Burroughs

B5500

Information Processing Systems

ESPOL
REFERENCE MANUAL

Burroughs
B 5500
INFORMATION PROCESSING SYSTEM

ESPOL

REFERENCE MANUAL

BUSINESS MACHINES GROUP
SALES TECHNICAL SERVICES
SYSTEMS DOCUMENTATION

Burroughs Corporation
Detroit, Michigan 48232

COPYRIGHT © 1966 BURROUGHS CORPORATION

AA 850936

TABLE OF CONTENTS

SECTION	TITLE	PAGE
	INTRODUCTION.	ix
1	ESPOL CONSTRUCTS VS. EXTENDED ALGOL CONSTRUCTS.	1-1
	General	1-1
	Constructs Common to Extended ALGOL and ESPOL	1-1
	Extended ALGOL Constructs Not Included in ESPOL.	1-1
2	PROGRAM DESCRIPTION	2-1
	Syntax.	2-1
	Semantics	2-1
3	DESCRIPTION OF VARIABLES.	3-1
	Syntax.	3-1
	Semantics	3-2
	Variables	3-2
	Elemental Variables	3-2
	Subscripted Variables	3-2
4	DESCRIPTION OF VARIABLE DECLARATIONS.	4-1
	Syntax.	4-1
	Semantics	4-2
	Simple Variable Declaration	4-2
	OWN	4-2
	Type.	4-2
	Identifier Expressions.	4-3
	Array Declarations.	4-5
	OWN	4-5
	Type.	4-5
	Identifier Expressions.	4-5
	Name Declaration.	4-8
	OWN	4-8
	Identifier Expressions.	4-8
	The Name Variable MEMORY.	4-8

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
5	ARITHMETIC AND BOOLEAN EXPRESSIONS.	5-1
	General	5-1
6	NUMBERS	6-1
	Syntax.	6-1
	Semantics	6-1
7	ESPOL PRIMARIES	7-1
	Syntax.	7-1
	Semantics	7-1
	⟨Octal Representation⟩.	7-1
	(⟨Expression⟩).	7-2
	*⟨Primary⟩.	7-2
	POLISH (⟨Polish String⟩).	7-2
8	WORD MODE SYLLABLES AND OPERATORS	8-1
	General	8-1
	Descriptions of Operations Caused by Polish Codes.	8-6
	Arithmetic Operators.	8-7
	ADD (+) - Add Single Precision. .	8-7
	SUB (-) - Subtract Single Precision	8-7
	MUL (x) - Multiply Single Precision	8-7
	/ - Divide Single Precision . . .	8-7
	DIV (IDV) - Integer Divide. . . .	8-7
	MOD (RDV) - Remainder Divide. . .	8-7
	Logical Operators	8-7
	AND (LND) - Logical AND	8-7
	OR (LOR) - Logical OR	8-7
	EQV - Logical Equivalence	8-7
	NOT (LNG) - Logical Negate. . . .	8-8
	Relational Operators.	8-8

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
8 (cont)	GTR (>) - Test Greater Than . . .	8-8
	GEQ (≥) - Test Greater Than or Equal	8-8
	EQL (=) - Test Equal.	8-8
	LEQ (≤) - Test Less Than or Equal	8-8
	LSS (<) - Test Less Than.	8-9
	NEQ (≠) - Test Not Equal.	8-9
	Branch Operator	8-9
	BRT - Branch Return	8-9
	Store Operators	8-10
	STD (←) - Store Destructive . . .	8-10
	SND (STN) - Store Non-Destructive	8-10
	ISD - Integer Store Destructive .	8-11
	ISN - Integer Store Non-Destructive	8-11
	Bit Operators	8-11
	DIA xx - Dial A	8-11
	DIB xx - Dial B	8-12
	TRB xx (TFR xx) - Transfer Bits .	8-12
	CFE xx (FCE xx) - Compare Field Equal	8-12
	CFL xx (FCL xx) - Compare Field Low	8-13
	RFB - Reset Flag Bit.	8-13
	SFB - Set Flag Bit.	8-13
	TOP - Test for Operand.	8-13
	SSP (RSB) - Set Sign Plus	8-13
	SSN (SSB) - Set Sign Negative . .	8-13
	CSB (CHS) - Change Sign Bit . . .	8-13
	CTC (CCK) - C TO C Transfer . . .	8-14
	CTF (CFX) - C TO F Transfer . . .	8-14

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
8 (cont)	FTC (FCX) - F TO C Transfer . . .	8-14
	FTF (FFX) - F TO F Transfer . . .	8-14
	Subroutine Operators.	8-14
	MKS - Mark Stack.	8-14
	XIT - Exit.	8-14
	RTN (RNO) - Return Normal	8-15
	RTS (RSP) - Return Special. . . .	8-16
	Stack Operators	8-18
	XCH - Exchange.	8-18
	DUP - Duplicate	8-18
	DEL - Delete.	8-18
	Miscellaneous Operators	8-18
	LOD - LOAD.	8-18
	INX - Index	8-19
	COC - Construct Operand Call. . .	8-19
	CDC - Construct Descriptor Call .	8-19
	COM - Communicate	8-19
	PRL - Program Release	8-19
	ZPI - Conditional Halt.	8-20
	STF - Set F	8-20
	STS - Set S	8-20
	RDF - Read F.	8-20
	RDS - Read S.	8-20
	LLL - Link List Lookup.	8-20
	RRR (IPS) - Read Ready Register .	8-21
	TIO - Test I/O.	8-23
	Control State Operators	8-24
	INI (ITI) - Interrogate Interrupt	8-24
	IOR - I/O Release	8-24
	IIO - Initiate I/O.	8-24
	IP1 (INA) - Initiate P1	8-24
	IP2 (INB) - Initiate P2	8-24

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
8 (cont)	HP2 (HLB) - Halt P2	8-25
	RTM (RTR) - Read Timer	8-25
9	LABELS	9-1
	General	9-1
10	IN-LINE CHARACTER MODE STATEMENT	10-1
	Syntax	10-1
	Semantics	10-1
11	PROCEDURES	11-1
	General	11-1
	Procedure Parameters	11-1
	Save Procedures Vs. Non-Save Procedures	11-2
12	SUBROUTINES	12-1
	General	12-1
	Subroutine Declaration	12-1
	Syntax	12-1
	Semantics	12-1
	Subroutine Call Statement and Subroutine Function Designator	12-2
	Syntax	12-2
	Semantics	12-2
	Subroutine Call Statement	12-2
	Subroutine Function Designator	12-3
13	SWITCH DESIGNATORS	13-1
	General	13-1
14	DESIGNATIONAL EXPRESSIONS	14-1
	Syntax	14-1
	Semantics	14-1
15	ESPOL INTRINSICS	15-1
	General	15-1

TABLE OF CONTENTS (cont)

SECTION	TITLE	PAGE
APPENDIX A -	ESPOL STATEMENT EXAMPLES.	A-1
APPENDIX B -	REFERENCE FOR EXAMPLES OF CODE GENERATED BY ESPOL.	B-1
APPENDIX C -	CHARACTERISTICS OF PROGRAMS GENERATED BY ESPOL.	C-1
APPENDIX D -	OPERATING CHARACTERISTICS OF THE ESPOL COMPILER.	D-1

LIST OF TABLES

TABLE	TITLE	PAGE
8-1	Operations Caused by Polish Codes	8-1
8-2	Association Between A-Register Bits and Peripheral Units.	8-22

INTRODUCTION

Executive Systems Problem Oriented Language (ESPOL) was designed to include many features of Extended ALGOL. There are differences between the languages, of course, since the purposes of the languages are different. For example, it is not the purpose of Extended ALGOL programs to deal with problems related to interrupt handling, storage allocation, overlay, etc., but this is the purpose of the DF MCP. However, since the languages are similar, this document describes ESPOL in terms of Extended ALGOL constructs, whenever possible.

Use of this document requires a thorough knowledge of Extended ALGOL. In addition, knowledge of the operational characteristics of the B 5500 is required. The reader is consequently assumed to be completely familiar with at least the information in the publications:

a. Extended ALGOL Reference Manual for the Burroughs B 5500.

b. A Narrative Description of the Burroughs B 5500 Disk File Master Control Program.

Because of the similarity between Extended ALGOL and ESPOL, constructs which are common to both languages are not always defined. (See Extended ALGOL Reference Manual for constructs not defined.) Constructs which are not common to both languages and constructs which have different definitions in ESPOL than in Extended ALGOL will be metalinguistically defined if such definitions are required for clarity. Metalinguistic formulas will be preceded by an underlined number. The numbers have the following meanings:

a. 1 - defined the same for ESPOL and Extended ALGOL.
b. 2 - defined differently for ESPOL than Extended ALGOL.
c. 3 - defined only for ESPOL.

SECTION 1
ESPOL CONSTRUCTS VERSUS EXTENDED ALGOL CONSTRUCTS

GENERAL.

Constructs which are in ESPOL in the same fashion as in Extended ALGOL and Extended ALGOL constructs which ESPOL does not include are noted in this section. ESPOL constructs which are only similar to Extended ALGOL and those which exist only in ESPOL are covered in the following sections.

CONSTRUCTS COMMON TO EXTENDED ALGOL AND ESPOL.

Constructs which are common to both Extended ALGOL and ESPOL are listed below (see Extended ALGOL Reference Manual for details).

a. The use of COMMENT.
b. Strings.
c. Partial word designators.
d. Function designators.
e. Assignment statements.
f. GO TO statements.
g. Dummy statements.
h. Conditional statements.
i. Iterative statements.
j. SWITCH declarations.
k. DEFINE declarations.
l. FORWARD reference declarations.
m. STREAM procedure declarations.
n. STREAM statements.

EXTENDED ALGOL CONSTRUCTS NOT INCLUDED IN ESPOL.

Extended ALGOL constructs which are not included in ESPOL are listed below.

a. Standard functions.
b. Type transfer functions.
c. Nested blocks.
d. I/O statements.
e. FILL statements.

f. FILE declarations.

g. FORMAT declarations.

h. LIST declarations.

i. Diagnostic declarations.

SECTION 2
PROGRAM DESCRIPTION

SYNTAX.

The syntax for ⟨program⟩ is as follows:

```
2  ⟨program⟩ ::= ⟨block⟩ . ⟨space⟩

2  ⟨block⟩ ::= ⟨unlabeled block⟩

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

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

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

2  ⟨declaration⟩ ::= ⟨variable declaration⟩ | ⟨label declaration⟩ |
                     ⟨switch declaration⟩ |
                     ⟨procedure declaration⟩ |
                     ⟨stream procedure declaration⟩ |
                     ⟨subroutine declaration⟩ |
                     ⟨define declaration⟩ |
                     ⟨forward reference declaration⟩

2  ⟨statement⟩ ::= ⟨unconditional statement⟩ |
                   ⟨conditional statement⟩ |
                   ⟨iterative statement⟩ |
                   ⟨in-line character mode statement⟩

2  ⟨unconditional statement⟩ ::= ⟨compound statement⟩ |
                                 ⟨basic statement⟩
```

SEMANTICS.

The general design of an ESPOL program is much the same as that of an Extended ALGOL program. Although ESPOL allows no nested blocks, it utilizes declarations, simple variables, variables with subscripts, assignment statements, iterative statements, labels, etc.

SECTION 3
DESCRIPTION OF VARIABLES

SYNTAX.

