CONTROL DATA CORPORATION • DEVELOPMENT DIV • SOFTWARE DOCUMENT

DOCUMENT CLASS ERS PAGE NO i

PRODUCT NAME FORTRAN Extended 2.0

PRODUCT NO. C012 VERSION ~~1.0~~ MACHINE SERIES ~~6400/6600~~ 6400/6500/6600

DEPT NO ~~474~~ 6231 PROJECT NO 4P6X1 3PC05 CHANGE NO DATE

SUBMITTED		REVIEWED				APPROVED	
PROJ MGR	DATE	DEPT MGR	DATE	DIR	DATE	MAB	DATE
12-6-66	12/5/66	12-9-66		12/12/66			12/20/66

Fred Magnus 12/6/66
DRB Chairman Date

JR Pawlett / JNA 12/6/66
QA Manager Date

12/6/66
DOC Manager Date

RC Gunther 12/6/66
SPM Date

RECEIVED
DEC 12 1966
INTERNAL DOCUMENT DISTRIBUTION

DISCLAIMER: THIS DOCUMENT IS A WORKING PAPER ONLY. AND DOES NOT NECESSARILY REPRESENT ANY OFFICIAL INTENT ON THE PART OF CONTROL DATA.

TABLE OF CONTENTS

1. INTRODUCTION

FORTRAN Extended Version 1.0 is a superset of ASA FORTRAN. This document establishes:

(1) The form of a program written in FORTRAN Extended Version 1.0.

(2) Rules for the interpretation of the program.

(3) The form of the input and output data.

Throughout this document all ASA extensions are made conspicuous by placing a dot in the margin where the extension is referenced or its definition begins.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an executable program (9.1.6).

A main program is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement (9.1.5).

A subprogram is similar to a main program but is headed by a BLOCK DATA, FUNCTION or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram (9.1.3, 9.1.4).

The term program unit will refer to either a main program or subprogram (9.1.7).

Any program unit except a specification subprogram may reference an external procedure (Section 9).

An external procedure that is defined by FORTRAN statements is called a procedure subprogram. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram. (Section 8 and 9).

Any program unit consists of statements and comments. A statement is divided into physical sections called lines, the first of which is called an initial line and the rest of which are called continuation lines (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (6.2).

The statements in FORTRAN fall into two broad classes - executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are names and operators. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the array name, deserves special mention. An array name must have the size of the identified array defined in an array declarator (7.2.1.1). An array name qualified only by a subscript is used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labeled with numbers. Input/output units are numbered (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE s $(a_1, a_2, \ldots, a_n)$

it is assumed that at least one symbolic name is included in the list within parentheses. A list is a set of identifiable elements each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term 'reference' is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1 THE FORTRAN CHARACTER SET

A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

The order in which the characters are listed does not imply a collating sequence.

3.1.1 Digits

A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7, These are only used in octal constants (4.2.7) and in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 Letters

A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 Alphanumeric Characters

An alphanumeric character is a letter or a digit.

3.1.4 Special Characters

A special character is one of the eleven characters blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, and currency symbol.

3.1.4.1 Blank Character

With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 4.2.6, 5.1.1.6, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 LINES

A line is a string of 72 characters. All characters must be from the FORTRAN character set except as described in 5.1.1.6 and 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1,2,3,...72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 Comment Line

The letter C or an asterisk or the currency symbol in column 1 of a line designates that line as a comment line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 End Line

An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each and in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 Initial Line

An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4 Continuation Line

A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and is not a comment line.

- A continuation line may only follow an initial or another continuation line or either of these followed by any number of comment lines.

3.2.5 Blank Lines

A line with blanks in columns 1 through 80 does not affect the program in any way, unless the succeeding line has a continuation punch in column 6 in which case it is treated as an initial line. The blank line will appear in the source program listing.

3.3 STATEMENTS

A statement consists of an initial line optionally followed by up to nineteen ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 STATEMENT LABEL

Optionally, a statement may be labeled so that it may be referenced in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5 SYMBOLIC NAMES

A symbolic name consists of from one to seven alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6 ORDERING OF CHARACTERS

An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statement exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

3.7 STATEMENT SEPARATOR

The currency symbol may be used to separate statements. The appearance of the currency symbol not in a Hollerith string serves to terminate the statement. The statement following the currency symbol may not be labeled nor may the preceding statement be a FORMAT statement.

4. DATA TYPES

Seven different types of data are defined. These are integer, real, double precision, complex, logical, octal and Hollerith. Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum

is involved. The data type of a function defines the type the datum it supplies to the expression in which it appears.

4.1 DATA TYPE ASSOCIATION

The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array has only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, double precision, complex, and logical types. This specific declaration overrides the implied association available for integer and real (5.3).

- If a symbolic name appears in more than one type statement in the same program unit the data type association for that name will be that given by the last-occuring such type statement.

- There exists no mechanism to associate a symbolic name with the Hollerith or octal data types. Thus data of these types are identified under the guise of a name of one of the other types. If an octal or Hollerith constant is not combined with a constant of some other type or a variable by an arithmetic (+,-,/,*,**) or any relational operator, it is treated as type integer, the result assuming the type of the replacement variable. If it is combined with an operand of another type, it is treated as the same type.

- ECS variables may only appear in the source program in the following circumstances:

(1) In a COMMON statement as an element of an ECS common block.

(2) In a CALL statement or function reference as an actual parameter.

(3) In a SUBROUTINE or FUNCTION statement as a dummy argument.

(4) In a TYPE ECS statement.

(5) In a DIMENSION statement.

4.2 DATA TYPE PROPERTIES

The mathematical and the representation properties for each of the data types are defined in the following sections. For real, double precision, and integer data, the value zero is considered neigher positive or negative.

The value zero is represented by a processor word with all bit positions equal to zero. If an all blank field is read using numeric conversion a -0 is produced.

4.2.1 Integer Type

An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

The value of an integer datum may be in the range -(2**48-1) through +(2**48-1). However, integer addition and subtraction take operands and give results in the range -(2**59-1) through +(2**59-1). When integers are used as DO parameters, they must be in the range $0 < I \leq 2^{17}-1$.

4.2.2 Real Type

A real datum is a processor approximation to the value of a real number. It may assume positive, negative and zero values. The nagnitude of non-zero real values may be in the range (10**322) to (10**(-293)) and will have a precision of fifteen (15) decimal digits.

4.2.3 Double Precision Type

A double precision datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

These data have the same magnitude range as real datum but have a precision of twenty-nine (29) decimal digits.

4.2.4 Complex Type

A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first of the pair represents the real part and the second, the imaginary part. Each part has, accordingly, the same degree of approximation as for a real datum.

4.2.5 Logical Type

A logical datum may assume only the truth values of true or false. True is represented by any negative value; false is any positive value.

4.2.6 Hollerith Type

A Hollerith datum is a string of characters. This string may consist of any characters capable of representation in the processor. The blank character is a valid and significant character in a Hollerith datum.

4.2.7 Octal Type

An octal datum is a bit pattern (60 bit word). It may consist of any pattern capable of representation in the processor.

4.2.8 ECS Type

An ECS datum occupies a sixty bit word and resides in Extended Core Storage.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term "reference" is used to indicate an identification of a datum implying that the current value of that datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be named. One case of special interest in which the datum is

named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 DATA AND PROCEDURE NAMES

A data name identifies a constant, a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1 Constants

A constant is a datum that is always defined during execution and may not be redefined. Rules for writing constants are given for each data type.

An integer, real, or double precision constant is said to be signed when it is written immediately following a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

O This also applies to Hollerith and octal constants. A signed Hollerith constant will cause an information diagnostic.

5.1.1.1 Integer Constant

An integer constant is written as a nonempty string of digits. The constant is the digit string interpreted as a decimal numeral.

The string may be up to 18 digits long.

5.1.1.2 Real Constant

A basic real constant is written as an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal part are strings of digits; either one of these strings may be empty but not both. The constant is an approximation to the digit string interpreted as a decimal numeral.

A decimal exponent is written as the letter, E, followed by an optionally signed integer constant. A decimal exponent is a multiplier (applied to the constant written immediately preceding it) that is an approximation to the exponential form ten raised to the power indicated by the integer written following the E.

The integer may be in the range -308 through +337.

A real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 Double Precision Constant

A double precision exponent is written and interpreted identically to a decimal exponent except that the letter, D, is used instead of the letter, E.

A double precision constant is indicated by writing a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent.

5.1.1.4 Complex Constant

A complex constant is written as an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The datum is an approximation to the complex number represented by the pair.

5.1.1.5 Logical Constant

The logical constants, true and false, are written .TRUE. and .FALSE. respectively.

• True and false constants may also be written .T. and .F.

5.1.1.6 Hollerith Constant

A Hollerith constant is written as an integer constant (whose value n is greater than zero) followed by the letter H, followed by exactly n characters which comprise the Hollerith datum proper. Any n characters capable of representation by the processor may follow the H. The character blank is significant in the Hollerith datum string. This type of constant may be written in the argument list of a CALL statement and in the data initialization statement.

• When n is not a multiple of 10, the last computer word is left justified with blank fill. Hollerith constants may also be used in function argument lists and arithmetic expressions. When used in arithmetic expressions, n should not be greater than 10. If the constant is used as an operand of an arithmetic operation, an information diagnostic will be given.

A Hollerith constant can also be written as an integer constant (whose value n is greater than zero) followed by either of the letters L or R followed by exactly n characters.

When n is not a multiple of 10 the last computer word is left justified with zero fill in the case of L, and right justified zero fill in the case of R.

5.1.1.7 Octal Constant

• An octal constant is written as a non-empty string of up to 20 octal digits followed by the letter B. An octal constant may be used in the same contexts as an integer constant, and is always operated on without conversion to another data type.

5.1.2 Variable

CA138-1 A variable is a datum that is identified by a symbolic name (3.5). Such

5.1.3 Array

An array is an ordered set of data of one, two or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 Array Element

An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2 Subscript

A subscript is written as a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1) determines the identified array element.

- The number of subscript expressions may be less than the declared dimensionality. For each missing expression the compiler will assume an expression value of one (1), thus if no subscript appears a subscript whose expressions are all equal to one (1) is implied.

5.1.3.3 Subscript Expression

A subscript expression is written as one of the following constructs:

c*v+k
c*v-k
c*v
v+k
v-k
v
k

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

o Further, a subscript expression may be any arithmetic expression. Subscripts of this more general class which do not conform to those described above are referred to as nonstandard subscripts. If the type of the expression is not integer, its value will be converted to integer before evaluating the array element successor function.

5.1.4 Procedures

A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 FUNCTION REFERENCES

A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See Section 10.2.1 for a discussion of requirements that apply to function references.

5.3 TYPE RULES FOR DATA AND PROCEDURE IDENTIFIERS

The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block, or a namelist name.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 DUMMY ARGUMENTS

A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8)

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Section 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

This section also includes formation and evaluation rules for masking expressions.

6.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, or complex. The arithmetic operators are:

Operator	Representing
+	Addition, positive value (zero + element)
-	Subtraction, negative value (zero - element)
*	Multiplication
/	Division
**	Exponentiation

The constituent elements of the expression may also include values of types octal and Hollerith.

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form

primary**primary

A term is a factor or a construct of one of the forms

term/factor

or

term*term

A signed term is a term immediately preceded by + or -.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or -.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or - immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary, and the resultant factor is of the same type as that of the lement being exponentiated. A real or double precision primary may be exponentiated by a real or double precision primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type double precision.

