CR-152289

A

FEASIBILITY STUDY FOR A NUMERICAL AERODYNAMIC SIMULATION FACILITY

(NASA-CR-152289) FEASIBILITY STUDY FOR A NUMERICAL AERODYNAMIC SIMULATION FACILITY. VOLUME 3: FMP LANGUAGE SPECIFICATION/USER MANUAL Final Report (Control Data Corp., St. Paul, Minn.) 263 p HC A12/MF A01 N79-26070 Unclas G3/09 28383

Volume III — FMP Language Specification/User Manual

Contributions by: B. G. Kenner
N. R. Lincoln

MAY 1979

Prepared under Contract No. NAS2-9896

CONTROL DATA CORPORATION
Research and Advanced Design Laboratory
4290 Fernwood Street
St. Paul, Minnesota 55112

for

AMES RESEARCH CENTER

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

A: Variables, arrays, non-generic function names, constant names, dynamic variables and dynamic arrays that appear in a STAR . . .

B: The appearance of a symbolic name other than the name of an intrinsic function in a type statement . . .

C: . . . statement, the type of a symbolic name other than an intrinsic function name is implied by the . . .

D: The predefined FORTRAN functions either possess predefined types or else take their type from the type of their operand(s). Implicit typing of any of these names has no effect. If the names of any of these functions appear in explicit type statements, the typing is ignored if it confirms the predefined type of the function; otherwise, the name ceases to reference the FORTRAN-supplied function. See Table 6-0 below for details.

Table 6-0. Effect of Typing an Intrinsic Function Name

Kind of Name	Typing	Result	Diagnostic
Generic only, variable type (MAX)	Any	Function becomes EXTERNAL	Warning
Generic only, fixed type (DBLE)	Confirms predefined type	No effect	None
	Contradicts predefined type	Function becomes EXTERNAL	Warning
Specific only (MAXO)	Confirms predefined type	No effect	None
	Contradicts predefined type	Function becomes EXTERNAL	Warning
Generic and specific, variable type (SQRT)	Confirms predefined type of specific function	No effect	None
	Contradicts predefined type of specific function	Function becomes EXTERNAL	Warning
Generic and specific, fixed type (INT)	Confirms predefined type	No effect	None
	Contradicts predefined type	Function becomes EXTERNAL	Warning

This page left blank intentionally

E: IMPLICIT statements must precede all other specification statements except PARAMETER statements. If the type of a named constant is specified by an IMPLICIT statement, the IMPLICIT statement must precede the PARAMETER statement which defines the value of the constant. Appearance of an IMPLICIT statement which specifies the default type of symbolic names beginning with some letter after a PARAMETER statement which defines a constant whose name begins with that letter is prohibited.

F: The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

G: CHARACTER *K $w_1/d_1/,w_2/d_2/,...,w_n/d_n/$

H: . . . element length in bytes of every w. This specifica- . . .

I: . . . byte is implied for every w not accompanied . . .

J: d_i Optional. Represents the initial value for v_i or w_i. If . . .

K: v_i A variable, array, array declarator, function, or constant name. If v_i is the name of a constant,, d_i must not appear

A $*k_i$ Optional. An integer constant or simple integer variable specifying the element length in bytes for v_i. If v_i is an array declarator, $*k_i$ must appear between the declarator name and dimensions. If k_i is a variable, v_i must be a dummy argument and k_i must either be a dummy argument or in common. A variable used in this way as an adjustable length specification must either be implicitly integer, or else must have appeared in an INTEGER type statement before it appears in a CHARACTER (or any other declaration) statement. If $*k_i$ is omitted, the length of v_i is determined by *K.

If the array declarator for an array appears in an explicit type statement, it cannot appear also in a ROWWISE, DIMENSION, or COMMON statement. However, the array name alone can appear in COMMON statements to include the array in a common block. (An array declarator must appear once and only once in a program unit.)

DIMENSION STATEMENT

The DIMENSION statement serves as a vehicle for one or more array declarators. For an array declared in a DIMENSION statement, subscripts are interpreted in the conventional manner. For a discussion of rowwise and conventional array element succession, see section 2.

Form:

DIMENSION $a_1,a_2, \ldots ,a_n$

a_i An array declarator.

If the array declarator for an array appears in a DIMENSION statement, it cannot also appear in a ROWWISE, COMMON, or explicit type statement. However, the array name alone can appear in an explicit type statement to type the array and in COMMON statements to include it in a common block. (An array declarator must appear once and only once in any program unit.)

ROWWISE STATEMENT

The ROWWISE statement serves as a vehicle for one or more array declarators. It should be used in much the same way that a DIMENSION statement is used, the difference lying in the fact that for an array declared in a ROWWISE statement, subscripts are interpreted in a rowwise manner. For a discussion of rowwise and conventional array element succession, see section 2.

Form:

ROWWISE $a_1,a_2, \ldots ,a_n$

a_i An array declarator.

If an array declarator for a particular array appears in a ROWWISE statement, it cannot appear also in a DIMENSION, COMMON, or explicit type declaration statement. However, the array name alone can appear in an explicit type statement to type the array and in COMMON statements to include it in a common block. (An array declarator must appear once and only once in a program unit.)

COMMON STATEMENT

The COMMON statement is a nonexecutable statement that allows specified variables and arrays to be referenced by more than one program unit. Elements in common storage can be referenced and defined in any program unit that contains a COMMON statement specifying common blocks containing those elements. An element can be included in only one common block.

Storage for arrays and variables listed in a COMMON statement is reserved in a common block in the order in which the elements appear in the statement, and starting on a double word boundary. The elements are strung together in such a way that, for example, for a common block containing a complex variable, a 10-integer array, and 64 bit variables, 13 logically consecutive words are reserved: the first two words for complex data are followed immediately by 10 words for the integer array, which is followed by one word for 64 variables of type bit. The assignment of storage is determined solely by consideration of data type and array declarations for the variables and arrays in the COMMON statement. One or more blocks can be specified with a single COMMON statement; the order of appearance of blocks in the statement is not significant.

Form:

COMMON /blk_1/$list_1$/blk_2/$list_2$ . . . /blk_n/$list_n$ B

blk_i A symbolic name denoting a labeled common block. Absence of blk denotes the blank common block; if the first block identified is blank common, then the first pair of slashes can be omitted as well.

$list_i$ A block specification list, a list of the elements whose storage locations are in the common block blk_i. The list has the form:

$u_1,u_2, \ldots ,u_m$

where u_i is a variable name, an array name, or an array declarator.

Only an entire array can be placed in a common block. An array declarator, but not an array element name, can appear in a COMMON statement. Dummy arguments cannot appear in COMMON statements.

A block name can appear more than once in a COMMON statement or in several COMMON statements in a program unit; the elements are stored cumulatively in the order of their occurrence in all COMMON statements in the program unit. Block names can also be used elsewhere in the program to identify other entities. a common block name can unambiguously identify a variable, statement function, or array in the same program. For example, a valid COMMON statement is COMMON/ONE/ONE,TWO,THREE.

Blank common generally can be used in the same way as labeled common, except that elements in blank common cannot be initialized in DATA or type statements as can elements in labeled common. Also, unlike any labeled common block, the blank common block need not have the same length in every program unit in which it is declared. For example, the declaration in one program unit could be COMMON//A(4),B/LAB/C,D and in another could be COMMON//A(4)/LAB/C,D.

The size of a common block is the sum of the storage required for the elements introduced into that block through

A: w_i One of the following forms, where a_i is a character array name, ds_i is a dimension specifier (that is, the string $a_i ds_i$ is an array declarator), and n_i is a character variable, function, or constant name. If n_i appears and is the name of a constant, d_i may not be used.

n_1		
n_1	$*k_i$	
a_i		
a_i	ds_i	
a_i	$*k_i$	
a_i	ds_i	$*k_i$
a_i	$*k_i$	ds_i

k_i An unsigned integer constant, an integer constant expression enclosed in parentheses, an asterisk enclosed in parentheses, or a simple integer variable. If k_i is a simple integer variable, the entity being typed (a_i or n_i) must be a dummy argument and k_i must appear in every dummy argument list which also contains the entity being typed or else k_i must be in common. If k_i is a simple integer variable, it must be of default type integer or else must have been previously typed, either by IMPLICIT statement or by explicit type statement. If $*k_i$ does not appear, the length of the entity being typed is determined by *K if it appears, or else defaults to 1 regardless of any default length declared for symbols of its initial letter in preceding IMPLICIT statements.

B: COMMON $/blk_1/list_1$, $/blk_2/list_2$, ... $/blk_n/list_n$

COMMON and EQUIVALENCE statements. A double precision or complex element requires two words; a logical, real, or integer element requires one word; a character element requires one byte times the length specified for the element; a bit element requires a single bit. Character elements must fall on byte boundaries and integer, complex, logical, real, and double precision elements must fall on word boundaries. Character and bit types can appear in a common block with other types, so long as the elements having the other types are not forced off word boundaries.

Although block names must be the same name if they are to refer to the same common block, the names and types of the elements in the common block can differ among program units. If two program units define a particular common block to have the same data type assigned to any two elements in corresponding positions in the common block, then the two elements refer to the same value. Otherwise, any data in the common area is treated as having the data type of the name used to refer to it, and no type conversion takes place.

If a program unit does not use all locations reserved in a labeled or blank common block, unused variables can be inserted in the COMMON declaration to force proper correspondence of the variables or arrays in the common areas. Alternatively, correspondence in blank common can be ensured by placing selected variables at the end of the block in such a way that they can be omitted in the COMMON declarations for a program unit that does not use them. However, a common block (other than blank common) must have the same length in every program unit in which it is declared.

If an array declarator for a particular array appears in a COMMON statement, it cannot appear also in a ROWWISE, DIMENSION, or explicit type statement. However, the array name alone can appear in explicit type statements to specify the array's data type. (An array declarator must appear once and only once in a program unit.)

In a subprogram, the dummy arguments for the subprogram cannot be placed in common. However, variable dimensions for a dummy array can be placed in common, so long as those variables are not also dummy arguments.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement is a nonexecutable statement that permits two or more variables in the same program unit to share storage locations. This arrangement of data can be contrasted with that of variables and arrays not mentioned in an EQUIVALENCE statement (which are generally assigned unique locations) and with that of variables and arrays declared in COMMON statements (the COMMON statement permits two or more variables, each in a different program unit, to share storage locations).

Form:

EQUIVALENCE(group$_1$), . . . ,(group$_n$)

group$_i$ A list of the form:

$v_1, \ldots, v_m$

where v_i is a variable, array element, or array name (array declarators are not permitted), and $m \geq 2$. Each comma separating two groups is optional.

All the elements in group$_i$ begin at the same storage location.

The naming of array elements is relatively flexible in an EQUIVALENCE statement. Unlike array names in most STAR FORTRAN statements, an array name in an EQUIVALENCE statement names only the first element of the array. Also, in an EQUIVALENCE statement any array element can be identified using an array element name containing a subscript having only a single subscript expression, where the value of the expression is the location of the element in the array as determined by the succession formulas given in table 2-2. However, if neither of these forms is used, then the subscript must conform to the ordinary subscript form. Each subscript expression in an EQUIVALENCE statement must be an integer constant; the number of subscript expressions must correspond in number to the dimensionality of the array or else must be one.

A storage location can be shared by variables having different data types. A logical, integer, or real variable equivalenced to a double precision or complex variable shares the same location with the real or most significant half of the complex or double precision variable. However, when one- or two-word variables are equivalenced to character or bit variables, they must begin on full word boundaries. Similarly, if a character variable is equivalenced to a bit variable, the character variable must be aligned on a byte boundary. Type is associated only with the name used to reference a location, and that name determines how data assigned to or read from the location is to be interpreted; no type is remembered and no conversion takes place. Consequently, if (for example) a real element is equivalenced to an integer element, defining the real element causes the integer element to become undefined, and vice versa.

A variable can appear in both EQUIVALENCE and COMMON statements in a program unit. However, a variable in common can be equivalenced to another variable only if that variable is not in any common block. A variable or array is brought into a common block if it is equivalenced to an element in common. It is acceptable for an EQUIVALENCE statement to lengthen a common block, so long as the common block is extended beyond the last assignment for that block and does not extend the block's origin. A dummy argument must not appear in any EQUIVALENCE statement.

Figure 6-1 illustrates some of these concepts. In figure 6-1A, array element A(2) in the labeled common block BLK1 is equivalenced to array element B(1), which is not in common. The EQUIVALENCE statement causes the entire array B to be brought into common, extending the length of common by two words and equivalencing other pairs of data elements as shown in figure 6-1B. If instead A(1) and B(2) has been equivalenced, an error would have resulted because this would have been an attempt to extend the common block's origin to P.

It is also incorrect directly or indirectly to cause a single storage location to contain more than one element of the same array. For example, adding a second EQUIVALENCE statement, EQUIVALENCE (A(4), B(2)), to the statements in figure 6-1 would constitute a request for A(4) and A(3) to share the same storage location.

A: . . . cannot be placed in common. However, dimension bound variables for adjustably dimensioned arrays and length variables for adjustable length character entities may be placed in common, so long as . . .

B. where v_i is a variable, array element, substring, or array name (array declarators are not permitted), and m is greater than or equal to 2. Each comma . . .

C: LEVEL STATEMENT

The level statement assigns variables or arrays to the different levels of FMP memory.

Form:

LEVEL n, a1,a2, . . . , am

ai	Variables, array names or array declarators, separated by commas.
n	Unsigned integer 1,2,3, or blank, or integer PARAMETER indicating to which memory list is to be allocated.
1	Main Memory
2	Intermediate Memory
3	Backing Storage

The Default Level is LEVEL 1. LEVEL statements must preceed the first executable statement in a program unit. Names of variables which do not appear in a LEVEL statement are allocated to Main Memory.

Type information may not be included in the LEVEL statement. Array declaratives of the form A(n1,n2)..) where ni is a simple integer are permitted. Array declarators perform the same function as if they appeared in a DIMENSION statement.

Variables and arrays appearing in a LEVEL statement can appear in DATA, DIMENSION, EQUIVALENCE, COMMON, type, DYNAMIC, SUBROUTINE, and FUNCTION statements. Data assigned to LEVEL 3 can only consist of arrays, and may only be referenced in COMMON, type, DIMENSION, EQUIVALENCE, DATA, CALL, SUBROUTINE, and FUNCTION statements. FORTRAN expressions involving LEVEL 3 data must reference the entire array or subarrays.

No restrictions are imposed on the way in which reference is made to variables or arrays allocated to LEVELS 1 and 2. DYNAMIC arrays may only be assigned to LEVELS 1 and 2.

If the level of any variable is multiply defined, the first level defined is assumed and a warning diagnostic is printed.

All members of a common block must be assigned to the same level: a fatal diagnostic is issued if conflicting levels are declared. If some, but not all members of a common block are declared in a LEVEL statement, all are assigned to the declared level and an informational diagnostic is printed.

If a variable or array name declared in a LEVEL statement appears as an actual argument in a CALL statement, the corresponding dummy argument must be allocated to the same level in the called subprogram. If a variable or array name appears in an EQUIVALENCE and a LEVEL statement, the equivalenced variables must all be allocated to the same level.

Example:

```
    PROGRAM DEMO
    DIMENSION A(100,B(200),C(300)
    LEVEL 2,A,B
    .
    .
    .
    C(I)=A(I)+B(I)
```

The LEVEL statement allocates arrays A and B to Intermediate Memory. The arithmetic statement will cause the fetching of the Ith element of A and B from Intermediate Memory, their summation, and the storage of the result into Main Memory in the Ith element of C.

DYNAMIC STATEMENT

The DYNAMIC statement identifies those variables whose dimensions, and perhaps memory allocation, will be determined during program execution.

Form:

```
    DYNAMIC v1,v2 . . . . vn
    vi    A variable, array declarator or array name of type REAL or INTEGER
```

All variables in the DYNAMIC statement list are declared to be dynamic pointer data for actual memory arrays, while arrays or array declarators signify that the named variable consists of an array of dynamic pointer data or a DYNAMIC ARRAY.

DYNAMIC variables and arrays may only appear in memory LEVELS 1 and 2. DYNAMIC variables may appear in COMMON, CALL, FUNCTION, SUBROUTINE, type, DIMENSION and arithmetic and input/output statements. DYNAMIC variables may not appear in EQUIVALENCE, DATA or as the parameters in DO statements.

If DYNAMIC variables or arrays are passed as parameters in FUNCTION and SUBROUTINE call statements, then the corresponding dummy arguments must also be declared DYNAMIC and possess the same dimensionality in the called subprogram unit.

The number of storage locations used by the pointer data is variable throughout program execution for DYNAMIC variables and arrays. The space required is a function of the dimensionality of the arrays being described by the DYNAMIC variable. Thus COMMON statements in two different program units which contain DYNAMIC variables must have identical format.

The following form is, therefore, illegal:

```
      PROGRAM DEMO
      DYNAMIC A(100)
      COMMON/B/A
      .
      .
      CALL C
      .
      .
      END


      SUBROUTINE C
      COMMON/B/A(100)

      .
      D=A(I)
      .
```

In this case the programmer is erroneously attempting to deal with the DYNAMIC array as an array itself. Since the storage of FORTRAN arrays consist of one element per memory word and DYNAMIC pointers take from 2 to 14 memory words per pointer element, the two COMMON statements imply a different memory allocation, and this is an illegal condition which cannot be detected by the compiler or at object time.

The values of DYNAMIC pointer variables can only be established by execution of expressions involving subarray references or by the DEFINE statement.

Example:

```
      PROGRAM DEMO
      DYNAMIC A,B(4)
      DIMENSION X(100),Y(10,20)
      .
      .
      A=X(1:100)

      B(2)=X(1:10)+Y(1:10,1)
```

The DYNAMIC statement declares variable A to be a dynamic pointer, and array B to be a dynamic array of pointers. The initial value of all pointers is set to an internally recognized value of NULL. This indicates that no data is pointed to, or the pointer is not yet defined.

The replacement statement A=X(1:100) causes the following actions:

One hundred words are allocated from dynamic space in Main Memory. The address of this space and the length 100 are then assigned to the pointer variable A. A map unit move is then performed to transfer the data from the array X to the newly defined array A. The attributes of address and length assigned by this dynamic activity will then be retained as the defined quantities for A until another expression is encountered which changes either the memory allocation or dimensionality.

If the statement A=X(1:100) is executed again, the pointer data is unchanged and the same memory space is reused. If another statement:

```
A=X(1:200)
```

is encountered, a new space allocation of two hundred elements is made from dynamic space, and a new length of 200 established for the pointer A. The data is then transferred.

If the statement:

```
A=X(1:20)
```

is encountered, then the original memory address is retained for the DYNAMIC variable A, but the length is changed to 20 and the data transferred from X. The remaining 80 elements that used to be part of the space pointed to by A become undefined.

The second example assigns 10 words of dynamic space to the pointer element B(2), and performs the arithmetic on the array elements X and Y, storing the results into the assigned dynamic space. The reallocation or contraction of space for DYNAMIC ARRAY elements follows the previous rules given for DYNAMIC variables.

DYNAMIC variables can also be established by DEFINE statements.

Example:

```
PROGRAM DEMO
DYNAMIC A
LEVEL 2,B(100)
.
.
DEFINE (A,B(10:20))
.
.
A=A*A
```

In this example, the DYNAMIC variable is assigned the starting address of B(10) and a length of 11. essentially describing a subarray of B. The operation A=A*A would then become a vector multiply of elements 10 through 20 of array B by themselves, with the results returned to elements 10 through 20.

When DYNAMIC variables appear as the objects of replacement statements their dimensionality is always redefined, with the following exception:

```
PROGRAM DEMO
DYNAMIC A
DIMENSION B(100)

.
A=B(1:00)+B(1:100)


A(31:50)=B(11:30)
.
```

In this case, the array pointed to by A begins as a 100 element array. The second assignment statement does not shrink the array to 20 elements, although that is all the data that is being moved. Instead, elements 31 to 50 are replaced and the memory address and dimensionality remain unchanged.

When a DYNAMIC variable is referenced as a subarray;i.e., A(1:m:n) in an executable expression and m is greater than the existing dynamically assigned dimensionality in that direction, a fatal object time diagnostic message is printed. Subarray references to DYNAMIC array elements, except in DEFINE statements, are not permitted and will cause the compiler to generate a fatal diagnostic message.

A.

```
CHARACTER*10 CH,DH
COMMON/BLK1/A(4),CH,DH
DIMENSION B(8)
EQUIVALENCE (A(2),B(1))
```

B.

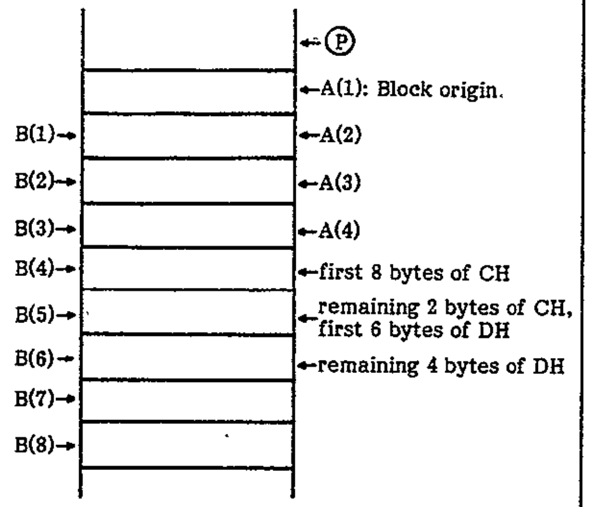


Figure 6-1. COMMON and EQUIVALENCE Statements

EXTERNAL STATEMENT

Before a subprogram name can be used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program unit.

Form:

EXTERNAL$p_1, \ldots ,p_n$

p_i A procedure name or entry point name.

The appearance of a name in an EXTERNAL statement declares that name to be an external procedure name rather than a data element name.

Any name used as an actual argument in a procedure call is assumed to name data unless it appears in an EXTERNAL statement. For example, any predefined FORTRAN function name must be declared in an EXTERNAL statement if it is to be used as an actual argument. A function reference in an actual argument list need not be declared in an EXTERNAL statement, however, because it is not the function, but the result of function evaluation, that is the argument.

The effect that placing a predefined FORTRAN function name in an EXTERNAL statement has on the kind of code generated is shown in table 6-1.

DATA STATEMENT

Only variables and array elements assigned values with a data initialization statement or in an explicit type statement are defined (possess a predictable value) when program execution begins. The DATA statement is a nonexecutable statement used to assign initial values to variables and array elements (including entire arrays).

Form:

DATA$v_1/k_1/,v_2/k_2/, \ldots ,v_n/k_n/$

v_i	A variable list of the form: $w_1, \ldots ,w_m$ where w_i is a variable, array element, array, or implied DO. Subscripts used to identify array elements must be integer constants, except within an implied DO.
k_i	A data list of the form: $j*d_1, \ldots ,j*d_m$ where d_i is an optionally signed constant. The constant can be preceded by an optional repeat specification j*, where j is an (unsigned) integer constant.

The comma after each second slash is optional. Except for certain variable list items of type bit, a one-to-one correspondence must exist between the items in the variable list and the constants in the data list. In particular:

An array of any type except bit must correspond to a number of items equal to the number of elements in the array.

A simple variable of type bit must correspond to a bit constant.

An implied DO specifying a number of elements of an array of any type except bit must correspond to a number of items equal to the number of array elements. The elements specified need not be contiguous.

A bit array must correspond to a list of one or more hexadecimal and bit constants whose total bit length is the number of elements in the bit array.

A contiguous portion (one or more elements) of a bit array must correspond to a list of one or more hexadecimal and bit constants whose total bit length is the number of elements in the bit array portion. Such a bit array portion is specified in the variable list by means of a single bit array element or an implied DO.

An implied DO might specify more than one contiguous portion of a bit array. For example, in the initialization:

```
ROWWISE DSB(4,4)
BIT DSB
DATA ((DSB(I,J), J=1,4), I=1,4,2)/2*B'1001'/
```

two contiguous portions disjoint from one another are specified:

DSB(1,1), DSB(1,2), DSB(1,3), DSB(1,4)

DSB(3,1), DSB(3,2), DSB(3,3), DSB(3,4)

In such a case, the correspondence rules must be applied individually to each of the portions. Hence, initializing the eight DSB array elements with a single constant B'10011001' (or X'99') would cause a fatal error.

A: EXTERNAL STATEMENT

The form of the EXTERNAL statement is

EXTERNAL $p_1,...,P_n$

where p_i is the name of an external procedure or block data subprogram. The appearance of a name in an EXTERNAL statement declares that name to be defined externally to the declaring program unit. Such an appearance implies that the name is not the name of an intrinsic function, statement function, variable, or array.

If the name of an external procedure appears in an actual argument list, an EXTERNAL declaration of that name is required. (If a reference to an external function appears in an actual argument list, EXTERNAL declaration is in an actual argument list, EXTERNAL declaration is permitted but not required.)

If the name of an intrinsic function appears in an EXTERNAL statement, the connection between the name and the intrinsic function is broken. Thus the EXTERNAL declaration provides the user a means to substitute his own function for the FORTRAN-supplied function.

(Note that the STAR FORTRAN object library provides an external version of every intrinsic function. See Table 6-1 for further details.)

INTRINSIC STATEMENT

The form of the INTRINSIC statement is

INTRINSIC $i_1,...,i_n$

where i_k is the name of an intrinsic function. The appearance of a name in an INTRINSIC statement declares that name to be the name of a FORTRAN-supplied intrinsic function. (Not all FORTRAN-supplied functions are intrinsic functions. For example, the LENGTH and UNIT functions used in connection with the BUFFER IN and BUFFER OUT statements are not intrinsic. See Section 15 for further details.) Names other than those of intrinsic functions may not appear in INTRINSIC statements. A name which appears in an INTRINSIC statement cannot be the name of a variable, array, statement function, or external procedure.