The syntax for ⟨variable⟩ is as follows:

2 ⟨variable⟩ ::= ⟨simple variable⟩ | ⟨array variable⟩ |
⟨name variable⟩

2 ⟨simple variable⟩ ::= ⟨elemental variable⟩

3 ⟨array variable⟩ ::= ⟨elemental variable⟩ |
⟨subscripted variable⟩

3 ⟨name variable⟩ ::= ⟨elemental variable⟩ |
⟨subscripted variable⟩

3 ⟨elemental variable⟩ ::= ⟨simple variable identifier⟩ |
⟨array identifier⟩ | ⟨name identifier⟩

3 ⟨simple variable identifier⟩ ::= ⟨variable identifier⟩

2 ⟨array identifier⟩ ::= ⟨variable identifier⟩

3 ⟨name identifier⟩ ::= ⟨variable identifier⟩

1 ⟨variable identifier⟩ ::= ⟨identifier⟩

2 ⟨subscripted variable⟩ ::= ⟨array identifier⟩
[⟨subscript list⟩] | ⟨name identifier⟩
[⟨subscript expression⟩] |
⟨array identifier⟩ [⟨row designator⟩] |

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

1 ⟨subscript expression⟩ ::= ⟨arithmetic expression⟩

2 ⟨row designator⟩ ::= * | ⟨subscript list⟩, *

SEMANTICS.

VARIABLES.

ESPOL provides three classes of variables: simple variables, array variables, and name variables. Basically, the different variables are characterized in the following manner. Simple variables are like simple variables in Extended ALGOL and are used to represent operands. Array variables when subscripted are like subscript variables in Extended ALGOL and are used with operations dealing with data descriptors having non-zero size fields. Array variables with no subscript have no ALGOL counterpart and are used when dealing with data descriptors with non-zero size fields and program descriptors. Name variables which also have no Extended ALGOL counterpart are used when dealing with data descriptors having zero size fields.

ELEMENTAL VARIABLES.

Elemental variables are related to exactly one memory location. In the case of simple variables, that one location is presumed to contain an operand, as in Extended ALGOL. In the case of array variables, the one location is presumed to contain a program descriptor or a data descriptor with a non-zero size field. In the case of name variables, the one location is presumed to contain a data descriptor with a zero size field.

When an elemental variable occurs as a primary, the value represented is the contents of the one memory location related to that variable. The occurrence of an elemental variable in a left-part list denotes that a value is to be stored in the location related to the elemental variable. For examples of codes generated by the ESPOL Compiler to perform the above operations, see appendix A, part A.

SUBSCRIPTED VARIABLES.

Subscripted variables provide a means to reference locations through use of indexed descriptors.

When an array identifier (or name identifier) followed by a bracketed ⟨subscript list⟩ is used as a primary, the value represented is the contents of the location for which the address can be derived through use of the descriptor related to the array identifier (or name

identifier) and the subscript list. In the case of an array variable, the value represented is accessed as an operand; in the case of a name variable, the value is accessed as a descriptor.

When an array identifier followed by a bracketed ⟨row designator⟩ is used as a primary, the value represented is the dope-vector element for which the address can be derived through use of the descriptor related to the array identifier and row designator.

The occurrence of a ⟨subscripted variable⟩ in a left-part list denotes that a value is to be stored in a location for which the address can be derived through use of the descriptor related to the array (or name identifier) and the bracketed subscript list.

For examples of code generated by the ESPOL Compiler to perform the above operations, see appendix A, part B.

SECTION 4
DESCRIPTION OF VARIABLE DECLARATIONS

SYNTAX.

The syntax for ⟨variable declaration⟩ is as follows:

3 ⟨variable declaration⟩ ::= ⟨simple variable declaration⟩ |
⟨array declaration⟩ |
⟨name declaration⟩

3 ⟨simple variable declaration⟩ ::= ⟨local or own type⟩
⟨identifier expression list⟩

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

1 ⟨type⟩ ::= REAL | INTEGER | ALPHA | BOOLEAN

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

3 ⟨identifier expression⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ =
⟨relative address expression⟩

3 ⟨relative address expression⟩ ::= ⟨variable identifier⟩ |
⟨variable identifier⟩
⟨adding operator⟩
⟨unsigned integer⟩ |
⟨rR or rF indicator⟩
⟨unsigned integer⟩

1 ⟨adding operator⟩ ::= + | -

3 ⟨rR or rF indicator⟩ ::= ⟨empty⟩ | + | -

2 ⟨array declaration⟩ ::= ⟨kind⟩ ARRAY ⟨array list⟩ |
⟨kind⟩ ARRAY ⟨fill array⟩ |
⟨kind⟩ ARRAY ⟨array list⟩, ⟨fill array⟩

3 ⟨kind⟩ ::= ⟨empty⟩ | ⟨local or own type⟩

2 ⟨array list⟩ ::= ⟨identifier expression⟩
[⟨dimension information⟩] | ⟨array list⟩,
⟨identifier expression⟩
[⟨dimension information⟩]

3 ⟨dimension information⟩ ::= ⟨save row size⟩ |
⟨dimensions indicator⟩

3 ⟨save row size⟩ ::= ⟨unsigned integer⟩

3 ⟨dimensions indicator⟩ ::= * | ⟨dimensions indicator⟩, *

3 ⟨fill array⟩ ::= ⟨identifier expression⟩ [⟨save row size⟩] ←
⟨value list⟩

1 ⟨value list⟩ ::= ⟨initial value⟩ | ⟨value list⟩, ⟨initial value⟩

1 ⟨initial value⟩ ::= ⟨number⟩ | ⟨string⟩ | OCT ⟨octal number⟩

3 ⟨name declaration⟩ ::= NAME ⟨name list⟩ | OWN NAME ⟨name list⟩ ·
ARRAY NAME ⟨name list⟩ | OWN ARRAY NAME
⟨name list⟩

3 ⟨name list⟩ ::= ⟨identifier expression list⟩

SEMANTICS.

SIMPLE VARIABLE DECLARATION.

OWN. The use of OWN in a declaration causes a variable to be assigned a permanent PRT location in cases where a stack location would normally be assigned. This ability may be desired when declaring variables in procedures.

TYPE. All simple variables in ESPOL have a ⟨type⟩, either REAL, ALPHA, BOOLEAN, or INTEGER, according to how they are declared. However, ESPOL pays less regard to a variable's type than does Extended ALGOL. The only distinction made in ESPOL because of type is when a value is to be stored. In this regard, an integer store operator is used when storing values for variables declared INTEGER, and regular store operators are used when storing values for variables declared BOOLEAN, ALPHA, or REAL. Consequently, other than for

storing options, the only advantage of the various "types" is for documentation.

IDENTIFIER EXPRESSIONS. Identifier expressions allow the programmer to specify the address or relative address of a variable. That is, the address specified by an identifier expression is the address of the location represented by a variable when it appears as an elemental variable.

In appearance, identifier expressions are used in ESPOL declarations as identifiers are used in Extended ALGOL declarations.

An identifier expression is interpreted as follows:

a. If the identifier expression is an identifier alone, the Compiler will reserve a location for the variable in the program's PRT or stack. The location reserved will be the next available PRT location if the variable is declared in the outer block or if the variable is declared as OWN in a procedure, the location reserved will be the next available stack location (the first procedure local variable is assigned the address rF* + 1); a zero will be placed in the location reserved upon procedure entrance.

b. If an identifier expression is of the form ⟨identifier⟩ = ⟨relative address expressions⟩, no location is reserved for the variable. This notation specifies that the declared variable is to be assigned the location determined by the relative address expression. The relative address expressions may be in any of the following forms:

   1) ⟨variable identifier⟩.

   2) ⟨variable identifier⟩ ⟨adding operator⟩ ⟨unsigned integer⟩.

---

* The notation rF stands for the F-register setting and rR the R-register setting.

3) ⟨rR or rF indicator⟩ ⟨unsigned integer⟩.

c. If form 1 is used, the variable being declared will be assigned the same address as that of the variable in the relative address expression.

d. If form 2 is used, the variable will be assigned the address of the variable in the relative address expression, adjusted according to the value of ⟨adding operator⟩ ⟨unsigned integer⟩.

e. If form 3 is used, the address is assigned in the following manner. If ⟨rR or rF indicator⟩ is empty, the address assigned will be the ⟨unsigned integer⟩, relative to rR. Otherwise, the address will be the value of ⟨relative address expression⟩, relative to rF.

The following examples illustrate the use of identifier expressions.

Example 1:

```
REAL A;
```

Comment: If this declaration were in the outer block, a PRT location would be reserved for the variable A. If the declaration were in a procedure, a stack location would be reserved for A.

Example 2:

```
REAL B = A;
```

Comment: This declaration would cause B to reference the same location as the variable A.

Example 3:

```
REAL C = A+3;
```

Comment: This declaration would cause C to reference the memory location for which the address is three greater than the address of the variable A.

Example 4:

```
REAL D = 15;
```

Comment: This declaration would cause D to reference the 15th location of the PRT (i.e., rR+15).

Example 5:

```
REAL E = +15;
```

Comment: This declaration would cause E to reference the stack location at rF + 15 (i.e., the 15th word above the word at which rF is pointing).

Example 6:

```
REAL F = -19;
```

Comment: This declaration would cause F to reference the stack location at rF-19 (i.e., the 19th word below the word at which rF is pointing).

ARRAY DECLARATIONS.

OWN. The use of OWN in an array declaration causes the location for the array descriptor to be assigned a permanent PRT address in cases where a stack location would normally be assigned.

TYPE. Type has the same meaning with respect to subscripted array variables as it does to simple variables. For INTEGER arrays, however, integer store operators are only used when storing values in subscripted array variables. Regular store operators are used when storing values in elemental array variables.

If no type is given for an array, REAL is assumed.

IDENTIFIER EXPRESSIONS. Identifier expressions have the same function in array declarations as they do in simple variable declarations. The identifier expression in an array declaration, of course, decides the address of the location which is represented by the array variable when used in its elemental form. For arrays which will be referenced through use of subscripted variables, the subject location should contain a data descriptor with a non-zero size field.

ESPOL arrays fall into two groups and can be referred to either as standard arrays or save arrays. Standard arrays are those for which the program must provide array descriptors and assign memory. Save arrays are those for which the Compiler provides array descriptors and assigns memory.

Arrays are distinguished by the way in which they are declared. That is, when declaring standard arrays, a dimension indicator (i.e., asterisk) is used to denote the number of dimensions; when a save array is declared, an unsigned integer is used to specify the save row size.

a. Standard Arrays.

1) When a standard array is declared in ESPOL, a memory location is reserved for the array in the PRT or stack, depending upon how and/or where it is declared. Actual placement of a descriptor in the reserved cell must be done programmatically.

2) Standard arrays may be declared to have one or more dimensions. When more than one dimension is specified, the program must, of course, supply all dope vectors (i.e., vectors of descriptors) needed. In fact, all information about a standard array, such as its size and the memory it uses, must be determined programmatically.

b. Save Arrays.

1) When a SAVE array is declared, ESPOL assigns an absolute core area for the array and provides the array descriptor. Because of this action, it is required that the memory location for a SAVE array descriptor be in the PRT.