- An integer primary may also be exponentiated by a real or double precision primary, and the resultant factor is of type real or double precision, respectively. An integer, real, or double primary may be exponentiated by a complex primary and the resultant factor is of type complex.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type. Further, an admissible real element may be combined with an admissible double precision or complex element; the resultant element is of type double precision or complex respectively.

- Further an admissible integer element may be combined with an admissible real, double precision, or complex element; the resultant element is of type real, double precision, or complex, respectively. An admissible double precision element may be combined with an admissible complex element; the resultant element is of type complex.

6.2 RELATIONAL EXPRESSIONS

A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. One arithmetic expression may be of type real or double precision and the other type real or double precision or both arithmetic expressions may be of type integer. If a real expression and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged. The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greather than or equal to

- The operands of a relational operator may take the same combinations of types as are defined for the arithmetic operators. Only the real part of complex elements are used by relational operators, except for .EQ. and .NE.

6.3 LOGICAL EXPRESSIONS

A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	Representing
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

- .OR., .AND., and .NOT. may be written .O., .A., and .N., respectively.

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4 MASKING EXPRESSIONS

A masking expression is formed with masking operators and masking elements. The masking operators are:

Operator	Representing
.AND. or .A.	Bit-by-bit logical multiplication
.OR. or .O.	Bit-by-bit logical addition
.NOT. or .N.	Bit-by-bit logical negation

The operators operate only on the most significant word of double precision elements and the real part of complex elements; no type conversion of elements is performed in the evaluation of the expression.

The masking elements are masking primary, masking factor, masking term, and masking expression.

A masking primary is a masking expression enclosed in parentheses, or an arithmetic expression.

A masking factor is a masking primary or .NOT. followed by a masking primary.

A masking expression is always of type octal.

A masking term is a masking factor or a construct of the form:

masking term .AND. masking term

A masking expression is a masking term or a construct of the form:

masking expression .OR. masking expression

6.5 EVALUATION OF EXPRESSIONS

A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expression imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence (except as modified in the following paragraph).

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The value of an integer factor or term is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term. The associative and commutative laws do not apply in the evaluation of integer terms containing division, hence the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement, or CALL statement in which the function reference appears, the type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real, double precision, or complex exponent. No factor may be evaluated that requires raising a zero valued primary to a zero

valued exponent.

No element may be evaluated whose value is not mathematically defined.

Expressions are evaluated generally from left to right with the binding strength of the operators and parentheses controlling the order of operations. All function references and exponentials which are not evaluated in-line are evaluated first. The heirarchy of operators from highest binding strength to lowest is: **, /, *, + and -, relationals, .NOT., .AND., and .OR.. However, if a divide operator has an integer operand, the operator will assume the same binding strength as the multiply operator. If an expression is mixed mode, conversion from one type to another occurs when the operands of an arithmetic or relational operator are of different types. The types of the operands of an operator depend on the type of elements and the order of evaluation up to the point of issuing the operator.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and classification of program units.

7.1 EXECUTABLE STATEMENTS

There are three types of executable statements:

(1) Assignment statements.
(2) Control statements.
(3) Input/Output statements.

7.1.1 Assignment Statements

There are four types of assignment statements:

(1) Arithmetic assignment statement.
(2) Logical assignment statement.
(3) GO TO assignment statement.
(4) Masking assignment statement.

7.1.1.1 Arithmetic Assignment Statement

An arithmetic assignment statement is of the form:

v=e

where v is a variable name or array element name of type other than logical and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

- Several variables may be set to the value of the same expression by using the following form of the arithmetic assignment statement:

$$V_1 = v_2 = \ldots v_m = e$$

Here the value of the expression e is converted to the type of v_m and stored there. V_m is then converted to the type v_{m-1} and stored there. This process is repeated until a value is stored in v_1.

7.1.1.2 Logical Assignment Statement

A logical assignment statement is of the form:

v = e

where v is a logical variable name or a logical array element name and e is a logical expression. Execution of this statement causes the logical expression to be evaluated and its value to be assigned
- to the logical entry. Multiple replacement is also available for logical assignment statements.

7.1.1.3 GO TO Assignment Statement

A GO TO assignment statement is of the form:

ASSIGN k TO i

- where k is a statement label and i is an variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement (Section 7.1.2.1.2) using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition (10.2) of the variable. The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears.

Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO

statement until it has been redefined (Section 10.2.3).

7.1.1.4 Masking Assignment Statement

A masking assignment statement is of the form:

v = e

where v is a variable name or array element name and e is a masking expression as defined in Section 6.4. The replacement variable v may be of any type, but if of type complex or double precision, the imaginary or least significant word will be replaced with a zero. Multiple replacement is available for masking assignment statements.

TABLE 1. RULES FOR ASSIGNMENT OF e to v

If v Type is	And e Type is	The Assignment Rule Is*
Integer	Integer	Assign
Integer	Real	Fix and Assign
Integer	Double Precision	Fix and Assign
Integer	Complex	P Fix and Assign Real Part
Real	Integer	Float and Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate and Real Assign
Real	Complex	P Assign Real Part
Double Precision	Integer	DP Float and Assign
Double Precision	Real	DP Evaluate, Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P DP Float Real Part and Assign
Complex	Integer	P Float and Assign to Real Part,I
Complex	Real	P Assign Real Part,I
Complex	Double Precision	P DP Evaluate and Real Assign to Real Part , I
Complex	Complex	Assign

* NOTES

(1) P means prohibited combination under ASA.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

(4) DP Evaluate means evaluate the expression according to the rules of 6.1 (or any more precise rules) then DP Float.

(5) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(6) Float means transform the value to the form of a real datum.

(7) DP Float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double datum can contain.

(8) Real Part refers to the real portion of the complex datum.

(9) I means imaginary part of the complex datum is set to zero.

7.1.2 Control Statements

There are eight types of control statements:

(1) GO TO statements.
(2) Arithmetic IF statement.
(3) Logical IF statement.
(4) CALL statement.
(5) RETURN statement.
(6) CONTINUE statement.
(7) Program control statements.
(8) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO statements

There are three types of GO TO statements:

(1) Unconditional GO TO statement.
(2) Assigned GO TO statement.
(3) Computed GO TO statement.

7.1.2.1.1 Unconditional GO TO statement

An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2 Assigned GO TO statement

An assigned GO TO statement is of the form:

GO TO i, $(k_1, k_2, \ldots, k_n)$

where i is an integer variable reference, and the k's are statement labels.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statement labels in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

7.1.2.1.3 Computed GO TO statement

A computed GO TO statement is of the form:

GO TO (k_1, k_2,...k_n), i

or

GO TO (k_1, k_2,...,k_n), e

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

As an extension to ASA i may be replaced by e, any arithmetic expression as described in Section 6.1. The value of e will be converted to integer and used in place of i.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the integer value of i or e at the time of the execution. If the value of i or e is less than one it will be treated as equal to one. If the value is greater than n it will be treated as equal to n. The comma following the right parenthesis is optional.

7.1.2.2 Arithmetic IF Statement

An arithmetic IF statement is of the form:

IF (e) k_1, k_2, k_3

or

IF (e) k_1, k_2

Where e is any arithmetic expression of type integer, real, or double precision, and the k's are statement labels.

As an extension to ASA e may be of type complex, in this event only the real part is used in selecting the branch.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

The second form is an extension to ASA, e may be a masking or arithmetic expression. Execution of this statement causes evaluation of the expression following which the statement identified by the statement label k_1 or k_2 is executed next as the value of e is nonzero or zero, respectively.

7.1.2.3 Logical IF Statement

A logical IF statement is of the form:

```
IF (e) s
   or
IF (e) k1, k2
```

where e is a logical expression and s is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement s is executed as though it were a CONTINUE statement. If the value of e is true, s is executed.

The second form is provided as an extension to ASA. If the logical expression is true the statement identified by statement label k_1 is executed next, if false the statement labeled k_2 is executed next.

7.1.2.4 CALL Statement

A CALL statement is of one of the forms:

CALL $s(a_1, a_2, \ldots a_n)$

or

CALL s

or

CALL s (a_1, a_2,...a_n), RETURNS (b_1, b_2,...b_m)

or

CALL s, RETURNS (b_1, b_2,...b_m)

where s is the name of a subroutine and the a's and b's are actual arguments (8.4.2). The b's are statement labels of statements in the current calling subprogram and indicate alternate return points (8.4.1).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5 RETURN Statement

- A RETURN statement if of one of the forms:

RETURN

or

RETURN a

- where a is a dummy argument which appears in the associated RETURNS list.

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

The statement RETURN a can only appear in a subroutine subprogram. Execution of this statement causes control to be returned at the statement number corresponding to a in the returns list.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6 CONTINUE Statement

A CONTINUE statement is of the form:

```
CONTINUE
```

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7 Program Control Statements

There are two types of program control statements:

(1) STOP statement.

(2) PAUSE statement.

7.1.2.7.1 STOP Statement

A STOP statement is of one of the forms:

```
STOP n
  or
STOP
```

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program. n is displayed in the dayfile.

7.1.2.7.2 PAUSE Statement

A PAUSE statement is of one of the forms:

PAUSE n

or

PAUSE

where n is an octal digit string of length from one to five. n is displayed in the dayfile. 1

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program, but if execution is resumed without otherwise changing the state of the processor, the completion of the PAUSE statement causes continuation of normal execution sequence.

7.1.2.8 DO Statement

A DO statement is one of the forms:

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where

(1) n is the statment label of an executable statement. This statement, called the terminal statement of the associated DO must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement, nor a logical IF containing any of these forms.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1 called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2 and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A completely nested nest is a set of DO statements and their ranges, and any DO statements contained with their ranges, such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement.

- A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following six steps:

1. The control variable is assigned the value represented by the initial parameter. This value should be less than or equal to the value represented by the terminal parameter; if this condition is not true, the DO loop will be executed once.
2. The range of the DO is executed.
3. If control reached the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.
4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

6. If any of the control parameters exceed $2^{17}-1$ or if integer constants used as control parameters exceed ten digits with or without leading zeroes, the performance of the loop is unspecified.

In the remainder of this section (7.1.2.8) a logical IF statement containing a GOTO or arithmetic IF statement form is regarded as a GOTO or arithmetic IF statement respectively.

Upon exiting from the range of a DO by execution of a GOTO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A DO is said to have an extended range if both of the following conditions apply:

(1) There exists a GOTO statement or arithmetic IF statement within the range of a DO nest that can cause control to pass out of that nest.

(2) There exists a GOTO statement or arithmetic IF statement not within the nest that, in the collection of all possible sequences of execution in the particular program unit could be executed after a statement of the type described in (1), and the execution of which could cause control to return to a statement in the nest such that the innermost DO which contains this statement also contains the statement described in (1).

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is. A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a DO of the same program unit that has an extended range. When a procedure reference occurs in the range of a DO the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.2.9 ENCODE/DECODE Statements

- Core to core statements. These statements are comparable to the formatted write and formatted read statements, respectively with the difference that no peripheral equipment is involved. Information is transferred under format specifications from one area of internal storage to another.

DECODE Statement. A DECODE statement is of the form:

DECODE (n,f,A) k

where k is a list, A is a variable or array element reference, n is an integer constant or variable, and f is a format specification.

Execution of this statement causes the transfer and editing of n characters starting at A. The information is scanned and converted as specified by the format specification identified by f. The resulting values are assigned to the elements specified by the list. If the number of characters specified by the list is greater than n , conversion will continue at the first full word following the n^{th} character.

ENCODE Statement

An ENCODE statement is of the form:

ENCODE (n,f,A) k

where k, A, n and f are as above.

