ESPOL language is defined as follows:

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

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨special character⟩ ::= . | , | [ | ] | ( | ) | + | - | / |
< | > | ≤ | ≥ | = | ≠ | ← | % | & | * | # | @ | : |
; | $

⟨string character⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨special character⟩
| ⟨single space⟩

⟨string bracket character⟩ ::= "

⟨single space⟩ ::= {one horizontal blank position}

⟨space⟩ ::= ⟨single space⟩ | ⟨space⟩ ⟨single space⟩

⟨invalid character⟩ ::= ?

⟨character⟩ ::= ⟨string character⟩ | ⟨string bracket character⟩
| ⟨invalid character⟩

SEMANTICS.

The Burroughs Common Language (BCL) character set consists of 64 characters: letters, digits, special characters, the space, the string bracket character, and the invalid character.

The invalid character is not used in the language. It may be used, however, as a character in an output string.

# SECTION 2
# BASIC SYMBOLS

GENERAL.

A symbol is a mark or a contiguous set of marks that represents an object, quality, process, quantity, etc. Basic symbols are symbols whose meaning is given as absolute within the context of the ESPOL language.

SYNTAX.

The syntax for ⟨basic symbol⟩ is:

⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨delimiter⟩ | ⟨empty⟩

⟨logical value⟩ ::= TRUE | FALSE

⟨empty⟩ ::= {a null string of characters}

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨specificator⟩

⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩ | ⟨concatenate operator⟩ | ⟨replacement operator⟩

⟨arithmetic operator⟩ ::= + | - | * | / | DIV | MOD| MULX

⟨relational operator⟩ ::= < | ≤ | = | ≥ | > | ≠ | IS

⟨logical operator⟩ ::= EQV | IMP | OR | AND | NOT

⟨sequential operator⟩ ::= GO | TO | IF | THEN | ELSE | FOR | DO | CASE

⟨concatenate operator⟩ ::= &

⟨replacement operator⟩ ::= ← | := | ←*

⟨separator⟩ ::= , | . | @ | : | ; | ⟨space⟩ | STEP | UNTIL
| WHILE | COMMENT | IN | IS | NULL | OF | BY

⟨space⟩ ::= ⟨single space⟩ | ⟨space⟩ ⟨ single space⟩

⟨single space⟩ ::= {one horizontal blank position}

⟨bracket⟩ ::= ( | ) | [ | ] | " | BEGIN | END | #

⟨declarator⟩ ::= OWN | BOOLEAN | INTEGER | REAL | ARRAY |
LABEL | FORWARD | PROCEDURE | FIELD | LAYOUT | QUEUE
| DEFINE | EVENT | POINTER | DOUBLE | REFERENCE|
PICTURE | USING | SAVE | MONITOR | ON | WORD

⟨specificator⟩ ::= VALUE

SEMANTICS.
Only upper-case letters are permitted. Individual letters do not have individual meanings.

An important function of delimiters is to separate the various entities that make up a program. Delimiters have a fixed meaning. If this meaning is not obvious, it is given at the appropriate place in this manual.

A space must separate any of the following:

a. Multicharacter Delimiter.
b. Logical Value.
c. Identifier.
d. Unsigned Number.

The phrase "reserved words" is used in this manual to denote the set of single-word delimiters. Other than noted above, the use of the space is discretionary.

# SECTION 3
# BASIC COMPONENTS

## GENERAL.

SYNTAX.

The syntax for ⟨basic component⟩ is:

⟨basic component⟩ ::= ⟨identifier⟩ | ⟨number⟩ | ⟨string⟩

SEMANTICS.

Basic components are the most primitive structures of the ESPOL language.

## IDENTIFIERS.

SYNTAX.

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ |
⟨identifier⟩ ⟨digit⟩

Examples:

X
A5
ID
X1
G76D3
NOTHINGTODO
ARITHMETICMEAN

SEMATICS.

Identifiers have no absolute meaning. They are used to name labels, variables, arrays, procedures, etc.

A reserved word may not be used as an identifier.

The maximum permissable identifier length is to be specified.

No space may appear within an identifier.

The same identifier cannot be used to denote two different entities simultaneously.

NUMBERS.

SYNTAX

The syntax for ⟨number⟩ is:

⟨number⟩ ::= ⟨sign⟩ ⟨unsigned number⟩

⟨unsigned number⟩ ::= ⟨decimal number⟩ ⟨exponent part⟩ |
⟨decimal number⟩ | ⟨octal number⟩

⟨decimal number⟩ ::= ⟨unsigned integer⟩ ⟨decimal fraction⟩ |
⟨unsigned integer⟩ | ⟨decimal fraction⟩

⟨integer⟩ ::= ⟨sign⟩ ⟨unsigned integer⟩

⟨sign⟩ ::= ⟨empty⟩ | + | -

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩ ⟨digit⟩

⟨exponent part⟩ ::= @ ⟨integer⟩ | @@ ⟨integer⟩

⟨decimal fraction⟩ ::= . ⟨unsigned integer⟩

⟨octal number⟩ ::= @ ⟨octal constant⟩

⟨octal constant⟩ ::= ⟨octal digit⟩ | ⟨octal constant⟩ ⟨octal digit⟩

⟨octal digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Examples:

| Unsigned numbers: | Decimal numbers: | Integers: |
|---|---|---|
| 1354.543 | 3.14 | +546 |
| @67 | 37 | -62256 |
| 1354.54@68 | .57 | +1 |
| 1.23@73 | 1354 | -565 |
| 27 | .546 | 23 |
| 1@-43 | 1354.543 | |

| Unsigned integers: | Exponent parts: | Decimal Fractions: |
|---|---|---|
| 5 | @68 | .5 |
| 69 | @-46 | .69 |
| 8 | @+54 | .7 |
| 73 | @-8 | .29 |
| | @727 | |
| | @+63 | |

SEMANTICS.

Numbers may be of two types: INTEGER or REAL. Integers are of type INTEGER; all other numbers are of type REAL (explicitly or implicitly -- by default).

The range of permissable real or integer values is to be specified.

The exponent part is a scale factor expressed as an integeral power of 10.

No space may appear within an unsigned number.

An exponent part with a double @@ signifies an extended precision value.

An octal number cannot have more than 16 octal digits.

STRINGS.

SYNTAX.

The syntax for ⟨string⟩ is:

⟨string⟩ ::= ⟨simple string⟩ | ⟨simple string⟩ ⟨string⟩

⟨simple string⟩ ::= ⟨numeric string⟩ | ⟨alpha string⟩

⟨numeric string⟩ ::= ⟨binary code⟩ " ⟨binary string⟩ " |
⟨quaternary code⟩ " ⟨quaternary string⟩ " |
⟨octal code⟩ " ⟨octal string⟩ " | ⟨hexadecimal code⟩
" ⟨hexadecimal string⟩ "

⟨alpha string⟩ ::= ⟨BCL code⟩ " ⟨BCL string⟩ " | ⟨ASCII code⟩
"⟨ASCII string⟩ " | ⟨EBCDIC code⟩ " ⟨EBCDIC string⟩"

⟨binary code⟩ ::= 1 | 10 | 12 | 13 | 14 | 16 | 17 | 18 | 120 |
130 | 140 | 160 | 170 | 180

⟨quaternary code⟩ ::= 2 | 20 | 24 | 26 | 28 | 240 | 260 | 280

⟨octal code⟩ ::= 3 | 30 | 36 | 360

⟨hexadecimal code⟩ ::= 4 | 40 | 48 | 480

⟨BCL code⟩ ::= 6 | 60 | ⟨empty⟩

⟨ASCII code⟩ ::= 7 | 70

⟨EBCDIC code⟩ ::= 8 | 80

⟨binary string⟩ ::= ⟨binary character⟩ | ⟨binary string⟩
⟨binary character⟩

⟨binary character⟩ ::= 0 | 1

⟨quaternary string⟩ ::= ⟨quaternary character⟩ |
⟨quaternary string⟩ ⟨ quaternary character⟩

⟨quaternary character⟩ ::= 0 | 1 | 2 | 3

⟨octal string⟩ ::= ⟨octal character⟩ | ⟨octal string⟩
⟨octal character⟩

⟨octal character⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨hexadecimal string⟩ ::= ⟨hexadecimal character⟩ |
⟨hexadecimal string⟩ ⟨hexadecimal character⟩

⟨hexadecimal character⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9 | A | B | C | D | E | F

⟨ASCII string⟩ ::= ⟨BCL string⟩

⟨EBCDIC string⟩ ::= ⟨BCL string⟩

⟨BCL string⟩ ::= " | ⟨BCL character⟩ | ⟨BCL string⟩
⟨BCL character⟩

⟨BCL character⟩ ::= ⟨string character⟩

SEMANTICS.

Strings may consist of:

a. 1-bit characters (binary).
b. 2-bit characters (quaternary).
c. 3-bit characters (octal).
d. 4-bit characters (hexadecimal).
e. 6-bit characters (BCL).
f. 7-bit characters (ASCII, in 8-bit format).
g. 8-bit characters (EBCDIC).

The string code determines the interpretation of the characters between the quotes. It specifies the character set, and if the string has fewer than 48 bits, the justification of the string within the word which contains it. The first digit specifies the character set in which the source string is written. The next non-zero digit specifies the character size of the internal string created by the compiler. If no second size is specified, it is the same as the initial size. A trailing zero indicates that the string is left-justified within a word if it contains fewer than 48 bits. If no zero is present, the string is right-justified.

An empty string code is treated as a code of 6 (right-justified BCL string).

An ASCII or EBCDIC code may be used only with characters from the BCL character set. The compiler produces (internally) a string of 8-bit characters which represent the same graphics as the given BCL characters. For characters which are not in the BCL character set, the string must be written as a hexadecimal string, where each pair of hexadecimal characters represents the internal code of one ASCII or EBCDIC character.

The quote character may appear only at the beginning of a simple string. Strings with internal quotes must be broken into separate simple strings by the use of three quotes in succession. The last two quotes must be contiguous.

The maximum permissible length of a string depends upon the context in which the string is used. Pointer operations (Replace and String Compare), FILL statements, and list elements which consist of only a string may be represented by strings up to 256 48-bit words in length.

Strings used as operands in expressions are limited to a length of 48 bits.

When a string is formed from simple strings of different character sizes, the following applies:

a. The justification specified by each string code after the first is ignored.

b. Every character in a string is aligned at a character boundary appropriate for that character's size. This may result in zero bits being inserted between simple strings. For example, 6"8" 4"8" produces 001000 001000. Note that two bits are inserted between the simple strings so that the 4"8" falls on a 4-bit character boundary. However, 8"8" 4"8" requires no such alignment.

c. When it is necessary for the compiler to know the length (in characters) of a string and the character size (e.g., Replace and String Compare), the character size is the maximum character size of all the simple strings and the length is the smallest number of characters (of the maximum character size) required to contain all the bits of the string.

# SECTION 4
# GENERAL COMPONENTS

GENERAL.

SYNTAX.

The syntax for ⟨general component⟩ is:

⟨general component⟩ ::= ⟨variable⟩ | ⟨item⟩ | ⟨value designator⟩

SEMANTICS.

General components are constituents of expressions. In principle, general components are less complex structures than expressions.

Because the syntactic definition of general components contains expressions, the definition of expressions and general components is recursive.

VARIABLES.

SYNTAX.

The syntax for ⟨variable⟩ is:

⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

⟨simple variable⟩ ::= ⟨identifier⟩

⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]

⟨array identifier⟩ ::= ⟨identifier⟩

⟨subscript list⟩ ::= ⟨subscript⟩ | ⟨subscript list⟩, ⟨subscript⟩

⟨subscript⟩ ::= ⟨arithmetic expression⟩

Examples:

Simple Variables:

A
ABSOLUTE
ABS2
EPSILON

```
DELTA
A17
ALPHAINFO
BETA4
Q
```

Subscripted Variables:

```
A[5]
QTY[Q+7,VxN,Z]
Q[7,2]
A[5]
A[ITH]
KRONECKER[ITH+2,JTH-ITH]
MAXQ [IF BETA=30 THEN -2 ELSE K+2]
```

Subscript Lists:

```
5
7,2
QV,9
IF J THEN 1 ELSE P*Q+S
ITH
ITH + 2, JTH - ITH
IF BETA=30 THEN-2 ELSE K+2
```

SEMANTICS.

A simple variable is an identifier used to reference some quantity.