2) SAVE array may have only <u>one</u> dimension. The size of the array is <u>equal</u> <u>to</u> the save row size designated at declaration time. Except in the case of fill arrays, a SAVE array will be initialized to zeros at run time.

c. Fill Arrays. A fill array is a save array which is declared together with the values to which it is to be initialized. A fill array must occur as the last array to be declared in any specific array declaration. Values occurring in the value list are used at run time to initialize the specified save array. Values from the value list are taken from left to right and assigned, starting at element zero. Zeros are assigned to high-order array elements if the value list is not of sufficient size.

The following examples demonstrate the use of array declarations.

Example 1:

ARRAY A

Comment: This declaration would cause a memory cell to be reserved for the A array. The reserved cell would be in the PRT if this declaration occurred in the outer block, or in the stack if the declaration occurred in a procedure.

Example 2:

ARRAY B

Comment: This declaration specifies that reference to B should cause reference to the memory location at rF + 15, and that B should be referenced as a two-dimension array.

Example 3:

ARRAY C = 4

Comment: This declaration specifies rR + 4 as the address for a two-dimension C array. It also causes a PRT cell to be set up for the 25-element D save array which will be initialized to zeros at run time, and causes a PRT cell to be set up for the fill-type E save array as well as specifying that E is to be initialized to the fill list at run time.

NAME DECLARATION.

A name declaration is much the same as a simple variable declaration in appearance. Name declarations, however, use the words NAME or ARRAY NAME rather than a type. The use of ARRAY NAME as opposed to NAME does not make a functional difference in a name declaration. The word ARRAY is allowed only for documentary purposes and would generally be used if the variable being declared were to be subscripted when used.

OWN. OWN has the same function in a name declaration as in a simple variable declaration.

IDENTIFIER EXPRESSIONS.

Identifier expressions have the same function in name declarations as in simple variable declarations. The only difference is that the location determined by the expression is in this case for a name descriptor.

The following example demonstrates the use of name declarations.

```
NAME A = +4, B = 15, C;
ARRAY NAME AA = +4, BB = 15, CC;
Comment: Reference to A and AA will cause reference to the
         location at rF + 4, B and BB will cause reference to
         rR + 15, and C and CC will each cause reference to
         reserved PRT locations. The word ARRAY in the second
         declaration is strictly for documentary purposes and
         has no other function. As is the case with other
         variable declarations (save array declarations ex-
         cluded), the required contents of the specified loca-
         tions must be provided programmatically.
```

THE NAME VARIABLE "MEMORY."

The identifier MEMORY has special significance in ESPOL in that it is recognized to be a NAME variable which is assigned memory location 200. The DF MCP initializes location 200 to a word containing 101 in the left-most three bits and zeros in all remaining bits. The identifier M has been made synonymous with MEMORY. The identifier M may be declared to have another meaning in a procedure.

SECTION 5
ARITHMETIC AND BOOLEAN EXPRESSIONS

GENERAL.

ESPOL expressions are in most ways much like Extended ALGOL expressions, but there are significant differences. Expressions in Extended ALGOL (excluding designational expressions) are considered to have either an arithmetic value or a logical value. Generally speaking, an ESPOL expression can be considered to have both an arithmetic value and a logical value. The arithmetic value of an expression involves only the right-most bit of the result. If the right-most bit is 1, the expression is TRUE; otherwise it is FALSE. Arithmetic, logical, or relation operators can be used with, or in, any expression because ESPOL recognizes both arithmetic and logical values for expressions. For example, consider the following statements:

a. S1: A ← B + C;
b. S2: A ← A > B;
c. S3: If A + B THEN A ← B ELSE B ← C;
d. S4: A ← NOT (A + B);
e. S5: A ← TRUE;

In S1, A is set to the sum B + C. In S2, A is set to 1 if A is greater than B; otherwise A is set to 0. In S3, the sum A + B is determined and if the right-most bit of the sum is 1, the condition is TRUE; otherwise it is FALSE. In S4, the sum A + B is determined and then negated. (That is, all bits in the sum that are 0 are set to 1 and all bits that are equal to 1 are set to 0; the flag bit is not affected.) In S5, A is set to 1. (FALSE has a value of 0.)

The sequence in which operations are performed is determined by the following rules of precedence.

Each operator has the following precedence associated with it.

a. First - NOT.
b. Second - x / MOD DIV.
c. Third - + - INX.
d. Fourth - < ≤ = ≠ ≥ >.

e. Fifth - AND.

f. Sixth - OR.

g. Seventh - EQV.

When operators have the same orders of precedence, the sequence of operation is determined from left to right.

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

No two operators may be adjacent.

It should be noted that there is neither an exponentiation operator nor an implication operator in ESPOL.

The operator INX is a binary operator, generally used for indexing purposes, and is not an Extended ALGOL operator. It is used as the other arithmetic operators (for example: X INX Y). Using this example, the function of INX is as follows: The low-order fifteen bits of X are added to the low-order fifteen bits of Y. Overflow is suppressed (i.e., the remaining bits in Y are unaffected). The consequent result is a modified value of Y.

SECTION 6
NUMBERS

SYNTAX.

The syntax for ⟨number⟩ is as follows:

2 ⟨number⟩ ::= ⟨decimal number⟩ | ⟨octal representation⟩

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

1 ⟨integer⟩ ::= ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ | - ⟨unsigned integer⟩

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

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

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

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

3 ⟨octal representation⟩ ::= @ ⟨octal number⟩

1 ⟨octal number⟩ ::= ⟨octal digit⟩ | ⟨octal number⟩ ⟨octal digit⟩

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

SEMANTICS.

Numbers in ESPOL are, in general, represented in the same fashion as numbers in Extended ALGOL. Two exceptions are:

a. ESPOL does not provide a "powers of ten notation."
b. ESPOL does provide an octal number notation.

The octal number notation, that is, the octal representation, consists of the symbol @ followed by an octal number containing from one to sixteen octal digits. If the octal number contains fewer than sixteen octal digits, leading zeros are presumed.

SECTION 7
ESPOL PRIMARIES

SYNTAX.
The syntax for ⟨primary⟩ is as follows:

```
2 ⟨primary⟩ ::= ⟨unsigned decimal number⟩ | ⟨octal representation⟩ |
                ⟨variable⟩ | ⟨function designator⟩ |
                ⟨assignment statement⟩ | ⟨concatenate expression⟩ |
                ⟨string⟩ | ⟨logical value⟩ | (⟨expression⟩) |
                * ⟨primary⟩ | [⟨variable⟩] | POLISH
                (⟨Polish string⟩) | ⟨subroutine function designator⟩

3 ⟨Polish string⟩ ::= ⟨Polish component⟩ | ⟨Polish string⟩,
                      ⟨Polish component⟩

3 ⟨Polish component⟩ ::= ⟨Polish code⟩ |
                         {⟨expression⟩ not beginning with x, /,
                         +, or -} | .⟨identifier⟩

3 ⟨subroutine function designator⟩ ::= ⟨subroutine identifier⟩

3 ⟨subroutine identifier⟩ ::= ⟨identifier⟩
```

SEMANTICS.
ESPOL, for the most part, recognizes the same forms of primaries as are recognized by Extended ALGOL. There are exceptions, of course, in that expressions are primaries and ESPOL expressions can differ from expressions in Extended ALGOL. Also, ESPOL has some primaries which do not exist in Extended ALGOL.

ESPOL primaries which do not have Extended ALGOL counterparts are described in the following paragraphs.

⟨OCTAL REPRESENTATION⟩.
⟨octal representation⟩ may be used as a primary and is recognized to have an octal value as described in section 6.

(⟨EXPRESSION⟩).
The ⟨expression⟩ enclosed within parentheses represents an Arithmetic-Boolean type expression as described in section 5.

*⟨PRIMARY⟩.
The * ⟨primary⟩ primary is a special notation which specifies that a load operation is to be executed after the ⟨primary⟩ (immediately following the asterisk) has been placed in the top of the stack. (See appendix B for examples of codes generated for this construct.)

The [⟨variable⟩] primary is a special notation which specifies that the ⟨variable⟩ enclosed within the brackets is to be obtained through use of a descriptor call operation. (See appendix B for examples of codes generated for this construct.)

POLISH (⟨POLISH STRING⟩).
The Polish primary provides the ability to explicitly generate object code, although no code for branching is allowed. ALGOL-type expressions are allowed in a ⟨Polish string⟩ and will generate the same code as they would as primaries outside of the Polish string. The identifier P is synonymous with POLISH unless it is declared by the programmer to have another meaning.

The construct .⟨identifier⟩ indicates that a literal call is to be performed which will place in the top of the stack the relative address of the ⟨identifier⟩'s location.

The word POLISH itself causes no code to be generated. For example, A ← POLISH; would cause the top word in the stack to be stored in A's location if the address of A was represented by a relative address. However, if the address of the ⟨left part⟩ was represented by a descriptor, then that descriptor would be stored in the location addressed by the word in the top of the stack.

Other codes may be generated by using the ⟨Polish code⟩s which are discussed in section 8.

SECTION 8
WORD MODE SYLLABLES AND OPERATORS

GENERAL.

Table 8-1 provides a list of Polish codes together with their operations.

Table 8-1
Operations Caused by Polish Codes

Polish Codes	Operation
ADD or +	Single Precision Add
AND or LND	Logical AND
BRT	Logical AND
CCX or CTC	rB. [33:15] ← rA. [33:15]
CDC	Construct Descriptor Call
CFE xx or FCE xx	Compare Field Equal (xx = no. of bits to compare)
CFL xx or FCL xx	Compare Field Low (xx = no. of bits to compare)
CFX or CTF	rB. [18:15] ← rA. [33:15]
CHS or CSB	Change Sign Bit
COC	Construct Operand Call
COM	Communicate
CSB or CHS	Change Sign Bit
CTC or CCX	rB. [33:15] ← rA. [33:15]

Table 8-1 (cont)
Operations Caused by Polish Codes

Polish Codes	Operation
CTF or CFX	rB. [18:15] ← rA. [33:15]
DEL	Delete
DIA xx	Dial rA (xx = bit to be dialed)
DIB xx	Dial rB (xx = bit to be dialed)
DIV or IDV	Integer Divide
DUP	Duplicate
EQL or =	Test rB = rA
EQV	Logical Equivalence
FCE xx or CFE xx	Compare Field Equal (xx = no. of bits to compare)
FCL xx or CFL xx	Compare Field Low (xx = no. of bits to compare)
FCX or FTC	rB. [33:15] ← rA. [18:15]
FFX or FTF	rB. [18:15] ← rA. [18:15]
FTC or FCX	rB. [33:15] ← rA. [18:15]
FTF or FFX	rB. [18:15] ← rA. [18:15]
GEQ or ≥	Test rB ≥ rA
GTR or >	Test rB > rA
HLB or HP2	Halt P2

Table 8-1 (cont)
Operations Caused by Polish Codes

Polish Codes	Operation
HP2 or HLB	Halt P2
IDV or DIV	Integer Divide
IIO	Initiate I/O
INA or IP1	Initiate P1
INB or IP2	Initiate P2
INI or ITI	Interrogate Interrupt
INX	Index
IOR	I/O Release
IP1 or INA	Initiate P1
IP2 or INB	Initiate P2
IPS or RRR	Interrogate Peripheral Status
ISD	Integer Store Destructive
ISN	Integer Store Non-Destructive
ITI or INI	Interrogate Interrupt
LEQ or $\leq$	Test rB $\leq$ rA Test
LLL	Link List Lookup
LND or AND	Logical AND
LNG or NOT	Logical NOT

Table 8-1 (cont)