Execution of this statement causes the transfer of information in the list, converted according to the format specification f, to locations starting at A, for n characters. If the number of characters converted is less than n, the remaining characters are made blanks. If the number of characters specified by the list is more than n characters, conversion will continue at the first full word following the n^{th} character. If n is not a multiple of 10, the remaining characters in the last word are converted to blanks.

7.1.3 Input/Output Statements

There are two types of input/output statements:

(1) READ and WRITE statements.

(2) Auxiliary input/output statements.

The first type consists of the statements that causes transfer of records of sequential files to and from the internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, u and f identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and u may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in Section 7.2.3. u may take on values from 1 to 99.

Either the statement label of a FORMAT statement or an array name may be represented by f. If a statement label, the identified statement must appear in the same program unit as the input/output statement. If an array name, it must conform to the specifications in 7.2.3.10.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the foregoing, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ and WRITE Statements

The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted reeord consists of a string of the characters that are permissible in Hollerith constants (5.1.1.6). The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 Input/Output Lists

The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$i = m_1, m_2, m_3$

or

$i = m_1, m_2$

The elements i, m_1, m_2, and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 Formatted READ

A formatted READ statement is of one of the forms:

READ (u,f) k

or

READ (u,f)

where k is a list.

Execution of this statement causes the input of the records from the unit identified by u. u may be an integer constant or simple integer variable. The information is scanned and converted as specified by the format specification identified by f. The resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

READ f,k

or

Read f

Execution of this statement causes the input of records from the INPUT file.

7.1.3.1.3 Formatted WRITE

A formatted WRITE statement is of one of the forms:

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

where k is a list.

Execution of this statement creates the next records on the unit identified by u. u may be an integer constant or simple integer variable, the k list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by f. See however, 7.2.3.4.

```
PRINT f,k
PUNCH f,k
```

Execution of these statements creates the next record on the OUTPUT and PUNCH file respectively.

7.1.3.1.4 Unformatted READ

An unformatted READ statement is of one of the forms:

```
READ  (u) k
   or
READ  (u)
```

where k is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

The effect of a READ statement without a list is to skip the next logical record on the I/O device.

7.1.3.1.5 Unformatted WRITE

An unformatted WRITE statement is of the form:

```
WRITE  (u)  k
```

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

The format of the record on the external device is described in Chapter 3 of the 6400/6600 SCOPE reference manual (Pub. No. 60173800).

7.1.3.1.6 BUFFER IN and BUFFER OUT statements.

Recording Mode

The recording mode k in the following statements designate equipment pecularities, it is inoperative for many peripheral devices. For magnetic tapes, k=0 designates even parity, k=1 designates odd parity.

BUFFER IN statement

A BUFFER IN statement is of the form:

```
BUFFER IN (u,k)  (A,B)
```

where k is the recording mode (integer constant or simple integer variable), A is the first word of the block to be transmitted, and B is the last word of the block to be transmitted.

Execution of this statement causes the input of the next record from unit u to be initiated. Only one physical record is read for each BUFFER IN statement.

BUFFER OUT statement

A BUFFER OUT statement is of the form:

```
BUFFER OUT  (u,k)  (A,B)
```

A is the first word of the data block to be output, and B is the last word of the block to be output.

Execution of this statement causes the output of words A through B, inclusive, on unit u to be initiated, and control returned to the program, permitting the performance of the other tasks while transmission is in progress. One physical record is written for each BUFFER OUT statement.

All buffer statements should be followed by an I/O status function. See Appendix C.

7.1.3.1.7 NAMELIST Statement

A NAMELIST statement is of the form:

NAMELIST $/y_1/a_1/y_2/a_2/.../y_n/a_n$

where each y is a symbolic name and each a is a list of the form $b_1,b_2,...,b_m$. Each b is a variable or array name.

Each y is a NAMELIST name, a name which must differ from all other names in the program unit. The name after being defined may appear only in READ or WRITE statements as described later in this section. A NAMELIST name may be defined only once in its program unit, and must precede any reference to it.

In any given NAMELIST statement, the list a of variable names or array names occurring between the NAMELIST name y and the next NAMELIST name (or the end of the statement if no NAMELIST name follows) is associated with the NAMELIST name y. A variable name or array name may be an element of more than one such list. In a subprogram, b may be a dummy argument identifying a variable or an array with the exception that the array may not have adjustable dimensions.

The NAMELIST statement permits the input and output of character strings consisting of names and values without a format specification. The statements:

```
READ   (u,x)
WRITE  (u,x)
```

where u is an integer variable or integer constant and x is a NAMELIST name, cause BCD input or output, on the device specified as a logical unit u, of the variables and arrays associated with the name x.

READ (u,x) causes unit u to scan the current file until it locates either an end of file or a record with a $ in column 2 followed immediately by the name x with no embedded blanks. On location of the name x the unit reads the following data items from the current file until it encounters a $.

Data items are separated by commas. They may take any of three forms:

(1) v=c

(2) $a=d_1,\ldots d_j$

(3) $a(n) = d_1,\ldots d_m$ where v is a variable name. c is a constant, a is an array name and n is a subscript consisting wholly of integer constants. The number of constants must equal the number of dimensions of the array a. The d_m are simple constants or repeated constants of the form k*c, where k*c indicates that the constant c is to be repeated k times.

The number of constants, including repetitions, given for an (unsubscripted) array name must equal the number of elements in that array. For a subscripted array name, the number of constants need not equal but may not exceed the number of array elements which follow the specified element, plus one.

v=c causes the variable v to be set to c.

$a=d_1,\ldots d_j$ causes the values $d_1,\ldots,d_j$ to be stored in consecutive elements of array a, in the order in which the array is stored internally.

$a(n) = d_1,\ldots,d_m$ fills elements consecutively starting at a(n).

The specified constants may be integers, real numbers, double precision numbers, complex numbers of the form (c_1,c_2) or logical constants of the forms, T, .TRUE., F, .FALSE. A logical or complex variable may be set only to a logical and complex constant , respectively. Any other variable may be set to an integer, real or double-precision constant. Such a constant will be converted to the type of its associated variable.

Constants and repeated constant fields may not include embedded blanks, blanks may appear elsewhere in data records.

Input records may not include more than 120 characters. More than one record may be used for input data. Each but the last record must end with a constant followed by a comma, and no serialization numbers may appear; the first column of each record will be ignored.

The set of data items may consist of any subset of the variable names associated with x. These names need not appear in the same order in which they appear in the defining NAMELIST statement.

WRITE (u,x) causes output on unit u of BCD information as follows:

(1) One record consisting of a $ in column 2 immediately followed by the name x.

(2) As many records as are needed to output the current values of all variables in the list associated with x. Simple variables are output as v=c. Elements of dimensioned variables are output in the linear order in which they are stored internally. The fields for the data are made large enough to include all significant digits. Logical constants appear as T and F. No data appears in column 1 of any record.

(3) One record consisting of a $ in column 2 immediately followed by the letters END.

The records output by such a WRITE statement may be read by a READ (u,x) statement where x is the same NAMELIST name.

7.1.3.2 Auxiliary Input/Output Statements

There are three types of auxiliary input/output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND statement.

A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE Statement

A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE Statement

An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is an unique record signifying a demarcation of a sequential file.

7.1.3.3 Input/Output with Extended Core Storage (ECS)

The two following subroutines are provided to permit the transfer of data between ECS and central memory. The form of the calls is:

CALL READEC (A,B,N)

and

CALL WRITEC (A,B,N)

Subroutines READEC and WRITEC effect the transfer of data from ECS to central memory, or from central memory to ECS respectively.

A is a simple or subscripted variable located in central memory, B is a simple or subscripted variable located in an ECS common block (see Section 7.2.1.3) and N is an integer constant or integer expression. When the subroutine is called, N consecutive words of data will be transferred between central memory and ECS beginning with location A in central memory and B in ECS.

7.1.3.3.1 Input/Output with Mass Storage (MS)

Four object time I/O subroutines are provided to control the transfer of records between central memory and a mass storage device. The forms of the calls are:

CALL OPENMS (u,ix,L,p)
CALL READMS (u,fwa,n,i)
CALL WRITMS (u,fwa,n,i)
CALL STINDX (u,ix,L)

u is the unit number
ix is the first word address of the index (in central memory)
L is the length of the index (for a name index,
$L \leq 2$(number of index entries) +1; for a number index,
$L \leq$ number of index entries +1)
P indicates that the file is referenced through a name index (p=1), or through a number index (p=0).
fwa is the central memory address of the first word of the record.
n is the number of central memory words transferred.
i is the record name or number.

OPENMS is called to open the mass storage file. OPENMS informs SCOPE that this file is a "random access" file and if the file exists, the master index is read into the area specified by the program.

READMS and WRITMS perform the actual transfer of data to or from central memory.

STINDX is called to change the file index to the base specified in the call (see Appendix J).

7.1.3.4 Printing of Formatted Record

When formatted records are prepared for printing, the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	Vertical Spacing Before Printing One Line
Blank	One line
0	Two lines
1	To first line of next page
+	No advance
Other	One line

7.2 NONEXECUTABLE STATEMENTS

There are six types of nonexecutable statements:

(1) Specification statements.

(2) Data initialization statement.

(3) FORMAT statement.

(4) Function defining statements.

(5) Subprogram statements.

(6) NAMELIST statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8. NAMELIST statements are defined in Section 7.1.3.1.7.

7.2.1 Specification Statements

There are five types of specification statements:

(1) DIMENSION statement.

(2) COMMON statement.

(3) EQUIVALENCE statement.

(4) EXTERNAL statement.

(5) Type-statements.

7.2.1.1 Array-Declarator

An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name , the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator form may be a type-statement, DIMENSION, or COMMON statement.

An array declarator has the form:

v (i)

where:

(1) v, called the declarator name, is a symbolic name.

(2) (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator. Doing so may result in referencing an element outside of the array.

7.2.1.1.1 Array Element Successor Function and Value of a Subscript. For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero or the result will be undefined.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. VALUE OF A SUBSCRIPT

Dimensionality	Subscript Declarator	Subscript	Subscript Value	Maximum Subscript Value
1	(A)	(a)	a	A
2	(A, B)	(a,b)	a + A•(b-1)	A•B
3	(A, B, C)	(a,b,c)	a + A•(b-1) + A•B•(c-1)	A•B•C

NOTES (1) a, b and c are subscript expressions.
(2) A, B, and C are dimensions.

7.2.1.1.2 Adjustable Dimension

If any of the entries in a declarator subscript is an integer variable name, the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may only appear in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6), there must be at least one constant array declaration associated through subprogram references.

In a subprogram, a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2 DIMENSION Statement

A DIMENSION statement is of the form:

DIMENSION $v_1(i_1), v_2(i_2),\ldots,v_n(i_n)$

where each v(i) is an array declarator.

7.2.1.3 COMMON Statement

A COMMON statement is of the form:

COMMON $/x_1/a_1/.../x_n/a_n$

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name, a number up to seven digits long, or it is empty. If x_1 is empty, the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement, the entities occuring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement if no block name appears, are declared to be in blank or unlabeled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The length of a common block other than BLANK COMMON must not be increased by any program unit that comprises an executable program after that length is initially established at load time. Any declaration of the same common block name in subsequently loaded program units must have a length less than or equal to the original declaraction. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1 Correspondence of Common Blocks

If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

(1) There is identity in type for all entities defined in the corresponding position from the beginning of that block.

(2) If the block is labeled and the same number of entities is defined for the block, then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

A double precision or a complex entity is counted as two logically consecutive storage units; a logical, real, or integer entity, as one storage unit.