Intrinsic functions are of two kinds, generic and specific. A specific intrinsic is one with well-defined argument and result types; for example, MAX1 has REAL arguments and returns an INTEGER result. A generic intrinsic is one which accepts more than one argument type. Some generic intrinsics return a result whose type depends upon the type of their operands; for example, MAX accepts arguments of type INTEGER, REAL, DOUBLE PRECISION, or HALF PRECISION and returns a result whose type is the same as the (common) type of its arguments. Other generic intrinsics return a result of fixed type, independent of the type of their operands; for example, CMPLX always returns a result of type COMPLEX regardless of its argument type, which may be INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, or COMPLEX.

Some intrinsic function names are the names only of generic functions (e.g., MAX); some are the names only of specific functions (e.g., MAX0); some are the names of both generic and specific functions (e.g., SQRT).

Some specific intrinsic functions may be passed as actual arguments. No generic intrinsic function may be passed as an actual argument. When a specific intrinsic is passed as an actual argument, its name must appear in an INTRINSIC statement in the passing program unit. It is permissible to pass a specific intrinsic whose name is also that of a generic intrinsic; the appearance of the intrinsic name in an actual argument list does not affect the generic properties of the name within the passing program unit. For example, the first reference to SQRT in the following sequence is a reference to the specific intrinsic function which returns the REAL square root of a REAL argument; the second is a reference to the generic intrinsic function which, among other things, returns the DOUBLE PRECISION square root of a DOUBLE PRECISION argument.

```
DOUBLE PRECISION D1, D2
INTRINSIC SQRT
CALL SUB(SQRT)
D2 = SQRT(D1)
```

The specific intrinsics for type conversion, lexical relationship and for choosing the largest or smallest value may not be passed as actual arguments. All other specific intrinsics may be passed provided they are declared INTRINSIC in the passing program unit.

The following table summarizes code generation for the various possible combinations of INTRINSIC/EXTERNAL declaration for intrinsic and non-intrinsic function names.

Table 6-1. Code Generation for Function References

Function Name	Declaration	Use	Generated Code
Not intrinsic	None	Referenced	Slow call to user routine
Not intrinsic	None	Passed	Compilation error
Not intrinsic	INTRINSIC	Referenced	Compilation error
Not intrinsic	INTRINSIC	Passed	Compilation error
Not intrinsic	EXTERNAL	Referenced	Slow call to user routine
Not intrinsic	EXTERNAL	Passed	Slow call to user routine
Specific only	None	Referenced	Fast call or inline
Specific only	None	Passed	Compilation error
Specific only	INTRINSIC	Referenced	Fast call or inline
Specific only	INTRINSIC	Passed	Slow call to library routine*
Specific only	EXTERNAL	Referenced	Slow call to user routine
Specific only	EXTERNAL	Passed	Slow call to user routine
Generic only	None	Referenced	Fast call or inline
Generic only	None	Passed	Compilation error
Generic only	INTRINSIC	Referenced	Fast call or inline
Generic only	INTRINSIC	Passed	Compilation error
Generic only	EXTERNAL	Referenced	Slow call to user routine
Generic only	EXTERNAL	Passed	Slow call to user routine
Generic and Specific	None	Referenced	Fast call or inline
Generic and Specific	None	Passed	Compilation error
Generic and Specific	INTRINSIC	Referenced	Fast call or inline
Generic and Specific	INTRINSIC	Passed	Slow call to library routine*

Generic and Specific	EXTERNAL	Referenced	Slow call to user routine
Generic and Specific	EXTERNAL	Passed	Slow call to user routine

*Assuming the intrinsic is passable.

Notes: If the use is "PASSED", the generated code column describes the code generated for a reference to the corresponding dummy procedure in the called routine.

In this table, "user routine" means a routine which is, or at least can be, written in FORTRAN. A routine named SQRT is such a routine. "Library routine" refers to a routine which cannot be written in FORTRAN. A routine named FT_XSQRT is such a routine.

PARAMETER STATEMENT

The names and values of named constants are declared with the PARAMETER statement. The form of a PARAMETER statement is

PARAMETER $(p_1=e_1,...,p_n=e_n)$

where p_i is the name of a constant and e_i is a constant expression which defines the value of p_i. Named constants have associated types; the possible types of a named constant are INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, CHARACTER, and LOGICAL. The type of p_i, and its length if it is of type CHARACTER, must have been specified, either by default, by IMPLICIT typing, or by explicit typing, before the PARAMETER statement which assigns p_i its value. If p_i is of default implied type, the default type of names beginning with its first letter may not be changed by an IMPLICIT statement which appears after the PARAMETER statement which assigns p_i its value. If the type of p_i is implied, either according to the default implied typing or according to some preceding IMPLICIT specification, then p may not appear in a subsequent type statement.

If p_i is type INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, or COMPLEX, then e_i must be one of these same types, though not necessarily the same as p_i. If the types of p_i and e_i are not the same, e_i is converted to the type of p_i before the value of p_i is assigned. This conversion is according to the same rules which obtain in arithmetic assignment statements. The expression e_i is an arbitrary arithmetic expression except that the right-hand operand of the exponentiation operator must be INTEGER and all the operands must be constant.

If p_i is CHARACTER or LOGICAL, then e_i must be, respectively, CHARACTER or LOGICAL. If p_i is CHARACTER, e_i may be an arbitrary character expression except that all the operands must be constant. If p_i is logical, then e_i may be an arbitrary LOGICAL expression except that all operands must be constant and, furthermore, any arithmetic expressions which are operands of relational operators in p_i must conform to the restriction on exponentiation previously stated.

The operands in e_i may include previously defined named constants, including those defined previously in the same PARAMETER statement (i.e., p_j for j less than i).

Named constants may be used in expressions and in the constant lists of DATA statements. They may not be used in FORMAT statements, as statement labels, or as parts of other constants (e.g., either half of a complex constant).

SAVE STATEMENT

The form of the SAVE statement is

SAVE $a_1,...,a_n$

where a_i is a variable name, array name, or common block name enclosed in slashes. An empty list is permitted: that is, n may legally be zero. a_i may not be the name of a dummy argument, a procedure, or an entity in common. It is illegal for a_i to be the same as a_j unless i is equal to j. Furthermore, if $a_1,...,a_n$ and $b_1,...,b_m$ are the lists of any two SAVE statements in the same program unit, it is illegal for any a_i to be the same as any b_j. If a program unit contains a SAVE statement with an empty list, then that must be the only SAVE statement in the program unit. If a common block name is specified in a SAVE statement in any subprogram of an executable program, then it must be specified in a SAVE statement in every subprogram in which it is referenced.

The purpose of the SAVE statement is to preserve the definition of variables which must remain defined beyond the execution of a RETURN or END statement in a subprogram. In the absence of SAVE statements, execution of a RETURN or END statement in a subprogram causes all variables and arrays known to the subprogram to become undefined (which means that their contents can no longer be depended upon) except those which are in blank common, are initially defined and are neither redefined nor undefined during execution of the subprogram, or are in a named common block which appears in at least one other program unit which is, either directly or indirectly, referencing the subprogram.

B: . . . substring, or implied DO. Subscript expressions and substring expressions used to identify array elements and substrings must be integer constant expressions except that subscript expressions may include references to the control variables of any containing implied DOs.

The data list item corresponding to the variable list item is the variable list item's initial value. The rules of correspondence apply to bit array initialization in BIT statements as well as in DATA statements.

The form j* before a constant in the data list indicates the number of times the constant is specified. The following two DATA statements are identical in effect:

```
DATA K,L,M/0,0,0/

DATA K,L,M/3*0/
```

A

TABLE 6-1. EXTERNAL DECLARATION OF A SUPPLIED FUNCTION

		Code
In-Line Function	declared external	external (user-provided)
	not declared external	in-line
External Function	declared external	external
	not declared external	external
Function Having Both an External and In-Line Version	declared external	external
	not declared external	in-line

IMPLIED DO IN DATA STATEMENT

An implied DO in the variable list of a DATA statement can be used as a shortened notation for specifying parts of an array.

Form:

$(p,i=m_1,m_2,m_3)$

B p	A subscripted array name, or another implied DO.
i	The implied-DO control variable, a simple integer variable. i cannot also be the implied-DO control variable of an implied-DO list containing this list.
C m_1	The initial value parameter, an (unsigned) integer constant, less than or equal to m_2.
m_2	The terminal value parameter, an (unsigned) integer constant, greater than or equal to m_1.
m_3	Optional. The incrementation value parameter, an (unsigned) integer constant. When omitted, the preceding comma must also be omitted and an increment of 1 is assumed.

Implied-DO loops in the DATA statement can be nested up to seven deep. Subscript expressions must be one of the following forms:

```
c      i-c
i      k*i+c
i+c    k*i-c
```

where c and k are unsigned nonzero integer constants, and i is the implied-DO control variable of this implied-DO list or of an implied-DO list that contains this list. D

The order in which elements are specified by an implied DO in a DATA statement is identical to that in which elements are specified by an implied DO in an input/output list (see section 9).

RULES FOR INITIALIZING VALUES

The rules for initializing values with the DATA statement also apply to data initialization with the type statements described earlier in this section: d_i in the explicit type statement form corresponds to the d_i in the DATA statement form. Nevertheless, several differences in form exist and are as follows:

In a DATA statement, a list of simple variables can be initialized by a list of constants. In a type statement, only an array can be initialized by a list.

Dimension declarators can occur in type statements, but only array elements can occur in DATA statements.

The implied DO is allowed in DATA statements, but not in type statements.

The DATA statement cannot be used to assign values to dummy arguments in a subprogram or to elements in blank common. Elements in a labeled common block can be initialized with a data initialization statement in any program unit that mentions the block in a COMMON statement; furthermore, different parts of a block can be initialized in different program units, as well as with different statements in the same program unit.

Character or hollerith constants used to initialize variable list items are padded with blank characters on the right or are truncated on the right to fit the variable length, E depending upon whether the number of characters in the constant is less than or greater than the number of characters defined by the variable list element. A warning message is issued if truncation occurs.

A list of bit and hexadecimal constants used to initialize a contiguous portion of a bit array, including possibly an entire bit array, must have a total bit length exactly the same as the length of the array portion. If the constant or constant list bit length is too short, the system issues the fatal diagnostic TOO LITTLE DATA IN HEX OR BIT CONSTANT. If the constant or constant list bit length is too long, the system issues the fatal diagnostic TOO MUCH DATA IN HEX OR BIT CONSTANT.

A bit or hexadecimal constant used to initialize a variable list item of any type other than type bit is either right-justified and padded on the left with zero bits or else truncated on the left to fit the length of the variable, depending on whether the number of bits in the constant is

A: delete

B: p A list of array element names and implied DOs.

C: m_1,m_2,m_3 The initial, limit, and increment expressions for the implied DO. m_3, together with the comma which precedes it, is optional. The expressions are arbitrary integer expressions except that non-constant references must be restricted to the control variables of containing implied DOs. If m_3 is omitted, a value of 1 is assumed.

D: . . . to seven deep. Subscript expressions appearing in implied DOs may be arbitrary integer expressions so long as the only variable constituents of those expressions are implied-DO-variables of containing implied DOs.

E: . . . list items are padded or truncated to fit the variable length . . .

less than or greater than the number of bits defined for the variable list item. A warning message is issued if truncation occurs.

Example:

Given the array declaration INTEGER I(2), the data statement:

```
DATA I /2*X'38'/
```

initializes each of the two elements of the array I with a 64-bit constant whose value is hexadecimal 38, equal to decimal 56. Since the number of bits required to represent X'38' (that is, 8 bits) is less than the number of bits required for integer data, the constant would be padded on the left with zero bits. The data statement in this example has the same effect as the statement:

```
DATA I /2*56/
```

containing an integer constant instead of a hexadecimal one.

Bit arrays are a special case. Initializing a bit array or a contiguous part of a bit array (the latter by means of an implied-DO variable list item) is unlike initializing other kinds of quantities, including other type bit items. Any contiguous part of a bit array – including a single element, several elements, or the entire array – can correspond to one data list item whose length matches exactly the length of the array part. For example, if B is a 10-element array of type bit, the following are allowable DATA statements:

```
DATA B(1) /B'0'/

DATA B /B'11 1111 1111'/

DATA B /X'FF', 2*B'1'/

DATA (B(I), I=1,8) /X'F0'/

DATA (B(I), I=3,10) /2*B'1', X'0', B'0', B'1'/

DATA (B(I), I=1,10,5) /2*B'0'/
```

Except for the last one, all of the above statements describe contiguous parts of array B. The last DATA statement identifies two parts of array B, elements B(1) and B(6); each element properly corresponds to a data list item having a length of 1 bit. The following statement would be incorrect:

```
DATA (B(I), I=1,10,5) /B'00'/
```

An attempt would be made to initialize B(1) with B'00', and the fatal diagnostic TOO MUCH DATA IN HEX OR BIT CONSTANT would be issued.

Although more than one constant can be used to initialize a single bit array portion, two bit array portions cannot be initialized by a single constant. For example, two bit arrays BA(2) and BB(4) might be initialized acceptably with either of the statements:

```
DATA BA, BB /B'10', X'A'/
DATA (BA(I),I=1,2),(BB(I),I=1,4) /B'10', X'A'/
```

but not with the statement:

```
DATA BA(1), BA(2), BB / B'10', X'A'/
```

An attempt would be made to initialize BA(1) with B'10' and a fatal diagnostic would be issued. Similarly, parts of two different bit arrays cannot be initialized with a single data list item. For example, the statement:

```
DATA BA,BB /B'10 1010'/
```

would be incorrect. An attempt would be made to initialize the two-element array BA with the bit constant B'101010', and a fatal diagnostic would be issued. The statement:

```
DATA BA,BB /B'10', B'1010'/
```

would, however, be acceptable.

The type of a variable list item and the constant used to initialize it can differ in some cases. The constant value is converted (if necessary) to the type of the variable when both the variable and the constant have numeric data types; by contrast, the variable is initialized with the unconverted constant value when the constant is one of the nonnumeric data types hexadecimal, character, Hollerith, or bit. A logical constant list item can initialize only a logical variable list item. Mixed mode data initialization rules are given in table 6-2. The conversion is the same as for assignment statements.

TABLE 6-2. DATA INITIALIZATION CONVERSIONS

Variable Type	Constant Type							
	Logical	Integer	Real	Double Precision	Complex	Character or Hollerith	Bit	Hexadecimal
Logical	nocon	n/a	n/a	n/a	n/a	nocon	nocon	nocon
Integer	n/a	nocon	c	c	c	nocon	nocon	nocon
Real	n/a	c	nocon	c	c	nocon	nocon	nocon
Double Precision	n/a	c	c	nocon	c	nocon	nocon	nocon
Complex	n/a	c	c	c	nocon	nocon	nocon	nocon
Character	n/a	n/a	n/a	n/a	n/a	nocon	nocon	nocon
Bit	n/a	n/a	n/a	n/a	n/a	n/a	nocon	nocon

The letter c indicates that conversion is performed; nocon, that conversion is not performed; and n/a, that the type combination is not allowed.

This page left blank intentionally.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

DEFINING PROGRAM UNITS AND FUNCTIONS STATEMENT 7

Discussed in this section are the statements used to define and reference the following user-written procedures:

Statement function	Not a program unit; one-statement definition; is referenced
Main program	Executable program unit; multistatement definition; is not referenced
Function subprogram	Executable program unit; multistatement definition; is referenced
Subroutine subprogram	Executable program unit; multistatement definition; is referenced with a CALL statement
Specification subprogram	Nonexecutable program unit; multistatement definition; is not referenced

Not discussed are the predefined functions supplied with FORTRAN; these are covered in section 15. Argument passing (under the heading Passing Arguments Between Subprograms) and file declaration (under the heading PROGRAM Statement) are also covered here. CALL and RETURN are covered in the flow control statement section. Interfacing with non-FORTRAN external procedures is discussed in section 12.

The category of procedure definition to be used is determined by its particular capabilities and the needs of the program being written. If the program requires the evaluation of a standard mathematical function, often a FORTRAN-supplied function can be used. If a single computation is needed repeatedly, a user-written statement function can be included in the program. If a number of statements are required to obtain a single result, a function subprogram is written. If a number of calculations are required to obtain several values, a subroutine subprogram should be written.

The first statement of a program unit defines the program unit to be a main program, subroutine subprogram, function subprogram, or specification subprogram. A program unit whose first statement is not a FUNCTION, SUBROUTINE, or BLOCK DATA statement is a main program. Normally, a main program begins with a PROGRAM statement, but this statement can be omitted if no input data is required and all output is performed with PRINT statements. A subprogram is a program unit that begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement. B

An executable FORTRAN program must contain one main program and can have any number of subprograms and references to other external procedures, including the predefined functions supplied with FORTRAN. A main program must not be referenced by another program unit; once defined, subprograms may or may not be so referenced. Any program unit must never directly or indirectly invoke itself.

THE MAIN PROGRAM

The PROGRAM statement defines the name that is used as the program's entry point name and as the object module name for the loader. It is also used to declare files that are used in the main program and in any subprograms that are called.

PROGRAM STATEMENT

The PROGRAM statement is the first statement in a main program. However, the statement is optional when no request for input is made within the program, and no output except using PRINT is performed. Only one PROGRAM statement can occur in any program.

Form:

PROGRAM p (fip$_1$, fip$_2$, . . . ,fip$_n$) A

p Optional when no fip list is present; the name of the program.

fip$_i$ Optional. A file information parameter that can assume one of the following forms:

```
UNITn=f
TAPEn=f
UNITn[p1,p2,p3,p4]=f
TAPEn[p1,p2,p3,p4]=f
INPUT
INPUT=f
OUTPUT
OUTPUT=f
PUNCH
PUNCH=f
RLP
RLP=m
```

The logical unit number n is an integer constant in the range 1 to 99. The parameter list [p$_1$,p$_2$,p$_3$,p$_4$] specifies the file to be an explicit file. The filename f, a string of one to eight letters or digits beginning with a letter, is the name of a file required by the main program or a subprogram. No more than 70 files can be declared (including OUTPUT, whether or not it is listed). The specification m is a positive integer. When no fip is required, the list including parentheses is omitted.

The name p must not appear in any other statement in the program unit. The program name p can be omitted from the statement when no file information parameter list is present, in which case the name M_A_I_N is supplied.

File Information Parameters

No file names can appear in a program. Instead, the forms UNITn=f and TAPEn=f are used interchangeably to associate the file named f with a logical unit number n. Whenever the file f needs to be referred to in subsequent statements, the

PRECEDING PAGE BLANK NOT FILMED

A: THE MAIN PROGRAM

PRECEDING PAGE BLANK NOT FILMED

A main program is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It may have a PROGRAM statement as its first statement.

There must be exactly one main program in an executable program. Execution of an executable program begins with the first executable statement of the main program.

THE PROGRAM STATEMENT

The PROGRAM statement defines the name that is used as the program's entry point name and as the object module name by the loader. It may also be used to declare files that are preconnected to units used anywhere in the program and to request the mapping of dynamic space into large pages. See chapter 8 for a description of preconnected files.

The form of the PROGRAM statement is:

```
PROGRAM pgm[([fp[,fp]...][,RLP[=m]])]
```

where: pgm is the symbolic name of the main program in which the PROGRAM statement appears.

fp is a preconnection specifier. It is either a file declaration specifier or an alternate unit specifier.

RLP specifies that dynamic space is to be mapped in m large pages. If m is omitted a value of 1 is assumed. If RLP is omitted dynamic space is mapped in small pages. The comma preceding the RLP list item must be omitted if it is the only item in the list.

Items enclosed in [] are optional and the ellipsis . . . means that the items may be repeated. The preconnection specifier list may contain a maximum of 70 specifier items fp.

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must be the first statement of a main program. If it is omitted the symbolic name M A I N is supplied for pgm. Units are preconnected as described in chapter 8 if the PROGRAM statement is omitted.

The symbolic name pgm is global to the executable program and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. The name pgm must not be the same as any local name in the main program, except that it may be the same as a file name or an alternate unit name.

FILE DECLARATION SPECIFIER

A file declaration specifier provides the means of specifying, for an external file, the file name, the unit(s) to which it is preconnected, and the input/output buffer length in small pages.

The forms of a file declaration specifier are:

```
fn
fn=bl
```

where: fn is a symbolic file name.

bl is a buffer length specifer, consisting of an unsigned integer constant in the range 1..24. If bl is omitted a value of 3 is assumed.

The appearance of a symbolic file name fn is a file declaration specifier has the same unit-file connection effect as the execution of an OPEN statement prior to the execution of any input/output statement that refers to the file defined by fn.

The OPEN statement has the form:

```
OPEN(UNIT=nHfn,FILE='fn',. . .)
```

where n is the number of characters in 'fn'. If the character string 'fn' has either the character string 'TAPE' or the character string 'UNIT' as an initial substring and has a digit string u, with a non-zero leading digit, as its only other characters, then the unit-file connection affect of a second OPEN statement of the form:

```
OPEN(UNIT=u,FILE='fn',. . .)
```

is also implied. Note that if 'fn' is either 'TAPE0' or 'UNIT0' then this second OPEN statement is equivalent to:

```
OPEN(UNIT=0,FILE='fn',. . .)
```

The scope of a symbolic name is the PROGRAM statement in which it appears.

If the file with file name 'fn' exists, the values of the other parameters required for the OPEN statement are determined from the attributes of the file. If the file does not exist, the values of the parameters will be determined from the first input/output statement that references the file and can cause it to be created. If necessary a value of 512 words (1 small page) will be supplied for the record length parameter. The OPEN statement is described in chapter 8.

The buffer length specifier bl specifies the length in small pages of a buffer to be supplied by the processor for input/output data transfers. An error condition exists if an attempt is made to change bl by means of an OPEN statement.

ALTERNATE UNIT SPECIFIER

An alternate unit specifier provides the means of specifying one, or more, external unit identifiers of units to be preconnected to an external file.

The form of an alternate unit specifier is:

```
an = fn
```

where: an is a symbolic name such that nHan, where the value of n is the number of characters in an, is an external unit identifier. an is an alternate unit name.

fn is a symbolic file name.

An alternate unit name an may appear only in one alternate unit specifier in a program. It must not appear in as file declaration specifier of that program.

A symbolic file name fn that appears in an alternate unit specifier must have appeared previously in a file declaration specifier in the same PROGRAM statement. Note that a particular symbolic file name may appear in more than one alternate unit specifier.

The appearance of an alternate unit specifier an=fn has the same unit-file connection effect as the execution of an OPEN statement prior to the execution of any input/output statement that refers to the file defined by fn.

The OPEN statement has the form:

```
OPEN(UNIT=nHan,FILE='fn', . . .)
```

where n is the number of characters in 'an'. If the character string 'an' has either the character string 'TAPE' or the characters string 'UNIT' as an initial substring and has a digit string u, with a non zero leading digit, as its only other characters, then the unit-file connection affect of a second OPEN statement of the form:

```
OPEN(UNIT=u,FILE='fn', . . .)
```

is implied. Note that if 'an' is either 'TAPE0' or 'UNIT0' the OPEN statement is equivalent to:

```
OPEN(UNIT=0,FILE='fn', . . .)
```

The scope of an alternate unit name is the PROGRAM statement in which it appears.

MAIN PROGRAM RESTRICTIONS

The PROGRAM statement may appear only as the first statement of a main program. A main program may contain any other statement except a BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN statement. The appearance of a SAVE statement in a main program has no effect.

A main program may not be referenced from a subprogram or from itself.

B: A main program may optionally begin with a PROGRAM statement.

logical unit number must be used instead of the name; therefore, the logical unit number must be associated with only one file name. Even files that are mentioned only in a subprogram must appear in the PROGRAM statement of the main program.

INPUT or INPUT=f declares the file read by a READ statement without a file designator. OUTPUT or OUTPUT=f declares the file written by a PRINT statement, and also declares the file to which diagnostics, as well as STOP and PAUSE messages, are written. If neither OUTPUT nor OUTPUT=f is specified, OUTPUT is declared implicitly. PUNCH or PUNCH=f declares the file written by a PUNCH statement.

Note that the declaration (OUTPUT=DIAG,UNIT6=OUTPUT) would send diagnostics and PRINT output to the file DIAG, and would send unit 6 output to the file OUTPUT. The declaration (OUTPUT=OUT,UNIT6=OUT) would send diagnostics, PRINT output, and unit 6 output to the file OUT.

Files are opened at run time upon processing of the PROGRAM statement. The file search order used to find a file with a particular name is:

1. If a private file (local or attached permanent) exists, the private file is opened and used.

2. If an attached pool file exists, the pool file is opened and used.

3. If no file is found, a local file is REQUESTed with a length of 128 blocks.

For example, if the user declares PUNCH in the PROGRAM statement, a file named PUNCH of length 128 is created unless it already exists. OUTPUT is also created with length 128 unless a file called PRFILE exists prior to execution. If it does, PRFILE is renamed as OUTPUT and used (or renamed as f if OUTPUT=f was declared). This allows the user to specify an output file length other than the default value of 128. Such an expedient is necessary, because the file named OUTPUT – unlike other files – cannot be precreated in a batch job.

At the end of execution, the length of a disk output file will be reduced if the last operation on the file was a write operation or an ENDFILE. The length of the file is reduced from 128 blocks (or the user-specified length) to the number of blocks actually written.

Declaration of Files for Explicit I/O

Files can be specified in the PROGRAM statement to be explicit files (see section 13 for a discussion of implicit and explicit I/O on SRM-structured files) by providing four parameters enclosed in brackets following the TAPE or UNIT specification. Tape files must be explicit, but disk files can be either explicit or implicit. The files INPUT, OUTPUT, and PUNCH cannot be accessed explicitly.