Operations Caused by Polish Codes

Polish Codes	Operation
LOD	Load
LOR or OR	Logical OR
LSS or <	Test rB < rA Test
MOD or RDV	Remainder Divide
MKS	Mark Stack
MUL or X	Single Precision Multiply
NEQ or ≠	Test rB ≠ rA
NOP	No-Op
NOT or LNG	Logical NOT
OR or LOR	Logical OR
PRL	Program Release
RDF	Read rF (rB. [18:15] ← rF)
RDS	Read rS (rB. [33:15] ← rS)
RDV or MOD	Remainder Divide
RFB	Reset Flag Bit
RNO or RTN	Return Normal
RRR or IPS	Interrogate Peripheral Status
RSB or SSP	Reset Sign Bit (positive)

Table 8-1 (cont)
Operations Caused by Polish Codes

Polish Codes	Operation
RSP or RTS	Return Special
RTM or RTR	Read Timer
RTS or RSP	Return Special
SFB	Set Flag Bit
SND or STN	Store Non-Destructive
SSB or SSN	Set Sign Bit (negative)
SSN or SSB	Set Sign Bit (negative)
SSP or RSB	Reset Sign Bit (positive)
STD or ←	Store Destructive
STF	Set rF (rF ← rB. [18:15])
STN or SND	Store Non-Destructive
STS	Set rS (rS ← rB. [33:15])
SUB or -	Single Precision Subtract
TFR xx or TRB xx	Transfer Bits (xx = number of bits)
TIO	Test I/O Channel
TOP	Test for Operand
TRB xx or TFR xx	Transfer Bits (xx = number of bits)
XCH	Exchange

Table 8-1 (cont)

Operations Caused by Polish Codes

Polish Codes	Operation
XIT	Exit
ZIP or ZP1	Conditional Halt
ZP1 or ZIP	Conditional Halt
+ or ADD	Single Precision Add
= or EQL	Test rB = rA
$\geq$ or GEQ	Test rB $\geq$ rA
> or GTR	Test rB > rA
$\leq$ or LEQ	Test rB $\leq$ rA
< or LSS	Test rB < rA
x or MUL	Single Precision Multiply
$\neq$ or NEQ	Test rB $\neq$ rA
$\leftarrow$ or STD	Store Destructive
- or SUB	Single Precision Subtract
	Real Divide (Single Precision)

DESCRIPTIONS OF OPERATIONS CAUSED BY POLISH CODES.

The following paragraphs describe the operations listed in table 8-1 which are caused by their corresponding Polish codes.

ARITHMETIC OPERATORS.

ADD (+) - ADD SINGLE PRECISION. The operands in rA and rB are added algebraically and the sum is left in rB.

SUB (-) - SUBTRACT SINGLE PRECISION. The operand in rA is algebraically subtracted from the operand in rB and the difference is left in rB.

MUL (x) - MULTIPY SINGLE PRECISION. The operands in rA and rB are algebraically multiplied and the result is left in rB.

/ - DIVIDE SINGLE PRECISION. The operand in rB is algebraically divided by the operand in rA and the quotient is left in rB.

DIV (IDV) - INTEGER DIVIDE. The operand in rB is algebraically divided by the operand in rA and the integer part of the quotient is left in rB.

MOD (RDV) - REMAINDER DIVIDE. The operand in rB is algebraically divided by the operand in rA to develop an integer quotient. The remainder after the division is left in rB.

LOGICAL OPERATORS.

AND (LND) - LOGICAL AND. Corresponding bits (excluding flag bits) of the words in rA and rB are compared. If corresponding bits are both 1, the bit in rA is set to 1. If corresponding bits are not both 1, the bit in rA is set to 0. After all bits are tested, the flag bit of the word in rB is transferred to the flag bit position in rA, and rB is marked empty.

OR (LOR) - LOGICAL OR. Corresponding bits (excluding flag bits) of the words in rA and rB are compared. If either of the corresponding bits is 1, the bit in rA is set to 1. If the corresponding bits are both 0, the bit in rA is set to 0. After all bits are tested, the flag bit of the word in rB is transferred to the flag bit position in rA, and rB is marked empty.

EQV - LOGICAL EQUIVALENCE. Corresponding bits (excluding flag bits) of the words in rA and rB are compared. If corresponding bits are

both 1 or both 0, the bit in rB is set to 1. If corresponding bits in rA and rB are different in value, the bit in rB is set to 0. After all bits are tested, rA is marked empty. The flag bit in rB is left unaltered.

NOT (LNG) - LOGICAL NEGATE. Each bit in rA (excluding the flag bit) is complemented; that is, all bits equal to 0 are set to 1 and all bits equal to 1 are set to 0. The flag bit is left unaltered.

RELATIONAL OPERATORS.

GTR (>) - TEST GREATER THAN. The operand in rB is algebraically compared with the operand in rA. If the value of the operand in rB is greater than the value of the operand in rA, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0. If the value of the operand in rB is less than or equal to the value of the operand in rA, all bits in rB are set to 0. In either case, rA is marked empty.

GEQ ($\geq$) - TEST GREATER THAN OR EQUAL. The operand in rB is algebraically compared with the operand in rA. If the value of the operand in rB is greater than or equal to the value of the operand in rA, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0. If the value of the operand in rB is less than the value of the operand in rA, all bits in rB are set to 0. In either case, rA is marked empty.

EQL (=) - TEST EQUAL. The operand in rB is algebraically compared with the operand in rA. If the value of the operands in rB and rA are equal, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0; otherwise, all bits in rB are set to 0. In either case, rA is marked empty.

LEQ ($\leq$) - TEST LESS THAN OR EQUAL. The operand in rB is algebraically compared with the operand in rA. If the value of the operand in rB is less than or equal to the value of the operand in rA, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0. If the value of the operand in rB is greater than the value of the operand in rA, all bits in rB are set to 0. In either case, rA is marked empty.

LSS (<) - TEST LESS THAN. The operand in rA is algebraically compared with the operand in rB. If the value of the operand in rB is less than the value of the operand in rA, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0. If the value of the operand in rB is greater than or equal to the value of the operand in rA, all bits in rB are set to 0. In either case, rA is marked empty.

NEQ (≠) - TEST NOT EQUAL. The operand in rB is algebraically compared with the operand in rA. If the value of the operands in rA and rB are not equal, bit [47:1] in rB is set to 1 and all other bits in rB are set to 0; otherwise, all bits in rB are set to 0. In either case, rA is marked empty.

BRANCH OPERATOR.

BRT - BRANCH RETURN. The presence bit of the word in rA is tested. If the presence bit is 0, the presence bit interrupt is set and the operation is terminated. If the presence bit is 1, the following operations are performed.

a. The S register is set to the value of the field at [18:15] in rA.

b. The C register is set to the value of the field at [33:15] in rA.

c. The L register is set to 0.

d. The Mark Stack Control Word addressed by rS is read from memory.

e. The R register and F register are set to the contents of their respective fields of the Mark Stack Control Word. Also, the mark stack flip-flops and program level flip-flop are set to the value of their respective positions of the Mark Stack Control Word.

f. The S register is decreased by 1.

g. The registers rA and rB are marked empty.

STORE OPERATORS.

STD (←) - STORE DESTRUCTIVE. If the flag bit and the presence bit of the word in rA are both 1, the contents of rB are stored in the memory cell addressed by the 15 low-order bits of rA. The A and B registers are marked empty.

If the flag bit of the word in rA is 1 and the presence bit is 0, the presence bit interrupt is set and the operation is terminated.

If the flag bit of the word in rA is 0, the 10 low-order bits of the word in rA are used as a relative address, except that no addressing relative to the C register takes place. If the syllable calls for addressing relative to the C register, the absolute address is constructed relative to the R register instead. The contents of rB are stored in the memory cell addressed after appropriate indexing of the relative address. The A and B registers are set to empty.

If the VARF flip-flop is set, the processor is set to subprogram level after the relative address operation and VARF is reset.

SND (STN) - STORE NON-DESTRUCTIVE. If the flag bit and the presence bit of the word in rA are both one, the contents of rB are stored in the memory cell addressed by the 15 low-order bits of rA. The A register is set to empty.

If the flag bit of the word in rA is 1 and the presence bit is 0, the presence bit interrupt is set and the operation is terminated.

If the flag bit of the word in rA is 0, the 10 low-order bits of the word in rA are used as relative address, except that no addressing relative to the C register takes place. If the syllable calls for addressing relative to the C register, the absolute address is constructed relative to the R register instead. The contents of rB are stored in the memory cell addressed after appropriate indexing of the relative address. The A register is set to empty.

If the VARF flip-flop is set, the processor is set to subprogram level after the relative address operation, and VARF is reset.

ISD - INTEGER STORE DESTRUCTIVE. If the flag bit and the presence bit of the word in rA are both 1 or if the flag bit of the word in rA is 0, the word in rB is made an integer.

If an integer overflow occurs, the integer overflow interrupt is set and the operation is terminated. If integer overflow does not occur, a store operation is performed.

If the flag bit of the word in rA is 1 and the presence bit is 0, the presence bit interrupt is set and the operation is terminated.

If the VARF flip-flop is set, the processor is set to subprogram level after the relative address operation, and VARF is reset.

ISN - INTEGER STORE NON-DESTRUCTIVE. If the flag bit and the presence bit of the word in rA are both 1 or if the flag bit of the word in rA is 0, the word in rB is made an integer.

If an integer overflow occurs, the integer overflow interrupt is set and the operation is terminated. If integer overflow does not occur, a store operation for the Store Non-Destructive operator is performed.

If the flag bit of the word in rA is 1 and the presence bit of the word in rA is 0, the presence bit is set in the interrupt register, and the operation is terminated.

If the VARF flip-flop is set, the processor is set to subprogram level after the relative address operation, and VARF is reset.

BIT OPERATORS.

DIA xx - DIAL A. If the six high-order bits of the operator (i.e., the bits equal to the binary representation of xx) are not 0, the three most significant bits of the operator are placed in the G register and the three next most significant bits of the operator are placed in the H register. If all of the six high-order bits are zero, no action takes place. If the H register is set to 110 or 111, the operation of subsequent operators using this register is not specified.

DIB xx - DIAL B. If the six high-order bits of the operator (i.e., the bits equal to the binary representation of xx) are not 0, the three most significant bits of the operator are placed in the K register and the three next most significant bits of the operator are placed in the V register. If all of the six high-order bits of the operator are 0, a set variant operator takes place. If the V register is set to 110 or 111, the operation of subsequent operators using this register is not specified.

TRB xx (TFR xx) - TRANSFER BITS. A field in rA, starting at the bit position addressed by the G and H registers, replaces a corresponding length field in rB, starting at the bit position addressed by the K and V registers, and proceeding towards the low-order bit positions.

The length of the field transferred is specified by the six high-order bits of the operator (i.e., the bits equal to the binary representation of xx). The transfer of bits is terminated by the transfer of the specified number of bits or when either the A or B register has been exhausted.

The contents of the G, H, K, and V registers after the operation are the same as prior to the operation. The A register is set to empty.

CFE xx (FCE xx) - COMPARE FIELD EQUAL. A field in rA, starting at the bit position addressed by the G and H registers, is compared with a corresponding length field in rB, starting at the bit position addressed by the K and V registers, and proceeding towards the low-order bit positions.

The length of the fields in the registers is specified by the six high-order bits of the operator (i.e., the bits equal to the binary representation of xx). The comparison is terminated by the comparison of the number of bits specified or by the comparison of the low-order bit of either register.