Then for common blocks with the same number of storage units or blank common:

(1) in all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

• If any of the elements of a COMMON block are of type ECS then all of the elements of that block must be of type ECS. No elements of type ECS may appear in the blank COMMON block.

7.2.1.4 EQUIVALENCE Statement

An EQUIVALENCE statement is of the form:

EQUIVALENCE (k_1), (k_2), ...,(k_n)

in which each k is a list of the form:

$a_1, a_2, \ldots, a_m$

Each a is either a variable name or an array element name (not a dummy argument or an ECS variable or array element), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two storage unit entity is equivalenced to a one storage unit entity, the latter will share space with the first storage unit of the former.

The assignment of storage to variables and array declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declaration statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5 EXTERNAL Statement

An EXTERNAL statement is of the form:

EXTERNAL $v_1, v_2, \ldots, v_n$

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6 Type-statements

A type-statement is of the form:

$tv_1, v_2, \ldots, v_n$

- where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or ECS optionally preceded by the characters TYPE and each v is a variable name, an array name, a function name, or an array declarator.

DOUBLE may be used in place of DOUBLE PRECISION.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of the type double precision, complex, or logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit. See, however, the restriction in 8.3.1 second paragraph.

7.2.2 Data Initialization Statement

A data initialization statement is of the form:

DATA $k_1 / d_1 /$, $k_2 / d_2 /$, ..., $k_n / d_n /$

where:

(1) Each k is a list containing names of variables and array elements. Each k can also be an array name that can have from one to three control subscripts (each of which may be variable) and from one to three integer constant control parameters. These control subscripts have no meaning except in the range of the array name within the DATA statement.

(2) Each d is a list of constants and optionally signed constants, any of which may be preceded by j*.

(3) j is an integer constant.

If a list contains more than one entry, the entries are separated by commas.

Dummy arguments may not appear in the list k.

When the form j* appears before a constant it indicates that the constant is to be specified j times. A Hollerith constant may appear in the list d.

The constants may be grouped by parentheses and optionally preceded by j*.

A data initialization statement is used to define initial values of variables or array elements. There must be a one-to-one correspondence between the list-specified items and the constants. By this correspondence the initial value is established.

If a list-specified item is an array name with no control subscript or parameters, the constant list will define values in the array to the maximum dimensional length or until the constant list is exhausted.

An initially defined variable or array element may not be in blank common.

7.2.2.1 Alternate Form of Data Statement

The alternate form of the data initialization statement has the form:

DATA $(r_1=d_1),(r_2=d_2),\ldots(r_n=d_n)$

where:

(1) Each r is a list containing names of variables and arrays or array element names that can have from one to three control subscripts (each of which may be variable) and from one to three integer constant control parameters. These control subscripts have no meaning except in the range of the array name within the DATA statement.

(2) Each d is a list of constants and optionally signed constants, any of which may be preceded by j*. The constants may be grouped by parenthesis and optionally preceded by j*.

(3) j is an integer constant.

7.2.3 FORMAT Statement

FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

FORMAT $(q_1t_1z_1t_2z_a\ldots t_nz_nq_2)$

where

(1) $(q_1t_1z_1t_2z_2\ldots t_nz_nq_2)$ is the format specification.

(2) Each q is a series of slashes or is empty.

(3) Each t is a field descriptor or group of field descriptors.

(4) Each z is a field separator.

(5) n may be zero.

A FORMAT statement must be labeled.

7.2.3.1 Field Descriptors

The format field descriptors are of the form:

srEw.d

srFw.d

srGw.d

srDw.d

rIw

rLw

rAw

n $Hh_1, h_2 \ldots h_n$

nX

rRw

rOw

...

Tn

where:

(1) The letters F,E,G,D,I,L,A,H,R,T, O, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string. n used in conjunction with T indicates a beginning column position for subsequent information.

(3) d is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) s is optional and represents a scale factor designator.

(6) Each h is one of the characters capable of representation by the processor.

(7) Asterisks are used to delimit Hollerith strings.

- For all descriptors, the field width must be specified. For descriptors of the form w.d, if the d is not specified, it is assumed zero. Further, w must be greater than or equal to d.

The phrase basic field descriptor will be used to signify the field descriptor unmodified by s or r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 Field Separators

The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an internal medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 Repeat Specifications

Repetition of the field descriptors (except n H and n X) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the

group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses.

7.2.3.4 Format Control Interaction with an Input/Output Output List

The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than nH, nX, nT, or Asterisks must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, G, D, A, R, O, or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/putput list, except that a complex element requires the interpretation of two F, E, or G basic descriptors. To each H, Asterisks, T, or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, O, R, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5 Scale Factor

A Scale factor designator is defined for use with the F, E, G, and D conversions and is of the form:

n P

when n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors, until another scale factor is encountered, and then that scale factor is estalished.

7.2.3.5.1 Scale Factor Effects

The scale factor n affects the appropriate conversions in the following manner.

(1) For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

externally represented number equals internally represented number times the quantity ten raised to the nth power.

(2) For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

(3) For E and D output, the basic real constant part (see Section 5.1.1.2) of the output quantity is multiplied by 10^n and the exponent is reduced by n.

(4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

7.2.3.6 Numeric Conversion

The numeric field descriptors, I, F, E, G, and D are used to specify input/output of integer, real, double precision, and complex data.

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

(5) The number of characters produced by an output conversion must not exceed the field width.

If the field width is exceeded a leading asterisk will be inserted.

7.2.3.6.1 Integer Conversion

The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 Real Conversions

There are three conversions available for use with real data: F, E, and G.

The numeric field descriptor Fw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.
(2) E followed by an integer constant.
(3) E followed by a signed integer constant.
(4) D followed by an integer constant.
(5) D followed by a signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor Ew.d indicates that the external field occupies w positions , the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is:

$$\overset{\wedge}{\mathrm{f}}.x_1 \ldots x_d \; Y$$

where:

(1) $x_1 \ldots x_d$ are the d most significant rounded digits of the value of the data to be output.

(2) Y is of one of the forms:

$$E \pm y_1 y_2$$

or

$$\pm \; y_1 y_2 y_3$$

and has the significance of a decimal exponent.

(3) Each y is a digit.

(4) ξ signifies no character position or minus in that position.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

(1) If $n \leq 0$, there will be exactly -n leading zeros and d+n significant digits after the decimal point.

(2) If $n > 0$, there will be exactly n significant digits to the left of the decimal point and d-n+1 to the right of the decimal point.

The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

Input processing is the same as for the F conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \leq N < 1$	F (w-4) .d, 4 X
$1 \leq N < 10$	F (w-4) .(d-1), 4X
$10^{d-2} \leq N < 10^{d-1}$	F (w-4) .1, 4 X
$10^{d-2} \leq N < 10^{d}$	F (w-4) .0, 4 X
Otherwise	sEw.d

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

7.2.3.6.3 Double Precision Conversion

The numeric field descriptor Dw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a double precision datum.

The basic form of the external input field is the same as for real conversions.

The external output field is the same as for the E conversion, except that the character D may replace the character E in the exponent.

7.2.3.6.4 Complex Conversion

Since a complex datum consists of a pair of separate real data, the conversion is specified by two successfully interpreted real field descriptors. The first of these supplies the real part. The second supplies the imaginary part.

7.2.3.7 Logical Conversion

The logical field descriptor Lw indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output filed consists of w-1 blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8 Hollerith Field Descriptor

Hollerith information may be transmitted by means of four field descriptors, nH, Aw, Rw, and Asterisks.

(1) The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

(2) The Aw descriptor causes w Hollerith characters to be read into, or written from, a specified list element.

(3) The asterisks descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) between two successive asterisks appearing in the format statement.

(4) The Rw descriptor causes w Hollerith characters to be read into or written from, a specified list element.

Let g be the number of characters representable in a single storage unit (7.2.1.3.1). If the field width specified for A or R input is greater than or equal to g, the rightmost g characters will be taken from the external input field. If the field width is less than g, the w characters will appear left justified with blank fill for A fields and right justified with zero fill for R fields.

If the field width specified for A output is greater than g, the external output field will consist of w-g blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g the external output field will consist of the leftmost w characters from the internal representation.

If the field width specified for R output is greater than g, the external output field will consist of w-g blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g the external output field will consist of the rightmost w characters from the internal representation.

7.3.9 Blank Field Descriptor

The field descriptor for blanks is nX. On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10 Column Selection Control

• The descriptor Tn is used to skip to a specific column. The column pointer is skipped to column n, and the next format descriptor is then processed.

n may be any unsigned integer, maximum of 137. If n=zero, column 1 is assigned.

The output line image is blanked prior to actual formulation of a line. The T does no blanking, so that with its use, theorder of a list need not be the same as the printed page or card input. Using card input, if n > 80, the column pointer will be moved to column n but a succeeding specification will read only blanks.

7.2.3.11 Format Specification in Arrays

Any of the formatted input/output statements may contain an array name in place of the reference to a FORMAT statement label. At the time an array is referenced in such a manner, the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification.

The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, begins with a left parenthesis and ends with a right parenthesis. As an extension to ASA H field descriptors may be included in formats so defined. An nH field descriptor may be part of a format specification within an array.

The format specification may be inserted in the array by use of a data initialization statement, or by use of a READ statement together with an A format.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement function, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. There are two categories of subprograms: procedure subprograms and specification subprograms. Function subprograms and subroutine subprograms are classified as procedure subprograms. Block data subprograms are classified as specification subprograms. Type rules for function procedures are given in 5.3. All functions and subroutines are required to have less than 64 parameters.

8.1 STATEMENT FUNCTIONS

A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic, masking, or logical assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1 Defining Statement Functions

A statement function is defined by a statement of the form:

$f\ (a_1, a_2, \ldots, a_n) = e$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1. The a's are distinct variable names, called the dummy arguments of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Non-Hollerith constants.

(2) Variable or array element references.

(3) Intrinsic function references.

(4) References to previously defined statement functions.

(5) External function references.

8.1.2 Referencing Statement Functions

A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2 INTRINSIC FUNCTIONS AND THEIR REFERENCE

The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD and MOD are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 EXTERNAL FUNCTIONS

An external function is defined externally to the program unit that references it. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 Defining Function Subprograms

A FUNCTION statement is of the form:

t FUNCTION f (a_1, a_2, ..., a_n)

where:

(1) t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, or is empty. The word "PRECISION" is optional.

(2) f is the symbolic name of the function to be defined.

(3) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in the subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any non-executable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON or DATA statement in the function subprogram.

(4) The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(5) The function subprogram may contain any statement except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

• (6) If flow reaches the END line a RETURN will be implied.