Form:

$[p_1, p_2, p_3, p_4]$

p_1 Omit this parameter for disk (p3=4). Number of tape tracks:

7 = 7-track tape

9 = 9-track tape

p_2 Omit this parameter for disk (p3=4). Tape recording density in bpi:

200 = 7-track tape, bpi density of 200

556 = 7-track tape, bpi density of 556

800 = 7- or 9-track tape, bpi density of 800

1600 = 9-track tape, bpi density of 1600

p_3 Recording mode:

0 = 7-track tape, BCD mode, even parity

1 = 7- or 9-track tape, binary mode, odd parity

2 = 7-track tape, CDC 64-character ASCII subset, odd parity

4 = Disk

For values of 0 and 2, conversion takes place from binary data into BCD and ASCII characters respectively.

p_4 Buffer size specified as the number of small pages in the buffer. The value can be from 1 to 24. Default is 3.

The commas must remain to indicate preceding parameters that are unspecified. For example, the statement PROGRAM P (TAPE5[,,4]=FILE1) declares the file FILE1 to be an explicit disk file with a default buffer size of three small pages.

For transferring data in quantities of over three small pages in length, explicit input/output is generally more efficient in that fewer system calls are generated and more data is passed per call. However, this efficiency is degraded if the system is overloaded with jobs to the degree that physical memory becomes filled and the system must start swapping pages in and out of memory. For transferring data in quantities of less than three small pages in length, implicit input/output is simpler and is comparable (with respect to efficiency) with explicit input/output. All buffer statement input/output should be (and all tape input/output must be) performed on explicit files.

Parameters must be supplied at the first reference within the PROGRAM statement to an explicit file and are not allowed for subsequent references to the same file. If TAPE7 is to be an explicit tape file associated with file name DATA1, the following statement is correct:

```
PROGRAM P (TAPE6 [7,800,1] = FIL1,TAPE7=FIL1)
```

The following statement is not correct:

```
PROGRAM P (TAPE6=FIL1,TAPE7[7,800,1]=FIL1)
```

The explicit parameters given with TAPE7 are ignored and TAPE7 becomes an implicit disk file, the same as TAPE6.

The RLP parameter is used to request the mapping of dynamic space into large pages. The number of large pages is specified by m. If m is omitted, one large page is assumed. The RLP parameter can be used to improve the performance of programs that use large vector temporaries. Dynamic space includes vector temporaries and vectors assigned with the ASSIGN statement using DYN.

A: delete page 7-2.

STATEMENT FUNCTIONS

A statement function is a procedure defined by a single statement. A statement function must be defined in the program unit that references it; consequently, the function cannot be referenced by any other program unit.

DEFINING STATEMENT FUNCTIONS

E The user defines a statement function with a single statement similar in form to an assignment statement. The statement function must precede the first executable statement in the program unit, and must follow all nonexecutable statements except DATA, FORMAT, or NAMELIST statements.

Form:

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

f	The function's symbolic name.
a_i	A Dummy argument, a simple variable name distinct from any of the other dummy arguments. The list must be present, and it must contain at least one dummy argument (that is, $n \geq 1$).
e	Any scalar expression.

B Since dummy arguments serve only to indicate type, length, number, and order of the actual arguments, the names of dummy arguments can be the same as variable names of the same type and length appearing elsewhere in the program unit. Besides the dummy arguments, the expression e can contain constants, variables, array elements (the array name cannot be dummy), references to external functions (function subprograms and FORTRAN-supplied functions, for instance), and previously-defined statement functions.

The type of the statement function result is determined by the type of the function name. Type must be assigned to the function name in the same way that type is assigned to a variable; that is, the function name can either appear in an explicit type statement or be typed implicitly. Although the function name can appear in a type statement, it must not appear in an EQUIVALENCE, COMMON, or EXTERNAL statement, and must not be dimensioned or given an initial value. Type conversion from the expression type to the function name type occurs as for assignment statements (see table 4-1).

REFERENCING STATEMENT FUNCTIONS

A statement function is referenced when the function name suffixed with an actual argument list appears in an arithmetic, logical, or character expression. The actual arguments, each of which can be any scalar expression of the same type as the corresponding dummy argument, must agree in order, number, and length with the dummy arguments.

Evaluation of a statement function occurs during evaluation of an expression that contains a reference to the function. The values of the actual arguments are the values they have at the time of each evaluation of the function, while any name in the function expression that is not a dummy argument retains the value it would have, had it occurred outside the function at that time.

Examples:

Definition	Reference
ADD(X,Y,C,D)=X+Y+C+D	RZLT=GROSS-ADD(TAX, FICA,INS,RES)
AVG(O,P,Q,R)=(O+P+Q+R)/4	GRADE=AVG(T1,T2,T3,T4) +MID
LOGICAL A,B,EQV EQV(A,B)=(A.AND.B).OR. (.NOT.A.AND..NOT.B)	TEST=EQV(MAX,MIN).AND. ZED
COMPLEX Z Z(X,Y)=(1.,0.)*EXP(X)*COS(Y) +(0.,1.)*EXP(X)*SIN(Y)	RZLT2=(Z(BETA,GAMMA (I+K))**2-1.)/SQRT(T2)

SUBPROGRAMS

A subprogram is a program unit that is defined by more than one statement but is not a main program. The differences between function and subroutine specification and use are summarized in table 7-1. All references in the table to function name and subroutine name apply also to function entry point name and subroutine entry point name, respectively.

An external procedure is a procedure defined externally to the program units that reference it. Function and subroutine subprograms are external procedures that are written in FORTRAN. In-line functions and statement functions are not external procedures. Because name C

TABLE 7-1. DISTINGUISHING FUNCTIONS AND SUBROUTINES

F

	Function	Subroutine
How referenced	The function name appears in an expression.	The subroutine name appears in a CALL statement.
Arguments	One or more arguments must appear with the function name.	The subroutine name can appear with or without an argument list.
Type and length	The type and length of a function name is the type and length of the function result.	No type or length is associated with the name.
Results	A function must return a value through the function name. It can also return any number of values through arguments and COMMON.	A subroutine can return any number of values through arguments and COMMON.

A: . . . arguments. The parentheses are required even if there are no dummy arguments.

B: . cannot be dummy), references to external, intrinsic, and dummy functions . .

C: . . . written in FORTRAN. Statement functions are not external procedures. Intrinsic functions are not external procedures even if they invoke routines in the FORTRAN library. Because name . . .

D: . . . agree in order and number with the dummy arguments.

For a character argument, the length of the actual argument must be at least as great as the length of the dummy argument. If the length of the actual argument is greater, the excess characters are ignored.

E: . . . nonexecutable statements except DATA, FORMAT, ENTRY and . . .

F: TABLE 7-1. DISTINGUISHING FUNCTIONS AND SUBROUTINES

	Function	Subroutine
How referenced	The function name appears in an expression. Parentheses after the name are required even if there are no arguments.	The subroutine name appears in a CALL statement. Parentheses after the name are optional.
Type and length	The type and length of the function name are the type and length of the function result.	No type or length is associated with the name.
Results	A function must return a value through the function name. It can also return any number of values through arguments and COMMON, so long as it does not alter the value of any thing which occurs elsewhere in the statement containing the function reference, and does not alter a value in COMMON which affects the value of any other function reference in the statement.	A subroutine can return any number of values through arguments and COMMON.
Alternate return	Alternate return specifiers may not occur as arguments.	Alternate return specifiers may occur as agruments.

definitions for data are local to the program unit in which the names appear, names within an external procedure can be used in other program units of the same executable program to refer to unrelated entities.

PASSING ARGUMENTS BETWEEN SUBPROGRAMS

A transfer of control out of a program unit takes place when a CALL statement or external function reference is executed. Argument associations are made, and the referenced program unit executes until a RETURN statement relinquishes control to the referencing program unit. Upon return, any definitions made of arguments persist. If a STOP statement is executed within the referenced subprogram, program execution is terminated without control being returned to the referencing program unit.

Values can be made available to an external procedure in two ways: through use of COMMON statements and by means of argument lists. See section 6 for a discussion of COMMON statement usage.

Dummy and actual argument lists are the mechanism that FORTRAN employs to pass values between subprograms. An argument's being dummy or actual depends upon the context in which the argument appears. An argument appearing in a FUNCTION, SUBROUTINE, or ENTRY statement is a dummy argument, while an argument appearing in a subprogram reference is an actual argument. At the time a subprogram reference is executed, each variable listed as a dummy argument is associated with the same storage location as the actual argument corresponding to it (call by address). Each definition of a dummy argument can change the value in that storage location. Thus, when control returns to the referencing program unit, the values of the actual arguments can be different from what they were before the subprogram reference.

Dummy arguments are variable names, array names, external subprogram names, or (for subroutine definitions only) multiple return statement label indicators (asterisks). They are assigned data types as appropriate and are used in the executable statements of the subprogram. Actual arguments can be expressions, variables (including descriptors, and double descriptors), vectors, constants, arrays, array elements, external procedures, or (for subroutine calls only) labels in the calling program unit. (A label is prefixed with an ampersand.) The dummy argument list for a subprogram and an actual argument list for a reference to the same subprogram must agree in argument order, number, data type, and length (length is applicable to type CHARACTER elements only). The only exception is that actual arguments which are character or Hollerith constants can also correspond to dummy arguments of a type other than character.

Dummy argument arrays, like all other arrays, must have their sizes declared. The declarator dimensions can be integer constants, or simple integer variables which either must be dummy arguments as well or else must be in common. A dummy argument must never appear in a COMMON, EQUIVALENCE, or DATA specification statement.

If an actual argument is an external subprogram name, the name must appear in an EXTERNAL statement in the referencing program unit. Furthermore, the corresponding dummy argument can only be used as an external subprogram reference or as an actual argument in a subprogram reference in the referenced subprogram. An example of this usage is shown in figure 7-1. As a result of the first call to S, SAM is executed on the call to SUB; on the second call to S, TIME is executed on the call to SUB. However, if the external subprogram name is suffixed with an argument list then the name is not an argument but a function reference; here, the function is executed and it is the result that becomes the actual argument. A function referenced in an argument list need not have its name appear in an EXTERNAL statement in order to act as an argument. An example of this usage is shown in figure 7-2. The value of RZLT is the type real value returned by the execution of SAM.

```
PROGRAM P
EXTERNAL SAM,TIME
   :
CALL S (X,Y,Z,SAM,I)
   :
CALL S (T,U,V,TIME,W)
   :
END

SUBROUTINE S (A,B,C,SUB,D)
   :
CALL SUB
   :
RETURN
END
```

Figure 7-1. Subprogram Name as Actual Argument

```
PROGRAM R
   :
CALL S (X,Y,Z,SAM(X),I)
   :
END

SUBROUTINE S (A,B,C,RZLT,D)
   :
DIMP = RZLT**2/NIM+1.
   :
RETURN
END
```

Figure 7-2. Subprogram Reference as Actual Argument

Kinds of actual arguments allowed to correspond with a particular type of dummy argument are listed in table 7-2. When a dummy argument is associated with an actual argument that is either a constant or an expression containing operators, the dummy argument must not be defined in the subprogram.

A: . . . arrays, array elements, external, intrinsic, or dummy procedures, or (for . . .

B· label is prefixed with an asterisk or an ampersand.) The dummy argument.

C. . . . their sizes declared. Dimension bound expressions for dummy argument arrays may contain integer variables which are in common or are dummy arguments. In the latter case, each such variable must occur in the argument list of every ENTRY, SUBROUTINE, and FUNCTION statement which contains the array name. The upper bound of the last dimension (first dimension for ROWWISE arrays) may be an asterisk. The array is then called an assumed-size array. An assumed-size array may not appear without subscripts in an I/O list or in an array assignment. A dummy argument must never appear in a . . .

D: If an intrinsic function is used as an actual argument, it must appear in an INTRINSIC statement. If a subroutine, external function, or dummy procedure is used as an actual argument, it must appear in an EXTERNAL statement. The corresponding dummy argument in the referenced subprogram may be used as an actual argument and/or in subprogram references. An . . .

E: Dummy arguments are variables, arrays, dynamic variables, dynamic arrays, dummy procedures, or (for subroutine definitions . . .

F: For a character argument, the length of the actual argument must be at least as great as the length of the dummy argument. For character arrays, what matters is the total length in characters of the entire array – it is irrelevant how many characters are in each array element. If the length of the actual arguments exceeds the length of the dummy argument, the excess cahracters are ignored

When an H constant occurs as an actual argument, the compiler appends blanks if required to fill out a whole number of words. If the corresponding dummy argument is complex or double precision, the H constant must contain at least 9 characters in order to be padded out to 2 words (16 characters).

G: . . . order, number, and data type. The only exception is . . .

TABLE 7-2. CORRESPONDENCE OF ACTUAL TO DUMMY ARGUMENTS

Dummy Argument	Actual Argument
Simple variable	Scalar expression
Descriptor	Descriptor Descriptor array element Vector
Double descriptor	Double descriptor Double descriptor array element
Simple array	Simple array Array element (simple)
Descriptor array	Descriptor array Descriptor array element
Double Descriptor array	Double descriptor array Double descriptor array element
External procedure name	External procedure name
* (asterisk denoting dummy label - for subroutines only)	Statement label, prefixed by an ampersand
* (asterisk denoting vector function result)	Descriptor Descriptor array element Vector

FUNCTION SUBPROGRAMS

A function subprogram is a program unit whose first line is a FUNCTION statement. A function subprogram must be referenced in at least one other program unit to be executed, and contains a RETURN statement to return control to the referencing program unit. Statements that cannot be included in a function subprogram are the PROGRAM, BLOCK DATA, and SUBROUTINE statements, and any statement that directly or indirectly references the function being defined. The execution of a STOP statement within the function terminates the program.

The FUNCTION statement defines the program unit to be a function and not a subroutine or the main program. Only one FUNCTION statement is allowed in a subprogram.

Forms:

t FUNCTION f $(a_1,a_2,\ldots,a_n)$

CHARACTER FUNCTION f*m $(a_1,a_2,\ldots,a_n)$

t	Optional. A declaration of the type of f; can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or, as in the second form, CHARACTER.
f	The function's symbolic name.
m	Length specification, in bytes, of the character function result returned as the value of f. When *m is not specified in the second form, the assumed length is 1.
a_i	A dummy argument that can be a variable, array, or external procedure name. No two dummy arguments can have the same name. At least one argument is required.

Within the function, the name f is treated as a variable. It must be given a value at least once during the execution of the function subprogram. Once defined, the function name can be referenced and redefined without an occurrence of the name being interpreted as a function self-reference. The value returned to the expression that referenced the function f is the value that f has upon execution of a RETURN statement within the function subprogram.

The type of the function name f must be the same as in any program unit that references the function. Type specification can be explicit – it can appear before the word FUNCTION or else appear in a type declaration statement within the function (f must not be initialized) or it can be implicit. Implicit type specification takes effect only when no explicit typing of the function name was used. The function name must not appear in any nonexecutable statements within the function, except for purposes of type declaration or in a list of identifier names in a NAMELIST statement.

If the function name f is the same as that of a predefined function, the predefined function is unavailable in the user-defined function. Throughout the rest of the program, a reference to a function named f causes execution of the user-defined function unless the predefined function f is in-line (see appendix E to determine whether f is in-line or external). The presence of an external declaration for f governs whether or not an in-line predefined function is executed.

A function subprogram can modify the value of one or more of its arguments to return extra (side effect) values to the referencing program unit, with one restriction: because the order of evaluation of the components of an expression or statement is not guaranteed, a function reference must not define any other entity occurring in the same statement. The function's capability for modifying its arguments also applies to individual elements of an argument which represents an array. Other values can be returned by altering the values of entities in COMMON (the same side effect restriction applies). For example, given the statement

```
X(T) = FN(T,I+N,Y) + 3*FN(I,N,Z) - R
```

where X is an array, FN is a function, and R is in common, the variables T, I, N, and R must not be defined by FN. However, Z and Y can be so defined.

A function is referenced by using its name suffixed by an argument list, including parentheses and commas, instead of a data element in any expression. Each dummy argument in the FUNCTION statement must correspond to an actual argument in the function reference argument list. See the heading Passing Arguments Between Subprograms in this section for a further description of actual and dummy arguments in function references.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is a program unit whose first line is a SUBROUTINE statement. To be executed, a subroutine subprogram must be referenced with a CALL statement in another program unit; a RETURN statement returns control to the calling program unit. Statements that cannot be included in a subroutine subprogram are the PROGRAM,

A:	Dummy procedure	Subroutine External function Intrinsic function Dummy procedure.
B:	* (asterisk denoting dummy label – for subroutines only)	Statement label, prefixed by an asterisk or an ampersand.
C:		Optional. May be INTEGER, REAL, HALF PRECISION, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER or CHARACTER*m.

D: Intrinsic function except in program units where the function name appears in an EXTERNAL statement.

E: . . . altering the values of entities in COMMON. A value in COMMON may not be altered if it occurs in the statement containing the function reference, or if its value affects the value of another function reference in the same statement. For example, given the . . .

F: . . . a data element in any expression. The parentheses are required even if there are no arguments. Each dummy argument in . . .

G: m may be any integer constant expression where value is greater than 0. Alternately, m may be an asterisk enclosed in parentheses: (*). In this case the length of the result is determined by the type declaration for f in the referencing program unit.

H: a_i A dummy argument that can be a variable, array, dynamic variable, dynamic array, or dummy procedure name. The parentheses are required even if there are no arguments. No two dummy arguments can have the same name.

I: . . . executed, and must contain a RETURN statement to return . . .

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

K: . . . of f. When *m is not specified the assumed length is 1.

L: If the function name f is the same as that of an intrinsic function, the intrinsic function is unavailable in the user- . . .

BLOCK DATA, and FUNCTION statements and any statement that directly or indirectly references the subroutine being defined. The execution of a STOP statement within the subroutine causes the program to terminate.

The SUBROUTINE statement defines the program unit to be a subroutine and not a function or the main program. Only one SUBROUTINE statement is allowed in a subprogram.

Form:

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

s	The subroutine's symbolic name.
a_i	Optional. A dummy argument that can be a variable, array, external procedure name, or an * denoting a return point specified by a statement label in the calling program unit. When the argument list is omitted, the parentheses and commas must also be omitted.

A
I

The SUBROUTINE statement contains the subprogram name s that indicates the subprogram's main entry point (the first executable statement in the subroutine). The name s is not used to return results to the calling program the way that function names do, is not associated with a data type, and must not appear in any statement in the subprogram except the SUBROUTINE statement. Results are returned to the calling program unit only through definition or redefinition of one or more of the dummy arguments or through common. Dummy arguments in a SUBROUTINE statement are discussed elsewhere in this section under Passing Arguments Between Subprograms.

Whenever an asterisk occurs as a dummy argument in the SUBROUTINE statement, there must be the statement label (preceded by an ampersand) of a statement in the calling routine as the corresponding actual argument. In the CALL statements used to reference subroutine subprograms, an argument is a statement label if it is a string composed of an ampersand followed by the digits required for the label.

B
C

BLOCK DATA SUBPROGRAMS

Besides having one or more executable program units, a program can contain nonexecutable BLOCK DATA subprograms. A BLOCK DATA subprogram is a STAR FORTRAN specification subprogram consisting of only the following kinds of statements:

BLOCK DATA statement
IMPLICIT statements
explicit type statements
EQUIVALENCE statements
DIMENSION statements
ROWWISE statements
COMMON statements
DESCRIPTOR statements
DOUBLE DESCRIPTOR statements
DATA statements

J
D

The order of the statements in a BLOCK DATA subprogram should be as shown in figure 1-2.

A subprogram is a specification subprogram if the first statement is a BLOCK DATA statement.

Form:

BLOCK DATA b

b	Optional. Symbolic name of subprogram.

The single function of a BLOCK DATA subprogram is to initialize the values of elements in labeled common blocks (but not blank common) prior to program execution. If any element in a given common block is being given an initial value in such a subprogram, a complete set of specification statements for the entire common block must be present (including any type, EQUIVALENCE, and DIMENSION statements required to fully specify the common block's organization), except that not all of the elements of the block need be initialized. Initial values can be entered into more than one block in a single subprogram. Different variables and array elements in a common block can be initialized in different program units, but no variable or array element can be initialized more than once.

MULTIPLE ENTRY SUBPROGRAMS

The first executable statement following a FUNCTION or SUBROUTINE statement is the main entry point to that subprogram. Other entry points can be defined in subroutine and function subprograms by using the ENTRY statement: the ENTRY statement in a subprogram specifies that the first executable statement following the ENTRY statement is a secondary entry point. More than one secondary entry point can be declared in a subprogram.

Like the FUNCTION and SUBROUTINE statements, an ENTRY statement is not executable and has no effect on the logical flow of subprogram execution other than to specify where subprogram execution is to begin when the subprogram is referenced; also, like those statements, an ENTRY statement must not be labeled. An ENTRY statement can occur anywhere within a subroutine or function subprogram except within the range of a DO; however, at least one executable statement must appear between an ENTRY statement and the END line in the subprogram. An ENTRY statement must not appear in a main program or in a BLOCK DATA subprogram.

E
F

Form:

ENTRY e (a_1,a_2, . . . ,a_n)

e	The symbolic name of the entry point.
a_i	Dummy argument that can be a variable, array, external procedure name, descriptor, or (in a subroutine subprogram) an * denoting a return point specified by a statement label in the calling program unit. Argument list is optional for an ENTRY statement in a subroutine subprogram. When argument list is omitted, the parentheses and commas must also be omitted. At least one argument is required for an ENTRY statement in a function subprogram.

G
H

Control passes to the first executable statement following the ENTRY statement when the entry point name s is used in a CALL statement or function reference. In a subroutine subprogram, the entry point name s is not associated with a data type and must not appear in any statement in the subprogram except the ENTRY statement. In a function subprogram, however, the entry point name s must be associated with a data type implicitly or with explicit type statements. The distinctions between entry points in functions and subroutines are shown in table 7-1.

FUNCTION SUBPROGRAM ENTRY POINT NAMES

An entry point name in a function subprogram must be associated with a data type and can be assigned values

A. . . . variable. array, dynamic variable, dynamic array, dummy procedure, or . . .

B: . . . SUBROUTINE statement, there must be the statement label of a statement in the calling . .

C: . . . an asterisk or an ampersand followed by the digits required for the label.

D: PARAMETER statements
SAVE statements
END statements

E: . . . gram is referenced. An ENTRY statement can

F: . . . except within the range of a DO, or between a block IF statement and its corresponding END IF statement. An ENTRY . . .

G: . . . array, dynamic variable, dynamic array, dummy procedure, or . . .

H: . . . omitted, the parentheses are optional.

I: . . these are optional.

J: Delete

during execution. The entry point name must not appear in any nonexecutable statement in the function except in a FUNCTION or ENTRY statement, explicit type statement, or in the list of names in a NAMELIST statement.

An entry point name need not be of the same data type as the main entry point name or any other secondary entry point names in the function; however, a function reference using that entry point name must have the same data type as the name. Also, STAR FORTRAN permits scalar function subprograms to have vector function entry points, and vector functions (see section 11) to have scalar function entry points.

All entry point names in a function are associated so that a definition of one causes definition of all others having the same type and length, and causes undefinition (unpredictable values) of those having a different type or length association. In effect, all entry point names are equivalenced as in an EQUIVALENCE statement.

During each execution of the subprogram, at least one of the entry point names must be assigned a value (become defined), and once defined can be referenced and redefined. (A reference to the entry point name within the function refers to this value and is not a reference to the function.) A An entry point name having the same type and length as the entry point name used to enter the subprogram must be defined at the time of execution of any RETURN statement in the subprogram; the value of the name at that time is the function value returned to the referencing program unit.

SECONDARY ENTRY POINT ARGUMENT LISTS

B An entry point to a function subprogram must have at least one argument, and an entry point to a subroutine subprogram need have no arguments. A subprogram can modify the value of one or more of the arguments in the argument list of the ENTRY statement associated with the current entry to return values to the calling program unit. See the heading Passing Arguments Between Subprograms earlier in this section for specifications for dummy arguments in ENTRY statements.

The list of arguments in an ENTRY statement need not contain the same elements as other argument lists in FUNCTION, SUBROUTINE, or other ENTRY statements in the same program unit. Nevertheless, no statement in the subprogram can be executed that would cause reference or definition of an argument not in the argument list of the current entry. C

REFERENCING SECONDARY ENTRY POINTS

A secondary entry point to a subroutine subprogram is referenced by a CALL statement containing the entry point name. An example of multiple subroutine entry points is shown in figure 7-3. In the example, the statement CALL CLEAR(SET1) references the primary entry point of the subroutine. Elements of the array are set to zero before values are read into the array. Later in the program, the statement CALL FILL(SET1) references the secondary entry point FILL. Values are read into the array without any initialization of the elements to zero.

A secondary entry point to a function is referenced in the same way that the main entry point is referenced. See the heading Passing Arguments Between Subprograms earlier in this section for actual argument list specifications. An example of multiple function entry points is shown in figure 7-4. In the example, the statement RT1 = FSHN(X,Y,Z) references the primary entry point of the function. The calculation of the FSHN value is performed, and control returns to the main program. Later in the program, the statement RT2 = FRED(R,S,T) references the secondary entry point FRED. Depending on the value of the first argument, the return value is either the calculated value of FRED or FSHN. Since multiple function entry point names are effectively equivalenced, either FRED or FSHN can be used to set the return value.

```
      PROGRAM T(INPUT)
      DIMENSION SET1(25)
          .
          .
          .
      CALL CLEAR(SET1)
          .
          .
          .
      CALL FILL(SET1)
          .
          .
          .
      END

      SUBROUTINE CLEAR(RA)
      DIMENSION RA(25)
      INTEGER P
C—MAIN ENTRY POINT
      DO 100 I = 1,25
100   RA(I) = 0.0
      ENTRY FILL(RA)
C—SECONDARY ENTRY POINT
300   READ 2, V,P
2     FORMAT(10X, F7.2, I4)
      RA(P) = V
      IF(P.LT.0.OR.P.GT.25) RETURN
      GOTO 300
      END
```

Figure 7-3. Multiple Entry Subroutine

```
      PROGRAM Q
          .
          .
          .
      RT1 = FSHN(X,Y,Z)
          .
          .
          .
      RT2 = FRED(R,S,T)
          .
          .
          .
      END

      FUNCTION FSHN(A,B,C)
C—MAIN ENTRY POINT
300   FSHN = A*B/C**2
      RETURN
      ENTRY FRED(A,B,C)
C—SECONDARY ENTRY POINT
      IF(A.LE.702) GOTO 300
      FRED = (C+A)/B
      RETURN
      END
```

Figure 7-4. Multiple Entry Function

Subroutines cannot reference their own main entry points or secondary entry points directly or indirectly. A function subprogram can reference any of its entry point names, so long as the name is not followed by an argument list, because this does not constitute a function reference.