If all of the corresponding bits of the fields compared are equal, the low-order bit of rA is set to 1 and all other bit positions of

rA are set to 0. If any of the corresponding bit positions of the fields compared are not equal, all bit positions of rA are set to 0. The contents of the B, G, H, K, and V registers after the operation are the same as prior to the operation.

CFL xx (FCL xx) - COMPARE FIELD LOW. A field in rA, starting at the bit position addressed by the G and H registers, is compared with a field in rB, starting at the bit position addressed by the K and V registers, and proceeding towards the low-order bit positions.

The length of the fields in the registers is specified by the six high-order bits of the operator (i.e., the bits equal to the binary representation of xx). The comparison is terminated by the comparison of the number of bits specified or by the comparison of the low-order bit position of either register.

The magnitude of the field compared in rB is less than the magnitude of the field compared in rA, the low-order bit of rA is set to 1 and all other bit positions are set to 0; otherwise, all bit positions of rA are set to 0. The contents of the B, G, H, K, and V registers after the operation are the same as prior to the operation.

RFB - RESET FLAG BIT. The flag bit of the word in rA is set to 0.

SFB - SET FLAG BIT. The flag bit of the word in rA is set to 1.

TOP - TEST FOR OPERAND. If rA is full, a stack push-down occurs. The flag bit of the word in rB is tested. If the flag bit is 0, bit [47:1] in rA is set to 1 and all other bits in rA are set to 0; otherwise, all bits in rA are set to 0. In either case, the word in rB is left unaltered and rA is marked full.

SSP (RSB) - SET SIGN PLUS. The sign bit of the word in rA is set to 0.

SSN (SSB) - SET SIGN NEGATIVE. The sign bit of the word in rA is set to 1.

CSB (CHS) - CHANGE SIGN BIT. The sign bit of the word in rA is complemented.

CTC (CCK) - C TO C TRANSFER. The field at [33:15] in rA is transferred to the field at [33:15] in rB. The remainder of rB is unaltered and rA is marked empty.

CTF (CFX) - C TO F TRANSFER. The field at [33:15] in rA is transferred to the field at [18:15] in rB. The remainder of rB is unaltered and rA is marked empty.

FTC (FCX) - F TO C TRANSFER. The field at [18:15] in rA is transferred to the field at [33:15] in rB. The remainder of rB is unaltered and rA is marked empty.

FTF (FFX) - F TO F TRANSFER. The field at [18:15] of rA is transferred to the field at [18:15] of rB. The remainder of rB is unaltered and rA is marked empty.

SUBROUTINE OPERATORS.

MKS - MARK STACK. The contents, if any, of rA and rB are pushed into the stack in memory. A Mark Stack Control Word is constructed and stored in the top of the stack in memory. The F register is set to the address of the cell in which the Mark Stack Control Word has been stored. If the mark stack flip-flop is 0 and the processor is in the subprogram level, the Mark Stack Control Word is also stored in the cell addressed by the contents of the R register plus seven. The mark stack flip-flop is set to 1.

XIT - EXIT. Registers A and B are marked empty. The word addressed by the F register, the Return Control Word, is placed in the B register.

If the flag bit of the word in rB is 1, the operation is continued. If the flag bit is 0 and the processor is in the normal state, the flag bit interrupt is set, and the operator is exited with the Return Control Word left at the top of the stack. If the flag bit is 0 and the processor is in the control state, the operator is terminated but the interrupt is not set.

The C, L, G, H, K, and V registers are set to the contents of their respective fields of the Return Control Word in the B register. The

S register is set to the contents of the F register field of the Return Control Word in the B register.

The word now addressed by the S register, the Mark Stack Control Word, is read from memory into the B register. The R and F registers are set to the contents of their respective fields of the Mark Stack Control Word. The mark stack flip-flop and the program level flip-flop are set to the contents of their respective positions of the Mark Stack Control Word. The S register is decreased by 1 and the A and B registers are set to empty.

The mark stack bit of the word in rB is examined. If this bit is 0, the operation is completed. If the mark stack bit is 1, the program level bit is examined. If the program level bit is 0, indicating program level, the operation is completed.

If the program level bit is 1, indicating subprogram level, the word addressed by the F register field of the Mark Stack Control Word, the previous Mark Stack Control Word, is placed in rB. The mark stack bit is examined. If the mark stack bit is 1, the process of reading the previous Mark Stack Control Word and examining its mark stack bit is repeated until a Mark Stack Control Word with the mark stack bit set to 0 is placed in rB. The contents of rB is stored in the cell addressed by the contents of the R register plus seven. The operation is completed.

RTN (RNO) - RETURN NORMAL. If rA is empty, a word is placed in the A register by stack adjustment and the A register is set to full. If both rA and rB are full, the B register is set to empty.

If the flag bit of the word in rA is 1 and the presence bit 0, the presence bit interrupt is set and the operation is immediately terminated.

The word addressed by the F register, the Return Control Word, is placed in the B register.

If the flag bit of the word in rB is 1, the operation is continued. If the flag bit is 0 and the processor is in the normal state, the

flag bit interrupt is set and the operator is exited, with rA and rB marked full. If the flag bit is 0 and the processor is in the control state, the operator is terminated but the interrupt is not set.

The C, L, G, H, K, and V registers are set to the contents of their respective fields of the Return Control Word in rB. The S register is set to the contents of the F register field of the Return Control Words in rB.

The word addressed by the S register, the Mark Stack Control Word, is read from memory. The R and F registers are set to the contents of their respective fields of the Mark Stack Control Word. The mark stack control flip-flop and the program level flip-flop are set to the word. The S register is decreased by 1.

The mark stack bit of the word in rB is examined. If this bit is 0, the operation is completed.

If the mark stack bit is 1, the program level bit is examined. If the program level bit is 0, indicating program level, the operation is completed.

If the program level bit is 1, indicating subprogram level, the word addressed by the F register field of the Mark Stack Control Word, the previous Mark Stack Control Word, is placed in rB. The contents of rB are stored in the cell addressed by the contents of the R register plus seven. The operation is completed.

The subsequent action of the return operation is similar to that of the operand or description call syllable. If the syllable indication in the Return Control Word indicates an operand call syllable, the subsequent action is as for an operand call. If the syllable indication in the Return Control Word indicates a descriptor call syllable, the subsequent action performed is as for a descriptor call.

RTS (RSP) - RETURN SPECIAL. If rA is empty, a word is placed in rA by stack adjustment and rA is set to full. If both rA and rB are full, rB is set to empty. If the flag bit of the word in rA is 1 and

the presence bit is 0, the presence bit interrupt is set and the operation is immediately terminated.

The word addressed by the S register, the Return Control Word, is placed in rB.

If the flag bit of the word in rB is 1, the operation is continued. If the flag bit is 0 and the processor is in the normal state, the flag bit interrupt is set and the operator exited, with rA and rB marked full. If the flag bit is 0 and the processor is in the control state, the operator is terminated but the interrupt is not set.

The C, L, G, H, K, and V registers are set to the contents of their respective fields of the Return Control Word in the B register. The S register is set to the contents of the F register field of the Return Control Word in rB.

The word addressed by the S register, the Mark Stack Control Word, is read from memory. The R and F registers are set to the contents of their respective fields of the Mark Stack Control Word. The mark stack flip-flop and the program level flip-flop are set to the contents of their respective positions of the Mark Stack Control Word. The S register is decreased by 1.

The mark stack bit of the word in rB is examined. If this bit is 0, the operation is completed. If the mark stack bit is 1, the program level bit is examined. If the program level bit is 0, indicating program level, the operation is completed.

If the program level bit is 1, indicating subprogram level, the word addressed by the F register field of the Mark Stack Control Word, the previous Mark Stack Control Word, is placed in rB. The mark stack bit is examined. If the mark stack bit is 1, the process of reading the previous Mark Stack Control Word with the mark stack bit set to 0 is placed in rB. The contents of rB are stored in the cell addressed by the contents of the R register plus seven. The operation is completed.

The subsequent action of the return operation is similar to that of the operand or descriptor call syllable. If the syllable indication in the Return Control Word indicates an operand call syllable, the subsequent action performed is as for an operand call. If the syllable indicator in the Return Control Word indicates a descriptor call syllable, the subsequent action performed is as for a descriptor call.

STACK OPERATORS.
XCH - EXCHANGE. The contents of rA and rB are exchanged one for the other.

DUP - DUPLICATE. Stack adjustment occurs, if necessary, so that one register (rA or rB) is full and the other is empty. Then the contents of the full register are copied in the empty register and both registers are left marked full.

DEL - DELETE. The top word in the stack is deleted in one of the following ways, depending upon existing conditions.

a. If rA is full, it is marked empty.
b. If rA is empty and rB is full, rB is marked empty.
c. If rA and rB are both empty, rS is decremented by 1.

MISCELLANEOUS OPERATORS.
LOD - LOAD. If the flag bit and the presence bit of the word in rA are both 1, the word in rA is replaced by the contents of the cell addressed by the 15 low-order bits of rA.

If the flag bit of the word in rA is 0, the 10 low-order bits of the word in rA are used as a relative address. The contents of rA are replaced by the contents of the memory cell addressed after appropriate indexing of the relative address.

If the flag bit of the word in rA is 1 and the presence bit is 0, the presence bit interrupt is set and the operation is terminated.

If the VARF flip-flop is set, the processor is set to subprogram level after the relative address operation, and VARF is reset.

INX - INDEX. The 15 low-order bits of the word in rB are arithmetically added to the 15 low-order bits of the word in rA. The remainder of the word in rA is left unchanged, overflow is lost, and the B register is marked empty.

COC - CONSTRUCT OPERAND CALL. The contents of rA and rB are exchanged one for the other. The flag bit of the word in rA is set to 1. The subsequent action of this operator is identical to that of an operand call syllable after the operand call syllable has caused a word to be read from memory.

CDC - CONSTRUCT DESCRIPTOR CALL. The contents of rA and rB are exchanged one for the other. The flag bit of the word in rA is set to 1. The subsequent action of this operator is identical to that of a descriptor call syllable after the descriptor call syllable has caused a word to be read from memory.

COM - COMMUNICATE. The word at the top of the stack is stored in the cell addressed by the contents of rR plus nine. The word is deleted from the stack and the communicate interrupt is set.

The operator is a no-op in the control state.

PRL - PROGRAM RELEASE. If the flag bit and the presence bit of the word in rA are both 1, the contents of the cell addressed by the 15 low-order bits of the word in rA are placed in rA. If the processor is in the control state, the presence bit of the word obtained from memory is set to 0, and the word is stored back into the cell from which it was obtained.

If the flag bit of the word in rA is 0, the 10 low-order bits of the word in rA are used as a relative, except that no addressing relative to the C register takes place. If the syllable calls for addressing relative to the C register, the absolute address is constructed relative to the R register instead. The contents of the cell addressed after appropriate indexing of the relative address are placed in rA. If the processor is in the control state, the presence bit of the word obtained from memory is set to 0 and the word is stored back into the cell from which it was obtained.

If the processor is in normal state, the continuity bit of the word obtained from memory is inspected. If the continuity bit is 1, the continuity bit interrupt is set. If the continuity bit is 0, the program release interrupt is set and the A register is set to empty.

If the processor is in normal state, the address just used is stored in the cell addressed by the contents of rR plus nine. The address is stored in the 15 low-order bits of the word and all other bits are set to 0.