A subscripted variable is an array identifier and a subscript list. The array identifier refers to a set of values. An array identifier and a subscript list refers to a single value or a subset of values.

The total number of subscripts in a subscript list must equal the number of dimensions given in the array declaration.

Each subscript expression in a subscript list is evaluated from left-to-right.

If, upon evaluation, a subscript expression yields a value of type REAL, it is rounded automatically as follows:

integral subscript value = ENTIER (value of subscript expression + 0.5)

If the value of a subscript falls outside the bounds declared for that dimension, there is an INVALID INDEX error termination of the program.

ITEMS.

SYNTAX.

The syntax for ⟨item⟩ is:

⟨item⟩ ::= ⟨item identifier⟩ ⟨reference part⟩

⟨item identifier⟩ ::= ⟨identifier⟩

⟨reference part⟩ ::= ⟨empty⟩ | [⟨subscript⟩] |
@ (⟨reference expression⟩)

Examples:

```
PANDORA@(IF MI*1>MI*2 THEN ZGLOT ← F(A) ELSE REFERENCE (B))
NIKE@(NULL)
NERC@(ITEMNEXT← REFERENCE (LOGUN))
AGAM@(NEXTIN)
FIRSTGO @ WAITCHANELQUE[CHANELNUMBER]
NONCHAL@(FORCENT)
FAOWST@(REFERENCE (A[2,1]))
GEHRTA@(F(A+B))
OHNO@(CASE (Z←Z+1) OF (A1; B2; B3; C4; D9));
```

SEMANTICS.

An ⟨item⟩ provides a means to refer to a particular element of a queue entry.

The ⟨item identifier⟩ behaves like a subscript to the ⟨reference part⟩.

The reference part should point to an area which has the form of an entry in the appropriate queue.

a. A reference part of empty points to the same entry as the reference name of the queue.

b. A reference part of [⟨subscript⟩] may only be used with an item identifier declared in a queue array declaration. It points to the same entry as ⟨reference name⟩ [⟨subscript⟩].

VALUE DESIGNATOR.

SYNTAX.

The syntax for ⟨value designator⟩ is:

⟨value designator⟩ ::= ⟨value array identifier⟩ [⟨subscript list⟩]

SEMANTICS.

A value designator references a particular element of a value array.

EVENT DESIGNATORS.

SYNTAX.

The syntax for ⟨event designator⟩ is:

⟨event designator⟩ ::= ⟨event identifier⟩ | ⟨event item⟩ |
⟨event array identifier⟩ [⟨subscript list⟩] |
⟨event array item⟩ [⟨subscript list⟩]

⟨event item⟩ ::= ⟨item⟩
⟨event array item⟩ ::= ⟨item⟩

SEMANTICS.

An event designator designates an event quantity.

The item identifier of an item used as an event item must be specified as an event.

The item identifier of an item used as an event array item must be specified as an event array.

Examples:

Event Designators:

```
E1
E4[2]
```

Simple Event Designators:

```
E2
XERXES
```

Subscripted Event Designators:

```
E4[2]
ZEUS[2,3]
```

# SECTION 5
# EXPRESSIONS

## GENERAL.

SYNTAX.

The syntax for ⟨expression⟩ is:

⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨reference expression⟩ | ⟨pointer expression⟩ | ⟨word expression⟩ | ⟨array expression⟩

SEMANTICS.

An expression is a structure used to obtain a value or values. Expressions are constituents of statements.

## ARITHMETIC EXPRESSIONS.

SYNTAX.

The syntax for ⟨arithmetic expression⟩ is as follows:

⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | ⟨arithmetic assignment⟩ | ⟨word expression⟩ | ⟨IF clause⟩ ⟨arithmetic expression⟩ ELSE ⟨arithmetic expression⟩

⟨simple arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ ⟨adding operator⟩ ⟨term⟩ | ⟨sign⟩ ⟨term⟩

⟨arithmetic assignment⟩ ::= ⟨arithmetic variable⟩ ⟨replacement operator⟩ ⟨arithmetic expression⟩

⟨adding operator⟩ ::= + | -

⟨term⟩ ::= ⟨primary⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩

⟨factor⟩ ::= ⟨primary⟩ | ⟨primary⟩ ** ⟨integer⟩[1]

⟨multiplying operator⟩ ::= * | / | DIV | MOD | MULX

⟨primary⟩ ::= ⟨base⟩ & ⟨layout⟩ | ⟨unsigned number⟩ | ⟨field designator⟩ | ⟨field operand⟩

---

1. The exponentiation operator ** is defined for integer-constant exponents only.

⟨base⟩ ::= ⟨primary⟩

⟨field designator⟩ ::= ⟨field operand⟩ , ⟨field identifier⟩ |
⟨array primary⟩ , ⟨field identifier⟩

⟨field operand⟩ ::= ⟨variable⟩ | ⟨function designator⟩ |
⟨value designator⟩ | ⟨arithmetic item⟩ |
(⟨expression⟩) | ⟨case head⟩ (⟨expression
list⟩)

⟨arithmetic item⟩ ::= ⟨item⟩

⟨layout⟩ ::= ⟨layout identifier⟩ (⟨field value list⟩)

⟨field value list⟩ ::= ⟨field value⟩ | ⟨field value list⟩ ,
⟨field value⟩

⟨field value⟩ ::= ⟨empty⟩ | ⟨expression⟩ | *

⟨expression list⟩ ::= ⟨expression list⟩ , ⟨expression⟩ |
⟨expression⟩

⟨arithmetic variable⟩ ::= ⟨variable⟩

Examples.

Arithmetic Expressions:

```
3
+3
Q
Q-V
HO ← (IF GONE THEN 2 ELSE Z/3)
IF JOY THEN X ELSE 4+Q
W*U-Q(S+CU)
IF Q>0 THEN S+3*Q/A ELSE Z*S+3*Q
IF A<0 THEN U+V ELSE IF A*B> 17 THEN U/V ELSE IF K≠Y
THEN V/U
0.57@12*A[N*(N-1)/2,0)
Q*V*2
P MOD 2
```

```
A[2, SIZ*2 DIV QUANT] ← FI BOOEX THEN Q←Q+1 ELSE 4
```

Simple Arithmetic Expressions:

```
Q+V
Q-V
-Q
3
+3
Q
P MOD 2
Y*3
4*R DIV S
A[I]-B[J]+5.3
```

Terms:

```
Q
Q MOD V
7.394@-8
SUM
W[I+2,8]
2*(X+Y)
Y*3
Q MOD V DIV 2
```

Primaries:

```
7
J.K
J
Q & R
VARINAME & LOOK (6, IF BOOEXP THEN Q←Q+1 ELSE 2, AN[3,5])
2 & SEET (X←F(A+B), 12, VO) & NAW (27, TRUE)
@-72
7
O&CONCAT ()
Q
```

Field Designators:

```
SIGNIFY (X) .Z
VARINAME.FIELDNAME
FUNC(A,TRUE) .F6
(X←ARITHEXP(27)+Q MOD 2).F711
```

Field Operands:

```
CASE X←X+1 OF (V, +27*F(Z,7),ARY[2,BOOQ])
Q
TALLYHO(TFX)
(Q+R*Z-T)
```

Layouts:

```
CHAS (Q+R*6, F(A))
GOT(ZYGLOT,7)
```

SEMANTICS.

An arithmetic expression defines a numeric value.

A variable, value designator, or function designator used as a primary in an arithmetic expression must be of an arithmetic type: INTEGER, REAL, or DOUBLE.

Each expression in a field value list or expression list used in an arithmetic expression must be of an arithmetic type.

The value of an arithmetic expression may be expressed in single or extended precision.

a. The precision is extended if any variable, function designator, or number is of type DOUBLE.

b. The precision of a case expression value is the precision of the first expression of its expression list. If necessary, the other expressions of the expression list are adjusted to conform.

c. The precision of the value of a conditional arithmetic expression is the precision of the arithmetic expression preceding the ELSE.

d. Extended precision values may not be used in a field designator or as a base.

The operator DIV denotes integer division.

Y DIV Z = SIGN(Y/Z)xENTIER(ABS(Y/Z))

The operator MOD denotes remainder division.

Y MOD Z = Y - (Zx(SIGN(Y/Z)xENTIER(ABS(X/Z))))

The exponentiation operator, **, is defined for integer-constant exponents only.

The sequence in which operations are performed is determined by the precedence of the operators. The order of precedence of operators is:

a. First: **
b. Second: *,/, MOD, DIV, MULX
C. Third: +,-

When operators are of the same order of precedence, the sequence of operation is determined by the left-to-right order of appearance of the operators.

An expression between parentheses is evaluated by itself and this value is used in subsequent calculations. That is, the normal order of precedence of operators can be overridden by the judicious placement of parantheses. Therefore, the desired order of execution within an expression can always be arranged by the appropriate positioning of parentheses.

No two operators may be adjacent.

An empty field causes the initial field as specified by the field value part in the declaration, to be assigned to the field. A field value of * causes the field to be ignored. If no initial value is specified, then ⟨empty⟩ is equivalent to *.

BOOLEAN EXPRESSIONS.

SYNTAX.

The syntax for ⟨Boolean expression⟩ is as follows:

⟨Boolean expression⟩ ::= ⟨Boolean assignment⟩ | ⟨simple Boolean expression⟩ | ⟨word expression⟩ ⟨IF clause⟩ ⟨Boolean expression⟩ ELSE ⟨Boolean expression⟩

⟨Boolean assignment⟩ ::= ⟨Boolean variable⟩ ⟨replacement operator⟩ ⟨Boolean expression⟩

⟨simple Boolean expression⟩ ::= ⟨simple Boolean expression⟩ EQV ⟨implication⟩ | ⟨implication⟩

⟨implication⟩ ::= ⟨implication⟩ IMP ⟨Boolean term⟩ | ⟨Boolean term⟩

⟨Boolean term⟩ ::= ⟨Boolean term⟩ OR ⟨Boolean factor⟩ | ⟨Boolean factor⟩

⟨Boolean factor⟩ ::= ⟨Boolean factor⟩ AND ⟨Boolean secondary⟩ | ⟨Boolean secondary⟩

⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ | NOT ⟨Boolean primary⟩

⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨relation⟩ | ⟨Boolean item⟩ | ⟨Boolean field operand⟩ | ⟨Boolean field designator⟩ | ⟨Boolean primary⟩ & ⟨Boolean layout⟩ | ⟨value designator⟩

⟨Boolean item⟩ ::= ⟨item⟩

⟨Boolean field designator⟩ ::= ⟨Boolean field operand⟩ . ⟨field identifier⟩

⟨Boolean field operand⟩ ::= (⟨Boolean expression⟩) | ⟨Boolean variable⟩ |
⟨Boolean function designator⟩ |
⟨case head⟩ (⟨Boolean expression list⟩)

⟨Boolean expression list⟩ ::= ⟨Boolean expression⟩ |
⟨Boolean expression list⟩ , ⟨Boolean expression⟩

⟨Boolean layout⟩ ::= ⟨layout⟩

⟨Boolean variable⟩ ::= ⟨variable⟩

⟨Boolean function designator⟩ ::= ⟨function designator⟩

Examples:

Boolean Expressions:

```
BOOLE ← A EQV B[J,1]
TRUE OR FALSE
IF K<1 THEN S>W ELSE L≤C
```

Simple Boolean Expressions:

```
UGO EQV IGO
NOT SO
WEGO OR HEGO EQV IGO EQV UGO
```

Implications:

```
ITISRAINING IMP GROUNDISWET
BOOVAR AND THIS IMP TOMOR[1,2]
THIS IMP THAT IMP THOSE
NOT SO
```

Boolean Terms:

```
(B>C) OR (D>E)
BOOV AND BOON OR BOOK
A[1,2] AND BVAR OR (NOT THT) OR YEST
NOT SO
```

Boolean Factors:

```
BOOV AND BOON
NOT (J>2) AND TRUD
A[J+1,Z-3] AND VARB AND NEXTM
NOT SO
```

Boolean Secondaries:

```
TRUE
NOT SO
```

Boolean Primaries:

```
FALSE
X>Y
(NOT SO)
BOOVAR.F2
TRUE & CONGLOM (TRUE,FALSE,Z+10,TRUE)
```

Boolean Field Designators:

```
BAHK.LEFTMOST
FRNT.FORTYON
```

Boolean Field Operands:

```
(IF K>1 THEN V<2 ELSE (V2←DYNAM))
BOOLVARB
BOOFUNC(Q>V)
CASE ZYGLOT OF (TRUE, NOT SO, ETCETERA<Q)
```

SEMANTICS.