A: An entry point name having the same type and the same or greater length as the . . .

B: A subprogram can modify the . . .

C: . . . current entry. If a dummy array is an argument in an ENTRY list, each variable which occurs in a dimension bound expression for the array must be in common or in the argument list of the same ENTRY statement. (The same rule also applies to FUNCTION and SUBROUTINE statements.)

Replace all of Section 8 with the following pages.

CDC CYBER 200 FORTRAN 77 INPUT/OUTPUT STATEMENTS

INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called reading. Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called writing. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to inquire about or describe the properties of the connection to the external medium.

There are 14 input/output statements:

1. READ
2. WRITE
3. PRINT
4. OPEN
5. CLOSE
6. INQUIRE
7. BACKSPACE
8. ENDFILE
9. REWIND
10. PUNCH
11. ENCODE
12. DECODE
13. Q7BUFIN
14. Q7BUFOUT

The READ, WRITE, PRINT, PUNCH, Q7BUFIN, and Q7BUFOUT statements are data transfer input/output statements. The OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, and REWIND statements are auxiliary input/output statements. The BACKSPACE, ENDFILE, and REWIND statements are file positioning input/output statements.

RECORDS

A record is a sequence of values or a sequence of characters. For example, a punched card is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of record:

1. Formatted
2. Unformatted
3 Endfile

FORMATTED RECORD

A formatted record consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only for formatted input/output statements.

Formatted records may be prepared by some means other than FORTRAN; for example, by some manual input device.

UNFORMATTED RECORD

An unformatted record consists of a sequence of values in a processor dependent form and may contain both character and non-character data or may contain no data. The length of an unformatted record is measured in processor dependent units and depends on the output list used when it is written, as well as on the processor and the external medium. The length may be zero.

The only statements that read and write unformatted records are unformatted input/output statements, Q7BUFIN and Q7BUFOUT statements.

Unformatted Records Containing Data of Type Bit

The smallest unit of storage that may be transferred to or from an external file is a character. When data of type bit are being transmitted at least one character will be read from or written on the external file.

Let _b_ be the number of bits occupied by the data of type bit.

On input, the leftmost _b_ bits will be transferred to the internal storage from the next INT((_b_+7)/8) characters read from the current record. Unused bits will be skipped.

On output, INT((_b_+7)/8) characters will be written on the external file. Unused bits will be undefined.

ENDFILE RECORD

An endfile record is written by an ENDFILE statement. An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

FILES

A _file_ is a sequence of records.

There are two kinds of file:

1. External
2. Internal

Internal files are also categorized by the type of storage provided for the file.

FILE EXISTENCE

At any given time, there is a processor determined set of files that are said to exist for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, security reasons may prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of the file.

All input/output statements may refer to files that exist. The INQUIRE, OPEN, CLOSE, WRITE, PRINT, PUNCH, Q7BUFOUT and ENDFILE statements may also refer to files that do not exist.

FILE PROPERTIES

At any given time, there is a processor determined set of allowed access methods, a processor determined set of allowed forms, and a processor determined set of allowed record lengths for a file.

Each external file has exactly one file name, and is called a named file. The name of a named file is a character string, consisting of one to eight letters or digits, the first of which must not be a digit.

An external file may have zero or more alternate file names. An alternate file name provides a means of referring to an external file by more than one unit identifier. An alternate file name has the form of a file name, and is specified by the PROGRAM statement.

Both a file name and an alternate file name are global to the executable program. However, the scope of the file name extends to the external environment of the program, such as the processor operating system; the scope of the laternate file name is restricted to the executable program.

Note that, unlike a file name, an alternate file name is not a property of an external file. For example, execution of an INQUIRE by file statement that refers to an external file by an alternate file name and inquires the NAME causes the specifier variable fn to become defined with the file name, not an alternate file name.

An internal file does not have a name.

FILE POSITION

A file that is connected to a unit has a position property. Execution of certain input/output statements affects the position of a file. Certain circumstances can cause the position of a file to become indeterminate.

The initial point of a file is the position just before the first record. The terminal point is the position just after the last record.

If a file is positioned within a record, that record is the current record; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \leq n$ and a file is positioned within the ith record or between the (i-1)th record and the ith record, the (i-1)th record is the preceding record. If $n \geq 1$ and a file is positioned at its terminal point, the preceding record is the nth and last record. If n=0 or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \leq i < n$ and a file is positioned within the ith record or between the ith and (i+1)th record, the (i+1)th record is the next record. If $n \geq 1$ and the file is positioned at its initial point, the first record is the next record. If n=0 or if a file is positioned at its terminal point or within the nth and last record, there is no next record.

FILE ACCESS

There are two methods of accessing the records of an external file. sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing a file is determined when the file is connected to a unit.

Sequential Access

When connected for sequential access, a file has the following properties:

- The order of the records is the order in which they were written. A record that has not been written since the file was created must not be read.

- The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record.

- The records of the file must not be read or written by direct access input/output statements.

Direct Access

When connected for direct access, a file has the following properties:

- The order of the records is the order of their record numbers. The records may be read or written in any order.

- The records of the file are either all formatted or all unformatted. The file must not contain an endfile record.

- Reading and writing records is accomplished only by direct access input/output statements.

- All records of the file have the same length.

- Each record of the file is uniquely identified by a positive number called the record number. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, it may be rewritten.

- Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected to a unit. For example, it is permissible to write record 3 even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record was written since the file was created.

- The records of the file must not be read or written using list-directed formatting.

INTERNAL FILES

Internal files provide a means of transferring and converting data from internal storage to internal storage.

There are two types of internal file, standard and extended. A standard internal file is a sequence of character storage units. An extended internal file is a sequence of numeric storage units. An extended internal file may only be accessed by the ENCODE and DECODE statements.

The standard or extended property of an internal file is established by the type of storage provided for the file.

Throughout the remainder of this document, the phrase internal file shall be interrupted to mean standard internal file, unless explicitly prefixed with extended.

Standard Internal File Properties

A standard internal file has the following properties:

- The file is a character variable, character array element, character array, or character substring.

- A record of an internal file is a character variable, character array element, or character substring.

- If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring, respectively. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of records in the file is the same as the ordering of the array elements in the array. Every record of the file has the same length, which is the length of the array element in the array.

- The variable, array element, or substring that is the record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

- A record may be read only if the variable, array element, or substring that is the record is defined.

- A variable, array element, or substring that is a record of an internal file may become defined (or undefined) by means other than an output statement. For example, the variable, array element, or substring may become defined by a character assignment statement.

- An internal file is always positioned at the beginning of the first record prior to data transfer.

<u>Standard Internal File Restrictions</u>

A standard internal file has the following restrictions:

- Reading and writing records is accomplished only by sequential access formatted input/output statements that do not specify list-directed formatting or NAMELIST formatting.

- An auxiliary input/output statement must not specify an internal file.

<u>Extended Internal File Properties</u>

An extended internal file has the following properties:

- The file is a non-character variable, non-character array element, or non-character array.

- A record of the file is one or more contguous numeric storage units.

- The length of a record of the file is measured in characters, and is equal to

$$\underline{a} * \underline{m}$$

where: a is the maximum number of characters that can be stored in a single numeric storage unit at one time

m is the number of numeric storage units in the reocrd.

- Every record of the file has the same length.
- The variable or array element that is a record of the file is defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.
- A record may be read only if the variable or array element(s) that is the record is defined.
- A variable or array element(s) that is a record of the file may become defined (or undefined) by means other than an input statement.
- An extended internal file is always positioned at the initial point of the first record prior to data transfer.

Extended Internal File Restrictions

An extended internal file has the following restrictions:

- Reading and writing records is accomplished only be DECODE and ENCODE statements. List-directed formatting must not be specified.
- The file must be accessed sequentially.
- An auxiliary input/output statement must not specify an extended internal file.

UNITS

A unit is a means of referring to a file.

UNIT EXISTENCE

At any given time, there is a processor determined set of units that are said to exist for an executable program. A unit exists for each allowed external unit idenfifier.

All input/output statements may refer to units that exist. The INQUIRE and CLOSE statements may also refer to units that do not exist.

CONNECTION OF A UNIT

A unit has a property of being connected or not connected. If connected, it referes to a file. A unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric: If a unit is connected to a file, the file is connected to the unit.

Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore may be referenced by input/output statements without the prior execution of an OPEN statement. Each unit that exists is preconnected to a file. The file name of the file to which a unit is preconnected may be specified by a preconnection specifier in the PROGRAM statement of the main program. Otherwise, the processor determines the file name from the unit specifier u as follows:

- If INT(u) has a value representable by the digit string n in the range 0. . .99, the file name is TAPE$_n$.
- If u has a value of the form nHf, where f is a valid system file name, the file name is f.

Otherwise, the unit specified does not exist.

All input/output statements except OPEN, CLOSE, and INQUIRE must reference a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist. An example is a preconnected new file.

A unit must not be connected to more than one file at the same time, but an external file may be connected to more than one unit at the same time. However, means are provided to change the status of the unit and to connect a unit to a different file.

After a unit has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same file or a different file. After a file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same unit or a different unit. Note, however, that the only means to refer to a file that has been disconnected is by its name in an OPEN or INQUIRE statement. Therefore, there may be no means of reconnecting an unnamed file once it is disconnected.

UNIT SPECIFIER AND IDENTIFIER

The form of a unit specifier is:

(UNIT=)u

where u is an external unit identifier or an internal file identifier.

An external unit identifier is used to refer to an external file. An internal unit identifier is used to refer to an internal file.

An external unit identifier is one of the following:

1. An integer expression i whose value must be either

   a. An integer in the range 0. .999, or

   b. Of the form nHf, where f is a valid system file name.

   In case (b), if f is of the form TAPE≤ or UNITk, where k is an integer in the range 0. .999 with no leading zero, it is equivalent ot the integer k for the purpose of identifying external units.

2. An asterisk, identifying a particular processor determined external unit that is preconnected for formatted sequential access.

The external unit identified by the value of i is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
READ (6) X

SUBROUTINE B
N=6
REWIND n
```

the value 6 used in both program units identifies the same external unit.

An external unit identifier in an auxiliary input/output statement must not be an asterisk.

An internal file identifier provides the means of referring to a standard or extended internal file. An internal file identifier for a standard internal file is the symbolic name of a character variable, character array, character array element, or character substring. An internal file identifier for an extended internal file is the symbolic name name of a non-character variable, a non-character array, or non-character array element.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in a list of specifiers.

FORMAT SPECIFIER AND IDENTIFIER

The form of a format specifier is:

(FMT=)f

where f is a format identifier.

A format identifier identifies a format. A format identifier must be one of the following:

- The statement label of a FORMAT statement that appears in the same program unit as the format identifier.
- An integer variable that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier.
- A character array name.
- Any character expression except a character expression involving conatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. Note that a character constant is permitted.
- An asterisk, specifying list-directed formatting.
- A NAMELIST group name specifying NAMELIST formatting.
- A non-character array name.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

RECORD SPECIFIER

The form of a record specifier is:

REC = rn

where rn is an integer expression whose value is positive it specifies the number of the record that is to be read or written in a file connected for direct access.

ERROR AND END-OF-FILE CONDITIONS

The set of input/output error conditions is processor dependent.

An end-of-file condition exists if either of the following events occurs:

- An endfile record is encountered during the reading of a file connected for sequential access. In this case, the file is positioned after the endfile record.
- An attempt is made to read a record beyond the end of an internal file.

[illegible]

If an error condition occurs during the execution of an input/output statement, execution of the input/output statement terminates and the position of the file becomes indeterminate.

If an error condition or an end-of-file condition occurs during execution of a READ statement, execution of the READ statement terminates and the entities specified by the input list and implied-DO-variables in the input list become undefined. Note that variables and array elements appearing only in subscripts, substring expressions, and implied-DO parameters in an input list do not become undefined when the entities specified by the list become undefined.

If an error condition occurs during the execution of an output statement, execution of the output statement terminates and implied-DO-variables become undefined.

If an error condition occurs during execution of an input/output statement that contains neither an input/output status specifier nor an error specifier, or if an end-of-file condition occurs during execution of a READ statement that contains neither an input/output status specifier nor an end-of-file specifier, execution of the executable program terminates.

INPUT/OUTPUT STATUS, ERROR, AND END-OF-FILE SPECIFIERS

The form of an input/output status specifier is:

IOSTAT = ios

where ios is an integer variable or integer array element.

Execution of an input/output statement containing this specifier causes ios to become defined:

- with a zero value if neither an error condition nor an end-of-file condition is encountered by the processor,
- with a processor dependent positive integer value if an error condition is encountered, or
- with a processor dependent negative integer value if an end-of-file condition is encountered and no error condition is encountered.

The positive integer value denoting an error condition is, for each error, the same value used as the runtime error number when the input/output status specifier is omitted.

ERROR SPECIFIER

The form of an error specifier is:

ERR = s

where s is the statement label of an executable statement that appears in the same program unit as the error specifier.

If an input/output statement contains an error specifier an the processor encounters an error condition during the execution of the statement:

- execution of the input/output statement terminates,
- the position of the file specified in the input/output statement becomes indeterminate,
- if the input/output statement contains an input/output status specifier, the variable or array element ios becomes defined with a processor dependent positive integer value, and
- execution continues with the statement labeled s.

END-OF-FILE SPECIFIER

The form of an end-of-file specifier is:

END = s

where s is the statement label of an executable statement that appears in the same program unit as the end-of-file specifier.

If a READ statement contains an end-of-file specifier and the processor encounters an end-of-file condition and no error condition during the execution of the statment:

- execution of the READ statement terminates,
- if the READ statement contains an input/output status specifier, the variable or array element ios becomes defined with a processor dependent negative integer value, and
- execution continues with the statement labled s.

READ, WRITE, PRINT, AND PUNCH STATEMENTS

The READ statement is the data transfer input statement. The WRITE, PRINT, and PUNCH statements are the data transfer output statements. The forms of the data transfer input/output statements are:

READ (cilist) (iolist)

READ f(.iolist)

WRITE (cilist) (iolist)

PRINT f(,iolist)

PUNCH f(,iolist)

where: cilist is a control information list that includes:

- A reference to the source or destination of the data to be transferred.
- Optional specification of editing processes.
- Optional specifiers that determine the execution sequence on the occurrence of certain events.
- Optional specification to identify a record.
- Optional specification to provide the return of the input/output status.

f is a format identifier.

iolist is an input/output list specifying the data to be transferred.

If the format identifier f is a NAMELIST group name, the iolist must not be present.

CONTROL INFORMATION LIST

A control information list, cilist, is a list whose list items may be any of the following

```
(UNIT = ) u
(FMT = ) f
REC = rn
IOSTAT = ios
ERR = s
END = s
```

A control information list must contain exactly one unit specifier, at most one format specifier, at most one record specifier, at most one input/output status specifier, at most one error specifier, and at most one end-of-file specifier.

If the control information list contains a format specifier, the statement is a formatted input/output statement: otherwise, it is an unformatted input/output statement.

If the control information list contains a record specifier, the statement is a direct access input/output statement; otherwise, it is a sequential access input/output statement.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

The unit specifier must not specify an extended internal file.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

A control information list must not contain both a record specifier and an end-of-file specifier.

If the format identifier is an asterisk, the statement is a list-directed input/output statement and a record specifier must not be present.

In a WRITE statement, the control information list must not contain an end-of-file specifier.

If the unit specifier specifies an internal file, the control information list must contain a format identifier other than an asterisk and must not contain a record specifier.

INPUT/OUTPUT LIST

An input/output list, iolist, specifies the entities whose values are transferred by a data transfer input/output statement.

An input/output list is a list of input/output list items and implied-DO list. An input/output list item is either an input list item or an output list item.

If an array name appears as an input/output list item, it is treated as if all of the elements of the array were specified in the order given by array element ordering. The name of an assumed-size dummy array must not appear as an input/output list item.

Input List Items

An input list item must be one of the following:

- A variable name.
- An array element name.
- A character substring name.
- An array name.

Only input list items may appear as input/output list items in an input statement.

Output List Items

An output list item must be one of the following:

- A variable name.
- An array element name.
- A character substring name.
- An array name.
- Any other expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

Implied-DO List

An implied-DO list is of the form:

(dlist, i= e1 , e2 [,e3])

where: i, e1, e2, and e3 are as specified for the DO statement

dlist is an input/output list.

The range of an implied-DO list is the list dlist. Note that dlist may contain implied-DO lists. The iteration count and the values of the DO-variable i are established from e1, e2, and e3 exactly as for a DO-loop. In an input statement, the DO-variable i, or an associated entity, must not appear as an input list item in dlist. When an implied- DO list appears in an input/output list, the list items in dlist are specified once for each iteration of the implied- DO list with appropriate substitution of values for any occurrence of the DO-variable i.

EXECUTION OF A DATA TRANSFER INPUT/OUTPUT STATEMENT

The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

- Determine the direction of data transfer.
- Identify the unit.
- Establish the format if any is specified.
- Position the file prior to data transfer.
- Transfer data between the file and the entities specified by the input/output list (if any), or identified by association with a NAMELIST group name (if one).
- Position the file after data transfer.
- Cause the specified integer variable or array element in the input/output status specifier (if any) to become defined.

DIRECTION OF DATA TRANSFER

Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if one is specified.

Execution of a WRITE, PRINT, or PUNCH statement causes values to be transferred to a file from the entities specified by the output list and format specification (if any). Execution of a WRITE, PRINT, or PUNCH statement for a file that does not exist creates the file, unless an error condition occurs.

IDENTIFYING A UNIT

A data transfer input/output statement that contains a control information list includes a unit specifier that identifies an external unit or an internal file. A READ statement that does not contain a control information list specifies a particular processor determined unit, which is the same as the unit identified by an asterisk in a READ statement that contains a control information list. A PRINT statement specifies some other processor determined unit, which is the same as the unit identified by an asterisk in a WRITE statement. A PUNCH statement identifies yet another processor determined unit. Thus each data transfer input/output statement identifies an external unit or an internal file.

Data transfer input/output statements that do not contain control information lists refer to units that are preconnected as follows:

Statement	Standard Unit	File Name
READ	INT(5HINPUT)	"INPUT"
PRINT	INT(6HOUTPUT)	"OUTPUT"
PUNCH	INT(5HPUNCH)	"PUNCH"

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins.

ESTABLISHING A FORMAT

If the control information list contains a format identifier other than an asterisk or NAMELIST group name, the format specification identified by the format identifier is established. If the format identifier is an asterisk, list-directed formatting is established. If the format identifier is a NAMELIST group name, NAMELIST formatting is established.

On output, if an internal file has been specified, a format specification that is in the file or is associated with the file must not be specified.

FILE POSITION PRIOR TO DATA TRANSFER

The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

Sequential Access

On input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

An internal file is always positioned at the beginning of the first record of the file. This record becomes the current record.

Direct Access

For direct access, the file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

DATA TRANSFER

Data are transferred between records and entities specified by the input/output list. The list items are processed in the order of the input/output list.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item. In the example,

```
READ (3) N, A (N)
```

two values are read; one is assigned to N, and the second is assigned to A(N) for the new value of N.

An input list item, or any entity associated with it, must not contain any portion of the established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file.

A DO-variable becomes defined at the beginning of processing of the items that constitute the range of an implied-DO list.

On output, every entity whose value is to be transferred must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all the entities specified by the input list to become undefined.

Unformatted Data Transfer

During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record.

On input, the number of values required by the input list must be less than or equal to the number of values in the record.

On input, the type of each value in the record must agree with the type of the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. If an entity in the input list is of type character, the length of the character entity must agree with the length of the character value.

On output to a file connected for direct access, the output list must not specify more values than can fit into a record.

On output, if the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

Formatted Data Transfer

During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. The current record and possibly additional records are read or written.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output formatted data transfer is prohibited.

USING A FORMAT SPECIFICATION

If a format specification has been established, format control is initiated and editing is performed as described in Chapter 9.

On input, the input list and format specification must not require more characters from a record than the record contains.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

On output, if the file is connected for direct access or is an internal file and the characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

On output, if the file is connected for direct access or is an internal file, the output list and format specification must not specify more characters for a record than can fit into the record.

LIST-DIRECTED FORMATTING

If list-directed formatting has been established, editing is performed as described in Chapter 9.

PRINTING OF FORMATTED RECORDS

The transfer of information in a formatted record to certain devices determined by the processor is called printing. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

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

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record, the vertical spacing is one line and no characters other than blank are printed in that line.

A PRINT statement does not imply that printing will occur, and a WRITE statement does not imply that printing will not occur.

FILE POSITION AFTER DATA TRANSFER

If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written and that record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If the file is positioned after the endfile record, execution of a data transfer input/output statement is prohibited. However, a BACKSPACE or REWIND statement may be used to reposition the file.

If an error condition exists, the position of the file is indeterminate.

INPUT/OUTPUT STATUS SPECIFIER DEFINITION

If the data transfer input/output statement contains an input/output status specifier, the integer variable or array element ios becomes defined. If no error condition or end-of-file condition exists, the value of ios is zero. If an error condition exists, the value of ios is positive. If an end-of-file condition exists and no error condition exists, the value of ios is negative.

AUXILIARY INPUT/OUTPUT STATEMENTS

OPEN STATEMENT

An OPEN statement may be used to connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit.

The form of an OPEN statement is:

OPEN (olist)

where olist is a list of specifiers:

```
[UNIT =] u
IOSTAT = ios
ERR = s
FILE = fin
STATUS = sta
ACCESS = acc
FORM = fm
RECL = rl
BLANK = blnk
BUFL = bl
```

olist must contain exactly one external unit specifier and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = ios

is an input/output status specifier. Execution of an OPEN statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor dependent positive integer value if an error condition exists.

ERR = s

is an error specifier.

FILE = fin

fin is a character expression whose value when any trailing blanks are removed is the name of the file to be connected to the specified unit. The file name must be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, it becomes connected to a processor determined file. The processor determines a file name from the unit specifier u as follows:

1. If INT(u) has a value representable by the digit string n in the range 0. .999, the file name is TAPEn.

2. If u has a value of the form nHf, where f is a valid system file name, the file name is f.

Otherwise, the unit specified does not exist.

STATUS = sta

sta is a character expression whose value when any trailing blanks are removed is OLD, NEW, SCRATCH, or UNKNOWN. If OLD or NEW is specified, a FILE= specifier must be given. If OLD is specified, the file must exist. If NEW is specified, the file must not exist. Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If SCRATCH is specified with an unnamed file, the file is connected to the specified unit for use by the executable program but is deleted at the execution of the CLOSE statement referring to the same unit or at the termination of the executable program. SCRATCH must not be specified with a named file. If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, a value of UNKNOWN is assumed.

ACCESS = acc

acc is a character expression whose value when any trailing blanks are removed is SEQUENTIAL or DIRECT. It specifies the access method for the connection of the file as being sequential or direct. If this specifier is omitted, the assumed value is SEQUENTIAL. For an existing file, the specified access method must be included in the set of allowed access methods for the file. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

FORM = fm

fm is a character expression whose value when any trailing blanks are removed is FORMATTED or UNFORMATTED. If specifies that the file is being connected for formatted or unformatted input/output respectively. If this specifier is omitted, a value of UNFORMATTED is assumed if the file is being connected for direct access, and a value of FORMATTED is assumed if the file is being connected for sequential access. For an existing file, the specified form must be included in the set of allowed forms for the file. For a new file, the processor creates a file with a set of allowed forms that includes the specified form.

RECL = rl

rl is an integer expression whose value must be positive. It specifies the length of each record in a file being connected for direct access. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted input/output, the length is measured in processor determined units. For an existing file, the value of rl must be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value. This specifier must be given when the file is being connected for direct access; otherwise, it must be omitted.

BLANK = blnk

blnk is a character expression whose value when any trailing blanks are removed is NULL or ZERO. If NULL is specified, all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks other than leading blanks are treated as zeros. If this specifier is omitted, a value of NULL is assumed. This specifier is permitted only for a file being connected for formatted input/output.

BUFL = bl

bl is an integer expression whose value must be in the range 1 . . 24. It specifies the buffer length for the unit in small pages. If the file is already connected to the unit and the buffer length is being changed, an error condition exists. If this specifier is omitted a value of three small pages is assumed.

The unit specifier is required to appear; all other specifiers are optional, except that the record length rl must be specified if a file is being connected for direct access. Note that some of the specifiers have an assumed value if they are omitted.

The unit specified must exist.

A unit may be connected by execution of an OPEN statement in any program unit of an executable program and, once connected, may be referenced in any program unit of the executable program.

Open of a Connected Unit

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in the OPEN statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist, but is the same as the file to which the unit is preconnected, the properties specified by the OPEN statement become part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without the STATUS= specifier had been executed for the unit immediately prior to the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK= specifier may have a value different from the one currently in effect. Execution of the OPEN statement causes the new value of the BLANK= specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is permitted. The effect is that the file becomes connected to more than one unit.

CLOSE STATEMENT

A CLOSE statement is used to terminate the connection of a particular file to a unit.

The form of CLOSE statement is:

CLOSE (cllist)

where cllist is a list of specifiers:

```
[UNIT =] u
IOSTAT = ios
.ERR = s
STATUS = sta
```

cllist must contain exactly one external unit specifier and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = ios

is an input/output status specifier. Executing of a CLOSE statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor dependent positive integer value if an error condition exists.

ERR = s

is an error specifier.

STATUS = sta

sta is a character expression whose value when any trailing blanks are removed is KEEP or DELETE. sta determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, the file will not exist after the execution of the CLOSE statement. If this specifier is omitted, the assumed value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the assumed value is DELETE.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement specifying the unit that does not exist or has no file connected to it is permitted and affects no file.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same file or to a different file. After a file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same unit or to a different unit, provided that the file still exists.

Implicit Close at Termination of Execution

At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without the STATUS= specifier were executed on each connected unit.

INQUIRE STATEMENT

An INQUIRE statement may be used to inquire about the properties of a particular file or of the connection to a particular unit. There are two forms of the INQUIRE statement; inquire by file and inquire by unit. All value assignments are done according to the rules for assignment statements.

The INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed.

INQUIRE by File

The form of an INQUIRE by file statement is:

INQUIRE (iflist)

where iflist is a list of specifiers that must contain exactly one file specifier and may contain other inquiry specifiers. The iflist may contain at most one of each of the inquiry specifiers described below.

The form of a file specifier is:

```
FILE = fin
```

where fin is a character expression whose value when any trailing blanks are removed specifies the name of the file being inquired about. The named file need not exist, or be connected to a unit. The value of fin must be of a form acceptable to the processor as a file name.

INQUIRE by Unit

The form of an INQUIRE by unit statement is:

```
INQUIRE (iulist)
```

where iulist is a list of specifiers that must contain exactly one external unit specifier and may contain other inquiry specifiers. The iulist may contain at most one of each of the inquiry specifiers described below. The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

Inquiry Specifiers

The following inquiry specifiers may be used in either form of the INQUIRE statement:

```
IOSTAT = ios
ERR = s
EXIST = ex
OPENED = od
NUMBER = num
NAMED = nmd
NAME = fn
ACCESS = acc
SEQUENTIAL = seq
DIRECT = dir
FORM = fm
FORMATTED = fmt
UNFORMATTED = unf
RECL = rcl
NEXTREC = nr
BLANK = blnk
```

The specifiers are described as follows:

IOSTAT = ios

is an input/output status specifier. Execution of an INQUIRE statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor dependent positive integer value if an error condition exists.

ERR = s

is an error specifier.

EXIST = ex

ex is a logical variable or logical array element. Execution of an INQUIRE by file statement causes ex to be assigned the value true if there exists a file with the specified name; otherwise, ex is assigned the value false. Execution of an INQUIRE by unit statement causes ex to be assigned the value true if the specified unit exists; otherwise, ex is assigned the value false.

OPENED = od

od is a logical variable or logical array element. Execution of an INQUIRE by file statement causes od to be assigned the value true if the file specified is connected to a unit, otherwise, od is assigned the value false. Execution of an INQUIRE by unit statement causes od to be assigned the value true if the specified unit is connected to a file; otherwise, od is assigned the value false.

NUMBER = num

num is an integer variable or integer array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If more than one unit is currently connected to the file, the choice of the unit used to assign a value to num is described below. If there is an external unit identifier u currently connected to the file such that either

a. INT(u) has a value in the range 0. . 999, or

b. u has a value of the form nHTAPEk or nHUNITk, where k is an integer in the range 0. . 999 with no leading zero,

then the value assigned to num will be in the range 0. . 999. (In case (a) the value assigned to num is the value of u, and in case (b) it is the value of k. If external unit identifiers of both types (a) and (b) are currently connected to the file, the processor may assign either a type (a) or a type (b) value.) Otherwise, the value assigned to num is of the form INT(nHf), where f is a valid system file name. If there is no unit connected to the file, num becomes undefined.

NAMED = nmd

nmd is a logical variable or logical array element that is assigned the value true if the file has a name; otherwise, it is assigned the value false.

NAME = fn

fn is a character variable or character array element that is assigned the value of the name of the file, if the file has a name; otherwise, it becomes undefined. Note that if this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. For example, the processor may return a file name qualified by a user identification. However, the value returned must be suitable for use as the value of a FILE= specifier in an OPEN statement.

ACCESS = acc

acc is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, and UNKNOWN if the processor is unable to determine the method of access. If there is no connection, acc becomes undefined.

SEQUENTIAL = seq

seq is a character variable or character array element that is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

DIRECT = dir

dir is a character variable or character array element that is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

FORM = fm

fm is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If the processor is unable to determine the form, fm is assigned the value UNKNOWN. If there is no connection fm becomes undefined.

FORMATTED = fmt

fmt is a character variable or character array element that is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

UNFORMATTED = unf

unf is a character variable or character array element that is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

RECL = rcl

rcl is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor dependent units. If there is no connection or if the connection is not for direct access, rcl becomes undefined.

NEXTREC = nr

nr is an integer variable or integer array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since connection, nr is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, nr becomes undefined.

BLANK = blnk

blnk is a character variable or character array element that is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, blnk becomes undefined.

A variable or array element that becomes defined or undefined as a result of its use as a specifier in an INQUIRE statement, or any associated entity, must not be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by file statement causes the specifier variables or array elements nmd, fn, seq, dir, fmt, and unf to be assigned values only if the value of fin is acceptable to the processor as a file name and if there exists a file by that name; otherwise, they become undefined. Note that num becomes defined if and only if od becomes defined with the value true. Note also that the specifier variables or array elements acc, fm, rcl, nr, and blnk may become defined only if od becomes defined with the value true.

Execution of an INQUIRE by unit statement causes the specifier variables or array elements num, nmd, fn, acc, seq, dir, fm, fmt, unf, rcl, nr, and blnk to be assigned values only if the specified unit exists and if a file is connected to the unit; otherwise, they become undefined.

If an error condition occurs during the execution of an INQUIRE statement, all of the inquire specifier variables and array elements except ios become undefined.

Note that the specifier variables or array elements ex and od always become defined unless an error condition occurs.

FILE POSITIONING STATEMENTS

The forms of the file positioning statements are:

```
BACKSPACE u
BACKSPACE (alist)

ENDFILE u
ENDFILE (alist)

REWIND u
REWIND (alist)
```

where: u is an external unit identifier.

alist is a list of specifiers:

```
[UNIT = ] u

IOSTAT = ios

ERR = s
```

alist must contain exactly one external unit specifier and may contain at most one of each of the other specifiers.

The external unit specified by a BACKSPACE, ENDFILE, or REWIND statement must be connected for sequential access.

Execution of a file positioning statement containing an input/output status specifier causes ios to become defined with a zero value if no error condition exists or with a positive integer value if an error condition exists.

BACKSPACE Statement

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed formatting is prohibited.

ENDFILE Statement

Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

REWIND Statement

Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

RESTRICTIONS ON FUNCTION REFERENCES AND LIST ITEMS

A function must not be referenced within an expression appearing anywhere in an input/output statement if such a reference causes an input/output statement to be executed. Note that a restriction in the evaluation of expressions prohibits certain side effects.

RESTRICTION ON INPUT/OUTPUT STATEMENTS

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements must not refer to the unit.

NAMELIST INPUT/OUTPUT

NAMELIST provides formatted input/output with processor determined editing.

A symbolic name is a NAMELIST group name if and only if it appears in a NAMELIST statement. A NAMELIST group name is local to a program unit.

A NAMELIST group name provides the means of referring to a NAMELIST input/output list. Usage of a group name is the means of specifying NAMELIST formatting. A NAMELIST statement is used to specify a NAMELIST group name and the input/output list to be subsequently associated with that group name.

NAMELIST formatting is established for an input/output data transfer by using a NAMELIST group name as the format identifier f in a READ, WRITE, PRINT, or PUNCH statement; the statement must not include an input/output list.

NAMELIST STATEMENT

The form of a NAMELIST statement is:

NAMELIST / grpname / niplist [/ grpname / niplist]

where: grpname is a NAMELIST group name. Only one appearance of a group name in all of the NAMELIST statements of a program unit is permitted.

niplist is a NAMELIST input/output list of one or more items, each of which must be one of the following:

1. A variable name.
2. An array name.

Each name in the list niplist may be of any data type. It may not be an assumed size array.

NAMELIST DATA TRANSFER

A NAMELIST block is one or more formatted records that consist of a sequence of characters in NAMELIST format. Execution of an input/output data transfer statement with NAMELIST formatting causes one NAMELIST block to be transferred.

Execution of a WRITE, PRINT, or PUNCH statement with NAMELIST formatting causes one NAMELIST block to be written to a file. Data are transferred from internal storage in the order specified by the input/output list associated with the NAMELIST group name that appears in the output data transfer statement.

Execution of a READ statement with NAMELIST formatting causes one NAMELIST block to be read from a file. The NAMELIST group name in the block read must be the same as the group name in the READ statement being executed. Each variable or array name in the block must appear in the input/output list associated with the group name. Item names in the block may occur in any order and number. Note that an item name may appear more than once in a block, possibly resulting in more than one definition of an entity. The block is transferred with NAMELIST editing to internal storage in the order of item name appearance. Values are transmitted to the entities specified by the item names. The definition status of entitites whose names do not appear in the block is unchanged upon completion of the transfer. Note that an entity named in the associated input/output list but not named in the block retains its prior definition status: it is unaffected by the transfer.

On input, an error condition exists if the file is not positioned at the beginning of a NAMELIST block.

The effect of executing a data transfer input/output statement with NAMELIST formatting is otherwise described under "executing a data transfer input/output statement."

ENCODE AND DECODE STATEMENTS

The ENCODE statement is the internal file data transfer output statement that permits access to both standard and extended internal files. The DECODE statement is the internal file data transfer input statement that permits access to both standard and extended internal files. The forms of the statements are:

ENCODE (k, f, u) [iolist]

DECODE (k, f, u) [iolist]

where: k is an unsigned integer constant or integer variable having a positive value. The value specifies the number of characters in each record of the internal file identified by u.

f is a format identifier that does not specify list-directed formatting.

u is an internal file identifier.

iolist is an input/output list specifying the data to be transferred.

Execution of an ENCODE statement causes values to be transferred to an internal file from the entities specified by the output list iolist (if any) and the format identifier I. The execution sequence, restrictions, and error conditions are as described for a formatted WRITE statement that transfers data to an internal file.

Execution of a DECODE statement causes values to be transferred from an internal file to the entities specified by the input list iolist (if any). Execution proceeds as described for a formatted READ statement that transfers data from an internal file.

Action is unspecified if the total length of all the records read or written exceeds the number of character storage units in the file.

Action is unspecified if any item of iolist is in the file or is associated with the file.

On output, a format specification that is in the file or is associated with the file must not be specified.

Note that an internal file may be defined or redefined by means other than an ENCODE statement. Such means must ensure that the record length is established as provided above.

CONCURRENT INPUT/OUTPUT STATEMENTS

The concurrent input/output statements using Q7BUFIN and Q7BUFOUT are described in the chapter on processor supplied subroutines.

Replace all of Section 9 with the following pages.

CDC CYBER 200 FORTRAN 77 FORMAT SPECIFICATION

A format used in conjunction with formatted input/output statements provides information that directs the editing between the internal representation and the character strings of a record or a sequence of records in the file.

A format specification provides explicit editing information. An asterisk (*) as a format identifier in an input/output statement indicates list-directed formatting.

FORMAT SPECIFICATION METHODS

Format specifications may be given:

- In FORMAT statements.
- As values of character arrays, character variables, or other character expressions.

FORMAT STATEMENT

The form of a FORMAT statement is:

FORMAT is

where is is a format specification, as described under "form of a format specification." The statement must be labeled.

CHARACTER FORMAT SPECIFICATION

If the format identifier in a formatted input/output statement is a character array name, character variable name, or other character expression, the leftmost character positions of the specified entity must be in a defined state with character data that constitute a format specification when the statement is executed.

A character format specification must be of the form described under "form of a format specification." Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification, with no affect on the format specification. Blank characters may precede the format specification.

If the format identifier is a character array name, the length of the format specification may exceed the length of the first element of the array; a character array format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering. However, if a character array element name is specified as a format identifier, the length of the format specification must not exceed the length of the array element.

NON CHARACTER ARRAY-FORMAT SPECIFICATION

If the format identifier in a formatted input/output statement is a non character array name, the first m elements of the array must be in a defined state such that the first m elements (for some positive integer m) constitute a valid format specification when the statement is executed.

A non character array format specification must be of the form described under "form of a format specification." Note that the form begins with a left parenthesis and ends with a right parenthesis. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification. Blank characters may precede the format specification.

FORM OF A FORMAT SPECIFICATION

The form of a format specification is:

([flist])

where flist is a list. The forms of the flist items are:

[r] ed

ned

[r] fs

where: ed is a repeatable edit descriptor.

ned is a non-repeatable edit descriptor.

fs is a format specification with a non empty list flist.

r is a non zero, unsigned, integer constant called a repeat specification.

The comma used to separate list items in the list flist may be omitted as follows:

- Between a P edit descriptor and an immediately following F, E, D, or G edit descriptor.
- Before or after a slash edit descriptor.
- Before or after a colon edit descriptor.

EDIT DESCRIPTORS

An edit descriptor is either a repeatable edit descriptor or a non-repeatable edit descriptor.

The forms of a repeatable edit descriptor are:

Iw
Iw. m
Fw
Fw. d
Ew. d
Ew. dEe
Dw. d
Gw. d
Gw. dEe
Lw
A
Aw
Rw
Zw
Zw. m
Bw

where: I, F, E, D, G, L, A, R, Z, and B indicate the manner of editing

w and e are non zero, unsigned, integer constants

d and m are unsigned integer constants.

The forms a non-repeatable edit descriptor are:

"h1h2 . . . hn"
nHh1h2 . . . hn
Tc
TLc
TRc
nX
/
:
S
SP
SS
kP
BN
BZ

where: apostrophe, H, T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing

h is one of the characters capable of representation by the processor

n and c are non zero, unsigned, integer constants

k is an optionally signed integer constant

INTERACTION BETWEEN INPUT/OUTPUT LIST AND FORMAT

The beginning of formatted data transfer using a format specification initiates format control. Each action of format control depends on information jointly provided by:

- The next edit descriptor contained in the format specification, and
- The next item in the input/output list, if one exists.

If an input/output list specifies at least one list item, at least one repeatable edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if no list items are specified; in this case, one input record is skipped or one output record containing no characters is written. Except for an edit descriptor preceded by a repeat specification, r ed, and a format specification preceded by a repeat specification r(flist), a format specification is interpreted from left to right. A format specification or edit descriptor preceded by a repeat specification r is processed as a list of r format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification. Note that an omitted repeat specification is treated the same as a repeat specification whose value is one.

To each repeatable edit descriptor interpreted in a format specification, there corresponds one item specified by the input/output list, except that a list item of type complex requires the interpretation of two F, E, D, or G edit descriptors. To each P, X, T, TL, TR, S, SP, SS, H, BN, BZ, slash, colon, or apostrophe edit descriptors, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the records, and then format control proceeds. If there is no corresponding item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor, the S, SP, or SS edit descriptor sign control, or the BN or BZ edit descriptor blank control.

POSITIONING BY FORMAT CONTROL

After each I, F, E, D, G, L, A, H, R, Z, B, or apostrophe edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, X, or slash edit descriptor is processed, the file is positioned as described under "positional editing" and "slash editing."

If format control reverts as described in the previous section, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed.

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

EDITING

Edit descriptors are used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data.

A field is a part of a record that is read on input or written on output when format control processes one I, F, E, D, G, L, A, H, R, Z, B, or apostrophe edit descriptor. The field width is the size in characters of the field.

The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type.

APOSTROPHE EDITING

The apostrophe edit descriptor has the form of a character constant. It causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself. An apostrophe edit descriptor must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting apostrophes. Within the field, two consecutive apostrophes with no intervening blanks are counted as a single apostrophe.

H EDITING

The *n*H edit descriptor causes character information to be written from the *n* characters (including blanks) following the H of the *n*H edit descriptor in the format specification itself. An H edit descriptor must not be used on input.

Note that if an H edit descriptor occurs within a character constant that includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character in specifying n.

POSITIONAL EDITING

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record.

The position specified by a T edit descriptor may be in either direction from the current position. On output, this allows portions of the record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by the T, TL, TR, or X edit descriptor, portions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning so that subsequent editing causes replacement.

T, TL, AND TR EDITING

The Tc edit descriptor indicates that the transmission of the next character to or from a record is to occur at the cth character position.

The TLc edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position c characters backward from the current position. However, if the current position is less than or equal to position c, the TLc edit descriptor indicates that the transmission of the next character to or from the record is to occur at position one of the current record.

The TRc edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position c characters forward from the current position.

X EDITING

The nX edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

SLASH EDITING

The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file.

Note that a record that contains no characters may be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

COLON EDITING

The colon edit descriptor terminates format control if there are no more items in the input/output list. The colon edit descriptor has no effect if there are more items in the input/output list.

S, SP, AND SS EDITING

The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that normally contains an optional plus. If an SS edit descriptor is encountered, the processor must not produce a plus in any subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, D, and G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

P EDITING

A scale factor is specified by a P edit descriptor, which is of the form:

$$\underline{k}P$$

where k is an optionally signed integer constant called the <u>scale factor</u>.

Scale Factor

The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered, and then that scale factor is established. Note that reversion of format control does not affect the established scale factor.

The scale factor k affects the appropriate editing in the following manner:

1. On input, with F, E, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor affect is that the externally represented number equals the internally represented number multiplied by 10**k.

2. On input, with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.

3. On output, with E and D editing, the basic real constant part of the quantity to be produced is multiplied by 10**k and the exponent is reduced by k.

4. On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

BN AND BZ EDITING

The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in the numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

NUMERIC EDITING

The I, F, E, D, and G edit descriptors are used to specify input/output of integer, half precision, real, double precision, and complex data. The following general rules apply:

1. On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier and any BN or BZ blank

control that is currently in effect for the unit. Plus signs may be omitted. A field of all blanks is considered to be zero.

2. On input, with F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of any edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of a datum.

3. On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors or the processor. The representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

4. On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field:

5. On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E_w_. _d_E_e_ or G_w_. _d_E_e_ edit descriptor, the processor will fill the entire field of width _w_ with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted. Note that when an SP edit descriptor is in effect, a plus is not optional.

Integer Editing

The T_w_ and T_w_. _m_ edit descriptors indicate that the field to be edited occupies _w_ positions. The specified input/output list item must be of type integer. On input, the specified list item will become defined with an integer datum. On output, the specified list item must be defined with an integer datum.

On input, an I_w_. _m_ edit descriptor is treated identically to a T_w_ edit descriptor.

In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks.

The output field for the I_w_ edit descriptor consist of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The output field for the I_w_. _m_ edit descriptor is the same as for the I_w_ edit descriptor, except that the unsigned integer constant consists of at lease _m_ digits and, if necessary, has leading zeros. The value of _m_ must not exceed the value of _w_. If m is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

Half Precision, Real, and Double Precision Editing

The F, E, D, and G edit descriptors specify the editing of half precision, real, double precision, and complex data. An input/output list item corresponding to an F, E, D, or G edit descriptor must be half precision, real, double precision, or complex. An input list item will become defined with a datum whose type is the same as that of the list item. An output list item must be defined with a datum whose type is the same as that of the last item.

F EDITING

The F_w_ and F_w_. _d_ edit descriptors indicate that the field to be edited contains _w_ positions. If . _d_ is specified, it indicates that the fractional part of the field consists of d digits; if omitted, there will be no fractional digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost d digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented . The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

1. Signed integer constant.

2. E followed by zero or more blanks, followed by an optionally signed integer constant.

3. D followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing a D is processed identically to an exponent containing an E.

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to _d_ fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

E AND D EDITING

The E_w_. _d_, D_w_. _d_, and E_w_. _d_E_e_ edit descriptors indicate that the external field occupies _w_ positions, the fractional part of which consists of _d_ digits, unless a scale factor greater than one is in effect, and the exponent part consists of _e_ digits. The _e_ has no effect on input.

The form of the input field is the same as for = editing.

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

[±] [0] . X1X2. . .Xdexp

where: ± signifies a plus or a minus.

X1X2. . .Xd are the d most significant digits of the value of the datum after rounding.

exp is a decimal exponent, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
Ew.d	| exp | ≤ 99	E±z1z2 or ±0z1z2
	99 ≤ | exp | ≤ 999	±z1z2z3
Ew.dEe	| exp | ≤ (10**e) - 1	E±z1z2. . .ze
Dw.d	| exp | ≤ 99	D±z1z2 or E±z1z2 or ±0z1z2
	99 < | exp | ≤ 999	±z1z2z3

where z is a digit. The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The forms Ew. d and Dw. d must not be used if | exp | > 999.

The scale factor k controls the decimal normalization. If -d < k ≤ 0, the output field contains exactly | ≤ | leading zeros and d - k significant digits after the decimal point. If 0 < k ≤ d + 2, the output field contains exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point. Other values of k are not permitted.

G EDITING

The Gw. d and Gw. dEe edit descriptors indicate that the external field occupies w positions, the fractional part of which consists of d digits, unless a scale factor greater than one is in effect, and the exponent part consists of e digits.

G input editing is the same as for F editing.

The method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If N < 0.1 or N ≥ 10**d, Gw. d output editing is the same as kPEw. d editing and Gw. dEe output editing is the same as kPEw. dEe output editing, where k is the scale factor currently in effect. If N is greater than or equal to 0.1 and is less than 10**d, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
0.1 ≤ N < 1	F(w-n) . d, n("b")
1 ≤ N < 10	F(w-n) . (d-1), n("b")
.	.
.	.
.	.
10**(d-2) ≤ N < 10**(d-1)	F(w-n) . 1, n("b")
10**(d-1) ≤ N < 10**d	F(w-n) . 0, n("b")

where: b is a blank.

n is 4 for Gw. d and e+2 for Gw. dEe.

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside the range that permits the effective use of F editing.

COMPLEX EDITING

A complex datum consists of a pair of separate real data; therefore, the editing is specified by two successively interpreted F, E, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Note that non repeatable edit descriptors may appear between the two successive F, E, D, or G edit descriptors.

L EDITING

The Lw edit descriptor indicates that the field occupies w positions. The specified input/output list item must be of type logical. On input, the list item will become defined with a logical datum. On output, the specified list item must be defined with a logical datum.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants . TRUE . and . FALSE . are acceptable input forms.

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

A EDITING

The A[w] edit descriptor indicates that the field occupies w positions. The specified input/output list item is treated as if it were of type character, regardless of its declared type, except that it must not be used with an input/output list item of type bit.

On input, the input list item will become defined with character data.

If a field width w is specified with the A edit descriptor, the field consists of w characters. If a field width w is not specified with the A edit descriptor, the number of characters in the field is the length of the input/output list item in characters.

Let len be the length in characters of the input/output list item. If the specified field width w for A input is greater than or equal to len, the rightmost len characters will be taken from the input field. If the specified field width is less than len, the w characters will appear left justified with len-w trailing blanks in the internal representation.

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

PROCESSOR DEPENDENT EDITING

The R, Z, and B edit descriptors are used to specify processor dependent editing. The editing consists of direct bit, hexadecimal, or character code conversion between internal storage and character strings of a record. Conversion proceeds on a bit-by-bit basis; no numeric or logical significance is attached to the data. Any data except type bit may be edited with the R and Z edit descriptors. Data of type bit may only be edited with a B edit descriptor.

Note that if an input/output list item is of type complex two repeatable edit descriptors are required for the item. These edit descriptors do not have to be the same.

R Editing

The Rw edit descriptor indicates that the field occupies w positions. The specified input/output list item is treated as if it were of type character, regardless of its declared type.

Let Len be the length in characters of the input/output list item. If the specified field width w for R input is greater than or equal to len, the rightmost len characters will be taken from the input field. If the specified field width is less than len the w characters will appear right-justified with len-w leading characters filled with binary zeros in the internal representation.

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

Z Editing

The Zw and Zw. m edit descriptors indicate that the field occupies w positions. The specified input/output list item is treated as a sequence of hexadecimal digits, each occupying 4 bits in the internal representation of the input/output list item.

On input, the Z_w_ and Z_w_._m_ edit descriptors are treated identically.

Let _a_ be the number of hexadecimal digits that may be stored in the input/output list item at one time.

If the specified field width _w_ for Z input is greater than or equal to _a_, then the rightmost a hexadecimal digits are transmitted to the input list item after conversion from their character representation. If the specified field width is less than _a_, the _w_ hexadecimal digits will appear right-justified and preceded by _a-w_ hexadecimal zeros in the internal representation after conversion from their character representation. Blanks which appear anywhere in the field are treated as zeros.

If the specified field width _w_ for Z_w_ output is greater than _a_, the output field will consist of _w-a_ blanks followed by the a hexadecimal digits from the internal representation. If the specified width is less than or equal to _a_, the output field will consist of the rightmost _w_ hexadecimal digits from the internal representation.

The output field for the Z_w_._m_ edit descriptor is the same as for the Z_w_ edit descriptor, except that at least _m_ hexadecimal digits will appear, with leading hexadecimal zeros if necessary. The value of _m_ must not exceed the value of _w_. If the value of _m_ is zero and the internal representation of the output list item consists of all hexadecimal zeros, the output field will consist of only blank characters.

B Editing

The B_w_ edit descriptor indicates that the field occupies _w_ positions. The specified input/output list item must be of type bit. On input, the list item will become defined with a bit datum. On output, the specified list item must be defined with a bit datum.

Both the input and output fields consist of _w_-1 blanks followed by a 0 or a 1.

LIST-DIRECTED FORMATTING

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or one of the forms:

_r_*_c_

_r_*

where _r_ is an unsigned, non zero, integer constant. The _r_*_c_ form is equivalent to _r_ successive appearances of the constant _c_, and the _r_* form is equivalent to _r_ successive null values. Neither of these forms may contain embedded blanks, except where permitted in the constant _c_.

A value separator is one of the following:

1. A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks.

2. A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks.

3. One or more contiguous blanks between two constants or following the last constant.

LIST-DIRECTED INPUT

Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type half precision, real, or double precision, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type character, the input form consists of a non-empty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

Let len be the length of the list item, and let w be the length of the character constant. If len is less than or equal to w, the leftmost len characters of the constant are transmitted to the list item. If len is greater than w, the constant is transmitted to the leftmost w characters of the list item and the remaining len-w characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in an assignment statement.

A null value is specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the r* form. A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

Note that all blanks in a list-directed input record are considered to be part of some value separator except for the following:

1. Blanks embedded in a character constant.

2. Embedded blanks surrounding the real or imaginary part of a complex constant.

3. Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma.

LIST-DIRECTED OUTPUT

The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of character constants, the values are separated by one of the following·

1. One or more blanks.

2. A comma optionally preceded by one or more blanks and optionally followed by one or more blanks.

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I w edit descriptor, for some reasonable value of w.

Real and double precision constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value and a range 10**d1 ≤ x < 10**d2, where d1 and d2 are processor dependent integer values. If the magnitude of x is within this range, the constant is produced using 0PFw. d; otherwise, 1PEw dEe is used. Reasonable processor dependent values of w, d, and e are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constants is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced are not delimited by apostrophes, are not preceded or followed by a value separator, have each internal apostrophe represented externally by one apostrophe, and have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

If two or more successive values in an output record produced have identical values, the processor has the option of producing a repeated constant of the for r*c instead of the sequence of identical values.

Slashes as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carriage control when the record is printed.

NAMELIST FORMATTING

The form of a NAMELIST block is:

&grpname namval[,namval] ... &END

where: grpname is the group name of the block.

namval is one of the forms:

vname = c

aname [(s)] = [r*] c[,[r*]c] . . .

where: vname is a variable name,

c is a constant,

aname is an array name,

s is an array subscript in which each subscript expression is an integer constant.

r is an unsigned, positive integer constant,

The optional form r*c is equivalent to r successive appearances of the constant c.

A NAMELIST block consists of one or more formatted records, the last character. other than the character blank, of each record must be one of the following:

1. A comma that occurs after the constant c. Note that a complex constant must begin and end in the same record.

2. The last character of the block terminator &END.

In each record of a NAMELIST block, column one is reserved for carriage control. On input, the character in column one is ignored. On output, a carriage control character is placed in column one of each record.

An embedded blank must not occur within the strings:

1. &grpname
2. vname
3. aname[(s)]
4. &END

A blank is otherwise not significant in a NAMELIST block.

NAMLIST INPUT

The group name of the NAMELIST block being transmitted must appear in the READ statement being executed. Each variable name and array name in the block must appear in the input/output list referred to by the READ statement.

Each constant c must agree with the type of the corresponding input list item as follows:

1. A bit, logical, character, or complex constant must be of the same type as the corresponding input list item. A character constant is truncated from the right or extended to the right with blank characters, if necessary, to yield a character constant the same length as the corresponding character variable, character array element, or substring.

2. An integer, half precision, real, or double precision constant may be used for an integer, half precision, real, or double precision input list item. The constant is converted to the type of the list item during transmission. For conversion to half precision, real, or double precision. an integer has an implied decimal point to the right of the rightmost digit.

The forms of a logical constant having the value true are:

T
.T.
.TRUE.
TRUE

The forms of a logical constant having the value false are:

```
F
.F.
.FALSE.
FALSE
```

A character constant must have the same form as if it appeared in a statement of an executable program (the delimiting apostrophes must be present).

The forms of integer, half precision, real, double precision, and complex constants are as described for list-directed input.

A bit constant must be either a 0 or a 1.

The character blank is ignored within a non-character constant. Use of the BLANK= specifier in an OPEN statement has no effect on NAMELIST editing.

An error condition exists if a constant has no characters other than the character blank. (A character constant is allow to have only blank characters between the delimiting apostrophes).

NAMELIST OUTPUT

On output, each NAMELIST block is terminated with the characters &END.

The processor begins a new record for each block transferred. Column one of the first record of each block contains the carriage control character blank.

The processor begins a new record for the group name, for each variable name, for each array name, and for the block terminator &END. Column one of each such record contains the carriage control character blank.

The processor begins a new record if the output field width of a constant would exceed the number of unfilled character positions remaining in the current record. The current record is instead filled with blank characters and terminated. A new record is begun with the carriage control character blank in column one and the leftmost character of the constant in column two.

Logical constants are produced as T for the value true and F for the value false.

Bit constants are either 0 or 1.

Character constants are produced with delimiting apostrophes.

Integer constants are produced with the effect of an I edit descriptor.

Except for the value zero, half precision, real, and double precision constants are produced with the effect of an E edit descriptor. The scale factor is zero; no significant digits are produced before the decimal point. The number of significant digits produced to the right of the decimal point is that minimum number appropriate to the precision of the internal datum. Trailing zeros are eliminated. The characters 0.0 are produced for the value zero.

Complex constants are produced as a pair of real constants enclosed in parentheses and with a comma separating the real and imaginary parts. Each real constant is produced as described in the preceding paragraph.

This page left blank intentionally.

PRECEDING PAGE BLANK NOT FILMED

ARRAY ASSIGNMENT

A The array assignment statement discussed in this section is neither a part of the standard set of FORTRAN statements (as defined by American National Standard X3.9-1966, FORTRAN) nor directly related to the vector programming capabilities of STAR FORTRAN. An array assignment statement, which is typified by one or more operands written in subarray notation, is a shorthand for FORTRAN DO loops. If the DO loop equivalent of an array assignment statement satisfies the criteria listed in section 11 for vectorizable loops, and if the V compile option of the FORTRAN system control statement is on, then the array assignment statement will be compiled into machine vector instructions.

SUBARRAY REFERENCES

A subarray is a cross-section of an array; it can be one element, several elements, or all of the elements of the array. A subarray is identified by an array name, or an array name qualified by a subscript containing one or more implied-DO subscript expressions plus any number of other subscript expression forms (see section 2). Implied-DO subscript expressions can appear only in array expressions which, in turn, can appear only in array assignment statements.

The three implied-DO subscript expression forms are shown below.

Forms:

$m_1:m_2:m_3$

*

$m_1:*:m_3$

m_1	Initial value of subscript expression; an integer constant or simple integer variable.
m_2	Terminal value of subscript expression; an integer constant or simple integer variable.
m_3	Optional incrementation value; an integer constant or simple integer variable. When m_3 is omitted, the colon immediately preceding it must also be omitted and a value of 1 is assumed for the incrementation value.
B *	Represents a constant with a value equal to the declared dimension size.

The first form indicates subscript expression values ranging from m_1 up through m_2, starting with m_1 and incremented by m_3. C The second implied-DO form is equivalent to the form $1:m_2:1$, where m_2 is equal to the declared size of the array dimension. The third implied-DO form indicates subscript expression values starting with m_1, up through the declared size of the array dimension and in increments of m_3. In every case, if the value $(m_2-m_1)/m_3$ is not integral, the subscript expression never takes on the terminal value m_2. D The initial value m_1 must be less than or equal to the terminal value m_2.

Example:

A(5,10,2) is the array declarator. Then,

A(*,*,1) designates one-half of the array elements, and

A(*,*,2) designates the other half.

A(1:2,1:2,1:2) names the following elements:

```
A(1,1,1)
A(2,1,1)
A(1,2,1)
A(2,2,1)
A(1,1,2)
A(2,1,2)
A(1,2,2)
A(2,2,2)
```

A(1:5:2,1,1) designates the following elements:

```
A(1,1,1)
A(3,1,1)
A(5,1,1)
```

An entire array can be designated by the unsubscripted array name.

Example:

A(10,10) is the array declarator. Then, the following implied-DO forms are equivalent:

```
A
A(1:10,1:10)
A(1:10,*)
A(*,1:10)
A(*,*)
```

The order in which the array elements are indicated by a subarray is always with the leftmost subscript expression varying through its range, then the next subscript expression being incremented and the first subscript varying through its range again, and so on until every implied DO has been run through its range at least once. This rule applies to all subarrays, regardless of whether an array is rowwise or columnwise. However, whether or not an array is rowwise does affect whether or not its elements are accessed consecutively in memory.

The association between an instance of the subarray notation and the values elicited by it is displayed in figure 10-1. For an array declared as A(10,3), the figure shows the transformation from a subarray A(1,*) to its equivalent in array element references, which in turn elicit different sets of values according to whether A is rowwise or columnwise. In contrast to the subarray A(1,*), the subarray A(*,1) would not identify consecutive elements in memory if the array declarator occurred in a ROWWISE statement. In general, only a single row of elements in a rowwise array (of any size) can be specified consecutively in memory at one time using the subarray notation.

A: . . . (as defined by American National Standard X3.9-1978, . . .

B: * Represents a constant with a value equal to the declared upper bound for the corresponding dimension.

C: . . . form m1:m2:1 where m1 and m2 are, respectively, the declared lower and upper bounds for the corresponding array dimension. The third form is equivalent to the form m1:m2:m3 where m2 is the declared upper bound for the corresponding array dimension. The second and third forms may not be used for the last (1st if ROWWISE) dimension of an assumed size array. In every case, if the value (m2-m1)/m3 is not integral, . . .

D: . . . m2. there is no restriction on the values of m1, m2, and m3. m3 may be negative, m1 may be greater than m2, or both. If m3 is negative and m1 is less than m2, or if m3 is positive and m1 is greater than m2, the subarray is empty.

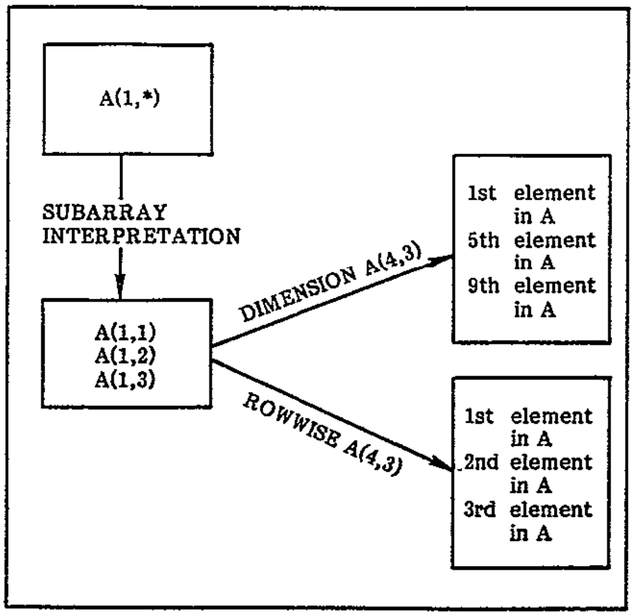


Figure 10-1. Meaning of a Subarray

CONFORMABLE SUBARRAYS

Two subarrays are called conformable if they satisfy both of the following conditions:

- The number of subscript expressions that are implied-DO subscript expressions must be the same for both.
- Scanning from left to right in the subscript, the i^{th} implied-DO subscript expression in one must be the same as the i^{th} implied-DO subscript expression in the other. Implied-DO subscript expressions are considered to be the same when the expansions of the subscript expressions into the following form are identical:

  initial value : terminal value : incrementation value

The subarrays need not have the same number of subscript expressions to be conformable, nor must the subarrays be the same data type. The number of entities specified in a subarray is the same as in the subarrays conformable with it.

Examples:

Given the array declarators A(5,3), B(8,5), and C(5,3,4), the following pairs of subarrays are conformable:

```
A   A(1:5,3)    A(1:5,3)    A(1:4,3)
A   B(1:5:1,2)  B(1,1:5)    C(1,2,1:4)

A           A(1:5,1:3)      A(1:5:2,2)
B(1:5,1:3)  C(1:5,2,1:3)    B(1:5:2,4)
```

and the following pairs of subarrays are nonconformable:

```
A   B(1:5,1:3)   A(1:4,3)
B   B(1:3,1:5)   C(1:1,2,1:4)
```

ARRAY EXPRESSIONS

An array expression has the form of any scalar expression – arithmetic, relational, or logical – except that it must contain at least one subarray. Any two subarrays in an array expression must be conformable.

Evaluation of an array expression proceeds with the stated operations being performed on corresponding elements of the array operands. Any scalar primaries are treated as arrays having the same number of elements as a subarray in the expression, with all elements containing the scalar value.

Examples:

Given the array declarators A(5,5), B(10,5), and C(5,10), the following are array expressions:

```
A+3.1
A(1:3,1)*A(1:3,2)/A(1:3,3)*A(1:3,4)
A(I,1:5)**2.0
B(10,1:5)+C(1:5,10)+1.0-A(1,1)
(A-B(1:5,*))/24.5*C(*,1:5)
```

ARRAY ASSIGNMENT STATEMENT

An array assignment statement has the following form:

a=expr

expr	An array expression, or any scalar expression.
a	A subarray conformable with the value of expr.

If the value of expr is a scalar (one value), execution of the assignment statement assigns that value to all identified elements of the subarray a . If the value of expr is a subarray (more than one value), the identified elements of a are replaced with the corresponding elements in the array expression result.

Data type conversion rules on assignment are identical to those described in section 4 for scalar assignment statements.

Examples:

Each of the statement pairs:

```
      DIMENSION X(5,3),Y(2,5)
      X(1:5,3) = Y(2,1:5)

      DIMENSION X(5,3), Y(2,5)
      X(*,3) = Y(2,*)
```

has the same effect as the statement:

```
      DIMENSION X(5,3),Y(2,5)
      DO 100 I=1,5,1
      X(I,3) = Y(2,I)
  100 CONTINUE
```

which in turn would accomplish the following set of assignments:

```
X(1,3) = Y(2,1)
X(2,3) = Y(2,2)
X(3,3) = Y(2,3)
X(4,3) = Y(2,4)
X(5,3) = Y(2,5)
```

This page left blank intentionally.

Similarly, the statement pair:

```
    DIMENSION X(5,3), Y(10,3,2)
    X(1:*:3,*) = Y(1:5:3,*,2)
```

has the same effect as the statements:

```
          DIMENSION X(5,3),Y(10,3,2)
          DO 200 I2=1,3,1
          DO 100 I1=1,5,3
          X(I1,I2) = Y(I1,I2,2)
      100 CONTINUE
      200 CONTINUE
```

which would accomplish the following set of assignments:

```
    X(1,1) = Y(1,1,2)
    X(4,1) = Y(4,1,2)
    X(1,2) = Y(1,2,2)
    X(4,2) = Y(4,2,2)
    X(1,3) = Y(1,3,2)
    X(4,3) = Y(4,3,2)
```

If any or all of the DIMENSION statements in these examples are changed to ROWWISE statements, the examples remain correct. Furthermore, if in the first example the array declarator for X appeared in the DIMENSION statement and the array declarator for Y appeared in a ROWWISE statement, the array assignment statement would be vectorizable because the elements of X and Y would be accessed consecutively in memory.

}A

PRECEDING PAGE BLANK NOT FILMED

A: DEFINE STATEMENT

Subarrays can be identified by single variable names or array element names through the use of the DEFINE statement. The DEFINE statement is an executable statement which establishes memory allocation and pointer data for DYNAMIC variables. A single DYNAMIC variable then can be made to describe a subarray, and that description can be changed throughout the program execution, if desired. DEFINE can also be used to dynamically establish the level of memory assigned to a arrays.

Forms:

```
DEFINE LEVEL i,dv1,dv2, . . . . .dvm
DEFINE (dv1,S1),(dv2,S2), . . . .(dvm,Sm)
```

i	An integer constant, simple integer variable or integer expression
dvn	Dynamic variable name or Dynamic array element
Sn	A subarray reference to an array, a dynamic variable or without array name (to establish dimensionality only)

Forms for Sn:

```
A(s1:s2:s3,s4:s5:s6, . . . . . .)
(s1:s2:s3,s4:s5:s6, . . . . .)
dV(s1:s2:s3,s4:s5:s6, . . . .)
dA(j1,j2, . .jm)
```

A	Name of a REAL, DOUBLE, HALF, COMPLEX, or INTEGER array
dV	Name of a dynamic variable
dA	Name of dynamic array
jn	Integer subscript expression, constant or integer variable
s1:s2:s3	Can optionally be s1:s2 or s1
s1	Initial value of subscription integer constant or simple integer variable
s2	If present, terminal value of subscript expression; an integer constant or simple integer variable
s3	Optional incrementation value; an integer constant or simple integer variable

In all cases, 1 to 7 dimensions are allowed. The first form establishes the level of memory to be used as dynamic space for the dynamic variables or dynamic array elements named in the list.

Example:

```
PROGRAM DEMO
DIMENSION A(100)
DYNAMIC C
```

```
.
.
I=2
.
.
DEFINE LEVEL I,C
.
C=A(1:100)
```

The dynamic variable C would be assigned to point to Intermediate (LEVEL 2) Memory. The replacement statement would then accomplish the transfer of 100 elements of A from Main Memory to Intermediate Memory, into 100 words of dynamic space there.

The integer expression i may have values from one to three. Any other value will cause the generation of a fatal object time diagnostic.

Changing memory level assignments will have no effect on the named variables when they appear as sources for operands:

```
DEFINE LEVEL 2,C
.
C=A(1:100)
.
DEFINE LEVEL 1,C
.
A(1:100)=C**2
```

The data moved to C will still be pointed at when C**2 is invoked. When C appears as an object of a replacement statement again, however, the dynamic variable C is redefined and a new memory allocation made, hence:

```
DEFINE LEVEL 1,C
.
A(1:100)=C**2
.
```

This would cause C to be defined as pointing to dynamic space in main memory, and the 10 elements of A moved there. It is then possible for the dynamic variable C to point to two different areas in the same FORTRAN statement:

```
DEFINE LEVEL 2,C
C=A(100)
DEFINE LEVEL 1,C
C=C**2
A=C
```

The first allocation for C would be in Intermediate (LEVEL 2) Memory. The second allocation 'pending' for C would be in Main Memory. The replacement statement would move 100 elements of A from main memory to Intermediate Memory. The arithmetic statement C=C**2 would multiply the data in Intermediate Memory by itself and store the result in the newly allocated area in Main Memory. The old data in LEVEL 2 for C would be abandoned.

Level reassignments of dynamic variables which are, in fact, pointers into subarrays of other arrays are ignored. The level of memory allocated to the main array, of which the subarray is part, is that level assigned to the dynamic variable.

The major purpose in a dynamic level assignment is to allow the most efficient use of the memories for program execution, although data base sizes and memory requirements may not be known until object time.

The second form assigns dynamic variable or dynamic array elements as pointers to subarrays or array components.

Example:

```
     PROGRAM DEMO
     DIMENSION A(100)
     DYNAMIC C
     .
     DEFINE (C,A(10:20))
     .
     C=3.14159
     .
```

The dynamic variable C becomes synonymous with the subarray A(10:20). The statement C=3.14159 results in the generation of a data transfer of the constant 3.14159 to elements 10 through 20 of A.

If the array name is omitted, then the subarray statement is used to establish implicit dimensions for C, and causes the allocation of dynamic space to C.

Example:

```
     PROGRAM DEMO
     DYNAMIC C,
     .
     DEFINE (C,(10,10,10))
     .
     C=3.1415926
```

This establishes C as an array in Main Memory dynamic space of dimensions 10 by 10 by 10. The replacement statement would transfer the constant 3.1415926 to all 1000 elements of C.

Dynamic variables and dynamic array element pointers establish the characteristics of the array they are describing. These dynamically established arrays may themselves be subdivided into subarrays:

Example:

```
PROGRAM DEMO
DYNAMIC C,D
.
.
DEFINE (C,(10,10,10)).
DEFINE (D,C(1,1,1:10))
.
.
D=3.1415926
```

The dynamic variable D would point to the first ten elements of the dynamically assigned array C. Dynamic array elements can be referenced in subarray form only in DEFINE statements.

The processing of dynamic variables and define statements is reserved for object time execution. All errors in conformability, memory allocation conflict and reuse of names and data space will result in the generation of fatal object time diagnostics.

Dynamic variables that are DEFINED as pointers into subarrays must be of the same data type as the subarray. A type mismatch will be the result in a fatal compile time diagnostic.

Section 11 is deleted entirely.

Section 12 is deleted entirely.

Section 13 is deleted entirely.

STAR FORTRAN-SUPPLIED SUBROUTINES 14

The following types of system-defined subroutines can be called from a STAR FORTRAN program:

Special calls:	Used to place specific STAR-100 machine instructions in the object code. Although a special call looks like a subroutine call, the special call generates in-line code.
Data Flag Branch Manager calls:	Used to trap special conditions and to branch to an interrupt-handling routine as a result of trapping such a condition.
MDUMP calls:	Used to dump specified areas in virtual memory during program execution.
System Error Processor calls:	Used to alter FORTRAN's run-time error processing so that, for example, execution halts when an error occurs that would normally have resulted in only a warning being issued.
Concurrent I/O calls:	Used to perform input and output of large arrays while at the same time leaving the CPU free for computational processing.

STAR FORTRAN SPECIAL CALLS

STAR FORTRAN users are able to have the compiler directly generate any instruction in the machine language repertoire. Such requests are made in the form of CALL statements to subroutines with special reserved names. The argument lists in the special call statements are used to provide label references, symbolic references, and literals to be included with the generated instruction. The user of special calls should be familiar with the hardware instructions or should have access to the STAR-100 Computer Hardware Reference Manual.

NOTE

The use of special calls is not recommended for the average FORTRAN user. Special calls should only be used when absolutely necessary for specific programming tasks.

Form:

CALL m(a_1, . . . ,a_n)

m	One of the special call names beginning with Q8.
a_i	An argument corresponding to one of the fields of the instruction format.

The special call formats are listed in appendix D.

ARGUMENTS

All arguments are either label references, symbolic references, or literals.

NOTE

The arguments for the special calls correspond to the fields of the hardware instructions. Arguments for the STAR Assembler instructions can appear different but are functionally the same. For example, the register to register hardware instruction (op code 78) is RTOR R,T in STAR Assembler but CALL Q8RTOR(R,T) in the special call format. The extra comma accounts for the missing S operand in the instruction.

The special call arguments must rigidly follow the instruction format because they represent the information associated with the instruction fields. Any missing argument must be indicated by a comma, except that trailing missing arguments can be omitted. With some exceptions, the arguments must appear in the order of the definable fields in the hardware instruction. An exception is that only one argument is allowed for an entire 8-bit G-designator field having 1-bit subfields. Another exception is that in indexed branch instructions (B0 through B5), the combined Y and B fields require only one argument, usually a label reference. If the combined fields represent two register designators, however, the user must use a 16-bit hexadecimal constant.

When an argument is a literal, the value of the literal goes in the instruction field. When an argument is a variable, the register number of the variable goes in the instruction field; the compiler generates a load before the designated instruction and a store afterwards, if required. Only registers 20 through FF (hexadecimal) are used for this purpose. The user is free to use the low-order temporary registers, but the contents are destroyed by generated object code when the user reverts to standard STAR FORTRAN statements.

Subfunction bits in the G field of formats 1, 2, and 3 are not cross-checked with the operands to assure validity of the instruction. Warnings are not generated if the user codes a jump into or out of range of a DO loop.

Label References

A label reference is designated by prefixing a statement label with the ampersand character. Label references can appear in the following instruction formats:

In the combined Y and B fields of a format C instruction.

In the 48-bit immediate (I) field of the format 5 instruction, except when only 24 bits of the field are used by certain instructions.

In the 8-bit immediate (I) field of format 9 and format B instructions.

A:
B: Deleted
C:

If the label reference occurs in the combined Y and B fields of a format C instruction, the label reference is translated into a code half-word offset from the special CALL to the statement within the program unit identified by the label. The labeled statement can be ahead of or behind the special CALL statement.

If the label reference occurs in the 48-bit immediate field of a format 5 instruction, the processor translates the label reference into a bit address of the statement tagged by the label. This bit address is a relative bit address with respect to the code base of the program unit in which the special CALL statement occurs.

If the label reference occurs in the 8-bit immediate field of a 2F, 32, or 33 instruction, the processor translates the label reference into a half-word offset from the special CALL statement, to the statement tagged by the label. If the resultant half-word offset exceeds a magnitude of 255, a zero is used to initialize the 8-bit immediate field, and the processor generates no warning to the user.

A label reference is the only permissible operand in the branch field of a relative branch instruction.

Symbolic References

A symbolic reference can be a simple variable of type real, integer, or logical; an array element of type real, integer, or logical; a descriptor; a descriptor array element; or a vector. Symbolic references can occur in any 8-bit register designator field (except in half-word registers). Registers modified by branch instructions cannot be referenced symbolically.

Literals

A literal can be a decimal, hexadecimal, bit, character, or Hollerith constant, and can be used for any instruction field. Any missing arguments are presumed to be zero constants. Generally, constants are taken to be register designators, rather than as data used by an instruction.

EXAMPLES OF SPECIAL CALL USAGE

The call to Q8BSAVE shown in figure 14-1 sets register 3 to the bit address of the next instruction, which has statement label 10. The call of Q8EX in statement 10 sets register 4 to the statement 10 bit offset from the code base address. In the next statement, the call to Q8SUBX sets integer variable CB to the code base address. The next call to Q8EX sets variable I to contain the statement 20 bit offset. Following that, variable L20 is set to the actual address of statement 20. This information is then used in the call to Q8BGE.

```
      INTEGER CB,L20
      CALL Q8BSAVE(3,,3)
   10 CALL Q8EX(4,&10)
      CALL Q8SUBX(3,4,CB)
      CALL Q8EX(I,&20)
      L20=I+CB
       .
       .
       .
      CALL Q8BGE(A,B,L20)
       .
       .
   20  .
```

Figure 14-1. Special CALL Statement

The calls in figure 14-2 produce identical results; each enters the character string AB in register 41 (hexadecimal). These examples are given to show how literals can be used as arguments; however, it should be noted that the use of register 41 would probably cause a program bug, because registers 20 to FF (hexadecimal) are assigned by the compiler.