8.3.2 Referencing External Functions

An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.
(5) The name of an external procedure.
• (6) A Hollerith constant.
• (7) An ECS variable name.
• (8) An ECS array name.
• (9) An ECS array element name.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

TABLE 3. INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering** (see note below)	$a_1 \pmod{a_2}$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max $(a_1,a_2,\ldots)$	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing smallest Value	Min $(a_1,a_2,\ldots)$	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive Difference	a_1 - Min (a_1,a_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain Most significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2 \sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex
*Logical product	$a_1 \wedge a_2$ *	2	AND	any type	Octal
*Logical sum	$a_1 \vee a_2$ *	2	OR	any type	Octal
*Complement	$-a_1$ *	1	COMPL	any type	Octal
*Shift	shift a_1 $\lvert a_2 \rvert$* bit positions: left circular if a_2 is positive; right with sign extension and end off if a_2 is negative. If a_2 is not constant, and $a_2 < 0$, and $\lvert a_2 \rvert > 63$, then the result is +0.	2	SHIFT	a_1: non-logical a_2: integer	Octal
*Random Number generation		1	RANF	(dummy)	Real

** The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

* If an argument is double precision or complex, only the high order or real part will be used.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3 Basic External Functions

The external functions listed in Table 4 are supplied. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Function
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e$ (a)	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10}$(a)	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	sin (a)	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	cos (a)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	tanh (a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Comples	Complex
Arctangent	arctan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	a_1 (mod a_2)	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real
Arccosine	arccos (a)	1	ACOS	Real	Real
Arcsine	arcsin (a)	1	ASIN	Real	Real
Trigonometric	tan (a)	1	TAN	Real	Real

*The function DMOD (a_1,a_2) is defined as $a_1 - (a_1/a_2)a_2$, where (x) is the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

8.4 SUBROUTINE

An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

8.4.1 Defining Subroutine Subprograms

A SUBROUTINE statement is of one of the forms:

SUBROUTINE s $(a_1,a_2,\ldots a_n)$

or

SUBROUTINE s

or

SUBROUTINE s $(a_1,a_2,\ldots a_n)$, RETURNS $(b_1,b_2,\ldots b_m)$

or

SUBROUTINE s, RETURNS $(b_1,b_2,\ldots b_m)$

where:

(1) s is the symbolic name of the subroutine to be defined.

(2) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

(3) The b's are variable names.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram need not contain at least one RETURN statement if the procedure is completed upon executing the END statement.

(6) The variable names $b_1, b_2, \ldots b_m$ may not be defined in a subroutine.

8.4.2.1 ENTRY Statement

An ENTRY statement is of the form:

ENTRY s

where s is an alternate entry point to a FUNCTION or SUBROUTINE. s may appear within the subprogram only in the ENTRY statement. Reference to the entry point is the same as described in Sections 8.3.2 and 8.4.2.

8.4.2 Referencing Subroutines

A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. The use of a Hollerith constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

(1) A Hollerith constant.
(2) A variable name.
(3) An array element name.
(4) An array name.
(5) Any other expression.
(6) The name of an external procedure.
(7) An ECS variable name.
(8) An ECS array element name.
(9) An ECS array name.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (5) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.5 BLOCK DATA SUBPROGRAM

A BLOCK DATA statement is of the form:

```
BLOCK DATA
    or
BLOCK DATA  d
```

where d is the symbolic name of the BLOCK DATA subprogram to be defined.

The d is necessary if the BLOCK DATA subprogram is to be included in a SEGMENT as defined for SCOPE 3.0.

This statement may only appear as the first statement of specification subprograms that are called block data subprograms, and that are used to enter initial values into elements of labeled common blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If an entry of a given common block is being given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values may be entered into more than one block in a single subprogram.

8.6 PROGRAM LINE

A program line has the form:

PROGRAM s (f_1, f_2,...f_n)

s is the symbolic name of the main program.

The parameters f_i are the names of all input/output files required by the main program and its subprograms. Although these arguments may be changed at execution time, they must, at compile time, satisfy the following conditions:

1. The file name INPUT must appear if any READ statements of the form READ f,k is included in the program or its subprograms.

2. The file name OUTPUT must appear if any PRINT statement is included in the program or its subprograms.

3. The file name PUNCH must appear if any PUNCH statement is included in the program or its subprograms.

4. The file name TAPE i, with i an integer constant in the range $1 \leq i \leq 99$, must appear if a READ (i,n), WRITE (i,n), READ (i), or WRITE (i), statement is included in the program or subprograms. Further, if i is a variable, there must be a file name TAPE i for each value i may assume.

5. Parameters for files which are used for Mass Storage, must, in addition to complying with the above conditions, specify the maximum number of records to be read or written on the file. The form of the entry is TAPE i (n), where n is the maximum number of records on file i in decimal.

   Files may be equivalenced at compile time. For example,

   (INPUT, OUTPUT, TAPE1 = INPUT, TAPE2 = OUTPUT)

   would cause all input normally provided by TAPE 1 to be extracted from INPUT and all listable output normally recorded on TAPE 2 to be transmistted to the OUTPUT file. Equivalenced file names must follow, in the list of parameters, those to which they are made equivalent. Their corresponding parameter positions may not be changed at the time the program is executed, although the names of the files to which they are made equivalent may be changed at this time.

   File buffers may be assigned a non-standard size at compile time. If no parameter is given in the Program Line, a buffer size of 2012B is assumed. An example of the form of the Program Line to change the buffer size is (OUTPUT=4000,TAPE4=OUTPUT). If the buffer is explicitly assigned a length the assignment must appear with the first reference to the file on the program card. The length will be interpreted as an octal number.

9. PROGRAMS

An executable program is a collection of statements, comment lines, a program line and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 PROGRAM COMPONENTS

Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 Program Part

A program part must contain at least one executable statement and may contain FORMAT statements, NAMELIST statements, and data initialization statements. It need not contain any statements from the latter three classes of statement. This collection of statements may optionally be preceded by statement function definitions, data initialization statements, and FORMAT statements. As before only some or none of these need be present.

9.1.2 Program Body

A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3 Subprogram

A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body, or is a block data subprogram.

9.1.4 Block Data Subprogram

A block data subprogram consists of a BLOCK DATA statement, followed by the appropriate (8.5) specification statements, followed by data initialization statements, followed by an end line.

9.1.5 Main Program

A main program consists of a program body optionally preceded by a program line.

9.1.6 Executable Program

An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 Program Unit

A program unit is a main program or a subprogram.

9.2 NORMAL EXECUTION SEQUENCE

When an executable program begins operation, execution commences with the execution of the first executable statement of the main program.

A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO completion of execution of that statement causes execution following execution of any of these statements is described in Section 7. A program part may not (in the sense of 9.1.1) contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA - AND INTERPROGRAM RELATIONSHIPS

10.1 SYMBOLIC NAMES

A symbolic name has been defined to consist of from one to seven alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g., GO TO, READ, FORMAT, etc. are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I	An array and the elements of that array.
Class II	A variable.
Class III	A statement function.
Class IV	An intrinsic function.
Class V	An external function.
Class VI	A subroutine.
Class VII	An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.
Class VIII	A block name.
Class IX	A NAMELIST name.
Class X	An ECS variable, array or elements of the array.
Class XI	A RETURNS name.

10.1.1 Restrictions on Class

A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI , VII, VIII, in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program, a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing. A symbolic name, once defined to be in Class XI may only be subsequently referred to in a RETURN statement.

10.1.2 Implications of Mentions in Specification and Data Statements

A symbolic name is in Class I if and only if it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I, Class X, or in Class II but not Class V. (8.3.1) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not in Class V. (8.3.1).

A symbolic name that appears in a type-statement cannot be in Class VI, VII or IX. Only one such appearance for a symbolic name in a program unit is permitted.

A Symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a DATA statement is in either Class I or in Class II but not Class V. (8.3.1) In an executable program, a storage unit (7.2.1.3.1) may have its value initialized one time at the most.

10.1.3 Array and Array Element

In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.

(2) In a list of dummy arguments.

(3) In the list of actual arguments in a reference to an external procedure.

(4) In a COMMON statement.

(5) In a type-statement.

(6) In the extended form of the DATA statement.

(7) Where subscripts are implied. (cf Sec. 5.1.3.2)

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name, the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

10.1.4 External Procedures

The only case when a symbolic name is in Class VII occurs when the name appears in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN or END statement in that procedure having intervened.

10.1.5 Subroutine

A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

(3) Immediately following the word ENTRY in an ENTRY statement when the ENTRY statement appears in a subroutine subprogram.

10.1.6 Statement Function

A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I..

(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.

(3) A function defining statement (8.1.1) is present for that symbolic name.

10.1.7 Intrinsic Function

A symbolic name is Class IV in a program unit if and only if it meets all four of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I or Class III.

(2) The symbolic name appears in the name column of the table in Section 8.2.

(3) The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.

(4) Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parenthesis.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 External Function

A symbolic name is in Class V if it:

(1) Appears immediately following the word FUNCTION in a FUNCTION statement.

(2) Is not in Class I, Class III, Class IV or Class VI and appears

immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, or an actual argument. There must be at least one such appearance in the program unit in which it is so used.

(3) Appears immediately following the word ENTRY in an ENTRY statement when the ENTRY statement appears in a function subprogram.

10.1.9 Variable

In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

(1) It is not in Class VI or Class VII.

(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

(3) It occurs other than in a Class VIII appearance.

10.1.10 Block Name

A symbolic name is in Class VIII if and only if it is used as a block name in a COMMON statement.

10.1.11 Namelist Name

A symbolic name is in Class IX if and only if it is used as a namelist name in a NAMELIST statement.

10.1.12 ECS Element

A symbolic name is in Class X if and only if it appears in a TYPE ECS statement.

10.1.13 RETURNS Name

A symbol is a RETURNS name if and only if it appears in a RETURNS list.

10.2 DEFINITION

There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variable only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other variables of definition that should be noted. The first, effected by GO TO assignment and referring to an integer variable being defined with other than integer value, is discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1 Definition of Procedures

If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 Associations that Effect Definition

Entities may become associated by:

(1) COMMON association.
(2) EQUIVALENCE association.
(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with

the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements association by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of a different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type, so the rule in this case is that an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3 Events That Effect Definition

Variables and array elements become initially defined if and only if their names are associated in a data initialization statement with a constant of the same type as the variable or array in question. Any entity not initially defined is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1, and 7.2.1.1.2) or certain entities in subprograms (6.5, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic, masking, or logical statement causes definition of the entity that precedes the equals.

(2) As execution of an input statement proceeds each entity which is assigned a value of its corresponding type from the input medium is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.

(3) Certain entities in function subprograms (10.2.9) become undefined.

(4) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(5) When an associated entity of different types becomes defined.

(6) When an associated entity of the same type becomes undefined.

10.2.4 Entities in Blank Common.

Entities in blank common and those entities associated with them may not be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

10.2.5 Entities in Labeled Common.

Entities in labeled common or any associates of those entities may be initially defined.

A program unit contains a labeled common block name if the name appears as a block name in the program unit. If a main program or referenced subprogram contains a labeled common block name, any entity in the block (and its associates) once defined remain defined

until they become undefined.

It should be noted that redefinition of an initially defined entity will allow later undefinition of that entity. Specifically, if a subprogram contains a labeled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block(and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6 Entities Not in COMMON.

An entity not in common except for a dummy argument or the value of a function may be initially defined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have not been redefined or become undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

Again, it should be emphasized, the redefinition of an initially defined entity can result in a subsequent undefinition of that entity.

10.2.7 Basic Block.

In a program unit, a basic block is a group of one or more executable statements defined as follows:

(1) DO statement.

(2) CALL statement.

(3) GO TO statement of all types.

(4) Arithmetic IF statement.

(5) STOP statement.

(6) RETURN statement.

(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.

(9) A READ statement with an integer variable in the list.

(10) A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

(1) The first executable statement of a program unit.

(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.8 Second Level Definition

Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over the range.

10.2.9 Certain Entities in Function Subprograms

If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.5), and if this factor contains a function reference, then all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3 DEFINITION REQUIREMENTS FOR USE OF ENTITIES

Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO -implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A or R or O. If the output is under control of a format specification, a correct association of conversion code with type of entity is required unless the conversion code is A or R or O. The following are the correct associations: I with integer; D with double precision; E, F, and G with real and complex and L with logical.

APPENDIX A

PARTIAL LIST OF ERROR MESSAGES

(1) A previous DO terminates on this DO statement.

(2) More storage required by DO statement processor.