ZPI - CONDITIONAL HALT.
If the OPERATOR switch on the maintenance panel is in the STOP position, the processor is halted by stopping the processor clock; otherwise, this operator is a no-op.

STF - SET F. The A register is set to the STF code. Then rF is set to the value of the field at [18:15] of rB, and rA and rB are marked empty. The processor is set to subprogram level if it is not already so set.

STS - SET S. The A register is set to the STS code. Then rS is set to the value of the field at [33:15] of rB, and rA and rB are marked empty.

RDF - READ F. The A register is set to the RDF code. Then the field at [18:15] of the rB is set to the value of rF, and rA is marked empty.

RDS - READ S. The A register is set to the RDS code. Then the field at [33:15] of rB is set to the value of rS, and rA is marked empty.

LLL - LINK LIST LOOKUP. This operator causes a scan of a linked list of indefinite length and tests a field in each link word against a corresponding test field in rA.

When the LLL operator is executed, the following conditions are required for rA and rB.

The word in rA must have the following format.

Field	Function	Contents
[0:9]	None	Irrelevant
[9:24]	Test Field	Test Value
[33:15]	None	Zeros

The word in rB must have the following format.

Field	Function	Contents
[0:33]	None	Irrelevant
[33:15]	Initial Link Address	Core Address

The complete test field, or any portion of the more significant end of the test field, in rA can be used. Bits on the less significant end are effectively eliminated from the test field by setting them to 0 in the test word in the top of the stack.

The word addressed by the initial link address is read from memory into rB. The test field of the word in rB is compared with the field in the corresponding position in rA.

If the field in rB is greater than or equal to the field in rA, the address that was used to access the link is placed in rA as a present Data Descriptor, with the remainder of the word set to 0. The list word is left in rB and the operator is exited.

If the field in rB is less than the field in rA, the link address in rB is used to access the next word from memory. The process continues indefinitely until a link word meeting the test condition is found.

RRR (IPS) - READ READY REGISTER. This operator places in the top of the stack a word representing the current Ready status of the peripheral equipment. One bit in the word is associated with each peripheral unit. This bit is set to 1 if the associated unit is Ready, or to 0 if the associated unit is Not Ready.

The stack is adjusted so that rA is empty. Then the low-order bits of rA are set to reflect the current status of the peripheral units, and the remaining bits are set to 0. Finally, rA is marked full and the operation is terminated.

Magnetic tape transport units are indicated as READY only when the tape is stationary and they are otherwise ready. They are indicated as NOT READY if the tape is still indexing to a stop following an operation.

Table 8-2 shows the association between the bits in rA, numbered from from right to left, and the peripheral units.

Table 8-2
Association Between A-Register Bits and Peripheral Units

A-Register Bit Position	Unit Des.	Peripheral Unit
1	1	(MTA) Magnetic Tape
2	3	(MTB) Magnetic Tape
3	5	(MTC) Magnetic Tape
4	7	(MTD) Magnetic Tape
5	9	(MTE) Magnetic Tape
6	11	(MTF) Magnetic Tape
7	13	(MTH) Magnetic Tape
8	15	(MTJ) Magnetic Tape
9	17	(MTK) Magnetic Tape
10	19	(MTL) Magnetic Tape
11	21	(MTM) Magnetic Tape
12	23	(MTN) Magnetic Tape
13	25	(MTP) Magnetic Tape
14	27	(MTR) Magnetic Tape
15	29	(MTS) Magnetic Tape
16	31	(MTT) Magnetic Tape

Table 8-2 (cont)

Association Between A-Register and Peripheral Units

A-Register Bit Position	Unit Des.	Peripheral Unit
17	4	(DRA) Drum 1
18	8	(DRB) Drum 2
19	6	(DKA) Disk File Control 1
20	12	(DKB) Disk File Control 2
21	22	(LPA) Printer 1
22	26	(LPB) Printer 2
23	10	(CPA) Card Punch
24	10	(CRA) Card Reader 1
25	14	(CRB) Card Reader 2
26	30	(SPO) SPO-Keyboard
27	18	(PPA) Paper Tape Punch 1
28	18	(PRA) Paper Tape Reader 1
29	20	(PRB) Paper Tape Reader 2
30	20	(PPB) Paper Tape Punch 2
31	16	(DCA) Data Communication Control

TIO - TEST I/O. This operator interrogates the I/O channels to determine which channel is currently in-line to be assigned next, that is, which is the lowest-numbered currently available input/output control unit. If necessary, pushdown occurs and a literal is placed in rA. The literal indicates the next assigned channel in the following way.

Literal	Channel
0	All channels are busy.
1	Channel 1 is due for assignment.
2	Channel 2 is due for assignment.
3	Channel 3 is due for assignment.
4	Channel 4 is due for assignment.

CONTROL STATE OPERATORS.

INI (ITI) - INTERROGATE INTERRUPT. If any interrupt bit is set, the C register is loaded with the 6-bit address which corresponds to the highest priority interrupt bit that is set. The interrupt bit creating this address is reset. The L register is cleared and the S register is set to 64.

If no interrupt bit is set, control continues in sequence.

IOR - I/O RELEASE. When this operator is executed, rA is assumed to contain a descriptor or a relative address. If a relative address is used, addressing is performed. If a descriptor with a 0 presence bit is used, the syllable exited without performing its described operation and the presence bit interrupt is not set.

Execution of the operator causes the word addressed by rA to be placed in rA. Then the presence bit of the word is set to 0 and returned to the cell from which it was obtained. Register A is marked empty.

IIO - INITIATE I/O. The word in rA is stored in location 8 and rA is set to empty. An initiate I/O signal is sent to central control for selection of an I/O channel. The processor proceeds to the next syllable.

IP1 (INA) - INITIATE P1. When this operator is executed, an initiate control word is required in rA. Execution of the operator causes rS to be set to the 15 low-order bits in rA. Bit [32:1] is transferred to the mode flip-flop. Then the word addressed by rS is read and remaining registers are restored using the information in the interrupt words.

IP2 (INB) - INITIATE P2. The initiate control word in rA is stored in memory location 8 and rA is set to empty. An initiate P2 signal is sent to the central control unit and processor 1 proceeds to the next syllable.

The central control unit sends a control signal to processor 2. Under control of this signal, the initiate control word is transferred from memory location 8 to rA in processor 2. Processor 2 then performs the initiate operations.

If processor 2 is not idle or not available, the P2 busy interrupt is set.

HP2 (HLB) - HALT P2. This operator causes processor 2 to store its registers just as if a P2 interrupt had occurred. If processor 2 is busy, the halt operator in processor 1 is held up. The operator in processor 1 is completed after all appropriate processor 2 registers are stored for interrupt. Processor 2 is left idle.

If processor 2 is not ready or is absent, the halt P2 operation is immediately terminated.

RTM (RTR) - READ TIMER. The 6-bit timer setting, together with the time internal interrupt setting as the 7th (most significant) bit, is placed in rA as an integer.

SECTION 9
LABELS

GENERAL.

ESPOL requires, as does Extended ALGOL, that labels be declared in a label declaration, with exception of labels used in in-line character mode statements. In ESPOL, however, labels have applications which are not provided, nor needed, in Extended ALGOL. Depending upon the application intended, ESPOL labels may appear in one of the following five forms.

a. ⟨label⟩ : ⟨statement⟩
b. ⟨label⟩ :: ⟨statement⟩
c. ⟨label⟩ : ⟨number⟩ : ⟨statement⟩
d. ⟨label⟩ : * : ⟨statement⟩
e. ⟨label⟩ ::: ⟨value list⟩

For general use in labeling statements throughout a program, forms a and b are used; forms c, d, and e have special functions. All labels, except those appearing as in form e, can be referenced for the purpose of transferring control.

Form a, the single colon label, is used only when it is not necessary that a label reference the beginning of a word; this is the case only when the labeled statement will receive control through means of a syllable branch. (Syllable branches may span up to 1023 syllables.)

Form b, the double colon label, is used when it is necessary that a label reference the beginning of a word; this is the case when the labeled statement may receive control through means other than a syllable branch. When necessary, no-ops are generated in the code string at compile time to adjust so that a double colon label will reference the beginning of a word.

Form c, the address label, is used when it is necessary that a label reference a specified word address within a program segment. The ⟨number⟩, between the colons in this form of label, specifies the relative address within a segment which the label references. The

⟨number⟩ in any given address label may never specify an address less than that specified in a previous address label in the same segment. This label form was implemented primarily for the purpose of labeling B 5500 interrupt locations; in the outer block of a program, the ⟨number⟩ between the colons specifies an absolute core address.

Form d, the asterisk label, is required by the ESPOL Compiler to specify the first executable statement in a program, following the code for interrupt locations.

Form e, the triple colon label, is in no way intended to be used for the purpose of transferring control. This form of the label was implemented to provide a facility for C-relative constants (i.e., constants that appear in a code segment and are referenced relative to rC). A triple colon label may be used only as an ⟨expression⟩ in a ⟨Polish string⟩; when so used, it will cause reference to the location marked by the label. It is the responsibility of the programmer to ensure that the area in a code segment occupied by constants is not entered for execution.

SECTION 10
IN-LINE CHARACTER MODE STATEMENT

SYNTAX.

The syntax for ⟨in-line character mode statement⟩ is as follows:

3 ⟨in-line character mode statement⟩ ::= ⟨stream parameter statement⟩ BEGIN ⟨stream statement⟩ END

3 ⟨stream parameter statement⟩ ::= STREAM (⟨stream statement parameter list⟩)

3 ⟨stream statement parameter list⟩ ::= ⟨-MKS parameters⟩: | ⟨+MKS parameters⟩ | ⟨-MKS parameters⟩ : ⟨+MKS parameters⟩

3 ⟨-MKS parameters⟩ ::= ⟨stream parameter list⟩

3 ⟨+MKS parameters⟩ ::= ⟨stream parameter list⟩

3 ⟨stream parameter list⟩ ::= ⟨stream parameter expression⟩ | ⟨stream parameter list⟩, ⟨stream parameter expression⟩

3 ⟨stream parameter expression⟩ ::= ⟨parameter identifier⟩ ← ⟨expression⟩ | ⟨parameter identifier⟩

3 ⟨parameter identifier⟩ ::= ⟨identifier⟩

1 ⟨stream statement⟩ ::= ⟨unlabeled stream statement⟩ | ⟨label⟩ : ⟨unlabeled stream statement⟩

SEMANTICS.

The in-line character mode statement provides a means for entering character mode operation without the use of a stream procedure. The ⟨stream statement⟩ used in an in-line character mode statement is defined as in Extended ALGOL. The ⟨stream parameter statement⟩ requires explanation.

The stream parameter statement causes the stack to be set up as indicated in the stream parameter list. The colon represents the Mark Stack Control Word. If no colon is present, a colon is assumed to be at the extreme left.

Each ⟨parameter identifier⟩ in a ⟨stream parameter list⟩ of a stream parameter statement represents a stack location which can be referenced through use of that parameter identifier; every variable in the following stream statement must appear as a ⟨parameter identifier⟩ in the stream parameter list.

The contents of a given stack location is initially determined by the ⟨stream parameter expression⟩.