A Boolean expression defines a logical value. A variable, value designator, or function designator used as a Boolean primary must be of type Boolean. The sequence in which operations are performed is determined by the precedence of the operators. The order of precedence is:

a. First: Arithmetic expressions
b. Second: Relations
c. Third: NOT
d. Fourth: AND
e. Fifth: OR
f. Sixth: IMP
g. Seventh: EQV

REFERENCE EXPRESSIONS.

SYNTAX.

The syntax for ⟨reference expression⟩ is:

⟨reference expression⟩ ::= NULL | ⟨reference assignment⟩ | ⟨IF clause⟩ ⟨reference expression⟩ ELSE ⟨reference expression⟩ ⟨reference designator⟩ | ⟨entry expression⟩ | ⟨reference item⟩ | REFERENCE (⟨variable⟩) | ⟨function designator⟩ | ⟨word expression⟩ | (⟨reference expression⟩) | ⟨case head⟩ (⟨reference expression list⟩)

⟨entry expression⟩ ::= ⟨queue name⟩ (⟨actual item list⟩)

⟨queue name⟩ ::= ⟨queue identifier⟩ | ⟨queue array identifier⟩

⟨actual item list⟩ ::= ⟨actual parameter list⟩

⟨reference assignment⟩ ::= ⟨reference designator⟩ ← ⟨reference expression⟩

⟨queue designator⟩ ::= ⟨queue identifier⟩ | ⟨queue array identifier⟩ [⟨arithmetic expression⟩]

⟨reference designator⟩ ::= ⟨reference identifier⟩ | ⟨reference name⟩ | ⟨reference array name⟩ [⟨subscript⟩]

⟨reference identifier⟩ ::= ⟨identifier⟩

⟨reference array name⟩ ::= ⟨array identifier⟩ | ⟨reference name⟩

⟨reference item⟩ ::= ⟨item⟩

⟨reference expression list⟩ ::= ⟨reference expression⟩ | ⟨reference expression list⟩ , ⟨reference expression⟩

Examples:

Reference Expressions:

```
IF A > B THEN NEXTONE ← LASTONE ELSE NULL
NULL
NESTONE ← LASTONE
NEXTONE
QUEUP (LOCAT,SIZEX)
```

```
ARY[2,3]
REFERENCE (BOOVARB)
REFFUNC(ARRAY[2,7])
CASE Q[V←7+ATT[NUM]] OF (FIRSTONE, NEXTONE, LASTONE)
```

Entry Expression:

```
QUNM (HERE,THERE,EVY)
```

Queue Designators:

```
QUNM
DYNAM[3]
```

SEMANTICS.
A reference expression points to some object or location. In particular, NULL points to nothing.

A reference name used as a reference array name must belong to a queue array.

An entry expression causes the creation of an entity having the format of an entry in the named queue or queue array. A pointer to this potential entry is returned as a reference expression.

The REFERENCE transfer function generates a reference expression pointing to a variable.

DESIGNATIONAL EXPRESSION.
SYNTAX.
The syntax for designational expression is:

⟨designational expression⟩ ::= ⟨label identifier⟩

SEMANTICS.
A designation expression defines a label.

RELATIONS.

SYNTAX.

The syntax for ⟨relation⟩ is:

⟨relation⟩ ::= ⟨arithmetic relation⟩ | ⟨reference relation⟩ | ⟨string relation⟩ | ⟨pointer relation⟩

⟨arithmetic relation⟩ ::= ⟨arithmetic expression⟩ ⟨arithmetic relational⟩ ⟨arithmetic expression⟩

⟨arithmetic relational⟩ ::= ⟨relational operator⟩ | IS

⟨reference relation⟩ ::= ⟨reference expression⟩ ⟨reference relational⟩ ⟨reference expression⟩

⟨reference relational⟩ ::= = | ≠

⟨string relation⟩ ::= ⟨pointer expression⟩ ⟨relational operator⟩ ⟨pointer expression⟩ FOR ⟨arithmetic expression⟩

⟨pointer relation⟩ ::= ⟨pointer expression⟩ ⟨reference relational⟩ ⟨pointer expression⟩

SEMANTICS.

Relations define the manner in which the various relational operators are used with the various expression types.

The IS operator compares all the bits (including tag bits) of two B 6500 words. If they all are equal, the result of the comparison is TRUE.

POINTER EXPRESSIONS.

SYNTAX.

The syntax for ⟨pointer expression⟩ is:

⟨pointer expression⟩ ::= ⟨simple pointer expression⟩ | ⟨if clause⟩ ⟨pointer expression⟩ ELSE ⟨pointer expression⟩

⟨simple pointer expression⟩ ::= ⟨pointer primary⟩ ⟨skip⟩
| ⟨pointer assignment⟩ | ⟨word array row⟩ |
⟨subscripted word variables⟩

⟨pointer primary⟩ ::= ⟨pointer identifier⟩ | (⟨pointer
expression⟩) | ⟨case head⟩ (⟨pointer expression
list⟩) | ⟨pointer designator⟩

⟨skip⟩ ::= ⟨empty⟩ | ⟨adding operator⟩ ⟨primary⟩

⟨pointer identifier⟩ ::= ⟨identifier⟩

⟨pointer designator⟩ ::= POINTER (⟨pointer parameters⟩)

⟨pointer parameters⟩ ::= ⟨array part⟩ | ⟨array part⟩,
⟨character size⟩

⟨character size⟩ ::= 4 | 6 | 8 | *

⟨array part⟩ ::= ⟨array row⟩ | ⟨subscripted variable⟩ |
⟨array identifier⟩

⟨subscripted word variable⟩ ::= ⟨subscripted variable⟩

⟨array row⟩ ::= ⟨array identifier⟩ [⟨row designator⟩]

⟨row designator⟩ ::= * | ⟨row⟩ , *

⟨word array row⟩ ::= ⟨array row⟩

⟨row⟩ ::= ⟨arithmetic expression⟩ | ⟨row⟩, ⟨arithmetic
expression⟩

⟨pointer expression list⟩ ::= ⟨pointer expression⟩ |
⟨pointer expression list⟩, ⟨pointer expression⟩

⟨pointer assignment⟩ ::= ⟨pointer variable⟩ ⟨replacement
operator⟩ ⟨pointer expression⟩

⟨pointer variable⟩ ::= ⟨variable⟩

SEMANTICS.

A pointer expression defines a character position within an array row. An identifier used as a pointer primary must be of type pointer. If a pointer expression is enclosed in parentheses, it is evaluated first and its value used as a primary.

If skip is not empty, the pointer value is adjusted by L characters to the right or left, where L is the absolute value of the arithmetic expression. If the adding operator is +, skipping is to the right. If the operator is -, skipping is to the left.

A pointer designator may be used to create a pointer value which references a specific character position in an array. The pointer designator has two forms:

a. POINTER (A,L) - yields a pointer value "pointing" to A. A is an array identifier which may either by subscripted or unsubscripted. L is A character length in bits (4, 6, or 8).

b. POINTER (A) - same as pointer (A,6).

c. POINTER (A,*) - yields a pointer with a size field equal to A.[42:3].

A pointer may be initialized either by a pointer assignment or by appearing as an update pointer in a SCAN or REPLACE statement or a string comparison.

ARRAY EXPRESSIONS.

SYNTAX.

The syntax for ⟨array expression⟩ is as follows:

⟨array expression⟩ ::= ⟨array assignment⟩ | ⟨array primary⟩ | ⟨IF clause⟩ ⟨array expression⟩ ELSE ⟨array expression⟩

⟨array primary⟩ ::= ⟨array designator⟩ | (⟨array expression⟩) | ⟨array primary⟩ & ⟨layout⟩ | ⟨word expression⟩

⟨array designator⟩ ::= ⟨array identifier⟩ ⟨subarray designator⟩

⟨array variable⟩ ::= ⟨simple variable⟩ | ⟨array item⟩

⟨subarray designator⟩ ::= ⟨empty⟩ | [⟨subscript part⟩
⟨subarray part⟩ ]

⟨subscript part⟩ ::= ⟨empty⟩ | ⟨subscript list⟩ ,

⟨subarray part⟩ ::= * | ⟨subarray part⟩ ,*

⟨array assignment⟩ ::= ⟨array designator⟩ ⟨replacement
operator⟩ ⟨array expression⟩

⟨array item⟩ ::= ⟨item⟩

Examples:

Array Expressions:

IF BOOVAR THEN ARRVAR [2,3,*] ELSE A2S3 ← A1S3
A2S2←A3S3
ARVR[3,*]

Array Primaries:

QVAR [3,*]
(A2S2←A3S3)
QVAR [2,*] & C2S

Array Designators:

NEXT
NXTON[2,3,*]

Array Variables:

DELTA
ARRAYNAME @ ARRAYDECQUE

Subarray Designators:

[*]
[2,*]

SEMANTICS.

An array designator references a data descriptor. In fact, an array designator is ESPOL for data descriptor. An array assignment initializes or changes the values of the various fields in the corresponding data descriptor.

WORD EXPRESSIONS.

SYNTAX.

The syntax for ⟨word expression⟩ is:

⟨word expression⟩ ::= ⟨word assignment⟩ | ⟨word variable⟩ | ⟨word item⟩ ⟨IF clause⟩ ⟨word expression⟩ ELSE ⟨word expression⟩ | WORD (⟨most expressions⟩) | ⟨function designator⟩ | ⟨case head⟩ ⟨word expression list⟩ | (⟨word expression⟩)

⟨word variable⟩ ::= ⟨variable⟩

⟨word item⟩ ::= ⟨item⟩

⟨most expressions⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨reference expression⟩ | ⟨array expression⟩

⟨word expression list⟩ ::= ⟨word expression⟩ | ⟨word expression list⟩, ⟨word expression⟩

⟨word assignment⟩ ::= ⟨word variable⟩ ⟨replacement operator⟩ ⟨expression⟩

SEMANTICS.

A word expression defines a word value. Word values are regarded as a 48-bit field with no type significance. A word transfer function behaves in much the same manner as the REAL and Boolean transfer functions, i.e., it suppresses syntax checking which would otherwise be invoked.

PRAGMATICS.

A word variable is accessed via a LODT. However, no such action must be expected for the expression associated with the word transfer function - the code applicable to the expression is compiled. There is no guarantee that a correctly compiled word expression produces valid B 6500 code.

# SECTION 6
# PROGRAMS, BLOCKS, AND COMPOUND STATEMENTS

## GENERAL.

SYNTAX.

The syntax for ⟨program⟩ is as follows:

⟨program⟩ ::= ⟨block⟩ . ⟨space⟩

⟨block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩

⟨compound statement⟩ ::= BEGIN ⟨compound tail⟩

⟨block head⟩ ::= BEGIN ⟨declaration⟩ | ⟨block head⟩ ; ⟨declaration⟩

⟨compound tail⟩ ::= ⟨statement⟩ END | ⟨statement⟩ ; ⟨compound tail⟩

Examples:

Program:

```
BEGIN REAL V; V←V+1;END.
```

Block:

```
BEGIN REAL Q; Q←Q+1;END
BEGIN
INTEGER I,K; REAL W;
FOR I←1 STEP 1 UNTIL M DO
FOR K←I+1 STEP 1 UNTIL M DO
BEGIN
W←A[I,K];
A[I,K]←A[K,I];
A[K,I]←W
END
END
```

Compound statement:

```
BEGIN X←0;FOR Y←1 STEP 1 UNTIL N DO X←X+A[Y];
IF X>Q THEN GO TO STOP ELSE IF X>W-2 THEN GO TO W;
AW←ST←X+BOB
END
BEGIN V←V+1 END
BEGIN Q←Q+1;V←V+1 END
BEGIN END
```

Block Head:

```
BEGIN REAL V
BEGIN REAL V; BOOLEAN Q
```

Compound Tail:

```
V←V+1 END
Q←Q+1;V←Q+V END
```

SEMANTICS.

Every block automatically introduces a new level of nomenclature. That is, any identifier occurring within the block may, through an appropriate declaration, be declared LOCAL to the block. The meaning of LOCAL is:

a. The entity represented by the identifier inside the block is not recognized by the identifier outside the block.

b. Conversely, any entity represented by the identifier outside the block is not recognized by that identifier inside the block.