```
CALL Q8ES(65,'AB')
CALL Q8ES(X'41',X'4142')
CALL Q8ES(B'1000001','AB')
CALL Q8ES('A','AB')
```

Figure 14-2. Q8ES Usage

The special calls in figure 14-3 generate the machine code shown in figure 14-4 provided J has been assigned to register 22 by the compiler.

```
CALL Q8ES(3,1)
CALL Q8ES(4,2)
CALL Q8ADDX(3,4,J)
```

Figure 14-3. Additional Q8 Usage

```
ES    R3,1
ES    R4,2
ADDX  R3,R4,R22
```

Figure 14-4. Generated Machine Code

If J has not been assigned any register by the compiler, the code shown in figure 14-5 would be generated.

```
ES     R3,1
ES     R4,2
ADDX R3,R4,T1
STO    (DATA BASE, RELATIVE
         LOCATION OF J),T1
```

Figure 14-5. Additional Generated Code

DATA FLAG BRANCH MANAGER

The data flag branch manager (DFBM) is a FORTRAN run-time library routine. A data flag branch is a hardware function of the STAR-100 computer. DFBM is software that processes data flag branches whenever they occur during execution of a FORTRAN program. Use of the data flag branch feature eliminates the time penalty that would be incurred if the FORTRAN user were compelled to perform explicit checks for special conditions. If the FORTRAN user takes no specific action with respect to data flag branches and DFBM, then any of the following causes a data flag branch to occur:

- A square root operation attempted with a negative operand
- A division operation attempted with a zero divisor
- An exponent overflow in computation of a number too large to be represented internally
- An operation attempted using an indefinite operand

A: delete page 14-2.

- Reduction of the job interval timer to zero (cannot occur unless the program sets the JIT)

- Execution of a hardware breakpoint instruction under certain usage conditions (cannot occur unless the program uses DEBUG or a BKP instruction)

Control passes to DFBM which performs interrupt processing for the condition. DFBM interrupts the executing FORTRAN program, issues an error diagnostic, dumps the contents of the data flag branch register, and aborts the program. If the program is running as part of a batch job, a post-mortem dump is produced. Default interrupt processing for other conditions that the user can specify does not cause the program to abort.

The FORTRAN user can select the special conditions which can cause a data flag branch and DFBM interrupt to occur. The user can also specify the processing that is to be performed as a result of the interrupt. Interrupt conditions and interrupt processing can be selected through calls to the DFBM entry points Q7DFSET, Q7DFOFF, Q7DFLAGS, and Q7DFCL1.

DATA FLAG BRANCH HARDWARE

For the FORTRAN user, the most significant part of the data flag branch hardware is the data flag branch (DFB) register. The 64-bit DFB register, located in the STAR-100 central processor, is formatted as shown in figure 14-6. Each interrupted task has a DFB register copy in its invisible package in the minus page.

The data flags are bits 35 through 47 of the DFB register. These bits indicate special conditions that have occurred. For example, the STAR hardware sets bit 41 at the end of a floating point divide fault (instruction in which the divisor is zero). Data flags remain set until the FORTRAN program or DFBM clears them.

The mask bits are bits 19 through 31 of the DFB register. They select the conditions which are to cause a data flag branch and DFBM interrupt. For example, bit 25 enables a data flag branch on a floating point divide fault. Bits 19, 20, 25, 29, 30, and 31 are set during FORTRAN run-time initialization; thereafter, the user can set and clear mask bits by calling DFBM entry points.

The product bits are bits 3 through 15 of the DFB register. Each is the dynamic logical product of a data flag and the associated mask bit. For example, the product bit for floating point divide fault is bit 9, which is set by STAR hardware if bits 25 and 41 are set. Bit 9 is cleared if either bit 25 or bit 41 is cleared. The product bits can be tested with a Q8BADF special call.

Bit 58 is the pipe 2 register instruction data flag. Setting of this bit indicates that one of the other data flags has been set by a pipe 2 instruction. STAR hardware sets the bit, which remains set until the FORTRAN program or DFBM clears it.

Bit 51 is the dynamic inclusive OR of all the product bits. Bit 52 is the data flag branch enable bit; if bit 52 is cleared, any further data flag branches of any kind are disabled until bit 52 is set again. DFBM and the STAR hardware clear and set bit 52. When both bit 51 and bit 52 are set, the STAR hardware initiates a data flag branch.

The condition indicated by each of the 13 data flags, along with a designator for the condition, is shown in table 14-1. Also given in the table are the mask and product bit associated with each data flag and a classification of I or III for each condition.

Default Conditions

At the time a FORTRAN program starts executing, six interrupt conditions are enabled. The conditions enabled as a result of run-time initialization are JIT, SFT, BKP, IND, SRT, and FDV.

The JIT, SFT, and BKP conditions do not occur unless the program takes specific action to cause the conditions.

An FDV condition occurs if a floating point division operation is attempted with a zero divisor. A zero divisor is either a machine zero or a floating point number having an

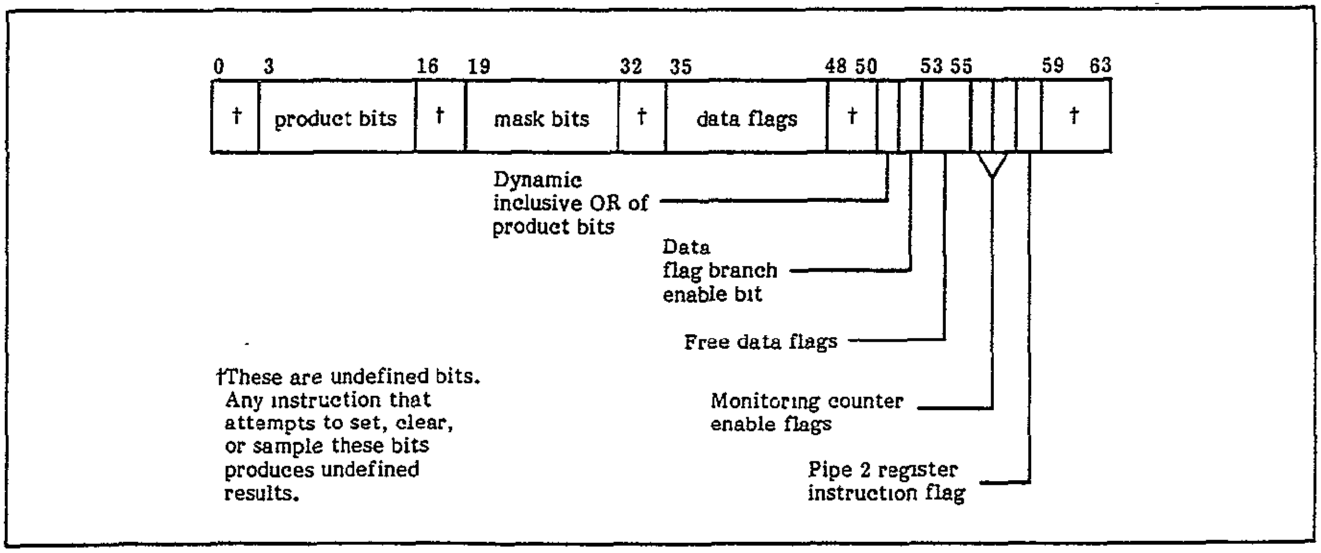

0
3
16
19
32
35
48
50
53
55
59
63
†
product bits
†
mask bits
†
data flags
†
†
Dynamic inclusive OR of product bits
Data flag branch enable bit
Free data flags
Monitoring counter enable flags
Pipe 2 register instruction flag
†These are undefined bits. Any instruction that attempts to set, clear, or sample these bits produces undefined results.

Figure 14-6. Data Flag Branch Register Format

TABLE 14-1. DATA FLAG BRANCH CONDITIONS

Class	Designator	Condition Description	Mask Bit	Data Flag Bit	Product Bit	Product Bit Search Order
I	SFT	(Reserved.)	19[†]	35	3	2
I	JIT	Job interval timer has reduced to zero.	20[†]	36	4	1
III	SSC	Selected condition has not been met. In search for masked key, there was no match; or count of nonzero translated bytes is greater than 65535_{10}.	21	37	5	11
III	DDF	Decimal data fault. A sign was found in a digit position, or vice versa.	22	38	6	12
III	TBZ	Truncation of leading nonzero digits or bits, or decimal or binary divide by zero.	23	39	7	13
III	ORD	Dynamic inclusive OR of the preceding three conditions (SSC, DDF, and TBZ). Enabling this condition permits an interrupt on any of the three conditions.	24	40	8	5
III	FDV	Floating point divide fault.	25[†]	41	9	8
III	EXO	Exponent overflow.	26	42	10	9
III	RMZ	Result is machine zero.	27	43	11	10
III	ORX	Dynamic inclusive OR of the preceding three conditions (FDV, EXO, and RMZ). Enabling this condition permits an interrupt on any of the three conditions.	28	44	12	4
III	SRT	Square root operation on negative operand.	29[†]	45	13	6
III	IND	Indefinite result or indefinite operand.	30[†]	46	14	7
I	BKP	Breakpoint flag was set on the breakpoint instruction (instruction 04).	31[†]	47	15	3

[†]Set during run-time initialization.

all-zero coefficient. A divisor having an indefinite value is not a zero divisor and does not cause a floating point divide fault. The result of a division by zero is an indefinite value which sets the IND data flag.

An SRT condition occurs if a square root operation is attempted with a negative operand. The square root of the absolute value of the negative operand is taken in this case, and the two's complement of this square root is stored as the result. The result, although meaningful, is not equivalent to the mathematical value of the square root of a negative number.

An IND condition occurs if an indefinite value is computed and stored into memory or into the register file. The condition also occurs if either or both of the operands of certain floating point operations have indefinite values (floating point arithmetic operations and floating point compare operations can set the IND data flag). Since an indefinite value results from a floating point operation in which either or both of the operands are indefinite values, indefinite values are likely to propagate. An FDV or EXO condition also sets the IND data flag.

Branches

When a data flag branch occurs, bit 52 is cleared, the address of the instruction that would have been executed next had the branch not occurred is stored in register 1, and control branches to the address in register 2. The address of a DFBM entry point is placed in register 2 during FORTRAN run-time initialization. Subsequent processing is determined by the bit settings in the DFB register and specifications made in any Q7DFSET, Q7DFOFF, and Q7DFCL1 calls.

The address in register 1 does not necessarily point to the instruction immediately following the instruction that caused the data flag branch. The hardware initiates a data flag branch only after all currently executing instructions have completed. Because instructions might be executing in parallel when the condition causing the data flag branch occurs, the branch can occur up to 35 instructions after the instruction that caused it. Also, the point at which control branches to DFBM can vary between executions of the same program because the load and store hardware operations can occur at different points as a result of the asynchronous nature of STAR I/O.

NOTE

The user can effect changes in the DFB register that conflict with DFBM. Use of the FORTRAN-supplied function Q8SDFB, the special calls Q8BADF and Q8LSDFR, or the system-provided utility DEBUG in a FORTRAN program that uses calls to DFBM entry points all should be done with great care.

DATA FLAG BRANCH SOFTWARE

A data flag branch, together with the subsequent processing performed by DFBM before the FORTRAN program resumes or aborts, is called a DFBM interrupt. A call to the DFBM entry point Q7DFSET can be used to enable and disable DFBM interrupts on specified conditions. Interrupt-handling routines are optional and can be specified through calls to one of the DFBM entry points Q7DFSET and Q7DFCL1, as described later in this section.

If the STAR hardware initiates a data flag branch during execution of a FORTRAN program, control branches to DFBM. DFBM checks the DFB register product bits in the following order:

1. JIT (bit 4)	5. ORD (bit 8)	9. EXO (bit 10)
2. SFT (bit 3)	6. SRT (bit 13)	10. RMZ (bit 11)
3. BKP (bit 15)	7. IND (bit 14)	11. SSC (bit 5)
4. ORX (bit 12)	8. FDV (bit 9)	12. DDF (bit 6)
		13. TBZ (bit 7)

Depending on the bits DFBM finds set and the interrupt-handling routines that the FORTRAN user has specified, DFBM calls the routine FT_ERMSG or passes control to an interrupt-handling routine established by the programmer.

Interrupt Classes

The DFBM interrupt conditions shown in table 14-1 can be divided into two classes, depending on whether the FORTRAN user can disable interrupts for the condition and how the interrupts are handled by DFBM. Interrupts on the class I conditions are always enabled; the corresponding mask bits are always set for the following conditions:

JIT
SFT
BKP

The FORTRAN user can enable or disable interrupts for all of the other conditions, which are class III conditions. Enabling or disabling of class III conditions is done using calls to one of the DFBM entry points Q7DFSET and Q7DFOFF as described later in this section.

DFBM processes the class III conditions as a group, as if they were all caused by a single event. Class I conditions are processed individually, as if they had been caused by separate events. A DFBM interrupt that processes a class I condition is called a class I interrupt, and one that processes class III conditions is called a class III interrupt.

Multiple Interrupts

The execution of a single hardware instruction can in some cases flag several class III conditions as well as one or more class I conditions. A number of product bits might be on when DFBM receives control as the result of a data flag branch. A single data flag branch could occur with enough product bits set that it would be translated into four DFBM interrupts, that is, three class I interrupts and one class III interrupt.

If a data flag branch occurs and more than one product bit is set, DFBM processes any class I interrupts first, one at a time, in the order JIT, SFT, and BKP. Then, if DFBM has been able to process the class I interrupts without aborting the program, it will process a class III interrupt. If a class I bit and a class III bit are set when DFBM gains control after a data flag branch, and if the specified interrupt-handling routines return after executing, the interrupt processing that would be performed is shown in table 14-2. Default processing for DFBM interrupts consists of issuing an error message and then either aborting or resuming the program, depending on whether the error was nonfatal, fatal, or catastrophic.

TABLE 14-2. MULTIPLE INTERRUPT PROCESSING

Class I Interrupt-Handling Routine Provided	Class III Interrupt-Handling Routine Provided	Processing Performed After Data Flag Branch Manager Gains Control
No	No	Class I error message issued, program aborted
Yes	No	Class I routine executed, class III error message issued, program aborted for fatal message and resumed otherwise
No	Yes	Class I error message issued, program aborted (class III routine not executed although class III condition flagged)
Yes	Yes	Class I routine executed, class III routine then executed, program resumed (no error messages issued by DFBM)

Default Interrupt Processing

In a typical DFBM interrupt, a class III interrupt might occur with one or more class III product bits set and with default processing being performed because no interrupt-handling routine has been specified. If the user does not specify any interrupt-handling routines and a data flag branch occurs, DFBM performs default interrupt processing as follows. Having gained control as a result of the data flag branch, and having checked the DFB register product bits in the order listed earlier, DFBM calls the routine FT_ERMSG to issue an error message for the condition indicated by the first product bit found to be on.

If the FT_ERMSG entry point SEP (System Error Processor, described in this section) was called previously in the FORTRAN program to specify an error exit subroutine for the error, FT_ERMSG calls the subroutine. An error message is issued (if applicable) before the user routine is called.

If the error message that FT_ERMSG issued was nonfatal, DFBM restarts the interrupted FORTRAN program at the address in register 1. If the error message was fatal or catastrophic, a dump of the contents of the DFB register is written onto the output file immediately following the error message, and the FORTRAN program aborts without return of control to DFBM. If the aborted program was being run as part of a batch job, the system utility DUMP writes a post-mortem dump onto the output file. The dump includes a full subroutine traceback in which DFBM appears to have been called by the interrupted routine (DFBM execution has actually been initiated by a hardware data flag branch). The system utility DUMP is described in the STAR Operating System Reference Manual, Volume 1.

Each class III condition has a separate error message, but only one message is issued when default processing is performed for a class III interrupt. The class III message issued is for the first class III product bit found on. For example, assume that the default class III interrupt conditions SRT, IND, and FDV are in effect at the time that a division operation is performed in which the divisor is zero. Also assume that the FORTRAN program is running in a batch job, has not disabled all data flag branches (has not cleared DFB register bit 52), and has not previously called SEP or Q7DFSET to specify a routine to handle division by zero. The division operation initiates a data flag branch. DFBM finds that bit 14 (IND product bit) of the DFB register is on and, since no class III interrupt-handling routine is available, calls FT_ERMSG. Since the user has not specified an error exit subroutine, FT_ERMSG issues a fatal error message for the IND condition, causes a DFB register dump to be written to the output file, and aborts the program. The error message and DFB register dump are shown in figure 14-7. Finally, since the job is a batch job, the DUMP utility produces a post-mortem dump. Note that no error message for the FDV condition is produced.

As another example, assume the same situation as in the previous example, with the exception that the FORTRAN program has called Q7DFSET to alter the class III interrupt conditions to ORX, SRT, and IND. The division operation with the zero divisor initiates a data flag branch. DFBM finds that bit 12 (ORX product bit) is on and calls FT_ERMSG, since no class III interrupt-handling routine is available. FT_ERMSG issues an error message for the ORX condition. Since the error is a warning, DFBM restarts the interrupted program at the address in register 1, even though a normally fatal condition (IND) has occurred.

CLASS III INTERRUPTS

If a class III interrupt occurs, DFBM performs default processing if the FORTRAN user has not provided a class III interrupt-handling routine through a Q7DFSET call. If the user has specified a class III interrupt-handling routine, DFBM takes the following actions:

1. Detects the condition by checking the DFB register product bits.
2. Saves a copy of the entire register file of the interrupted routine.
3. Clears the data flags (this also clears the product bits), leaving the mask bits as they are.
4. Sets bit 52, re-enabling data flag branches.
5. Calls the class III interrupt-handling routine.

In a class III interrupt where an interrupt-handling routine is called, no standard error message is issued by DFBM. DFBM manages class III interrupts according to the following rules:

- Any routine or subroutine of a FORTRAN program can specify and respecify class III interrupt conditions and interrupt-handling routines as frequently as desired. Q7DFSET calls are used to make the specifications.
- When a routine calls a subroutine, the class III interrupt conditions and class III interrupt-handling routines in effect in the calling routine are put into effect in the subroutine.
- When a routine returns to its caller, the class III interrupt conditions and class III interrupt-handling routines in effect at the time of the call are reinstated.

Each subroutine in a FORTRAN program can make different specifications of how class III interrupts are to be handled locally and in lower-level routines, without those specifications affecting how class III interrupts are handled by higher-level routines.

The rules of scope are illustrated in figure 14-8. In the figure, the main program begins execution with the default conditions in effect and executes until a call to Q7DFSET alters the default selection. A new set of conditions is selected by the second call to Q7DFSET and remains in effect until subroutine K is called. Selections remain in effect until subroutine K calls Q7DFSET. This newest set of conditions continues in effect when subroutine D is called and when the return to subprogram K occurs. When K completes execution and control returns to the main program, conditions in effect at the time subroutine K was called are reestablished and persist through the call to subprogram Z and the return to the main program.

```
   ERROR  124 DATA FLAG BRANCH - INDEFINITE RESULT - REGISTER 1 ADDRESS 000000012260

 DATA FLAG BRANCH REGISTER

 00000000 01000010 00011000 01000111 00000000 01001010 00010000 00100000

                         SFT JIT SSC DDF TBZ ORD FDV EXO RMZ ORX SRT IND BKP
 PRODUCT BITS (3-15)      0   0   0   0   0   0   1   0   0   0   0   1   0
 MASK BITS    (19-31)     1   1   0   0   0   0   1   0   0   0   1   1   1
 DATA FLAGS   (35-47)     0   0   0   0   0   0   1   0   0   1   0   1   0
```

Figure 14-7. DFB Register Dump Example

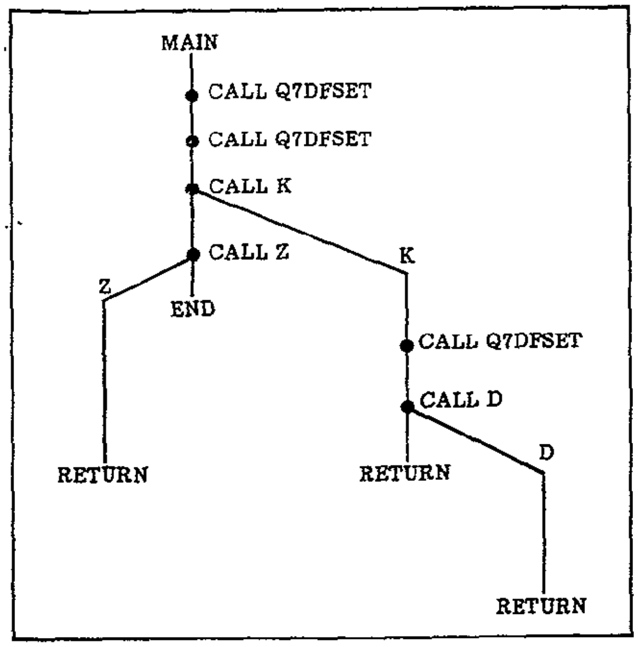


Figure 14-8. Scope of Selected Conditions

Interrupt-Handling Routines

A class III interrupt-handling routine can appropriately be written in FORTRAN. The routine must have no arguments. Any communication with higher-level routines must be through the use of COMMON statements.

At the time that the class III interrupt-handling routine gains control, all interrupts that were enabled at the time of the data flag branch are still enabled (the mask bits have not been altered, and bit 52 has been set). If a class III interrupt occurs while the interrupt-handling routine or any lower-level routine is executing, DFBM causes a catastrophic error message to be issued and the program to be aborted. The interrupt-handling routine can disable class III interrupts for the period of time that it is executing by calling Q7DFSET. Any class I interrupts occurring in a class III interrupt-handling routine are handled immediately.

All data flags in the DFB register have been cleared when the class III interrupt-handling routine receives control from DFBM. The routine can learn the status of the data flags as they were at the time of the data flag branch, as well as certain other information about the interrupt, by calling Q7DFLAGS.

If the class III interrupt-handling routine executes a RETURN statement, DFBM restarts the interrupted FORTRAN program or subprogram at the address in register 1. DFBM leaves the DFB register mask bits exactly as they were at the time of the data flag branch unless the class III interrupt-handling routine has made a call to Q7DFOFF. An interrupt-handling routine can call Q7DFOFF to disable specified conditions in the interrupted FORTRAN program at the time that the program is restarted. A call to Q7DFOFF might be advantageous if the conditions causing a data flag branch would cause a large number of other data flag branches to occur.

Q7DFSET

A call to Q7DFSET can be used to do either or both of the following:

- Specify the conditions on which a class III interrupt is to occur (that is, alter DFB register mask bits).
- Specify the name of a user-provided interrupt-handling routine to be called in the event of a class III interrupt.

Default class III interrupt conditions can be reestablished using Q7DFSET, either by specifying the SRT, IND, and FDV conditions or by specifying 'STD' as an argument. Default class III interrupt processing can also be reestablished with a Q7DFSET call.

Forms:

CALL Q7DFSET (ihr)

CALL Q7DFSET (ihr, 'NUL')

CALL Q7DFSET (ihr, 'mb_1', . . . , 'mb_n')

ihr	Zero, or the name of a user-provided interrupt-handling routine that is to be called if a class III interrupt occurs. Zero indicates that default processing is to be performed for class III interrupts (zero reestablishes the specification in effect at the time that the FORTRAN program began executing).
'NUL'	Indicates that all class III mask bits are to be cleared, disabling all class III interrupts.
'mb_i'	'STD', or one of the class III interrupt condition designators given in table 14-1. The designator must be enclosed in apostrophes. A designator from table 14-1 indicates that the corresponding mask bit is to be set. 'STD' indicates that the default class III mask bits - corresponding to the SRT, IND, and FDV conditions - are to be set. 'STD' can be used in combination with other designators in the same argument list.

No mask bits are altered from their current settings when Q7DFSET is called with only one argument, ihr. When Q7DFSET is called with two or more arguments, any class III mask bits not indicated by the argument list are cleared. The user must remember to declare any subroutine name used in a Q7DFSET call with an EXTERNAL statement.

For example, given the declaration EXTERNAL USRRTN, the following are valid Q7DFSET calls:

CALL Q7DFSET (USRRTN)

CALL Q7DFSET (USRRTN, 'EXO', 'IND', 'SRT', 'FDV')

CALL Q7DFSET (USRRTN, 'EXO', 'STD')

CALL Q7DFSET (0, 'STD')

CALL Q7DFSET (0, 'NUL').

The first call specified USRRTN to be the class III interrupt-handling routine. The second or third call has the effect of specifying that USRRTN is to be the class III interrupt-handling routine, that mask bits 25, 26, 29, and 30 are to be set, and that mask bits 21, 22, 23, 24, 27, and 28 are to be cleared. The fourth call restores the default set of conditions and default class III interrupt processing. The fifth call restores default class III interrupt processing but disables all data flag branches on all class III conditions.

Q7DFLAGS

The user can obtain information about the most recent class III interrupt by calling Q7DFLAGS.

Form:

```
    CALL Q7DFLAGS(pb,fb,ad,rf)
```

pb	A type logical array, declared to be a one-dimensional array of ten elements, in which DFBM returns the ten class III product bits (bits 5 through 14). Values returned are .FALSE. for bits that are cleared and .TRUE. for bits that are set. The order of the values in the array is the same as for the class III conditions listed in table 14-1.
fb	A type logical array, declared to be a one-dimensional array of eleven elements, in which DFBM returns the ten class III data flags (bits 37 through 46), followed by the pipe 2 register instruction data flag as the eleventh value. Values returned are .FALSE. for bits that are cleared and .TRUE. for bits that are set. The order of the values in the array is the same as for the class III conditions shown in table 14-1.
ad	A variable of type integer in which DFBM returns the address contained in register 1 at the time of the data flag branch.
rf	Optional. A type integer or real array (or a descriptor array of type integer or real) of size 256 in which DFBM returns the register file contents as they were at the time of the data flag branch.

If Q7DFLAGS is called before any class III interrupts have occurred, all of the data flags and product bits are shown to be .FALSE. and all other values returned are zero.

For example, the statements