When a stream parameter expression appears as ⟨parameter identifier⟩ ← ⟨expression⟩, the value in the stack location represented by the parameter identifier is the value of the expression. It is permissible for the ⟨parameter identifier⟩ in such an expression to be identical to an identifier used outside of the in-line character mode area; no relationship is assumed. Identifiers, if any, in the ⟨expression⟩ part of a ⟨stream parameter list⟩ must, of course, have meaning outside of the in-line character mode area.

No declarations are required or allowed in an in-line character mode statement. Labels may be used and are considered declared by default.

An example of an in-line character mode statement is as follows:

```
STREAM (P1 ← [A], P2 ← B, G);
    BEGIN
        SI ← LOC P2;
        SI ← SI + 7;
        IF SC = "1" THEN GO TO FINI;
        SI ← LOC G;
        SI ← SI + 7;
        SKIP 2 SB;
```

```
            DI ← P1;
            DI ← DI + 5;
            IF SB THEN BEGIN
                        DI ← DI + 1;
                        DS ← 2 LIT "ON";
                        END
                  ELSE DS ← 3 LIT "OFF";
FINI:
END;
```

SECTION 11
PROCEDURES

GENERAL.

Procedure declarations in ESPOL are essentially the same as procedure declarations in Extended ALGOL. There are, however, areas of principle difference.

a. The ⟨procedure body⟩ of an ESPOL procedure must be a compound statement or a block.

b. The ⟨procedure body⟩ of an ESPOL procedure cannot contain procedure declarations other than stream procedure declarations.

c. A program segment is generated for every procedure declaration even though it may not contain declarations.

d. ESPOL has SAVE procedures and non-save procedures.

e. ESPOL requires that the specifications part of a procedure declaration contains only:

   1) ⟨type⟩ ⟨identifier list⟩.
   2) NAME ⟨identifier list⟩.
   3) ⟨array specification⟩.

PROCEDURE PARAMETERS.

In ESPOL, as in Extended ALGOL, procedure parameters may be call-by-value or call-by-name. When a procedure parameter is a call-by-value parameter, the actual parameter in the procedure statement may be any expression. When a procedure parameter is a call-by-name parameter, the actual parameter in the procedure statement must be provided according to the following rules.

a. If a formal parameter is specified to be REAL, BOOLEAN, ALPHA, or INTEGER, then its corresponding actual parameter may be any expression (e.g., a ⟨primary⟩).

b. If a formal parameter is specified to be an array, then its actual parameter may be an array identifier, an array row, or a POLISH primary (i.e., a construct of the form POLISH (⟨Polish string⟩).

c. If a formal parameter is specified NAME, then its corresponding actual parameter may be a ⟨variable⟩ or a POLISH primary.

SAVE PROCEDURES VERSUS NON-SAVE PROCEDURES.

A SAVE procedure is declared if a ⟨procedure declaration⟩ is preceded by the word SAVE. If the word SAVE does not precede a ⟨procedure declaration⟩, the procedure is a non-save procedure.

When a SAVE procedure is declared, ESPOL assigns an absolute area in core for the procedure and generates an appropriate procedure descriptor for the PRT area.

When a non-save procedure is declared, no absolute area in core is reserved; however, a procedure descriptor is provided. The procedure descriptor for a non-save procedure has the following special characteristics:

a. The address field at [33:15] contains the core address of the area which was reserved for the first SAVE segment declared in the program (i.e., the address of the first SAVE procedure or SAVE array declared).

b. The field at [18:15] contains the disk address of the disk area reserved for the procedure.*

* The disk addresses placed in procedure descriptors are relative to a zero base. This is done under the assumption that during execution, the program will be residing on the first module of disk, starting at address zero.

It is assumed that the first SAVE segment in an ESPOL Program will be a SAVE procedure designed to bring overlayable segments into core from disk, if such action is required. For further explanation, see the section covering the ESPBIT procedure in A Narrative Description of the Burroughs B 5500 Disk File Master Control Program.

SECTION 12
SUBROUTINES

GENERAL.

The subroutine facility provides a means whereby a specified code stream can be entered, and returned from, through use of branching operations. The code stream utilized, i.e., the subroutine, is defined in a subroutine declaration. Subroutine entrance is caused due to the occurrence of a subroutine call statement or a subroutine function designator.

SUBROUTINE DECLARATION.

SYNTAX.

The syntax for ⟨subroutine declaration⟩ is as follows:

3 ⟨subroutine declaration⟩ ::= SUBROUTINE
⟨subroutine identification⟩
⟨subroutine body⟩ | REAL SUBROUTINE
⟨subroutine identifier⟩
⟨subroutine body⟩ |
BOOLEAN SUBROUTINE
⟨subroutine identifier⟩
⟨subroutine body⟩

3 ⟨subroutine identifier⟩ ::= ⟨identifier⟩

3 ⟨subroutine body⟩ ::= ⟨statement⟩

SEMANTICS.

A subroutine declaration associates a subroutine identifier with a program statement which is to be performed whenever the subroutine identifier appears as such in the program. If a subroutine is not declared REAL or BOOLEAN, then the subroutine identifier must subsequently appear only as a ⟨subroutine call statement⟩. If a subroutine is declared REAL or BOOLEAN, then the following requirements exist:

a. The last statement in the subroutine body must be an assignment statement with the ⟨subroutine identifier⟩ as the ⟨left part⟩.

b. The subroutine identifier must subsequently appear only as a ⟨subroutine function designator⟩.

It should be noted that when control is transferred to the body of a subroutine, an operand is placed in the stack for use when branching back after completion of the subroutine. Consequently, if a subroutine is exited by a means other than "falling out" in the normal fashion (e.g., if exit were gained through use of a GO TO statement), then the branching operand must be explicitly deleted from the stack.

SUBROUTINE CALL STATEMENT AND SUBROUTINE FUNCTION DESIGNATOR.

SYNTAX.

The syntax from ⟨subroutine call statement⟩ is as follows:

⟨subroutine call statement⟩ ::= ⟨subroutine identifier⟩
⟨subroutine function designator⟩ ::= ⟨subroutine identifier⟩

SEMANTICS.

The subroutine call statement and the subroutine function designator are used to establish conditions necessary for branching to and from code defined by subroutine declarations.

SUBROUTINE CALL STATEMENT.

The subroutine call statement is used when calling a subroutine which was not declared REAL or BOOLEAN. The subroutine call statement is used with a SUBROUTINE like a procedure statement is used with a procedure which was not declared with a ⟨type⟩.

SUBROUTINE FUNCTION DESIGNATOR.

The subroutine function designator is used when calling a subroutine which was declared REAL or BOOLEAN. The subroutine function designator is a form of ⟨primary⟩ and has the value assigned to the subroutine identifier by the last statement in the subroutine body. The subroutine function designator is used with subroutines as a function designator is used with a procedure declaration with a type.

SECTION 13
SWITCH DESIGNATORS

GENERAL.

Switches in ESPOL are declared in the same fashion as in Extended ALGOL. The first valid value for a switch designator in ESPOL is zero. It should be noted, however, that ESPOL allows any particular switch declaration to be referenced by a maximum of one switch designator.

SECTION 14
DESIGNATIONAL EXPRESSIONS

SYNTAX.

The syntax for ⟨designational expression⟩ is as follows:

```
2 ⟨designational expression⟩  ::= ⟨simple designational expression⟩

2 ⟨simple designational expression⟩ ::= ⟨label⟩ |
                                        ⟨switch designator⟩ |
                                        (⟨designational
                                        expression⟩) |
                                        POLISH (⟨Polish string⟩)

1 ⟨switch designator⟩ ::= ⟨switch identifier⟩
                          [⟨subscript expression⟩]

1 ⟨switch identifier⟩ ::= ⟨identifier⟩

2 ⟨label⟩ ::= ⟨identifier⟩
```

SEMANTICS.

ESPOL designational expressions differ from Extended ALGOL designational expressions in two ways.

a. ESPOL allows only simple designational expressions.
b. ESPOL allows a Polish primary as a designational expression.

The simple designational expressions which are common to both Compilers are used in ESPOL and Extended ALGOL in the same fashion. The Polish primary as a designational expression requires explanation.

When a statement of the form GO TO POLISH (⟨Polish string⟩) occurs in a program, the Polish primary is evaluated and a "syllable branch forward unconditional" operator is executed. Consequently, the expression POLISH (⟨Polish string⟩) should place in the top of the stack a word to be used by the syllable branch operator (i.e., a literal for an rC relative branch or a descriptor for an absolute branch).

SECTION 15
ESPOL INTRINSICS

GENERAL.

The intrinsics which are provided in ESPOL are listed below, with appropriate definitions. Given that XP is an expression, the intrinsics are defined as follows:

a. ABS (XP) - produces the absolute value of XP.

b. NABS (XP) - produces negative the absolute value of XP.

c. FLAG (XP) - produces the value of XP with the flag bit set to 1.

d. NFLAG (XP) - produces the value of XP with the flag bit set to 0.

e. SIGN (XP) - produces one of three integerized values depending upon the value of XP (+1 for XP > 0, 0 for XP = 0, -1 for XP < 0).

f. HUNT (XP) - causes a flag bit search starting at the absolute address specified by the 15 low-order bits of XP and continuing, if necessary, to consecutively higher addressed locations. The final result is a word with the absolute address in the low-order 15 bits of the first location which contained a value with a flag bit of 1; the high-order three bits of the result are set to 101.

APPENDIX A
ESPOL STATEMENT EXAMPLES

Shown below are examples of ESPOL statements together with the code generated by them.

For all examples, the following declarations are assumed to have preceded.

```
REAL R;
INTEGER I;
NAME N;
ARRAY NAME AN;
ARRAY RA [*];
INTEGER ARRAY IA [10];
```

PART A

ESPOL Statement	Code Generated
R ← I;	OPDC I LITC R STD
I ← R;	OPDC R LITC I ISD
RA ← IA;	LITC IA LOD LITC RA STD
IA ← RA;	LITC RA LOD LITC LA STD

ESPOL Statement	Code Generated
N ← RA;	LITC RA LOD LITC N STD
RA ← N;	DESC N LITC RA STD
IA ← AN;	DESC AN LITC IA STD

PART B

ESPOL Statement	Code Generated
R ← RA [I];	OPDC I OPDC RA LITC R STD
R ← AN [I];	OPDC I DESC AN INX LOD LITC R STD
R ← RA [0];	LITC 0 OPDC RA LITC R STD
R ← AN [0];	OPDC AN LITC R STD

ESPOL Statement	Code Generated
RA [I] ← R;	OPDC I DESC RA OPDC R XCH STD
AN [I] ← R;	OPDC I DESC AN INX OPDC R XCH STD

APPENDIX B

REFERENCE FOR EXAMPLES OF CODE GENERATED BY ESPOL

Shown below are examples showing the majority of ESPOL constructs, together with the code generated for them.