An identifier occurring within a block and not declared within that block is global to the block. This means the identifier represents the same entity inside the block and in the level (or levels) outside it, up to and including the level at which it is declared.

The general design of an ESPOL program is similar to that of an ALGOL program.

The remarks in two previous paragraphs above specifically do not apply where the entity is an interrupt declaration.

# SECTION 7
# STATEMENTS

GENERAL.

SYNTAX.

The syntax for ⟨statement⟩ is as follows:

⟨statement⟩ ::= ⟨conditional statement⟩ | ⟨unconditional statement⟩

⟨conditional statement⟩ ::= ⟨label⟩ : ⟨conditional statement⟩ | ⟨IF clause⟩ ⟨unconditional statement⟩ ELSE ⟨conditional statement⟩ | ⟨IF clause⟩ ⟨statement⟩ | ⟨conditional iteration⟩

⟨unconditional statement⟩ ::= ⟨label⟩ : ⟨unconditional statement⟩ | ⟨block⟩ | ⟨compound statement⟩ | ⟨basic statement⟩ ⟨IF clause⟩ ⟨unconditional statement⟩ ELSE ⟨unconditional statement⟩

⟨label⟩ ::= ⟨label identifier⟩

⟨IF clause⟩ ::= IF ⟨Boolean expression⟩ THEN

Examples:

Statements:

```
IF X>B THEN X←X+1 ELSE GO TO B2
X←A+B
```

Conditional Statements:

```
B2:IF X>0 THEN N←N+1
IF TRUE THEN V:Q←N+M ELSE IF NOT 800 THEN Q:=M/N
IF B←F(A) THEN GO TO START
WHILE TRUE DO IF X←Q+M>2 THEN GO TO L6
```

Unconditional Statements:

```
LBL: GO TO NEXT
```

```
IF Z>X THEN GO TO FLAS ELSE GO SCND
BEGIN Y←X+1;Z←Y+2 END
BEGIN REAL X; LABEL Q; IF Z>P THEN GO TO Q ELSE X←FC(C);
      Q:END
LBL:
```

IF Clause:

```
IF B>A THEN
IF GATE[1,2] AND GATE[1,3] THEN
```

SEMANTICS.

Statements are the units of operation of the language. The definition of statement is recursive because statements may be grouped in compound statements and blocks. A conditional statement causes certain statements to be executed or skipped depending upon the value produced by a Boolean expression.

BASIC STATEMENTS.

SYNTAX.

The syntax for ⟨basic statement⟩ is as follows:

⟨basic statement⟩ ::= ⟨go to statement⟩ | ⟨procedure statement⟩ | ⟨unconditional iteration⟩ | ⟨assignment statement⟩ | ⟨dummy statement⟩ | ⟨case statement⟩ | ⟨string transfer statement⟩

⟨go to statement⟩ ::= GO ⟨to part⟩ ⟨designational expression⟩ | GO ⟨to part⟩ ⟨case head⟩ (⟨designational expression list⟩)

⟨to part⟩ ::= ⟨empty⟩ | TO

⟨designational expression list⟩ ::= ⟨designational expression⟩ | ⟨designational expression list⟩ , ⟨designational expression⟩

⟨dummy statement⟩ ::= ⟨empty⟩

⟨case statement⟩ ::= ⟨case head⟩ ⟨compound statement⟩

⟨case head⟩ ::= CASE ⟨arithmetic expression⟩ OF

Examples:

GO TO Statements:

```
GO TO START
GO NEXT
GO TO CASE ARITHEXP-1 OF (LABEL1, NEXT, EXIT, START)
```

Dummy Statements:

```
L1:
EXIT: NEXT:
```

CASE Statements:

```
CASE V OF BEGIN X←X+1;Z←Z+1 END
```

CASE Head:

```
CASE Z←Q*V-B MOD 7 OF
```

SEMANTICS.

The GO TO statement transfers control to the label which is the value of the designational expression or the designational expression list.

In the case statement, the arithmetic expression in the case head is evaluated and is used to select one of the statements in the compound statement following the case head. The selected statement and only the selected statement is executed.

The statements in the compound statement are numbered starting with zero, and the arithmetic expression in the case head is interpreted as the number of the statement to be executed. Dummy statements should be used to arrange convenient numbering of the statements.

## PROCEDURE STATEMENTS AND FUNCTION DESIGNATORS.

SYNTAX.

The syntax for ⟨procedure statement⟩ is:

⟨procedure statement⟩ ::= ⟨procedure identifier⟩ ⟨actual parameter part⟩

⟨function designator⟩ ::= ⟨function identifier⟩ ⟨actual parameter part⟩

⟨function identifier⟩ ::= ⟨procedure identifier⟩

⟨actual parameter part⟩ ::= ⟨empty⟩ | ((⟨actual parameter list⟩))

⟨actual parameter list⟩ ::= ⟨actual parameter⟩ | ⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩

⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨procedure identifier⟩ | ⟨event designator⟩

⟨parameter delimiter⟩ ::= )" {any sequence of letters, including spaces} "(/,

Examples:

Procedure Statements:

```
ALGORITHM123 (A+2)
ALGORITHM546 (A+2)"AVERAGE PLUS TWO" (CALCRULE)
GETESPDISK
```

Function Designators:

```
J(A,B+2,Q[I,L])
GASVOL(K)"TEMPERATURE"(T)"PRESSURE"(P)
RANDOMNO
```

Actual Parameter Parts:

```
(A,B+2,Q[I,J])
(A+2)
(K)"TEMPERATURE"(T)"PRESSURE"(P)
```

Actual Parameters:

```
A+2
A
CHECKOUT
A[2]
```

Event Designators:

```
E1
XERXES
ZEUS[2,3]
```

Parameter Delimiters:

```
)"TEMPERATURE"(
```

SEMANTICS.

A procedure statement causes a previously defined procedure to be executed.

A function designator returns a value. However, when a function designator is used as a procedure statement, this value is lost.

The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading.

Formal and actual parameters must correspond in type and kind of quantities. The correspondence is obtained by taking the entries of these two lists in the same order.

ITERATION.

SYNTAX.

The syntax for ⟨unconditional iteration⟩ is:

⟨unconditional iteration⟩ ::= ⟨do statement⟩ | ⟨iteration clause⟩ ⟨unconditional statement⟩

⟨conditional iteration⟩ ::= ⟨iteration clause⟩ ⟨conditional statement⟩

⟨do statement⟩ ::= DO ⟨statement⟩ UNTIL ⟨Boolean expression⟩

⟨iteration clause⟩ ::= ⟨while part⟩ DO | ⟨for clause⟩ DO | ⟨thru clause⟩ DO

⟨while part⟩ ::= WHILE ⟨Boolean expression⟩

⟨thru clause⟩ ::= THRU ⟨arithmetic expression⟩

⟨for clause⟩ ::= FOR ⟨controlled variable⟩ ⟨replacement operator⟩ ⟨for part⟩

⟨controlled variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

⟨for part⟩ ::= ⟨initial part⟩ ⟨step part⟩ ⟨final part⟩

⟨initial part⟩ ::= ⟨arithmetic expression⟩

⟨step part⟩ ::= STEP ⟨arithmetic expression⟩ | BY ⟨arithmetic expression⟩

⟨final part⟩ ::= UNTIL ⟨arithmetic expression⟩ | ⟨while part⟩

Examples:

Iteration Clauses:

```
FOR V←Q STEP 1 UNTIL 90 DO
WHILE A>B DO
FOR FIRST ← BY 1 UNTIL LAST DO
```

While Parts:

```
WHILE NOT A ≠ C EQV GATE [1,2]
WHILE TRUE
```

For Clauses:

```
FOR ATLST ← J-2 STEP A ← Q ← J-K UNTIL F(A)
```

Controlled variables:

```
ATLST
V[2,3]
```

For Parts:

```
X←2 STEP X←Y+FUNC(A)UNTIL X←Y MOD 9
```

Initial Parts:

```
P MOD 2
+3
Q
```

Step Parts:

```
STEP IF B = 0 THEN X ELSE Y+2
BY 29
```

Final Parts:

```
UNTIL X-7
WHILE B>8
```

SEMANTICS.

The iteration clause provides the means of forming loops in a program. If BY ⟨arithmetic expression⟩ is used instead of STEP ⟨arithmetic expression⟩, the loop will be constructed using the Step and Branch operator to optimize execution time.

## ASSIGNMENT STATEMENTS.

SYNTAX.

The syntax for ⟨assignment statement⟩ is:

⟨assignment statement⟩ ::= ⟨arithmetic assignment statement⟩ | ⟨Boolean assignment statement⟩ | ⟨queue assignment⟩ | ⟨word assignment⟩ | ⟨reference assignment⟩ | ⟨pointer assignment⟩ | ⟨array assignment statement⟩

⟨arithmetic assignment statement⟩ ::= ⟨arithmetic assignment⟩ | ⟨arithmetic field assignment⟩

⟨arithmetic field assignment⟩ ::= ⟨arithmetic variable⟩ . ⟨field identifier⟩ ⟨replacement operator⟩ ⟨arithmetic expression⟩

⟨Boolean assignment statement⟩ ::= ⟨Boolean assignment⟩ | ⟨Boolean field assignment⟩

⟨Boolean field assignment⟩ ::= ⟨Boolean variable⟩ . ⟨field identifier⟩ ⟨replacement operator⟩ ⟨Boolean expression⟩

⟨queue assignment⟩ ::= ⟨queue designator⟩ ⟨replacement operator ⟨reference expression⟩

⟨array assignment statement⟩ ::= ⟨array assignment⟩ | ⟨array field assignment⟩

⟨array field assignment⟩ ::= ⟨array designator⟩ . ⟨field identifier⟩ ⟨replacement operator⟩ ⟨arithmetic expression⟩

Examples:

Arithmetic Assignment Statements:

V←BxQ-R

JOY.BOY← V←Q

Arithmetic Assignments:

```
ALTHO ← N+1
S[V,K+2]:=3-FUNC(Q=2)
```

Arithmetic Field Assignments:

```
IOQUEUE,ERR ← 6
V[7].LNK := QUED-GONEx7
```

Boolean Assignment Statements:

```
RDY←TRUE
PARTWAY.NXT ← B AND GONE > 6
```

Queue Assignments:

```
QUED ← QUEB(HERE,THERE,EVYWHR)
```

Array Assignment:

```
NEXT←IF A[1,2]AND B[Q,3] THEN VARY[2,*] ELSE A253←A234
```

SEMANTICS.
The assignment statement causes the expression to the right of the replacement operator to be evaluated. The value is assigned (transferred) to the variable or field on the left.

A queue assignment causes the entry denoted by the value of the reference expression to be made available to the insertion part of the designated queue, and this insertion part to be executed.

PRAGMATICS.
⟨word assignment⟩s invoke OVRD action.

There is no guarantee that a correctly compiled WORD ASSIGNMENT produces valid B 6500 code.

STRING STATEMENTS.

SYNTAX.

The syntax for ⟨string transfer statement⟩ is:

⟨string transfer statement⟩ ::= REPLACE ⟨destination⟩ BY
⟨source list⟩

⟨string scan statement⟩ ::= SCAN ⟨source⟩ ⟨scan part⟩

⟨destination⟩ ::= ⟨update pointer⟩ ⟨pointer expression⟩

⟨source list⟩ ::= ⟨source part⟩ | ⟨source part⟩ , ⟨source list⟩

⟨source⟩ ::= ⟨pointer source⟩ | ⟨arithmetic source⟩

⟨scan part⟩ ::= ⟨scan count⟩ ⟨condition⟩ | ⟨condition⟩

⟨source part⟩ ::= ⟨source⟩ ⟨transfer part⟩ | ⟨string⟩
⟨transfer part⟩ ⟨arithmetic source⟩ ⟨arithmetic
transfer part⟩ | ⟨string⟩

⟨transfer part⟩ ::= ⟨scan count⟩ ⟨condition⟩ | ⟨condition⟩
⟨final count⟩ ⟨units⟩ | WITH ⟨picture designator⟩ |
⟨final count⟩ WITH ⟨translate table⟩

⟨arithmetic transfer part⟩ ::= ⟨digit count⟩ | ⟨correct count⟩
| ⟨correct count⟩ ⟨condition⟩

⟨scan count⟩ ::= FOR ⟨update count⟩ ⟨arithmetic expression⟩

⟨final count⟩ ::= FOR ⟨arithmetic expression⟩

⟨digit count⟩ ::= FOR ⟨arithmetic expression⟩ DIGITS

⟨correct count⟩ ::= ⟨scan count⟩ CORRECTLY

⟨pointer source⟩ ::= ⟨update pointer⟩ ⟨pointer expression⟩

⟨arithmetic source⟩ ::= ⟨update variable⟩ ⟨arithmetic expression⟩

⟨condition⟩ ::= WHILE ⟨relational operator⟩ ⟨arithmetic expression⟩ | UNTIL ⟨relational operator⟩ ⟨arithmetic expression⟩ | WHILE IN ⟨table⟩ | UNTIL IN ⟨table⟩

⟨table⟩ ::= ⟨array row⟩ | ⟨subscripted variable⟩

⟨translate table⟩ ::= ⟨table⟩

⟨picture designator⟩ ::= ⟨picture identifier⟩ ⟨repeat parameters⟩

⟨repeat parameters⟩ ::= ⟨empty⟩ | (⟨unsigned integer list⟩)

⟨unsigned integer list⟩ ::= ⟨arithmetic expression⟩ | ⟨arithmetic expression⟩ , ⟨unsigned integer list⟩

⟨update pointer⟩ ::= ⟨empty⟩ | ⟨pointer identifier⟩ :

⟨update variable⟩ ::= ⟨empty⟩ | ⟨simple variable⟩ :

⟨update count⟩ ::= ⟨empty⟩ | ⟨simple variable⟩ :

⟨units⟩ ::= ⟨empty⟩ | WORDS | OVERWRITE

SEMANTICS.