(3) A constant parameter of a DO exceeds 131K.

(4) Parameter of a DO or DO implying loop must be unsigned integer.

(5) The initial value of a DO or DO implying loop must not exceed upper bound if both are constant.

(6) The control variable of a DO or DO implying loop must be a simple integer variable.

(7) ENTRY statement occuring in DO nest.

(8) A DO loop may not terminate on this type of a statement.

(9) A DO loop which terminates at this statement includes an unterminated DO.

(10) The terminal statement of this DO precedes it.

(11) The terminal label of a DO must be an integer constant.

(12) The increment of a DO or DO implying loop is zero.

(13) Attempt to redefine a DO parameter or a control variable of a DO within its range.

(14) Too many subscript indices.

(15) Adjacent commas in parameter string.

(16) No left parens after array name.

(17) No right parens in subscript.

(18) Left/right parens not matching.

(19) Subscripted variable not previously dimensioned.

(20) Illegal sequence or use of operators.

(21) Illegal or missing operator.

(22) Illegal replacement in arithmetic statement.

(23) Operator missing before or after parens.

(24) Equal-sign appears in expression.

(25) Two or more relational operators in the same relational sub-expression.

(26) Logical expression incorrectly formed.

(27) Relational subexpression incorrectly formed.

(28) The .NOT. operator must be followed either by an operand or (.

(29) Logical connections must be followed by (or an operand.

(30) Logical subexpression begins with an operator.

(31) Excess left parenthesis in logical expression.

(32) Function called incorrectly.

(33) Masking expression incorrectly formed.

(34) The first element of logical expression not an operand, (, or .NOT.

(35) Operators .AND., .OR., not followed by either .NOT., or an operand.

(36) Replacement variable for a logical expression not type logical.

(37) Illegal use of logical or relational operator.

(38) Right paren preceded by comma.

(39) Left paren followed by a comma.

(40) Empty parenthetical expression.

(41) ASF parameters do not agree in number.

(42) Arithmetic expression too long for compiler tables.

(43) Unreferenced parameter (s) in ASF.

(44) No right paren following parameter list.

(45) No = after parameter list in ASF.

(46) Parameter (s) missing or too many commas.

(47) ASF name multiply defined.

(48) Empty parameter list.

(49) Multiply defined dummy parameter.

(50) Illegal statement label.

(51) Logical IF is formed incorrectly.

(52) Adjacent commas.

(53) Right paren preceded by comma.

(54) Left paren followed by comma.

(55) Left/right parens not matching.

(56) An entry in a RETURNS list is not a statement label.

(57) I is not an integer variable.

(58) k is not a statement label.

(59) Variable or arithmetic expression of type other than integer in computed or assigned GO TO statements.

(60) Other than statement label is contained in assigned GOTO list.

APPENDIX "B"
CONTROL CARD

CONTROL CARD FORMAT

The control card which calls for the compilation of a FORTRAN source program consists of the characters FTN optionally followed by a parameter list enclosed in parentheses. If no parameter list is given, a period must follow the characters FTN. The card columns following the right parenthesis may be used for comments and are transcribed to the DAYFILE. If no parameter list is given, a period must be used to separate the characters FTN from any comments which appear on the card. Blank columns may be used anywhere for readability and will be squeezed out and ignored by the system. The formats for the control card are given below.

FTN (P_1, P_2, P_3, P_4, P_5) comments

FTN. comments

CONTROL CARD PARAMETERS

SOURCE INPUT PARAMETER

If no source input parameter is present the FORTRAN source input will be assumed on INPUT. If the source input is on some file other than INPUT, a source input parameter of the following form must be provided:

I=fn

where fn is the file name where the source input appears.

Source input parameters of the forms I=INPUT, and I, are equivalent to not providing a source input parameter.

BINARY OUTPUT PARAMETER

If no binary output parameter is present a relocatable binary file will be

written on a file named LGO. If binary output is instead desired on some other file, a binary output parameter of the following form must be provided.

B=fn

where fn is the file name on which the binary output is to be written. An fn of 0 will cause the suppression of binary output. Binary output parameters of the form B=LGO, or B, are equivalent to not providing a binary output parameter.

LIST PARAMETER

If no parameter is provided a listing hereinafter referred to as "normal listing" including source language and major diagnostics will be provided on OUTPUT. Other list options may be selected by use of one of the following parameters.

L=fn

where L may be one of the following:

L normal listing.

X listing of diagnostics which indicate non-ASA usage.

R Assembler cross reference table

O listing of generated object code

fn file name on which list output is to be written. An fn of 0 will cause suppression of all list output. = fn if not present will cause the list option to be on OUTPUT.

Any combination of the above parameters provides the features indicated.

OPTIMIZATION PARAMETER

If no parameter is present the second level of optimization will be assumed.

An F on the control card indicates the first or lower level of optimization is requested.

ERROR TRACEBACK AND CALLING SEQUENCE PARAMETER

If a parameter is present, calls to library functions will be made by the call by name sequence and maximum error checking will be done. Full error traceback will be done if an error is detected.

The omission of the T parameter on the control card, will cause the compiler to generate calls to library functions using the call by value sequence (SA1 parameter, RJ function). Minimum error checking will be done and no traceback will be provided in the case of error detection. A significant saving in memory space and execution time will be realized. This mode of compilation is not intended for use when programs are in the debug stages.

EXAMPLES

```
FTN  (LR)
```

This control card calls for compilation with normal listing plus the assembler cross reference table on OUTPUT, relocatable binary on LGO, source input from INPUT.

```
FTN (I=MYTAPE, B=0, LO)
```

This control card calls for compilation with no binary output, normal listing plus object code listing on OUTPUT and source input from the file named MYTAPE.

APPENDIX "C"

LIBRARY SUBPROGRAMS

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering* (see note below)	a_1 (mod a_2)	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max (a_1, a_2,...)	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing smallest value	Min (a_1,a_2,...)	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive Difference	a_1 - Min (a_1,a_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain Most significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real

*The function MOD or AMOD (a_1,a_2) is defined as $a_1 - [a_1/a_2]\, a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2 \sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex
Shift	Shift a_1 (only the first word if a double word element) a_2 bit positions: left circular if a_2 is positive; right with sign extension if a_2 is negative	2	SHIFT	a_1: Non-logical a_2: Integer	Octal
Logical Product	$a_1 \wedge a_2$	2	AND	Single word	Octal
Logical Sum	$a_1 \vee a_2$	2	OR	Single word	Octal
Complement	a	1	COMPL	Single Word	Octal

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e$ (a)	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10}$ (a)	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	sin (a)	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	cos (a)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	tanh (a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Arctangent	arctan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	a_1 (mod a_2)	2	DMOD	Double	Double
Modulus	$\sqrt{AIMAG^2(a)+REAL^2(a)}$	1	CABS	Complex	Real
Arccosine	Arccos (a)	1	ACOS	Real	Real
Arcsine	arcsin (a)	1	ASIN	Real	Real
Trigonometric Tangent	tan (a)	1	TAN	Real	Real
Random Number Generator	ranf (a)	1	RANF	Dummy	Real

*The function DMOD (a_1,a_2) is defined as $a_1 - [a_1/a_2]\,a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Address of argument a	loc (a)	1	LOCF	Symbolic	Integer
I/O status on buffer unit	-1. unit ready no error +0. unit ready, EOF encountered +1. unit ready, parity error encountered	1	UNIT	Integer	Real
I/O status on non-buffer unit	=0 no end of file encountered in previous read	1	EOF	Integer	Integer
LENGTH	Number of central memory words read on the previous I/O request for a particular file	1	LENGTH	Integer	Integer
Variable Characteristic	-1 = indefinite +1 = out of range 0 = Normal	1	LEGVAR	Real	Integer
Parity status on non-buffer unit	0 = no parity error on previous read	1	IOCHEC	Integer	Integer

Subroutine	Definition	Number of Arguments	Symbolic Name	Type of Argument
Set Sense Light	$1 \leq i \leq 6$ turn sense light i on i = 0 turn off all sense lights	1	SLITE	Integer
Test Sense Light	SLITET (i,j) If sense light i is on j = 1 If sense light i is off j = 2 Always turn sense light i off	2	SLITET	Integer
Test Sense Switch	SSWTCH (i,j) If sense switch i is down j = 1 If sense switch i is up j = 2	2	SSWTCH	Integer
Terminate	Terminate program execution and return control to the monitor	0	EXIT	
Console Comment	Place a message of up to 40 characters on the dayfile	1	REMARK	Hollerith
Console Value	Display the value of an argument in the dayfile	2	DISPLA	a_1=Hollerith a_1=real or integer
Random Access Positioning	Position disc heads to reference the next record in a random access file a_1 - unit number a_2 - record number	2	LOCATE	Integer

Subroutine	Definition	Number of Arguments	Symbolic Name	Type of Argument
Ranget (a)	obtain current generative value of RANF between 0 and 1	1	RANGET	Symbolic
Ranset (a)	Initialize generative value of RANF	1	RANSET	Real
Time Used	obtain CPU time used in seconds to the nearest milli-second	1	SECOND	Real

APPENDIX "D"

Intermixed COMPASS Subprograms

Subprograms written in COMPASS may be intermixed with FORTRAN coded subprograms in the source deck. COMPASS subprograms must begin with an IDENT card with the word IDENT punched in columns 11-15, and terminate with an END card with the word END punched in columns 11-13. Columns 1-10, and 14 must be blank.

Calling Sequence

When the FORTRAN compiler encounters a reference to an external subprogram, subroutine or function the following calling sequence is generated:

```
SA1     Argument List (if there is a parameter list)
RJ      Subprogram Name
```

where the argument list consists of consecutive words of the form:

```
VFD 60/Argument_i
```

followed by a zero word.

Returning

The COMPASS subprogram is responsible for preserving the contents of A0 in A0 upon returning control to the subprogram which called it. If the COMPASS subprogram was entered via a function reference the result of that function must be in X6 or X6 and X7 with the least significant or imaginary part of the double precision or complex result appearing in X7.

APPENDIX "E"

OPTIMIZATIONS

The following is a list of optimizations which will be carried out if the user selects the second level of optimization, although those followed by an asterisk will be performed if the first level of optimization is selected.

(1) Elimination of redundant operations where possible.

(2) Evaluation of array element address by the index function method.*

(3) Critical path analysis of instruction sequences to maximize parallel operation.

(4) Reformation of subexpressions to permit extended parallelism.*

(5) Evaluation of constant subexpressions at compile time.*

(6) Determination for each reference to a formal parameter at compile time as to whether it should be referenced by address substitution indirect addressing.*

(7) Elimination of common remote parameter lists.

(8) It will be noted whether or not an innermost DO loop fits in the stack and the resultant code will reflect this change in timing.

(9) Simple constants will be formed by sets rather than loads.*

(10) Branches to the next instruction will be eliminated.*

(11) Presetting of arrays with a constant pattern will be specially handled at load time to avoid the generation of a large binary deck.*

(12) Inline evaluation of some functions.*

(13) Branch evaluation of relationals.*

(14) Temporary evaluation of subscripts within DO loops.

APPENDIX "F"
STATEMENT FORMS

Subprogram Statements

Entry Points.