For all examples, the following declarations are assumed to have preceded.

```
REAL R;
INTEGER I;
NAME N;
ARRAY NAME AN;
ARRAY RA [*];
INTEGER ARRAY IA [10];
DEFINE P = POLISH #;
```

ESPOL Statement	Code Generated
R ← I;	OPDC I LITC R STD
I ← R;	OPDC R LITC I ISD
R ← N;	DESC AN LITC R STD
R ← AN;	DESC AN LITC R STD
R ← P;	LITC R STD

ESPOL Statement	Code Generated	
R ← I;	DESC R OPDC I XCH STD	If R is call by name in a PROCEDURE
R ← P;	DESC R XCH STD	If R is call by name in a PROCEDURE
R ← RA [0];	LITC 0 OPDC RA LITC R STD	
R ← RA [I];	OPDC I OPDC RA LITC R STD	
R ← AN [0];	OPDC AN LITC R STD	
R ← AN [I];	OPDC I DESC AN INX LOD LITC R STD	
R ← * I;	OPDC I LOD LITC R STD	

ESPOL Statement	Code Generated
R ← * N;	DESC N LOD LITC R STD
R ← * RA;	LITC RA LOD LOD LITC R STD
R ← * AN [0];	OPDC AN LOD LITC R STD
R ← * AN [I];	OPDC I DESC AN INX LOD LOD LITC R STD
R ← * RA [0];	LITC 0 OPDC RA LOD LITC R STD
R ← * RA [I];	OPDC I OPDC RA LOD LITC R STD

ESPOL Statement	Code Generated
R ← [I];	DESC I LITC R STD
R ← [N];	DESC N LITC R STD
R ← [AN];	DESC AN LITC R STD
R ← [RA];	LITC RA LOD LITC R STD
R ← [AN[0]];	DESC AN LITC R STD
R ← [AN[I]];	OPDC I DESC AN INX LITC R STD
R ← [RA[0]];	LITC 0 DESC RA LITC R STD
R ← [RA[I]];	OPDC I DESC RA LITC R STD

ESPOL Statement	Code Generated
R ← * [I];	DESC I LOD LITC R STD
R ← * [N];	DESC N LOD LITC R STD
R ← * [AN];	DESC AN LOD LITC R STD
R ← * [RA];	LITC RA LOD LOD LITC R STD
R ← * [AN[0]];	DESC AN LOD LITC R STD
R ← * [AN[I]];	OPDC I DESC AN INX LOD LITC R STD
R ← * [RA[0]];	LITC 0 DESC RA LOD LITC R STD

ESPOL Statement	Code Generated
R ← * [RA[0]];	LITC 0 DESC RA LOD LITC R STD
R ← * [RA[I]];	OPDC I DESC RA LOD LITC R STD

APPENDIX C
CHARACTERISTICS OF PROGRAMS GENERATED BY ESPOL

The information in programs generated by the ESPOL Compiler can be considered to fall into two categories: save information and non-save information. The save information includes, in addition to SAVE arrays and SAVE procedures, the code for interrupt locations, the stack, and the PRT. The non-save information includes non-save procedures.

FORMAT OF INFORMATION ON DISK.
When the program generated by the ESPOL Compiler is written on disk, it is written on a file on the user disk which has the ⟨file identification prefix⟩ MCPbbbb and the ⟨file identification⟩ DISKbbb.

FORMAT OF SAVE INFORMATION. The appearance of the save information in the file MCP/DISK is a word for word image of the save information as it is required in core. That is, the save information starts in the first word of the file (i.e., word location zero of the file) and continues through consecutively higher addressed words as far as required. Specifically, the information is located in the following manner.

The code for interrupt locations is contained in the file at the locations which correspond with the interrupt location addresses.

The PRT information starts at word location 200.

The outer block code starts immediately following the PRT information. It should be noted that, since branch operations may be performed to transfer control from interrupt locations to the outer block, the distance from interrupt locations to the outer block should not exceed 1023 words.

The SAVE procedure and SAVE array information starts immediately following the outer block code. Within the area containing the SAVE arrays and SAVE procedures, the arrays and procedures occur in the order in which they occurred in the source language program.

FORMAT OF NON-SAVE INFORMATION. The non-save procedures follow the save information. Each non-save procedure starts at the beginning of a disk segment. The first disk segment in the file MCP/DISK is considered to be at disk address zero. Using this base, the disk segment address at which any non-save procedure starts is the address specified at [18:15] in the procedure's descriptor in the PRT.

FORMAT OF PROGRAMS ON CARDS.
When the program generated by ESPOL is punched on cards, the cards are punched in alpha mode (i.e., the cards are not binary cards) and they have the following format:

Columns	Field Content
1-3	For save information: zeros.
	For non-save information: the digits following DECK on the $ card.
4-8	Zeros.
9-72	One to eight words of code, in alpha-numeric code.
73	Zero.
74-75	Number of words of code in this card, in octal.
76-80	For save information: starting core address of first word of code on this card is increased by 96.
	For non-save information: starting address of first word of code on this card; the address of the first word of each non-save segment is set to zero.

FORMAT OF SAVE INFORMATION. Save information appears in the first cards of the card output file; each card is formatted as shown above. The address field in columns 76 through 80 contains a value 96 greater than the actual address of the first word of code on the card; the machine language one-card ESPOL DECK LOADER which loads an ESPOL generated deck requires the address be increased by 96.

The code address specified in columns 76 through 80, decreased by 96, is the address of the core locations in which the code is to be placed for execution. The appearance of the save information on cards, according to its specified addresses, is a word-for-word image of the save information as it is required in core. The required format of save information in core is as described above under FORMAT OF INFORMATION ON DISK.

TRANSFER CARD. Immediately following the cards containing save information is a transfer card. This card is used by the ESPOL DECK LOADER. The transfer card provides the final "boot strap" that transfers control to the save information after it has been read into core.

APPENDIX D
OPERATING CHARACTERISTICS OF THE ESPOL COMPILER

INITIATING THE ESPOL COMPILER.

The ESPOL Compiler is initiated through use of an EXECUTE card containing ? EXECUTE ESPOL/DISK.

$ CARD FOR ESPOL.

ESPOL has two types of $ cards: (1) a $ card which specifies I/O options, such as is used with the Extended ALGOL Compiler and (2) a $ VOID card.

The format of I/O option $ cards for ESPOL is the same as that for $ cards in Extended ALGOL. That is, a $ must appear in column 1 followed either by the word CARD or the word TAPE. The remainder of the card may contain any of the following option words, in a free field format:

LIST
PRT
NEW TAPE
DECK ⟨digit⟩ ⟨digit⟩
DEBUGN
STUFF
INTRINSIC

The words CARD and TAPE are used with the ESPOL Compiler for the same purpose as with the Extended ALGOL Compiler. That is, CARD is used when the source program is on cards alone and TAPE is used when compiling from tape using the card file as a patch deck. Also, the words LIST, PRT, NEW TAPE, and DEBUGN provide the same actions for ESPOL as they do for the Extended ALGOL Compiler. The word DECK, which is not an ALGOL option, may be used to specify that the program generated by the compilation should be punched on cards rather than written on disk. Once the DECK option is set, it will not be revoked due to subsequent $ cards which do not specify the option.

The use of the STUFF option causes a card to be punched, with the following format, for each procedure, array, or variable assigned to the PRT by the Compiler.

Columns	Field Content
1-4	Decimal class, where: 10 = Procedure 12 = Stream Procedure 13 = Boolean Stream Procedure 14 = Real Stream Procedure 15 = Integer Stream Procedure 17 = Boolean Procedure 18 = Real Procedure 19 = Integer Procedure 21 = Boolean 22 = Real 23 = Integer 25 = Boolean Array 26 = Real Array 27 = Integer Array 30 = Name
5-8	Decimal PRT address
9-80	Identifier, left justified, with blank fill

The production of these output cards is initiated by the appearance of the word STUFF on a $ card. The introduction of another $ card not containing the word STUFF will inhibit the punching of these cards.

The INTRINSIC option is turned on by the appearance of the word INTRINSIC on a $ control card signifying to ESPOL that it is compiling an intrinsic file and enabling it to produce a disk file in the format expected by the MCP.

An intrinsic program (i.e., the symbolic input to ESPOL when using the $INTRINSIC option) is subject to the following restrictions:

a. The outer block of the program may contain only declarations; no statements are allowed.

b. Only Non-Save Procedures may be declared. SAVE Procedures and Stream Procedures are prohibited.

c. No SAVE array may be declared.

d. Any variable declared in the outer block must be equated to an R or F relative address through use of a ⟨relative address expression⟩ in its declaration.

It may be noted that the $INTRINSIC option is required when compiling the symbolic file for the intrinsics of the B 5500 Programing System. When compiling these intrinsics, the ESPOL file DISK should be equated to INT/DISK.

The $ VOID card has the format $ VOID ⟨sequence number⟩ where the $ is in column 1 and the ⟨sequence number⟩ is recognized to be the field of eight consecutive characters starting with the first non-blank character following the word VOID. A VOID card in an ESPOL patch deck causes cards on the symbolic input tape to be deleted from the compilation and, if a NEW TAPE is being created during the compilation, the voided cards are not written on the new tape. Card images on a symbolic input tape which are affected by a VOID card are those with sequence numbers equal to or greater than the sequence number of the VOID card and <u>less than</u> the value in the ⟨sequence number⟩ field.

FILES IN ESPOL.
For purposes such as creating label equation cards, it is of value to have information about file declarations. ESPOL has seven files.

The card input file is declared:

```
FILE IN CARD (5, 10);
```

The tape input file is declared:

```
FILE IN TAPE "OCRDIMG" (2, BUFFSIZE);
```

The disk output file for compiled code is declared:

```
FILE OUT DISK DISK [990] "MCP" "DISK" (2, 30, SAVE 999);
```

The card output file for compiled code is declared:

```
FILE OUT DECK 0 (2, 10);
```

The line printer file is declared:

```
FILE OUT LINE 1 (2, 15);
```

The new-tape output file for updated source programs is declared:

```
SAVE FILE OUT NEWTAPE "OCRDIMG" (1, BUFFSIZE, SAVE 1);
```

The disk input/output file used during the first pass of a compilation is declared:

```
FILE OUT CODE DISK SERIAL [1:1] (1,1023);
```

NOTIFICATION OF SYNTAX ERRORS.
During an ESPOL compilation, syntax errors are printed in the same format as during an ALGOL compilation. The meanings of the syntax error messages are provided as the first information on the symbolic card image tape for the ESPOL Compiler.

It should be noted that if any syntax errors occur during a compilation, a DIV BY ZERO error will terminate the compilation before the second pass of the compilation begins. (The second pass organizes the save and non-save information into its proper format and causes the compiled programs to be written on disk or cards.)

EXECUTING A PROGRAM GENERATED BY ESPOL.
To execute a program generated by the ESPOL Compiler, one of the following procedures must be followed.

EXECUTING A PROGRAM WRITTEN ON DISK. If the program generated during an ESPOL compilation is written on disk, it must be DUMPed, through use of a DUMP control card, to a tape with the ⟨library tape name⟩ SYSTEMb. Then the DF MCP LOADER must be executed.

The DF MCP LOADER searches for a tape with the ⟨library tape name⟩ SYSTEM which contains the file MCP DISK. It then reads the tape and copies the program information onto disk starting at word zero of the first disk module.

After the DF MCP LOADER has loaded the program from tape to disk, the one-card DISK LOAD BUTTON must be executed. The DISK LOAD BUTTON causes the first 133 disk segments of information to be read into core from the first disk module, places that information in core starting at word zero, and then causes control to be transferred to core location 16 (decimal).

EXECUTING A PROGRAM WRITTEN ON CARDS. If the program generated during an ESPOL compilation is written on cards, it may be loaded to core and executed through use of the ESPOL DECK LOADER.

The first card of the DF MCP LOADER deck is a copy of the ESPOL DECK LOADER. The ESPOL DECK LOADER reads all of the ESPOL cards which contain save information (and the transfer card), places the save information in its designated core locations, and then transfers control to core location 16 (decimal).

1032638 10-67 Printed in U.S. America