A ⟨string transfer statement⟩ tranfers information from the source to the destination. The unit of information is either words or characters, and the number of units transferred is specified by a count or determined by a condition. If the unit of information is words, then both the source and the destination are right-adjusted to a word boundary before the transfer begins.

A ⟨string scan statement⟩ scans the information in the source. Characters are the unit of information: also, a condition must be present. The number of characters scanned may be determined by the count or the condition.

Characters are the default unit for the transfer statement. For example, WORDS and OVERWRITE both mean units of one word. In addition, OVERWRITE ignores memory protection on the destination.

At the end of a scan or transfer the following updated information is available:

a. The destination update pointer, points to the next position to be filled. If the unit of transfer is words, it points to the left-most character of the next word to be filled.

b. The source update pointer, points to the next unit which is scanned or transferred. If the unit is a word, the update pointer points to the left-most character of the word.

c. For scans and transfers an arithmetic source is thought of as being circular, with the high-order and low-order ends contiguous. The source update variable returns the original expression rotated in such a way that the next character used is in the high-order position.

d. Each time a unit of information is scanned or transferred, the original count, as given by an arithmetic expression, is decremented by 1. This continues until the count reaches zero, if no condition is imposed. If a condition is imposed, the count may not reach zero, and the update count returns the value of the count at the end of the transfer. The reserved-word TOGGLE is true, iff the update count is zero.

The ⟨digit count⟩ converts the source arithmetic expression into an integer in decimal form. The designated number of low-order decimal digits is transferred to the destination and the source update variable returns the original expression DIV the designated power of 10.

The ⟨correct count⟩ rotates the source arithmetic expression so that the appropriate number of low-order characters appear in the high-order of the source. After this rotation a normal transfer occurs. If for example, one character were to be transferred

correctly, the low-order character would be moved to the high-order position before the transfer occurred. Since transfers work from left-to-right, this has the effect of allowing the transfer of right-justified characters in an arithmetic expression.

Scans and transfers always work from left-to-right on destination, source, and arithmetic source. Exceptions are correct count and digit count.

The ⟨translate table⟩ works as follows:

a. Each source character is used to find an 8-bit translation character in an array row. The high-order part of this character is discarded to make it fit the destination character set. It is then stored in the destination.

b. The 8-bit translation character is found in the following way. The low-order 2-bits of the source character are used as the character index and the remaining high-order bits are used as the word index. The word index is used as a subscript to the array row, or it is added to the right-most subscript of the subscripted variable. In the resulting word-character number (2 + (character index)) is the translation character. As usual, the characters are numbered left-to-right, so that the translation character is one of the four low-order 8-bit characters in the word.

The WHILE ⟨relation operator⟩ form of condition causes termination of the SCAN or TRANSFER when the source character ceases to have the designated relation to the low-order character of the condition arithmetic expression. The character which causes termination of the SCAN or TRANSFER is not scanned or transferred. Thus the source update pointer or update variable is pointing to this character.

The UNTIL ⟨relational operator⟩ form of condition causes termination of the SCAN or TRANSFER when the source character has the

designated relation to the low-order character of the condition arithmetic expression. The character which causes termination of the scan or transfer is not scanned or transferred.

The conditions involving the table construct use the bits of the source character to find a test bit in an array row. The character is in the table iff the test bit is on. The test bit is found in the following way. The low-order 5 bits of the source character (4 bits if the source character has only four) are used as the bit index, and the remaining bits, if any, are used as the array row, or it is added to the right-most subscript of the subscripted variable. In the resulting word-bit number (31 - (bit index)) is the test bit. As usual, bits are numbered right-to-left, so that the test bit is one of the low-order 32 bits in the word.

If the source is a string, the string is transferred to the destination under the control of a count. If the count, C, is not greater than the string length, L, then C characters are copied into the destination. If the count, C, is greater than the length, L, and the string is more than 48 bits long, the behavior is unpredictable. If C is greater than L, and the string is shorter than 48 bits, then the string is concatenated with itself until the count is exausted. For example, 8 "ABCD" for 10 transfers "ABCDABABCD", not "ABCDABCDAB".

# SECTION 8
# DECLARATIONS

## GENERAL.

### SYNTAX.

The syntax for ⟨declaration⟩ is as follows:

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨label declaration⟩ |
⟨array declaration⟩ | ⟨field declaration⟩ |
⟨layout declaration⟩ | ⟨queue declaration⟩ |
⟨queue array declaration⟩ | ⟨procedure declaration⟩ |
⟨define declaration⟩ | ⟨event declaration⟩ |
⟨event array declaration⟩ | ⟨picture declaration⟩ |
⟨value array declaration⟩ | ⟨monitor declaration⟩ |
⟨interrupt declaration⟩

### SEMANTICS.

Declarations define certain properties of entities and relate these entities with identifiers.

The entities dealt with in the ESPOL language are:

a. Variables.
b. Labels.
c. Procedures.
d. Fields.
e. Strings.
f. Texts.

Every identifier has a "scope." The scope of the identifier is usually the block in which it is declared. The exceptions are:

a. Formal symbols in a define declaration whose scope is the defines.

b. Formal parameters in a procedure declaration whose scope is the procedure declaration.

c. Invisible formal items in a queue declaration whose scope is the queue declaration.

An identifier is said to be "local" to the block in which it is declared. That is, the entity represented by the identifier inside the block is not recognized by the identifier outside the block. Conversely, any entity represented by the identifier outside the block is not recognized by that identifier inside the block.

An identifier is said to be "global" to a block if:

a. It is not declared in the block.
b. It is declared in an exterior block.

Entry into a block must be through the BEGIN. When the block is entered, all identifiers declared for the block assume the significance implied by the nature of the declarations given, in the order of their appearance in the block head.

Exit from a block may be through the END or by a GO TO statement. At the time of exit from the block, all identifiers which are declared for the block lose their significance.

Some identifiers may be declared with the declarator OWN. This declarator causes the identified quantity to retain its value(s) from one exit in a block to the next entry into that block.

An identifier may not be declared to represent more than one entity in a single block head, formal symbol list, formal parameter list, or formal item list.

TYPE DECLARATIONS.

SYNTAX.

The syntax for ⟨type declaration⟩ is as follows:

⟨type declaration⟩ ::= ⟨type⟩ ⟨type list⟩ | OWN ⟨type⟩ ⟨type list⟩

⟨type⟩ ::= REAL | INTEGER | BOOLEAN | DOUBLE | REFERENCE | POINTER | WORD

⟨type list⟩ ::= ⟨type part⟩ | ⟨type list⟩ , ⟨type part⟩

⟨type part⟩ ::= ⟨identifier⟩ ⟨address part⟩ | ⟨identifier⟩
⟨replacement operator⟩ ⟨initial value⟩

⟨address part⟩ ::= ⟨empty⟩ | = ⟨address⟩

⟨address⟩ ::= ⟨identifier⟩ | ⟨address couple⟩ | ⟨identifier⟩
⟨adding operator⟩ ⟨unsigned integer⟩

⟨address couple⟩ ::= (⟨level⟩ ⟨displacement⟩)

⟨level⟩ ::= ⟨unsigned integer⟩ | - ⟨unsigned integer⟩

⟨displacement⟩ ::= ⟨empty⟩ | , ⟨unsigned integer⟩

⟨initial value⟩ ::= ⟨expression⟩

Examples:

Type Declaration:

```
OWN REAL V,Q;
OWN INTEGER B,A;
INTEGER Q ← 1, R, S=T,V=(3,4);
```

Type List:

```
Q
C=W
GENERAL=(1,2),F12T←5,Z=(3)
```

Type Part:

```
ZERO←1
QUOTE = (3,4)
TERIF=GRT
```

Address:

ZQW
(3,4)

Initial Value:

3
IF B THEN XxQ ELSE BOD+NOTHING

SEMANTICS.
A type declaration defines the type of value of each type identifier.

A type declaration may also either assign the address of the identifier, or assign the initial value of the identifier, or specify that the identifier has the OWN property.

An address which is an identifier assigns the type identifier to the same location as the identifier. The identifier must have been declared previously, and must identify a quantity having a stack address.

An empty address part results in the assignment of an address couple by the compiler.

An address couple is an addressing level and a displacement from the base of that level. The addressing level may range from 0 to 31.

A negative integer used as a level directs the compiler to determine the level referenced by subtracting the integer from the level of the block being compiled.

An empty displacement directs the compiler to use the next available displacement for the level indicated. The range of displacement depends upon the value of level:

| Level | Displacement Range |
|---|---|
| 0,1 | 0-8191 |
| 2,3 | 0-4095 |
| 4-7 | 0-2047 |
| 8-15 | 0-1023 |
| 16-31 | 0-511 |

The specified level must be less than or equal to the current level.

It is possible for two or more identifiers to have the same address couple (directed by the programmer). It is the responsibility of the programmer to see that the antecedents of an address couple, used as an address part, are correct.

An initial value expression assigns the value that the identifier has upon entering the block in which the type declaration appears. The expression is evaluated upon each entry to the block. All initial value expressions are evaluated in the order in which they appear. An initial value expression must be of the same type as the declaration. If an initial value is not specified for any type identifier, the initial value of that identifier is not defined.

The OWN property action is essentially the same as that which would occur if the programmer were to specify an address couple referencing the zero level with an empty displacement.

LABEL DECLARATIONS.

SYNTAX.

The syntax for ⟨label declaration⟩ is:

⟨label declaration⟩ ::= LABEL ⟨label list⟩

⟨label list⟩ ::= ⟨label identifier⟩ | ⟨label list⟩ , ⟨label identifier⟩

⟨label identifier⟩ ::= ⟨identifier⟩

Examples:

Label Declarations:

```
LABEL FOG;
LABEL L7,L8;
```

Label List:

```
START,EXIT,LOOP
NEXT
```

SEMANTICS.
A declared label declaration defines each identifier in its label list as a label identifier.

A label identifier must appear in a label declaration in the head of the block in which it is used to label a statement.

A label identifier need not be declared if its first appearance in the block labels a statement, or if its first appearance in the block is in a GO TO statement and there is no non-local label with the same identifier previously appearing.