PROGRAM s

PROGRAM s (f_1, f_2,...,f_n)

SUBROUTINE s

SUBROUTINE s (a_1,a_2,...,a_n)

SUBROUTINE s, RETURNS (b_1,b_2,...,b_m)

SUBROUTINE s (a_1,a_2,...,a_n), RETURNS (b_1,b_2,...,b_m)

FUNCTION f (a_1,a_2,...,a_n)

REAL FUNCTION f (a_1,a_2,...,a_n)

DOUBLE FUNCTION f (a_1,a_2,...,a_n)

COMPLEX FUNCTION f (a_1,a_2,...,a_n)

INTEGER FUNCTION f (a_1,a_2,...,a_n)

LOGICAL FUNCTION f (a_1,a_2,...,a_n)

DOUBLE PRECISION FUNCTION f (a_1,a_2,...,a_n)

ENTRY s

Specification Program Declaration

BLOCK DATA

BLOCK DATA d

Inter-subroutine

EXTERNAL v_1, v_2,...,v_n

Inter-subroutine Transfer Statements

CALL s

CALL s (a_1,a_2,...,a_n)

CALL s, RETURNS (b_1,b_2,...,b_m)

CALL s (a_1,a_2,...,a_n), RETURNS (b_1,b_2,...,b_m)

RETURN

RETURN a

Data Declaration and Storage Allocation

Type Declaration

REAL $v_1,v_2,\ldots,v_n$

DOUBLE $v_1,v_2,\ldots,v_n$

COMPLEX $v_1,v_2,\ldots,v_n$

INTEGER $v_1,v_2,\ldots,v_n$

LOGICAL $v_1,v_2,\ldots,v_n$

DOUBLE PRECISION $v_1,v_2,\ldots,v_n$
ECS $v_1, v_2, \ldots v_n$
TYPE REAL $v_1,v_2,\ldots,v_n$

TYPE DOUBLE $v_1,v_2,\ldots,v_n$

TYPE COMPLEX $v_1,v_2,\ldots,v_n$

TYPE INTEGER $v_1,v_2,\ldots,v_n$

TYPE LOGICAL $v_1,v_2,\ldots,v_n$

TYPE DOUBLE PRECISION $v_1,v_2,\ldots,v_n$
TYPE ECS $v_1, v_2, \ldots v_n$

Storage Allocation

DIMENSION $v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$
COMMON $/x_1/a_1/\ldots/x_n/a_n$

EQUIVALENCE $(k_1), (k_2),\ldots,(k_n)$
DATA $k_1/d_1/, k_2/d_2/,\ldots, k_n/d_n/$
DATA $(r_1=d_1), (r_2=d_2),\ldots, (r_n=d_n)$

Statement Function

$f(a_1,a_2,\ldots,a_n) = e$

Symbol Manipulation and Control

Replacement Statements

v=e { Arithmetic
Logical
Masking

Intra-program Transfers

GO TO k

GO TO i, $(k_1,k_2,\ldots,k_n)$

GO TO $(k_1,k_2,\ldots,k_n)$, i

IF (e) k_1,k_2,k_3

IF (e) k_1,k_2

IF (e) s

Loop Control

DO n i = m_1,m_2,m_3

Miscellaneous Program Controls

ASSIGN k TO i

CONTINUE

PAUSE

PAUSE n

STOP

STOP n

Input/Output

I/O Format

FORMAT $(q_1t_1z_1t_2z_2\ldots t_nz_nq_2)$

I/O Control Statements

READ f,k

READ (u) k

READ (u)

READ (u,f) k

READ (u,f)

WRITE (u) k

WRITE (u,f)

WRITE (u,f) k

PRINT f,k

PUNCH f,k

BUFFER IN (u,k) (A,B)

BUFFER OUT (u,k) (A,B)

Internal Manipulation

ENCODE (n,f,A) k
DECODE (n,f,A) k

Tape Handling

ENDFILE u
REWIND u
BACKSPACE u

Miscellaneous

NAMELIST $/y_1/a_1/y_2/a_2/.../y_n/a_n$

Program Termination

END

APPENDIX "G"

SYSTEM ROUTINE SPECIFICATIONS

The SYSTEM routine has the capability of handling error tracing, diagnostic printing, termination of output buffers, and transfer to specified non-standard error procedures. All the FORTRAN mathematical routines rely on SYSTEM to complete the above mentioned tasks. A FORTRAN coded routine also has the capability of calling SYSTEM. Any of the parameters used by SYSTEM relating to a specific error may be changed by a user routine during execution. The END processor also makes use of SYSTEM to dump the output buffers and print an error summary. Since the initialization routine (Q8NTRY), the end processors (END, STOP, and EXIT), and SYSTEM must always be available, these routines are combined into one with multiple entry points.

The calling sequence to SYSTEM passes the error number as the first parameter and an error message as the SO card parameter. Therefore, one error number may have several different messages associated with it. The error summary given at the termination of the program lists the total number of times each error number was encountered.

If the error number is zero, this is accepted as a special call only to end the output buffers and return. If no OUTPUT file was defined before SYSTEM was called, then there will be no error printing and an appropriate message will appear in the dayfile. Each line printed is subjected to the line limit of the OUTPUT buffer, so when that limit is exceeded, the job is aborted.

The error table is ordered serially, i.e., the first error corresponds to the error number 1, and is expandable at assembly time. The last entry in the table is a catch-all for any error number which exceeds the table length. An entry in the error table is:

PRINT FREQUENCY	PRINT FREQUENCY INCREMENT	PRINT LIMIT	ERROR DETECTION TOTAL	F/ NF	A/ NA	NON-STANDARD RECOVERY ADDRESS
8	8	12	12	1	1	18

Use of PRINT FREQUENCY:

PRINT FREQUENCY = PF
PRINT FREQUENCY INCREMENT = PFI

1. If PF = 0 and PFI = 0, the diagnostic and trace back information are never listed.
2. If PF = 0 and PFI = 1, the diagnostic and trace back information are always listed until the print limit is reached.
3. If PF = 0 and PFI = n, the diagnostic and trace back information are listed only the first n times unless the print limit is reached first.
4. If PF = n, the diagnostic and trace back information are listed every n^{th} time until the print limit is reached.

Use of FATAL (F) / non-FATAL (NF):

1. If the error is non-fatal and no non-standard recovery address is specified, then the error messages are printed according to PRINT FREQUENCY and control is returned to the calling routine.
2. If the error is fatal and no non-standard recovery address is specified, then the error messages are printed according to PRINT FREQUENCY, an error summary is listed, all the output buffers are terminated, and the job is aborted.
3. If a non-standard recovery address is specified, see NON-STANDARD RECOVERY.

Use of NON-STANDARD RECOVERY:
SYSTEM will supply the non-standard recovery routine with the following information:

A1 address of parameter list passed to the routine which detected the error.

X1 first element of above parameter list e.g., the address of the first parameter.

A0 address of parameter list of the routine which called the routine which detected the error.

B1 address of a secondary parameter list. This list contains, in successive words:

1. The error number passed in SYSTEM .
2. address of the diagnostic word available to SYSTEM.
3. address within an auxiliary table if A/NA bit is set, otherwise zero.
4. instruction consisting of the RJ to SYSTEM in the upper 30 bits and trace back information in the lower 30 bits for the routine which called SYSTEM.

FORTRAN-coded routines will be unable to make use of the information in the secondary parameter list.

1. Non-fatal error:
The routine which detected the error and SYSTEM are delinked from the calling chain and the non-standard recovery routine is entered. When this routine exits in the normal fashion, control will return to the routine which called the routine which detected the error.

Thus, any faulty arguments may be corrected, and the recovery routine is allowed to call the routine which detected the error, providing corrected arguments. By not correcting the faulty arguments in the recovery routine, a three routine loop could develop between the routine which detects the error, SYSTEM, and the recovery routine. No checking is done for this case.

2. Fatal Error:
SYSTEM calls the non-standard recovery routine in the normal fashion, with the registers set as above. If the non-standard recovery routine exits in the normal fashion, thus returning control to SYSTEM the non-standard recovery routine is free, of course, to transfer to some other point and continue computation.

Use of the A/NA bit:
The A/NA bit is for use only when a non-standard recovery address is specified.

If this bit is set, then the address within an auxiliary table is passed in the third word of the secondary parameter list to the recovery routine. This bit allows more information than is normally supplied by SYSTEM to be passed to the recovery routine. Only during assembly of SYSTEM may this bit be set, because an entry must also be made into the auxiliary table. Each word in the auxiliary table must have the error number in its upper 10 bits so that the address of the first error number match is passed to the recovery routine. An entry in the auxiliary table for an error is not limited to any specific number of words.

The traceback information is terminated as soon as one of the following three variables is detected:

1) the calling routine is a program.
2) the maximum trace back limit is reached.
3) no trace back information is supplied.

In order to change an error table during execution a FORTRAN type call is made to SYSTEMC with the following parameters:

1) error number
2) list containing the consecutive locations:
   Word 1 - fatal/non-fatal (fatal - 1, non-fatal = 0)
   Word 2 - print frequency
   Word 3 - print frequency increment (only significant if word 2 =0)
            special values:
            word 2 = 0, word 3 = 0 never list error
            word 2 = 0, word 3 = 1 always list error
            word 2 = 0, word 3 = X list error only the first X times
   Word 4 - print limit
   Word 5 - non-standard recovery address
   Word 6 - maximum trace back limit

If any word within the parameter list is negative, then the value already in table entry will not be altered.

(The auxiliary table bit may only be set during assembly of SYSTEM so only then can an auxiliary table entry be made).

ERROR LISTING:

I. "message supplied by calling routine"
ERROR NUMBER XXXX DETECTED BY ZZZZZZZ at YYYYYY
CALLED FROM CCCCCC at WWWWWW
.
.
.
where the addresses are absolute and error number is decimal

where ZZZZZZZ and CCCCCCC are routine names, YYYYYY and WWWWWW are absolute addresses

II. ERROR SUMMARY

XXXXX ERROR YYYY TIMES
.
.
.
where all numbers are decimal

III. NO OUTPUT FILE FOUND

IV. OUTPUT FILE LINE LIMIT EXCEEDED

Functions of entry points:

1) Q8NTRY - initialize I/O buffer parameters
2) STOP - enter "STOP" in the dayfile and begin END processing
3) EXIT - enter "EXIT" in the dayfile and begin END processing
4) END - terminate all output buffers, print an error summary, transfer control to the main overlay if within an overlay or in any other case exit to monitor.

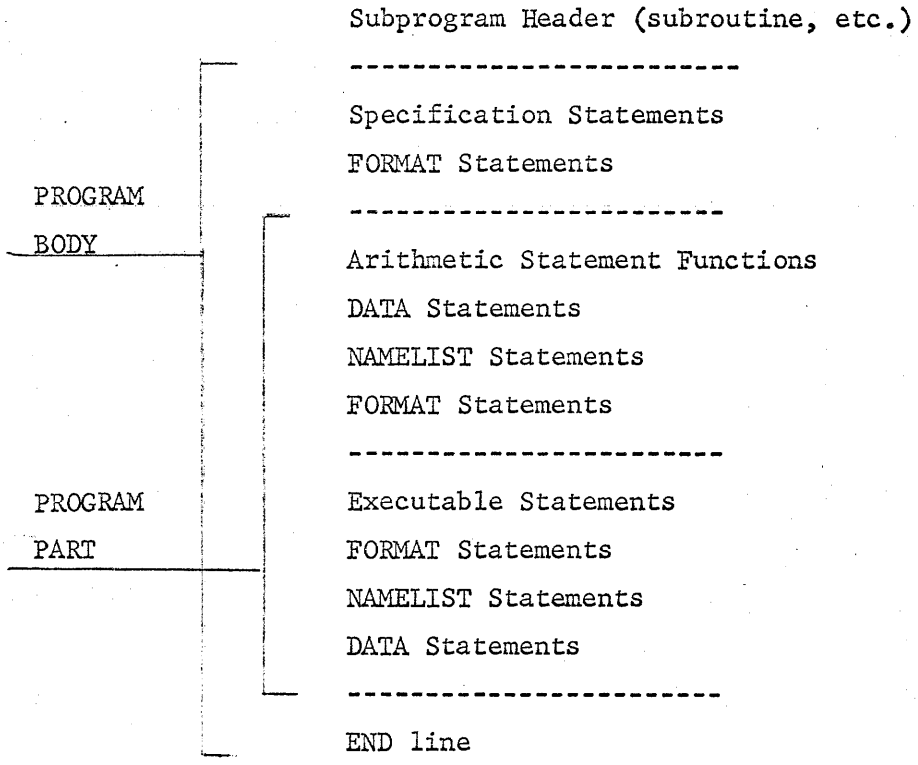
5) SYSTEM - handles error tracing, diangostic printing, termination of output buffers, and either transfers or specified non-standard error recovery address, aborts the job, or returns to calling routine depending on type of error.

6) SYSTEMC - changes entry to SYSTEM's error table according to arguments passed.

APPENDIX "H"

SUBPROGRAM STRUCTURE

The table below shows the general form of a FORTRAN subprogram. The statements within a group may appear in any order; however, the order of groups must be as shown. Comment lines may appear anywhere within the body.



Arrangement of code and data within PROGRAM, SUBROUTINE and FUNCTION subprograms.

SUBROUTINE and FUNCTION Structure

The code within procedure subprograms is arranged in the following blocks {relocation bases} in the given order.

START. The code for the primary entry and the saving of A0.

VARDIM. The address substitution code and the variable dimension initialization code.

ENTRY. Either a full word of N0's or nothing.

CODE. The code generated by compiling executable statements followed by parameter lists for external procedure references within the current procedure and storage for statement, D0 and optimizing temporary use.

DATA. Storage for usage declared variables, FORMAT statements and program constants.

DATA.. Storage for local dimensioned variables.

H0L. Storage for Hollerith constants.

Formal Parameters. One local block for each formal parameter in the order in which they appear in the subroutine headercard, to hold tables used in address substitution for processing references to dummy arguments.

Main subprograms consist of the following blocks.

START. The I/O file buffers and a table of file names specified in the program header card.

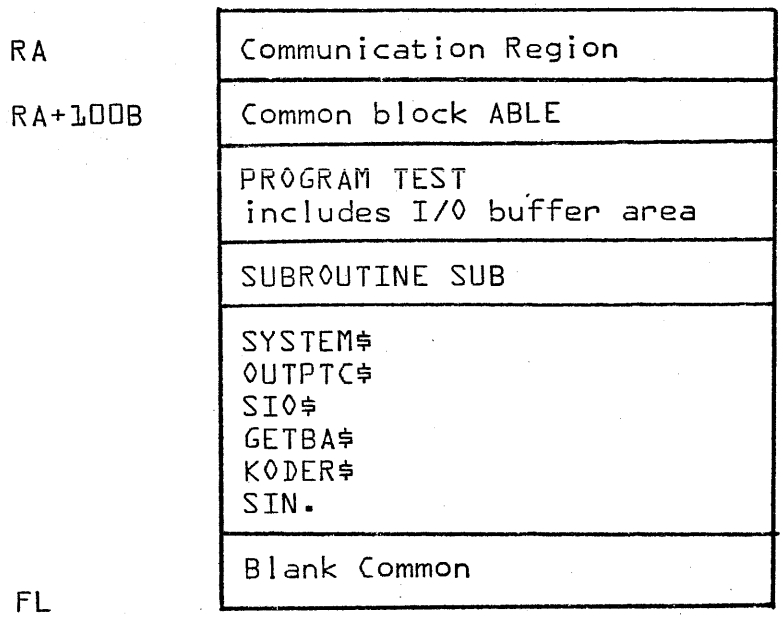
CODE. The transfer address code plus the code specified for the CODE. block above.

DATA. }
DATA.. } Same as above
H0L. }

Memory Structure

Subprograms are loaded as encountered in the input file from RA+100B toward FL. Labeled common blocks are loaded prior to the subprogram in which they first occur. Library routines are loaded immediately after the last encountered subprogram and these are followed by blank common.

The following is a typical memory layout.



APPENDIX "I"
OVERLAYS AND SEGMENTS

I.1 INTRODUCTION

Overlay and segment processing is provided in order to allow the programmer some execution time control over the programs in memory. He may desire this control for any of several reasons, such as:

1. his total program may be too large to fit in memory all at once.
2. he may desire to load, as execution proceeds, new versions of some subprograms and have them serviced by the same statements used to call the old versions.
3. he may wish to save time by only loading certain rarely used sections of program in case they are actually needed. etc.

Segments are groups of subprograms that are loaded from relocatable form when desired, giving the programmer explicit control over inter-program links established. An overlay is a program together with a cluster of subprograms which is converted to absolute form and written on mass storage prior to execution. During execution overlays are called into memory and executed as desired.

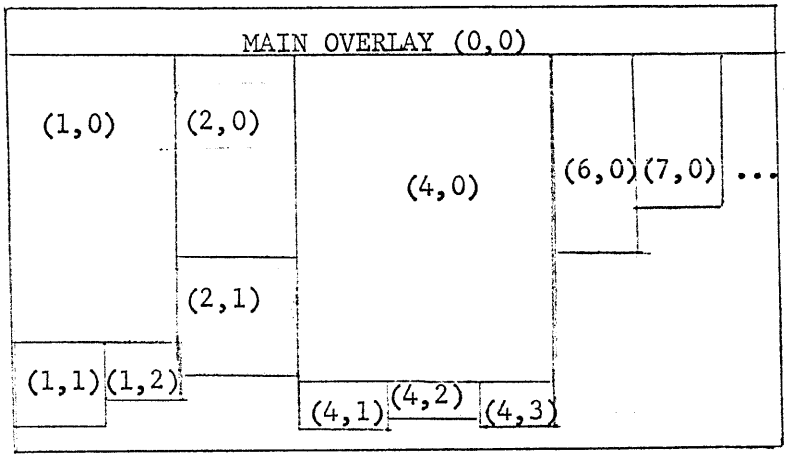
I.2 OVERLAYS

Overlay processing allows programs to be divided into independent parts which may be called and executed as needed. Each part (an overlay) must consist of a single PROGRAM together with any necessary subprograms.

Each overlay is numbered with an ordered pair of numbers (I,J), each of which is in the range $0\text{-}77_8$. The first number, I, denotes the primary level. The second number, J, denotes the secondary level. An overlay with a non-zero secondary level is called a secondary overlay and is associated with and subordinate to the overlay which has the same primary level and a zero secondary level, called the primary overlay. The initial or main overlay which always remains in memory has levels (0,0). The significance of this distinction appears in the order in which overlays are loaded. A secondary overlay may be loaded only from the main overlay or from its associated primary overlay. See example.

Overlay level numbers, (0,1), (0,2), (0,3)... are illegal. Primary overlays are all origined at the same point immediately following the main overlay (0,0). Secondary overlays are origined immediately following the primary overlay. For any given program execution all overlay identifiers must be unique. The loading of any primary overlay will destroy any other primary overlay. For this reason, no primary overlay may load other primary overlays. Secondary overlays may only be loaded by the associated primary overlay or main overlay. There are thus only two levels of overlay available to the programmer.

EXAMPLE:



Overlays (1,1) and (1,2) are secondary to overlay (1,0)
Overlay (2,1) is secondary to overlay (2,0)
Overlay (2,1) may not be called from (1,0) or (1,1) or (1,2) but only from (2,0), or (0,0)

Overlays (1,0), (2,0), (4,0) ... may be called only from the main overlay (0,0)

Overlays are called by

CALL OVERLAY (fn, I,J, p)

OVERLAY is a FORTRAN execution time subroutine which translates the FORTRAN call into a call to the loader.

fn - variable name of a location which contains the name of the file (left justified display code) the overlay is on

I - is the primary level of the overlay

J - is the secondary level of the overlay

p - is the recall parameter. If p exists and equals 6HRECALL, the overlay is not reloaded if it is already in memory.

This call will cause the loading and execution of the referenced overlay (which has previously been absolutized and written on file fn with control cards). When an END statement in main program is encountered, control returns to the statement following the CALL OVERLAY.

I.3 SEGMENTS

Four definitions are needed:

1. Entry point - a named location within a subprogram that can be referenced by another program - created by the SUBROUTINE, FUNCTION and ENTRY statements.
2. External reference - a reference within a program or subprogram to the entry point of some other subprogram - created by explicit CALL statements, function references, I/O statements, implicit functions, etc.
3. Link - the connection established between an external reference and an entry point when the programs are loaded into memory.
4. Unsatisfied external - an external reference for which no matching entry point can be found, and therefore no link established.

A segment is a group of subprograms (possible 1 subprogram) which are loaded together when specified by the programmer. Segments are loaded at "levels"; these levels range from $0\text{-}77_8$. Level zero is reserved for the initial or main segment. Level zero, which must contain a PROGRAM, remains in memory during execution. Segments may be loaded at any level ($0\text{-}77_8$).

When the segment is loaded, external references will be linked to entry points in previously loaded segments (those at a lower level).

Similarly, entry points in the segment will be linked to unsatisfied external references in previously loaded segments. Unsatisfied external references in the segment will remain unsatisfied; subsequent segment loading may include entry points to satisfy the external references. Unsatisfied external references may be satisfied, if possible, from the system library. If a segment is to be loaded at a requested level which is less than or equal to the level of the last loaded segment, all segments will be removed/delinked which are at levels down to and including the requested level. Delinking a segment at a given level requires that the linkage of external references in lower levels to entry points in the delinked segment be destroyed so that the external references are unsatisfied once again. References to labeled COMMON blocks follow the same rules. Maximum blank COMMON length is established in the first segment which declares blank COMMON.

Once the delinking is complete, the segment will be loaded. Only one occurrence of a given subprogram or entry point is necessary since all levels may eventually link to the subprogram. However, a user may force loading of a subprogram by explicitly naming it in another segment at a higher level. Thereafter, all external references in higher levels would be linked to the new version. In this manner, a subprogram and/or entry point can effectively replace an identical one which was already loaded at a lower level. Note, however, once a linkage is established, it is not destroyed unless the segment containing the entry point is removed.

EXAMPLE: The SINE routine is loaded in a segment at level 1. The user wished to try an experimental version of SINE. He loads a segment containing the new SINE at level 2. Segments loaded at level 3 or higher will now be linked to SINE at level 2 until a new level 2 or a new SINE is loaded.

COMMON blocks may be loaded with any segment, however, the maximum area for a given labeled COMMON block must be declared in the first segment which references that COMMON. Labeled COMMON blocks cannot be referenced acorss segments.

Segments may be loaded by

CALL SEGMENT (fn, e, a, lib, m)

where:

fn - variable name of the location which contains the file name (left justified display code) from which the segment load should take place.

e - is the level of the segment load ($1 \leq e \leq 63$).

a - variable name of array containing a list of SEGMENTS, SECTIONS and/or SUBPROGRAMS that are to be loaded with this call. This list must have the name in left justified display code and the list must be terminated by a zero entry. If the first entry in the list is zero, this signals a segment load of all subprograms remaining on the file fn.

lib - if zero, or blank, unsatisfied externals will be satisfied, if possible, from the system library.

m - if m is zero or blank, a map of the segment load will not be produced.

lib and m need not be specified.

Once the named subprograms are loaded control returns to the statement following the CALL SEGMENT. The programmer is free to call on the loaded subprograms as he chooses.