ARRAY DECLARATIONS.
SYNTAX.
The syntax for ⟨array declaration⟩ is as follows:

⟨array declaration⟩ ::= ⟨array kind⟩ ARRAY ⟨array list⟩

⟨array kind⟩ ::= ⟨empty⟩ | ⟨local or own type⟩ | SAVE ⟨local or own type⟩

⟨local or own type⟩ ::= ⟨array type⟩ | OWN ⟨array type⟩

⟨array list⟩ ::= ⟨array segment⟩ | ⟨array list⟩ , ⟨array segment⟩ | ⟨initialized array⟩ | ⟨array list⟩ , ⟨initialized array⟩

⟨array segment⟩ ::= ⟨array identifier⟩ ⟨address part⟩ [⟨bound list⟩] | ⟨array identifier⟩ ⟨address part⟩ , ⟨array segment⟩

⟨bound list⟩ ::= ⟨bound⟩ | ⟨bound list⟩ , ⟨bound⟩

⟨bound⟩ ::= ⟨upper bound⟩ | *

⟨upper bound⟩ ::= ⟨arithmetic expression⟩

⟨initialized array⟩ ::= ⟨array identifier⟩ ⟨replacement operator⟩ (⟨constant⟩)

⟨array type⟩ ::= ⟨empty⟩ | REAL | INTEGER | BOOLEAN | REFERENCE | WORD | EVENT

Examples:

Array Declarations:

```
OWN REAL ARRAY AZ, BZ = (3,4), CZ [27]
ARRAY BZ[10]
```

Array Lists:

```
AZ,BZ=(3,4),CZ[27]
PQ,RQ[31]
PC[15],RC[31]
```

Array Segments:

```
C7C=(-2,5),C8C=,C7C,C9C=(4)[*]
C27[1022]
```

Bound Lists:

```
1,56
7
1,*
```

Bounds:

27
*
IF A THEN 1 ELSE Q-7

SEMANTICS.
An array declaration defines one or several identifiers to represent arrays of subscripted variables and gives:

a. The dimension of the array.
b. The bounds of the subscripts.
c. The types of the variables.

The value of an array identifier is a data descriptor representing the ordered set of values of the corresponding array of subscripted variables.

An empty array kind means a default declaration of REAL.

The location specified by the address part must contain a data descriptor or an Indirect Reference Word pointing to a data descriptor.

The bound list gives the maximum value of each subscript, taken in order from left-to-right.

A bound of * means that the programmer is responsible for the allocation of the space for the array.

Expressions used as bounds are evaluated once, from left-to-right, upon entrance into the block. These expressions can depend only on variables and procedures which are non-local to the block for which the array declaration is valid or which are local and have initial value parts. Arrays declared in the outermost block must use constant or * bounds.

Dynamic OWN arrays are not permitted.

A bound of * may not appear to the left of an expression used as a bound in the same bound list.

When an array segment includes non-empty address parts, a bound of * must be used.

FIELD AND LAYOUT DECLARATIONS.

SYNTAX.

The syntax for ⟨field declaration⟩ is as follows:

⟨field declaration⟩ ::= FIELD ⟨field part list⟩

⟨layout declaration⟩ ::= LAYOUT ⟨layout part list⟩

⟨field part list⟩ ::= ⟨field part⟩ | ⟨field part list⟩ , ⟨field part⟩

⟨layout part list⟩ ::= ⟨layout part⟩ | ⟨layout part list⟩ , ⟨layout part⟩

⟨layout part⟩ ::= ⟨layout identifier⟩ (⟨layout item list⟩)

⟨layout item list⟩ ::= ⟨layout item⟩ | ⟨layout item list⟩ , ⟨layout item⟩

⟨layout item⟩ ::= ⟨layout field⟩ ⟨field value part⟩

⟨field value part⟩ ::= ⟨empty⟩ | ⟨replacement operator⟩ ⟨unsigned integer⟩

⟨layout field⟩ ::= ⟨field part⟩ | ⟨field⟩ | ⟨field identifier⟩

⟨field part⟩ ::= ⟨field identifier⟩ = ⟨field⟩ | TAG

⟨field identifier⟩ ::= ⟨identifier⟩

⟨field⟩ ::= ⟨arithmetic expression⟩ : ⟨arithmetic expression⟩

⟨layout identifier⟩ ::= ⟨identifier⟩

Examples:

Field Declarations:

```
FIELD A1 = 3:1, AZ = B:1
FIELD QUIZ = 20:21
```

Layout Declarations:

```
LAYOUT C2S (7:6 ← 5, QA = B:6 ← 92, QUITE),C3S (4:2 ←3)
LAYOUT LOOK (6:42 ← 9,QA = BxQ-R:2)
```

Field Part Lists:

```
A1 = 2:3 , A2 = B:A, A3 = C←F(Q):IF TROL THEN QxZ ELSE
     P-Q/R

B1=B:6
```

SEMANTICS.

A field declaration defines each identifier in its field part list as a field identifier and specifies the field.

A layout declaration identifies each identifier in its layout part list as a layout identifier and specifies a layout item list. A layout item list is composed of one or more fields, referred to as layout items. A layout item may be:

a. A previously declared field identifier.
b. An unidentified field.
c. A field part.

A layout item may specify a default value for the field. If no default value is given and no value is assigned, the field is ignored.

The expressions in the field designation are evaluated whenever the field identifier or layout identifier is used. Where the field or layout identifier appears, the expressions are compiled by textual replacement.

The basic B 6500 word consists of 51 bits:

a. Bits 50, 49, and 48 are referred to as tag bits. These tag bits cannot be addressed directly, however, they may be addressed by the reserved field identifier TAG.

b. The balance of the B 6500 word is referred to as the information field. The information field is address 47-0, that is, left-to-right where 47 is the bit on the far left and 0 is the bit on the far right.

TAG is an intrinsic field. It is equivalent to the field 51:3, except that the latter is not explicitly permitted.

QUEUE AND QUEUE ARRAY DECLARATIONS.

SYNTEX.

The syntax for ⟨queue declaration⟩ is as follows:

⟨queue declaration⟩ ::= QUEUE ⟨queue head⟩ ⟨queue body⟩

⟨queue array declaration⟩ ::= QUEUE ARRAY ⟨queue array head⟩ [⟨index bound⟩] ⟨queue body⟩

⟨queue head⟩ ::= ⟨queue identifier⟩ ⟨reference name part⟩

⟨queue array head⟩ ::= ⟨queue array identifier⟩ ⟨reference name part⟩

⟨index bound⟩ ::= ⟨arithmetic expression⟩

⟨reference name part⟩ ::= ⟨empty⟩ | : ⟨second name ⟩ ⟨address part⟩

⟨reference name⟩ ::= ⟨identifier⟩

⟨queue body⟩ ::= ⟨entry description⟩ ; ⟨algorithm part⟩

⟨entry description⟩ ::= (⟨entry item list⟩); ⟨value part⟩; ⟨specification part⟩

⟨entry item list⟩ ::= ⟨item list⟩ ⟨invisible item list⟩

⟨item list⟩ ::= ⟨item identifier⟩ | ⟨item list⟩ ⟨parameter delimiter⟩ ⟨item identifier⟩

⟨invisible item list⟩ ::= ⟨item list⟩

⟨algorithm part⟩ ::= ⟨empty⟩ | USING ⟨algorithm list⟩

⟨algorithm list⟩ ::= ⟨algorithm⟩ | ⟨algorithm⟩ : ⟨algorithm list⟩

⟨algorithm⟩ ::= ⟨boolean algorithm identifier⟩ IF ⟨Boolean expression⟩ | ⟨reference algorithm identifier⟩ IS ⟨reference expression⟩ | TO ⟨algorithm identifier⟩ , ⟨statement⟩ | ⟨lock specification⟩ | ⟨integer algorithm identifier⟩ = ⟨arithmetic expression⟩

⟨lock specification⟩ ::= LOCKED | LOCKED ⟨queue name⟩

⟨queue identifier⟩ ::= ⟨identifier⟩

⟨queue array identifier⟩ ::= ⟨identifier⟩

⟨reference algorithm identifier⟩ ::= ALLOCATE | NEXT | LAST | FIRST | PRIOR

⟨algorithm identifier⟩ ::= INSERT | REMOVE | DELINK | ⟨identifier⟩

⟨Boolean algorithm identifier⟩ ::= EMPTY | FULL

⟨integer algorithm identifier⟩ ::= POPULATION

SEMANTICS.
A queue is an ordered list of entries. Each entry has the form of an array row with one word for each item identifier in the entry item list.

A queue array is an array of queues. The index bound designates the number of queues involved, and is a strict upper bound for the queue array subscripts.

The entry item list is in effect a declaration for each of the item identifiers appearing in it. Those item identifiers appearing in the invisible item list are understood to be local to the queue declaration, and may not be referenced except in the queue body. The remaining item identifiers are local to the block in which the queue is declared.

In creating an entry expression, the elements in the actual item list must be in a one-to-one correspondence with the item identifiers in the item list of the entry description for the queue in question. The invisible item identifiers do not have corresponding actual parameters.

The reference name of a queue points to a specific entry in the queue. The exact position of this entry in the queue is determined by the queue algorithms. Similarly, the reference name of a queue array points to an array of entries, one for each queue of the queue array.

The reserved word ENTRY is local to the queue body. It is used to refer to the entry which is currently being placed in, or removed from the queue. Also, for the queue array declarations there is the reserved word INDEX which is local to the queue body. INDEX is the current subscript to the queue array.

The ALLOCATE algorithm is invoked whenever it is necessary to allocate space for an entry or an entry expression. If no ALLOCATE algorithm is given, then no entry expression using the queue name may be used.

The INSERT algorithm is used to insert new entries in the queue. It is called each time a queue assignment is used. Thus, no queue assignment may be used for a queue which has no INSERT algorithm.

The queue algorithms used by a queue are local to the block in which the queue is declared. In an explicit call on a queue algorithm, the actual parameter list has the following general format: ⟨queue name⟩ , ⟨reference expression⟩ , ⟨arithmetic expression⟩. The queue name parameter must always be present. The reference expression is not required for all algorithms, and when used is passed to ENTRY in the queue body. The arithmetic expression is only required for queue arrays, and even then, it may not be necessary. The arithmetic expression is passed to INDEX in the queue body of the queue array declaration. The following table should be helpful.

| ALGORITHM | REFERENCE | INTEGER |
|---|---|---|
| REMOVE | NO | YES |
| INSERT | YES | YES |
| DELINK | YES | YES |
| ALLOCATE | NO | NO |
| NEXT | NO | YES |
| LAST | NO | YES |
| FIRST | NO | YES |
| PRIOR | YES | YES |
| EMPTY | NO | YES |
| FULL | NO | YES |
| POPULATION | NO | YES |
| ALL OTHERS | YES | YES |

EXAMPLE:

```
REFERENCE LASTREADY;
QUEUE READYLIST:FIRSTREADY(STKNR,PRIORITY:NEXTREADY,PREADY);
          VALUE  NEXTREADY,PREADY,STKNR,PRIORITY;
          INTEGER STKNR,PRIORITY;
          REFERENCE NEXTREADY,PREADY;

USING
TO INSERT, IF LASTREADY=NULL
                EADY@(LASTREADY←FIRSTREADY←ENTRY)←PREADY←NULL
     ELSE IF PRIORITY@ENTRY ≥ PRIORITY@(PREADY@(ENTRY)←LASTREADY)
                    THEN
          NEXTREADY@(NEXTREADY@(LASTREADY)←LASTREADY←ENTRY)←NULL
     ELSE IF PRIORITY@ENTRY < PRIORITY THEN COMMENT GOES AT HEAD;
          PREADY@(PREADY@(NEXTREADY@(ENTRY)←FIRSTREADY(←FIRST-
                    READY←ENTRY)←NULL

ELSE
BEGIN
     WHILE PRIORITY@ENTRY<PRIORITY@PREADY@ENTRY DO
          PREADY@(ENTRY)←PREADY@(PREADY@(ENTRY));
```

```
            NEXTREADY@(ENTRY)←NEXTREADY@PREADY@ENTRY;
            NEXTREADY@(PREADY@ENTRY)←PREADY@(NEXTREADY@ENTRY)←
                  ENTRY
      END:
EMPTY IF FIRSTREADY=NULL:
TO REMOVE,
      IF ENTRY=FIRSTREADY THEN
            IF FIRSTREADY=LASTREADY THEN FIRSTREADY←LASTREADY←NULL
            ELSE PREADY@(FIRSTREADY←NEXTREADY)←NULL
      ELSE IF ENTRY=LASTREADY THEN
            NEXTREADY@(LASTREADY←PREADY@ENTRY)←NULL
      ELSE PREADY@(NEXTREADY@(PREADY@ENTRY)←NEXTREADY@ENTRY)
            ←PREADY@ENTRY;
```

PROCEDURE DECLARATIONS.

SYNTAX.

The syntax for ⟨procedure declaration⟩ is as follows:

⟨procedure declaration⟩ ::= ⟨save part⟩ ⟨procedure type⟩
PROCEDURE ⟨procedure heading⟩ ⟨procedure body⟩

⟨save part⟩ ::= ⟨empty⟩ | SAVE | SAVE 1

⟨procedure type⟩ ::= ⟨empty⟩ | ⟨type⟩

⟨procedure heading⟩ ::= ⟨procedure identifier⟩ ⟨address part⟩
⟨formal parameter part⟩;

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨formal parameter part⟩ ::= ⟨empty⟩ | (⟨formal parameter list⟩);
⟨value part⟩ ⟨specification part⟩

⟨formal parameter list⟩ ::= ⟨formal parameter list⟩
⟨parameter delimiter⟩ ⟨formal pararemter⟩ | ⟨formal
parameter⟩

⟨formal parameter⟩ ::= ⟨identifier⟩

⟨value part⟩ ::= ⟨empty⟩ | VALUE ⟨identifier list⟩ ;

⟨specification part⟩ ::= ⟨specification part⟩ ; ⟨specification⟩
| ⟨specification⟩

⟨specification⟩ ::= ⟨specifier⟩ ⟨identifier list⟩ |
⟨array specification⟩

⟨array specification⟩ ::= ⟨array type⟩ ARRAY
⟨array specifier list⟩

⟨array specifier list⟩ ::= ⟨array specifier list⟩, ⟨array
specifier⟩ | ⟨array specifier⟩

⟨bound specifier⟩ ::= * | ⟨bound specifier⟩ , *

⟨specifier⟩ ::= ⟨type⟩ | ⟨procedure type⟩ PROCEDURE | QUEUE |
EVENT | PICTURE

⟨procedure body⟩ ::= ⟨statement⟩ | FORWARD | EXTERNAL

⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩ ,
⟨identifier⟩

⟨array specifier⟩ ::= ⟨array identifier list⟩ [ ⟨bound
specifier⟩ ]

⟨array identifier list⟩ ::= ⟨array identifier⟩ | ⟨array
identifier list⟩, ⟨array identifier⟩

Examples:

Procedure Declaration:

```
PROCEDURE FAKE;X←X+1
PROCEDURE NEXT(A); VALUE A; REAL A; X←A+1
PROCEDURE NEXT(A); VALUE A; REAL A; FORWARD
```

Procedure Heading:

```
COMMUNE = INTRIN (A); REAL A;
```

Formal Parameter Part:

```
(A,B,C); VALUE A; REAL A,B,C;
(ABSTRACT,DEGENERATE); VALUE ABSTRACT; REAL ABSTRACT;
PROCEDURE DEGENERATE;
```

Formal Parameter List:

```
A,B,C
X)"VALUE OF EXPRESSION X PLUS 2 "(CALCRULE
```

Value Part:

```
VALUE X,Y,Z
```

Specification Part:

```
INTEGER N; ARRAY A,B,C,X1,X2[*];ALPHA ARRAY X3[*];
```

Specification:

```
INTEGER N,O,P,Q
PROCEDURE MIN,MAX,FIX
QUEUE X,Y,Z
EVENT A,B,C
QUEUE ARRAY FRED [*]
```

Array Specification:

```
REAL ARRAY GYM [*,*]
BOOLEAN ARRAY WADSUP [*,*,*],CUMON[*,*]
```

Array Specifier List:

```
GIGL[*]
HERE[*],THERE[*,*,*,*]
```

Array Specifier:

```
X1,X2,X3[*]
```

Procedure Body:

```
BEGIN I←X+QxR;V←I+RAY(Q);END SAMPLE
FORWARD
```

SEMANTICS.
A procedure declaration defines the procedure identifier as the name of a procedure.

The prescriptions stated above for the use of address part apply to the use with procedure declarations. Also, the address specified must be that of a Program Control Word or an Indirect Reference Word pointing to a Program Control Word.

The value part specifies which formal parameters are to be called by value. When a formal parameter is called by value, the formal parameter is set to the value of the corresponding actual parameter. Thereafter, the formal parameter is handled as a variable that is local to the procedure body. That is, any change of value of the variable cannot ramify outside the procedure body.

Only arithmetic, Boolean, pointer, and reference expressions may be given as actual parameters to be called by value. These expressions are evaluated once, before entry into the procedure body.

Formal parameters not in the value part are called by name (an exception is the event formal parameter discussed below). This means that wherever a formal parameter (called by name) appears in the procedure body, the formal parameter is replaced by the actual parameter. The meaning of "replaced by" is discussed below for each possible type of actual parameter.

Event formal parameters are called by reference (not in any way connected with the ESPOL reference type). Call by reference differs from call by name in that when the actual parameter is subscripted variable, the referenced array element is determined at the time of call. It is this array element which is accessed at each appearance of the formal parameter within the procedure body.

Every formal parameter must appear in the specification part.

An * must appear in a bound specifier for each dimension of the array.

Procedures may be called recursively.

In certain situations where procedures are called recursively, it is necessary to call a procedure that has not been declared. The declarator FORWARD is used for this circumstance. That is, there is first an appropriate procedure declaration with the procedure body replaced by FORWARD, then the call on the procedure, and later a complete procedure declaration.

A save part of SAVE indicates that the code for the procedure currently being declared is to be in the same segment as the block in which it is declared. In other words, the compiler can not create a new segment for the procedure.

A save part of SAVE 1 indicates that the procedure being declared is an INITIALIZATION procedure. The code for the procedure is at the end of the initial block of information which is loaded by the hardware. There will be three words of information between the SAVE 1 code and the rest of the initial block to facilitate relocation of the area after initialization is complete.

## DEFINE DECLARATIONS AND INVOCATIONS.

SYNTAX.

The syntax for ⟨define declaration⟩ is:

⟨define declaration⟩ ::= DEFINE ⟨definition list⟩

⟨definition list⟩ ::= ⟨definition⟩ | ⟨definition list⟩, ⟨definition⟩

⟨definition⟩ ::= ⟨defined identifier⟩ ⟨formal symbol part⟩ = ⟨text⟩ #

⟨defined identifier⟩ ::= ⟨identifier⟩

⟨formal symbol part⟩ ::= ⟨empty⟩ | (⟨formal symbol list⟩)

⟨formal symbol list⟩ ::= ⟨formal symbol⟩ | ⟨formal symbol list⟩
,⟨formal symbol⟩

⟨formal symbol⟩ ::= ⟨identifier⟩

⟨text⟩ ::= {any sequence of valid symbols not including #}

⟨invocation⟩ ::= ⟨defined identifier⟩ ⟨actual text part⟩

⟨actual text part⟩ ::= ⟨empty⟩ | (⟨closed text list⟩) |
[⟨closed text list⟩]

⟨closed text list⟩ ::= ⟨closed text⟩ | ⟨closed text list⟩ ,
⟨closed text⟩

⟨closed text⟩ ::= {an actual text not containing unmatched
bracketing symbols or unbracketed commas}

Examples:

Define Declaration:

```
DEFINE FORI = FOR I←1 STEP 1 UNTIL#,ADDUP = AxB+C/D#
```

Definition List:

```
MOVER = ← #
SPLIT = GO TO#,FOOL(GRANTED, MAYBE) = IF GRANTED THEN MAYBE#
```

Definition:

```
GRANTED (ARITHEXP) = ARITHEXP > NOTSO #
```

Formal Symbol List:

```
IDENTIFIERONE, TWO
ONLY
```

Text:

```
(
PROCEDURE
ANYID
IF A THEN GO TO SOUTH ELSE BEGIN X←ZxQ; GO TO NORTH END EG;
```

Invocation:

```
GUARANTY ( X←Y+1 )
```

Actual Text Part:

```
(ERGO)
[X←1;GO TO L;]
```

SEMANTICS.

The define declaration assigns the meaning of the defined identifiers. An invocation causes the replacement of the invocation by the text associated with defined identifier. If the definition of a defined identifier included any formal symbols, any appearance of these symbols in the text of the definition (but not in a string or comment) is replaced by the corresponding actual texts.

The word COMMENT is recognized in a text. It and all characters up to and including the next semicolon are deleted from the text. No text may include an incomplete comment.

In a closed text list, the closed texts are separated by commas, and the closed text list is terminated by a right parenthesis or bracket. In a closed text, a comma may appear only between matching bracketing symbols. No unmatched (unpaired) bracketing may appear.

The scope of a formal symbol is the text of the definition in which the formal symbol appears.

Bracketing symbols are [ ], ( ), and the group consisting of:
DEFINE = # ;

EVENT AND EVENT ARRAY DECLARATIONS.

SYNTAX.

The syntax for ⟨event declaration⟩ is:

⟨event declaration⟩ ::= EVENT ⟨event list⟩

⟨event list⟩ ::= ⟨event identifier⟩ ⟨address part⟩ |
⟨event list⟩ , ⟨event identifier⟩ ⟨address part⟩

⟨event identifier⟩ ::= ⟨identifier⟩

⟨event array declaration⟩ ::= EVENT ARRAY ⟨event segment list⟩

⟨event segment list⟩ ::= ⟨event segment⟩ | ⟨event segment list⟩
, ⟨event segment⟩

⟨event segment⟩ ::= ⟨event array list⟩ [⟨bound list⟩]

⟨event array list⟩ ::= ⟨event array identifier⟩ ⟨address part⟩
| ⟨event array list⟩, ⟨event array identifier⟩
⟨address part⟩

⟨event array identifier⟩ ::= ⟨identifier⟩

Examples:

Event Declaration:

```
Event E1,E2 = (3,4)
```

Event Array Declaration:

```
EVENT ARRAY E3(NONCE),E4[73],E5(-2)[6]
```

SEMANTICS.

An event declaration defines the identifier of a quantity which may be used to record an occurrence. An event array declaration defines the identifier of an array of these quantities. The quantities are used to report an occurrence to an asynchronous process.

INTERRUPT DECLARATIONS.

SYNTAX.

The syntax for ⟨interrupt declaration⟩ is:

⟨interrupt declaration⟩ ::= INTERRUPT ⟨interrupt list⟩

⟨interrupt list⟩ ::= ⟨interrupt segment⟩ | ⟨interrupt list⟩
,⟨interrupt segment⟩

⟨interrupt segment⟩ ::= ⟨interrupt identifier⟩ : ⟨on part⟩

⟨interrupt identifier⟩ ::= ⟨identifier⟩

⟨on part⟩ ::= ON ⟨event designator⟩, ⟨interrupt statement⟩

⟨interrupt statement⟩ ::= ⟨statement⟩

Examples:

```
INTERRUPT I1:  ON E1,A←A+B,
           I2:  ON EVNT, GO TO LBL;
```

SEMANTICS.

Interrupt declarations provide a means of forcing a process to depart from its current point of control and execute the statement associated with the interrupt declaration.

If the process is inactive at the time of the causation of the event, more than one interrupt statement may be pending when the process is reactivated. In this case, the interrupt statements are processed in the chronological order of the causation of their associated events before return is made to the reactivation point.

An interrupt must be enabled via the ENABLE intrinsic before it can have any effect. The DISABLE intrinsic renders the associated interrupt(s) ineffective.

The scope of an interrupt declaration does not follow the usual ALGOL conventions and is defined as follows:

a. Within any block, two interrupt declarations may not reference the same event entity except when a block is an independent process.

b. In all other respects, the interrupt declaration conforms to the rule of scope related in section 5.

PRAGMATICS.
The problems of scope arise from implementation difficulties. If one is allowed to redefine the action required of a particular process (stack) upon the causation of a particular event, then the stack may be required to carry a great deal of information necessary for correct redefinition. The rule is this--only one interrupt action per event per stack.

PICTURE DECLARATIONS.
SYNTAX.
The syntax for ⟨picture declaration⟩ is as follows:

⟨picture declaration⟩ ::= ⟨save or own⟩ PICTURE ⟨picture part list⟩

⟨save or own⟩ ::= SAVE | OWN

⟨picture part list⟩ ::= ⟨picture part⟩ | ⟨picture part list⟩ , ⟨picture part⟩

⟨picture part⟩ ::= ⟨picture identifier⟩ ( ⟨picture⟩ )

⟨picture identifier⟩ ::= ⟨identifier⟩

⟨picture⟩ ::= ⟨picture symbol⟩ | ⟨picture⟩ ⟨picture symbol⟩

⟨picture symbol⟩ ::= "⟨hexadecimal string⟩" | "⟨BCL string⟩" | , "⟨ASCII string⟩" | "⟨EBCDIC string⟩" | ⟨picture character⟩ ⟨repeat part⟩ | ⟨control character⟩ | ⟨introduction⟩ |

⟨skip character⟩ ⟨repeat part⟩ |
⟨single picture character⟩

⟨repeat part⟩ ::= ⟨empty⟩ | ( ⟨unsigned integer⟩ ) | (*)

⟨control character⟩ ::= 4 | 6 | 7 | 8 | :

⟨skip character⟩ ::= < | >

⟨introduction⟩ ::= ⟨introduction code⟩ ⟨new character⟩

⟨introduction code⟩ ::= B | P | N | C | U | N

⟨new character⟩ ::= ⟨string character⟩ | "

⟨single picture character⟩ ::= J | S

⟨picture character⟩ ::= A | 9 | X | Z | E | I | S | D | F | Q

SEMANTICS.

The PICTURE declaration provides a construct for generalized character editing. The following editing operations may be performed:

a. Unconditional character moves.

b. Move characters with leading zero editing.

c. Move characters with leading zero editing and floating character insertion.

d. Move characters with conditional character insertion.

e. Move characters with unconditional character insertion.

f. Move numeric part of characters only.

g. Skip source characters (forward and reverse).

h. Skip destination characters forward.

i. Insert overpunch sign on the previous character.

A picture consists of a named string of editing symbols which are enclosed in parentheses. The picture editing symbols listed below may be combined in any order to perform a wide range of editing functions. OWN pictures produce in-line code.

If the repeat part is empty, it is assumed to be equal to one. If the repeat part is of the form (*), an unsigned integer value is expected from an edit repeat list in a string statement.

The following output characters are assumed for the introduction codes. Another character may be substituted for the assumed character by the use of the introduction phrase, as defined in the syntax.

| Output Character | Introduction Code | Normal Use |
|---|---|---|
| Space(blank) | B | Replacement of leading zeros. |
| , | C | Conditional insert characters. |
| . | N | Unconditional insert character. |
| - | M | Character insertion if minus. |
| + | P | Character insertion if plus. |
| $ | U | Floating character insertion. |

The control characters shown below cause the following action:

a. 4 - set the default character size of inserted characters and strings to 4 bits.

b. 6 - set the default character size of inserted characters and strings to 6 bits.

c. 7 - set the default character size of inserted characters and strings to 7 bits.

d. 8 - set the default character size of inserted characters and strings to 8 bits.

e. : re-initiates leading zero replacement.

Quoted strings are inserted unconditionally in the destination string. The control characters 4,6,7,8 tell the compiler what character set to expect. BCL is the default character set.

The single picture characters perform the following action:

a. J - if a move with a float (E or F) has not inserted a float character, terminate the float and insert the U character, otherwise perform no operation.

b. S - insert a single P character if the sign is plus, otherwise insert a single M character.

The picture characters listed below perform the following action:

a. A - move the number of characters specified by the repeat field.

b. 9 - move the numeric part only of the number of characters specified by the repeat field.

c. E - move the numeric part only for the number of characters specified by the repeat field. Suppress leading zeros by substituting the B character. If the sign is plus, insert a P character in front of the first non-zero number. Otherwise, insert an M character. End the float action.

d. F - perform a move numeric with leading zeros replaced by the B character. Insert a U character in front of the first non-zero number. End the float action.

e. D - if an E or F float has not ended, insert the B character. Otherwise, insert the C character.

f. Q - back up the number of characters indicated by the repeat part and insert a sign overpunch.

g. R - if an E or F float has not ended, insert the P character. Otherwise, insert the M character.

h. I - insert the N character unconditionally.

i. X - skip the destination pointer forward by the number of characters specified in the repeat field and replace any leading zeros with the B character.

The picture skip characters perform the following action:

a. < - skip the source pointer backward by the number of characters specified in the repeat field.

b. > - skip the source pointer forward by the number of characters specified in the repeat field.

VALUE ARRAY DECLARATIONS.

SYNTAX.

The syntax for ⟨value array declarations⟩ is:

⟨value array declaration⟩ ::= ⟨array save part⟩ ⟨value type⟩ VALUE ARRAY ⟨value array part list⟩

⟨array save part⟩ ::= SAVE | ⟨empty⟩

⟨value type⟩ ::= REAL | INTEGER | DOUBLE | BOOLEAN

⟨value array part list⟩ ::= ⟨value array identifier⟩ ← (⟨constant list⟩)

⟨value array identifier⟩ ::= ⟨identifier⟩

⟨constant list⟩ ::= ⟨constant⟩ | ⟨constant list⟩ , ⟨constant⟩

⟨constant⟩ ::= ⟨unsigned integer⟩ (⟨constant list⟩) | ⟨number⟩ | ⟨logical value⟩ | ⟨string⟩

Examples:

```
SAVE BOOLEAN VALUE ARRAY DISPLAYOPT←(TRUE, FALSE, FALSE, TRUE,
FALSE)
```

SEMANTICS.

The value array declaration defines a 1-dimensional array of values.

MONITOR DECLARATIONS.

SYNTAX.

The syntax for ⟨monitor declaration⟩ is:

⟨monitor declaration⟩ ::= MONITOR ⟨procedure identifier⟩ (⟨monitor list⟩)

⟨monitor list⟩ ::= ⟨monitored item⟩ | ⟨monitor list⟩ , ⟨monitored item⟩

⟨monitored item⟩ ::= ⟨simple variable⟩

SEMANTICS.

Within the scope of a monitor declaration, the procedure identified in the monitor declaration is executed whenever a value is to be assigned (via the assignment statement) to a monitored item. The procedure must have the same type as the item and must return the value to be assigned to the item. The procedure may only have two parameters. The parameters must be specified as value. The first parameter corresponds to the first eight characters of the identifier of the monitored item. The second parameter corresponds to the value of the expression.

# SECTION 9
# INTRINSICS

## GENERAL.

### SYNTAX.

The syntax for ⟨intrinsic⟩ is as follows:

⟨intrinsic⟩ ::= ⟨array intrinsic⟩ | ⟨procedure intrinsic⟩ | ⟨function intrinsic⟩

⟨function intrinsic⟩ ::= ⟨arithmetic intrinsic⟩ | ⟨Boolean intrinsic⟩

⟨array intrinsic⟩ ::= MEMORY | M | STACK | WORDSTACK | STACKVECTOR | REGISTERS

⟨procedure intrinsic⟩ ::= EXIT | ALLOW | DISALLOW | PAUSE | HEYOU | HALT | RETURN | TIMER | MOVESTACK | SCANOUT | IIO | MASKSEARCH | LISTLOOKUP | BUZZ | BUZZCONTROL | WAIT | CAUSE | SET | RESET | FREE | DISABLE | ENABLE | HOLD | STOREITEM

⟨arithmetic intrinsic⟩ ::= NAME | SECONDWORD | MYSELF | BINARY | JOIN | ENTER | ONES | FIRSTONE | ABS | NAKS | DECIMAI | SCANIN | SET | RESET | READLOCK | SIZE | XSIGN

⟨Boolean intrinsic⟩ ::= TOGGLE | OVERFLOW | LOCK | UNLOCK | BUSY | HAPPENED | FIX | AVAILABLE

### SEMANTICS.

The ⟨array intrinsic⟩ has the following significance:

a. MEMORY and M are equivalent. They are 1-dimensional arrays referencing memory.

b. STACK is a 2-dimensional array. As used by the MCP stack, [m,n] refers to the nth word in stack number m. WORDSTACK if equivalent to STACK, except that it is a word array.

c. STACKVECTOR is a 1-dimensional array. As used by the MCP, STACKVECTOR references a particular row of STACK.

d. REGISTERS is a 1-dimensional array which references the various machine registers. For example, REGISTERS [34] is the current setting of the PIR.

The following intrinsics generate simple operators in the code string which may or may not return a value.

| Intrinsic | Operator | Result |
|---|---|---|
| MYSELF | WHOI | The number of the processor which is running. |
| ALLOW | EEXI | Enable external interrupts. |
| DISALLOW | DEXL | Disable external interrupts. |
| PAUSE | IDLE | Idle. |
| HEYOU | HEYU | Interrupt other processors. |
| XSIGN | SXSN | Set external-sign flip-flop equal to sign of top of stack and return the value of the top of stack. |
| TOGGLE | RTFF | Return value of true-false flip-flop. |
| OVERFLOW | ROFF | Return value of overflow flip-flop. |
| STOP | HALT | Halt. |
| RETURN(n) | RETN | Exit and return the value n as a result. |
| ENTIER(n) | NTIA | Round the absolute value of n down to an integer, return the result with proper sign. |

| Intrinsic | Operator | Result |
|---|---|---|
| ONES(n) | CBON | Return number of non-zero bits in n. |
| FIRSTONE(n) | LOG2 | Return number of most significant non-zero bit in n. |
| ABS(n) | BSET | Return absolute value of n. |
| NABS(n) | BRST | Return negative absolute value of n. |
| SCANIN(n) | SCNI | Get the information indicated by n and leave it in top of stack. |
| TIMER(n) | SINT | Set interval timer to value n. |
| MOVESTACK(n) | MVST | Transfer control to stack number n. |
| DECIMAL(n) | SCRF | Convert binary representation of n to decimal representation. |
| BINARY(n) | ICVD | Convert decimal representation of n to binary representation. |
| SCANOUT(n,m) | SCNO | n indicates the type of scanout and m is the value scanned. |
| IIO(a,m) | SCNO | Initiate I/O using unit specified by m and array row a. |
| SET(v,m) | DBST | Set bit number m in variable v. |
| RESET(v,m) | DBRS | Reset bit number m in variable v. |
| READLOCK(v,m) | RDLK | Interchange value of variable v and memory address m. |

| Intrinsic | Operator | Result |
|---|---|---|
| JOIN(n,m) | JOIN | Make a double operand whose first word is n and whose second is m. |
| LISTLOOKUP (n,r,m) | LLLU | Search the linked list starting at word n of the array row r until an entry greater than m is found. |
| MASKSEARCH (n,r,m) | SRCH | Search the array row r for an entry equal to n in all bits not masked by m. |

The following intrinsics are not guaranteed to produce simple inline code.

a. LOCK(m) and BUSY(m) are true iff the entity m was previously LOCKed; UNLOCK(m) is TRUE iff m was previously UNLOCKed. LOCK leaves m LOCKed. UNLOCKed leaves it UNLOCKed, and each may be used as a procedure intrinsic, in which case the value returned by them is ignored.

   BUZZ(m) and BUZZCONTROL(m) cause the current process to suspend operation until the entity m is UNLOCKed by some other process. The variable m is LOCKed and the process allowed to continue.

   It is guaranteed that if more than one process attempts to LOCK or BUZZ an entity previously UNLOCKed, only one process can succeed.

   The variable m must be a variable or a queue designator. If the entity m is a queue designator, the LOCKed algorithm must have been specified for m.

b. WAIT(e) deactivates the current process until the event e is CAUSEd. CAUSE(e) sets the event e to HAPPENED, activates processes waiting on e, and interrupts processes which have interrupt declarations referencing e.

SET(e) and RESET(e) set the event e to HAPPENED and not HAPPENED respectively.

FIX(e) and FREE(e) make e not available and available respectively. FIX(e) returns the value of the previous state of availability.

HAPPENED(e) is TRUE if e has happened and AVAILABLE(e) is TRUE iff e is available.

c. ENABLE(int) and DISABLE(int) cause the interrupt int to be enabled and disabled respectively.

HOLD causes the current process to be deactivated. A process deactivated by a HOLD is reactivated only if an event is caused which corresponds to some interrupt declared by the process.

d. SECONDWORD(ent), where ent is an event or a double precision operand, returns to the second word of ent.

e. STOREITEM(fitm,aitm) passes the actual item aitm to the formal item fitm.

f. NAME(var) returns the address of the variable var.

# INDEX

## METALINGUISTIC VARIABLES

The syntactical definition of each ESPOL metalinguistic variable is found in the pages indicated.

## METALINGUISTIC VARIABLES

tear along dotted line

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: _____ FORM: _____
_____ DATE: _____
_____

CHECK TYPE OF SUGGESTION:

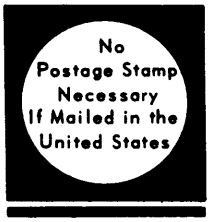
☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME _____ DATE _____
TITLE _____
COMPANY _____
ADDRESS _____
_____

STAPLE

FOLD DOWN SECOND FOLD DOWN


BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
6071 Second Avenue
Detroit, Michigan 48232

attn: Sales Technical Services
Systems Documentation

FOLD UP FIRST FOLD UP


1042744

1-70

Printed in U.S. America