```
    LOGICAL P(10), DF(11)
    INTEGER ADDR, REGS(256)
    CALL Q7DFLAGS (P,DF,ADDR,REGS)
```

place the product bits in logical array P, the data flags in logical array DF, the register 1 address in integer variable ADDR, and the register file in integer array REGS.

Q7DFOFF

By calling Q7DFOFF, a class III interrupt-handling routine can cause class III interrupt conditions to be disabled at the time that the interrupted FORTRAN program is restarted. A Q7DFOFF call issued from a routine other than an interrupt-handling routine or lower-level routine has no effect.

Form:

CALL Q7DFOFF ('mb_1', . . . , 'mb_n')

'mb_i'	'ALL', 'STD', or one of the class III interrupt condition designators given in table 14-1. A designator from table 14-1 indicates that the corresponding mask bit is to be cleared at the time that the interrupted routine is restarted. 'ALL' indicates that all class III interrupts are to be disabled. 'STD' indicates that the SRT, IND, and FDV class III interrupts are to be disabled.

Any mask bits not specified in the call are left unaffected by the call. If a class III interrupt-handling routine executes a RETURN statement after calling Q7DFOFF, DFBM gains control and disables the specified class III interrupts. The interrupts remain disabled until a new call to Q7DFSET is made. The scope of a Q7DFOFF call is the same as the scope of its associated Q7DFSET call.

For example, the following are valid Q7DFOFF calls:

```
    CALL Q7DFOFF('IND','FDV')

    CALL Q7DFOFF('ALL')
```

The first call will cause DFB register bits 25 and 30 to be cleared at the time that DFBM restarts the interrupted FORTRAN program. The second call would cause all of the class III mask bits to be cleared at that time.

CLASS I INTERRUPTS

Class I interrupts are always enabled; the class I mask bits are always on, and the FORTRAN program cannot be used to clear them. A FORTRAN user can specify class I interrupt-handling routines. A separate routine can be specified for each of the three class I conditions.

A user-specified interrupt-handling routine for handling a class I interrupt must be written in a lower-level language such as an assembler language. FORTRAN is not a sufficiently low-level language for the purpose of handling class I interrupt conditions. Class I interrupts do not occur unless the user takes specific action to cause them, such as utilizing the breakpoint feature of the DEBUG system utility or issuing the special call Q8WJTIME to set the job interval timer.

If a class I interrupt occurs, DFBM performs default processing unless the FORTRAN user has provided an interrupt-handling routine for the class I condition and made it known by means of a Q7DFCL1 call. If the user has specified an appropriate class I interrupt-handling routine, DFBM takes the following actions:

1. Detects the condition by checking the DFB register product bits.

2. Turns off the data flag associated with the interrupt (this also clears the associated product bit).

3. Branches to the address specified in the most recently executed Q7DFCL1 call for the specific condition.

Bit 52, the data flag enable bit, was cleared as part of the data flag branch and is not set by DFBM before the branch to the class I interrupt-handling routine occurs.

DFBM manages class I interrupts according to the following rules:

- Any routine or subroutine in a FORTRAN program can specify and respecify an interrupt-handling routine for a class I interrupt condition as frequently as desired. Q7DFCL1 calls are used to make the specification.

- Subroutine levels are not considered in managing class I interrupts in the way that they are in the managing of class III interrupts. The specification of a class I interrupt-handling routine is in effect for the duration of the program or until another Q7DFCL1 call is issued.

Interrupt-Handling Routines

A class I interrupt-handling routine is responsible for most of the interface between itself and DFBM. Since DFBM does not execute a standard call sequence, but instead simply branches to an address in the interrupt-handling routine, the address of the data base of the class I interrupt-handling routine is not available in register 1E. The interrupt-handling routine is responsible for saving registers 1 through FF and restoring them before branching back to DFBM. The address to which the class I interrupt-handling routine must branch is returned in a parameter of the Q7DFCL1 call that was most recently issued by the FORTRAN program. At the time that control branches to the class I interrupt-handling routine, all interrupts have been disabled.

Q7DFCL1

A call to Q7DFCL1 can be used to specify the name of a user-provided class I interrupt-handling routine to which DFBM must branch if the specified class I interrupt occurs. Q7DFCL1 returns the address in DFBM to which the interrupt-handling routine must return upon completion.

Form:

CALL Q7DFCL1(ihr, return, 'mb')

ihr	A one-word variable containing the virtual bit address of an interrupt-handling routine to which DFBM is to branch in the event that the specified class I interrupt condition, mb, occurs.
return	A one-word variable in which Q7DFCL1 returns the virtual bit address in DFBM to which the interrupt-handling routine for the condition mb must branch upon completion.
'mb'	One of the class I interrupt condition designators JIT, SFT, and BKP. The designator must be enclosed in apostrophes.

At least one Q7DFCL1 call must be made for each of the class I conditions for which the user desires other than default processing to be performed.

MDUMP

MDUMP is an object module callable by FORTRAN programs or META subroutines of a FORTRAN program. The module can be called as often as necessary to perform dumps of specified areas of virtual memory.

Form:

CALL MDUMP(first,len;dtype,u)

first	Simple variable, array, or array element with which the area to be dumped begins.
len	Length (in words) of area to be dumped.
dtype	Dump format:

'Z'	Hexadecimal dump
'I'	Integer dump
'Ew.d' or 'Fw.d'	Floating point dump, where w is the field width and d is the fractional decimal digit count

If dtype has a value other than one of the above, a hexadecimal dump is made.

u	Logical unit number of file to which dump is to be written. If u=0, the dump is written to OUTPUT.

The dump is written to a file or files defined in the PROGRAM statement or in the statement that requests execution of a FORTRAN program. For example, if a call to MDUMP is made, indicating that the dump is to be written to logical unit 3, then a file declaration UNIT3=filename must also be made. See section 7 for UNITn=f parameters in the PROGRAM statement.

MDUMP can be called from META subroutines of a FORTRAN program using the standard calling sequence conventions described in section 12. The logical unit referenced in the call must be defined in the same way as for calls made to MDUMP from a FORTRAN routine.

Sample output from a call to MDUMP is given in figure 14-9. An array I was declared and initialized with the two statements

```
DIMENSION  I(20)
DATA       I/5*7,15*12/
```

and then using the statement

```
CALL MDUMP(I,20,'Z',0)
```

a call to MDUMP was made. The output generated by this call shows 20 words of memory, four words per line of output. As 'Z', that is, a hexadecimal dump, was requested

```
        HEX DUMP      TIME 22.33.02         CALL ADDRESS  000000082C0

 BIT ADDRESS                                     C-O-N-T-E-N-T-S                                   WORD ADDRESS                ASCII


000000070180  00000000 00000007  00000000 00000007  00000000 00000007  00000000 00000007  00000001C06
000000070280  00000000 00000007  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000001C0A
000000070380  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000001C0E
000000070480  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000001C12
000000070580  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000000 0000000C  00000001C16
```

Figure 14-9. MDUMP Output

in the parameter list of the call, the 15 elements with value of 12 appear in the dump as hexadecimal C.

SYSTEM ERROR PROCESSOR (SEP)

The function of the STAR System Error Processor (SEP) is to enable the user to change certain run-time error attributes. FORTRAN run-time error conditions can belong to one of three classes: warning (W) for nonfatal but probably undesirable conditions, fatal (F) for conditions that cause abnormal termination of the program during execution, and catastrophic (C) for conditions that are not subject to user control. By using SEP, the user can set fatal error conditions to nonfatal status, and warning conditions can be made fatal. SEP is called as a subroutine by an executing program.

Form:

CALL SEP(p_1,p_2,p_3,p_4,p_5,p_6,p_7)

p_1 The error number of the run-time error (see appendix B). When p_1 is zero, then all other parameters must be zero except p_4, which refers to the global nonfatal error count.

p_2 Indicates the error class to which p_1 is to be changed. Parameter p_2 can be one of the following:

'F' Sets the error class to fatal. Program execution is terminated abnormally when this condition occurs.

'W' Sets the error class to warning. Execution continues when this nonfatal condition occurs.

0 No error class change is to take place.

When a fatal error is changed to a warning error, parameter p_4 should also be specified to change the maximum error count to a nonzero number.

p_3 The error exit subroutine entry point name (which must be included in an EXTERNAL statement in the same program unit). If the error p_1 occurs, entry point p_3 is called and execution continues from there. If p_3 is zero, no error exit is implied and processing continues if the error is nonfatal. If p_1 is a fatal error and the subroutine p_3 executes a RETURN, the program aborts; if p_1 is nonfatal and p_3 executes a RETURN, program execution continues.

p_4 An integer constant indicating the maximum error count for nonfatal errors; if the number of nonfatal error condition occurrences reaches p_4 then execution terminates. An infinite error count is indicated by a value of -1. If p_4 is zero, no change for this parameter is indicated (p_4 might have been assigned a value in a previous SEP call).

The maximum error count for a warning error for which SEP has not been called is 25. The maximum error count for a fatal error for which SEP has not been called is zero. When p_2 changes a fatal error to a warning error, p_4 should also be specified.

p_5 The error display suppression argument, applying only to nonfatal errors. p_5 can assume one of the following values:

'S' Indicates that the error message, normally sent to the user's output file and to the terminal, is to be suppressed.

0 No message suppression is to take place.

p_6 The number of characters in p_7, excluding bracketing apostrophes. The name of the routine or file in which the error occurred is appended automatically to the message string whenever applicable.

p_7 A character string that replaces the standard message associated with p_1. The string must be enclosed by apostrophes to form a character constant. Parameter p_6 must appear when p_7 appears.

Parameter p_1 and at least one additional parameter must be included in the call. Any parameter other than p_1 must be indicated as zero if that one is not to be specified; however, trailing zero parameter list entries can be omitted.

Calls to SEP can appear as frequently as required in a program, and the error attributes change any number of times during program execution. The SEP routine is especially useful during program checkout, enabling traps to be set for error conditions that could prove difficult to diagnose. Care should be exercised when altering fatal errors to nonfatal status.

Examples:

```
CALL SEP(26,'W',SUB,5,0,38,'ATTEMPT TO
    READ INTEGER UNDER D FORMAT')
```

Use of the above call causes the standard message for error 26, INTEGER MODE, CONVERSION CODE D, to be replaced with the error message ATTEMPT TO READ INTEGER UNDER D FORMAT, and the error level altered from fatal to warning. If error 26 occurs during program execution, the program issues the message, then branches to a subroutine named SUB, and processing continues from that point. When the error condition occurs for the fifth time, program execution is aborted.

```
CALL SEP(75,'F')
```

This call means that if the condition associated with error 75 occurs at any time in the program, it is considered fatal and the program execution is aborted.

```
CALL SEP(26,'W',0,10)
```

In the above call, error condition 26 is made nonfatal. When the error occurs for the tenth time, program execution is aborted.

```
CALL SEP(72,'W',0,100,'S')
```

This call means that error 72 can occur up to 100 times without the error message appearing on the user's terminal or output file.

CONCURRENT I/O SUBROUTINES

The mass storage input/output subroutines for concurrent I/O transmit data in an optimal manner between main memory and unstructured files on mass storage. No buffers are required and no structuring information is processed when a concurrent I/O routine is used. The routines also allow overlapping of computation with input or output of large data arrays, thus maximizing the use of system resources. Unless these routines are being used, processing of a FORTRAN program is suspended while an input/output request is being honored.

The four concurrent I/O routines and their functions are:

Q7BUFIN	Transfer data from mass storage to main memory
Q7BUFOUT	Transfer data from main memory to mass storage
Q7WAIT	Test or wait for input/output completion; obtain error status of operation
Q7SEEK	Reset page address at which data is to be transferred

Any file referenced in a call to the concurrent I/O routines must be declared in the PROGRAM statement to be an explicit mass storage file. The file cannot be referenced in any of the FORTRAN input/output or unit positioning statements. Once input or output is performed on a file using concurrent I/O routines, all input and output on that file must be performed only by means of those routines.

The user is responsible for the correspondence between the data record size and the size of the physical block to or from which the data is transferred. Any padding required to reconcile record size with block size is also the user's responsibility, as is the determination of any logical end-of-file that might exist before the physical end of the mass storage assigned to the file. (The concurrent I/O routines recognize the physical end of a file but no logical end-of-file.) The user is also responsible for checking for the existence of error conditions resulting from the transfer. No notification of the user is made of error conditions although certain conditions are flagged so that the user can query the system about them by calling Q7WAIT.

The greatest efficiency in input/output using the concurrent I/O routines may be obtained when overlap of input/output and computational operations is maintained throughout execution. When computational activity continues until completion of the previous input/output request, maximum overlap has been achieved.

ARRAY ALIGNMENT CONSIDERATIONS

The user must align the arrays named in the Q7BUFIN and Q7BUFOUT calls on small page boundaries, and must define the arrays to be multiples of small pages (padding must be added by the user if necessary). At the time a concurrent I/O call is executed, the program aborts if the array has not been aligned on a page boundary. Alignment can be accomplished by declaring the arrays to reside in one or more labeled common blocks, then using the GRSP parameter of the LOAD system control statement to load the common blocks on small page boundaries.

If the size of an array is greater than 24 small pages (that is, 12 288 words), the array should be placed on a large page to obtain the I/O efficiency that is derived from using concurrent I/O. The GRLP parameter of the LOAD system control statement can be used to load a labeled common block containing the large array on a large page boundary. More than one array can be defined within the 65 536 words of a large page. If necessary, a single array can overlap a large page boundary; however, this results in decreased efficiency because multiple explicit I/O requests must be issued by the system to transfer that array. When multiple explicit I/O requests are issued, concurrent processing ceases after the first of the multiple requests completes and cannot resume during the remainder of the I/O for that call. If the array did not overlap a large page boundary, a single explicit I/O request would initiate transfer of the array and control would return immediately to the program so that computation could continue.

For example, suppose that in a FORTRAN program a 20-page array BIGRAY and a 100-page array RA2 are used in calls to the concurrent I/O routines. The program then should also contain the statement

```
COMMON/ANAME/BIGRAY(10240),RA2(51200)
```

which declares an array BIGRAY with 10 240 words and an array RA2 with 51 200 words to reside in the labeled common block ANAME. After the program is compiled (using the system control statement FORTRAN.), loading is performed using the system control statement

```
LOAD,BINARY,CN=XECUTE,GRLP=*ANAME
```

which produces the executable virtual code file XECUTE from the file BINARY, and loads the common block ANAME on a large page boundary.

Whether or not an array has been placed on a large page, a call to Q7BUFIN or Q7BUFOUT transfers exactly the number of small pages specified in the call. The user can aid the I/O routines in deciding how an array was mapped by specifying 'SMALL' or 'LARGE' for the map parameter of the Q7BUFIN or Q7BUFOUT call (specification of the parameter does not itself cause the alignment to be performed).

SUBROUTINE CALLS

Two Q7BUFIN calls, two Q7BUFOUT calls, or a Q7BUFIN and a Q7BUFOUT call can be active at one time for a given file. If a third call is made for data transmission before a Q7WAIT call is issued, the program is aborted. The programmer is responsible for assuring that the specified portions of a file on which there are two outstanding I/O requests do not overlap.

The file address to which data is written or from which data is read can be specified in either of two ways. The Q7BUFIN or Q7BUFOUT call can specify a relative page address as a parameter. Alternatively, the Q7SEEK call can establish a relative page address for a succeeding Q7BUFOUT or Q7BUFIN call. In the absence of either specification of page address, the file is scanned sequentially, beginning at page zero of the file when it is first referenced by the program. Each Q7BUFIN or Q7BUFOUT call moves the current read/write position forward by a specified amount (equal to the value of the len parameter).

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

Q7BUFIN

The Q7BUFIN subroutine transfers data from a mass storage file to an array in main memory by means of explicit I/O. The first time it is called by the program, Q7BUFIN defines the array specified in the call to be the buffer for explicit input/output and initiates data transfer from the file. Control then returns immediately to the program unless the user aligned the array in such a way that the system is forced to issue multiple I/O requests. The array must not be referenced until a call to Q7WAIT has established that the transfer was successfully completed.

Form:

CALL Q7BUFIN(u,a,len,map,faddr)

u	Logical unit number of the mass storage file from which data is to be read. An integer constant or integer variable having a value of from 1 to 99, associated with the file by means of the PROGRAM statement.
a	Array element or array name (an array name indicates the first element of the array). Data from u is stored beginning at a, which must lie on a small page boundary.
len	An integer constant or integer variable indicating the number of small pages to be transferred.
map	Optional. The character (or Hollerith) constant 'SMALL' (or 5HSMALL) or 'LARGE' (or 5HLARGE), indicating that the array a was mapped onto a small page or large page, respectively. Recommended when array a has a length greater than 24 but was not mapped onto a large page (map would be 'SMALL').
faddr	Optional (if faddr is specified, map must also be specified). An integer constant or integer variable to whose value the current read position on u is modified before the read begins. A variable faddr is defined and redefined only by the user. If faddr is omitted, default is the current read position.

Depending on the value of len, a Q7BUFIN call might transfer data into only part of the array named by a, or it might transfer data to the words located beyond the end of the array.

Q7BUFOUT

The Q7BUFOUT subroutine transfers data from an array in main memory to a mass storage file by means of explicit I/O. The first time it is called by the program, Q7BUFOUT defines the array specified in the call to be the buffer for explicit input/output and initiates data transfer to the file. Control then returns immediately to the program unless the user aligned the array in such a way that the system is forced to issue multiple I/O requests. The array must not be referenced until a call to Q7WAIT has established that the transfer was successfully completed.

Form:

CALL Q7BUFOUT(u,a,len,map,faddr)

u	Logical unit number of the mass storage file to which data is to be written. An integer constant or integer variable having a value of from 1 to 99, associated with the file by means of the PROGRAM statement.
a	Array element or array name (an array name indicates the first element of the array). Data from the block starting at a, which must lie on a small page boundary, is output to u.
len	An integer constant or integer variable indicating the number of small pages to be transferred.
map	Optional. Same as the map parameter for Q7BUFIN.
faddr	Optional (if faddr is specified, map must also be specified). An integer constant or integer variable to whose value the current write position is modified before the write begins. A variable faddr is defined and redefined only by the user. If faddr is omitted, default is the current write position.

Depending on the value of len, a Q7BUFOUT call might transfer only part of the array named by a, or it might transfer data located beyond the end of the array.

Q7WAIT

The Q7WAIT subroutine must be called to determine whether or not input/output operations have completed without transmission error for a prior Q7BUFIN or Q7BUFOUT call for the specified file. I/O errors are reported to the user only through the stat parameter of this call. Each time Q7WAIT executes, it returns a status value (stat) that indicates data transmission status. When data transmission is still in progress, control either returns immediately to the program or is relinquished by the program until the data transfer is complete, depending on the parameters in the call. Q7WAIT can also be used to determine when the physical end of the mass storage assigned to a file has been reached.

Form:

CALL Q7WAIT(u,a,stat,ret,len)

u	Logical unit number of the file associated with the array a in a concurrent I/O operation in progress. An integer constant or integer variable having a value of from 1 to 99, associated with the file by means of the PROGRAM statement.
a	Array element or array name (an array name indicates the first element of the array) involved in a Q7BUFIN or Q7BUFOUT operation.
stat	An integer variable whose value is returned by the call to Q7WAIT. The value returned indicates the status of the I/O operation:

0 = Normal completion

1 = Physical end-of-file reached

2 = Data transfer error due to hardware failure

3 = I/O operation not yet completed

ret Optional. Integer constant or integer variable specifying action to be taken upon return from Q7WAIT call:

0 = If I/O is still in progress at time of call, program should wait (computation should cease) until I/O is completed normally or abnormally. Default.

1 = If I/O is still in progress at time of call, program should not wait but control should be returned to it immediately.

len Optional. If len is specified, ret must also be specified. An integer variable whose value is returned by the call to Q7WAIT. The value returned is the number of pages actually transmitted during the I/O operation. (If the physical end of the mass storage was reached, len might be less than the number of small pages requested to be transferred.)

Q7SEEK

The Q7SEEK subroutine resets the page address at which data transmission is to occur. It is an alternative to a faddr parameter in a Q7BUFIN or Q7BUFOUT call.

Form:

```
CALL Q7SEEK(u,faddr)
```

u	Logical unit number of unit to be referenced in a subsequent Q7BUFIN or Q7BUFOUT call. An integer constant or integer variable having a value of from 1 to 99, associated with the file by means of the PROGRAM statement.
faddr	Optional. If faddr is zero or omitted, the current read/write position of u is repositioned at the beginning of the file (a REWIND is executed). Otherwise, faddr has the same effect as the faddr parameter of a Q7BUFIN or Q7BUFOUT call.

A CALL Q7SEEK(u,0) or CALL Q7SEEK(u) statement performs a rewind on u.

Q8WIDTH SUBROUTINE

The subroutine Q8WIDTH enables a program to set a fixed record length for an ASCII output file. The default record length for a PUNCH file is 80 characters. For all other files, the default record length is variable, with trailing blanks removed from the end of each line.

Form:

```
CALL Q8WIDTH(u,width)
```

u	Logical unit number of the file
width	Record length for subsequent ASCII output to the file. The width must not exceed 137. If width is specified as zero, trailing blanks are removed from each line and the record length is variable.

SUPPLIED SUBROUTINES

A number of predefined subroutines are provided with the STAR FORTRAN compiler. The predefined subroutines are referenced by CALL statement. The subroutines are listed in alphabetic order.

DATE

This subroutine generates the same result as the DATE function. The form is

```
CALL DATE(a)
```

The result is stored in the argument a, which can be any 8-byte variable. Within any particular routine, DATE must be consistently called either as a function or a subroutine.

RANGET

This subroutine obtains the current value of the seed in the random number generator. The form is

```
CALL RANGET(n)
```

The argument n must be of type integer.

RANSET

This subroutine sets the seed in the random number generator. The form is

```
CALL RANSET(n)
```

The argument n must be integer. The current seed is set to the specified value if the argument is an odd positive integer. If the specified value is an even positive integer, the value is increased by 1 to an odd value. If the specified value is zero or negative, the current seed is set to the default value X'0000 54F4 A3B9 33BD'.

SECOND

This subroutine generates the same result as the SECOND function described in section 15. The form is

```
CALL SECOND(a)
```

The result is stored in the argument a, which can be any real variable. Within any particular routine, SECOND must be consistently called either as a function or a subroutine.

TIME

This subroutine generates the same result as the TIME function described in section 15. The form is

```
CALL TIME(a)
```

The result is stored in the argument a, which can be any 8-byte variable. Within any particular routine, TIME must be consistently called either as a function or a subroutine.

VRANF

This subroutine generates a vector of random numbers. The form is

```
CALL VRANF(v,n)
```

The argument v is a real array that is to contain the generated vector of random numbers. The argument n is an integer that specifies the length of argument v.

STACKLIB ROUTINES

The STACKLIB routines can be called for the purpose of optimizing certain loop constructs that cannot be vectorized. A loop construct that can be optimized is coded as a subroutine call. The subroutine name establishes the type of operation, and the arguments specify the operands to be used. In all cases, a STACKLIB call can be considered as replacing an equivalent DO loop.

The efficiency of STACKLIB routines is gained through maximum use of the instruction stack and through optimal use of the register file. For example, a STACKLIB routine can use a large part of the register file to hold elements of a vector operand. STACKLIB routines typically contain unrolled loops that produce more than one result per loop iteration.

The STACKLIB naming conventions allow for a large number of possible routine names. The routines currently supported represent a selection of the most useful STACKLIB constructs. The available STACKLIB routines are listed in table 14-3 and table 14-4.

Dyadic form:

CALL Q8fbrm(res,v2,v1,num)

f	One of the four arithmetic operations (A=add, S=subtract, M=multiply, D=divide).
b	Broadcast mask indicating whether either operand is invariant, that is, scalar (0=both vectors, 1=operand v1 scalar, 2=operand v2 scalar).
r	Recursion mask (0=no recursion, 1=recursive v1, 2=recursive v2).
m	Miscellaneous designator (currently always 0).
res	Result operand first address. A vector must be of type real.
v2	Left operand first address. A vector must be of type real.
v1	Right operand first address. A vector must be of type real.
num	The number of results to be produced. The value must be a positive integer.

Triadic form:

CALL Q8fsbrm(res,v4,v2,v1,num)

f	One of the four arithmetic operations (A=add, S=subtract, M=multiply, D=divide) used as the first operator.
s	One of the four arithmetic operators used as the second operator.
b	Broadcast mask indicating any invariant operands (0=no scalar operands; 1, 3, or 5=scalar v1; 2, 3, or 6=scalar v2; 4, 5, or 6=scalar v4).
r	Recursion mask (0=no recursion; 1, 3, or 5=recursive v1; 2, 3, or 6=recursive v2; 4, 5, or 6=recursive v4)
m	Miscellaneous designator (0 or 2=forward count; 1 or 3=backward count; 0 or 1=forward order of operations; 2 or 3=reverse order of operations)

TABLE 14-3. STACKLIB CALLS WITH FORWARD COUNT

Description	Type	STACKLIB Call With Sample Arguments	Equivalent Statement Contained In The Loop DO xx I = 2,N Where I Ranges From 2 Through N
Add, recursive v1	Dyadic	CALL Q8A010(A(2),B(2),A(1),N-1)	A(I)=B(I)+A(I-1)
Add, recursive v2	Dyadic	CALL Q8A020(A(2),A(1),B(2),N-1)	A(I)=A(I-1)+B(I)
Multiply add, recursive v2	Triadic	CALL Q8MA020(A(2),B(1),A(1),C(2),N-1)	A(I)=(B(I-1)*A(I-1))+C(I)
Multiply add, recursive v4	Triadic	CALL Q8MA040(A(2),A(1),B(1),C(2),N-1)	A(I)=(A(I-1)*B(I-1))+C(I)
Multiply add, recursive v1, reverse order	Triadic	CALL Q8AM011(A(2),B(2),C(1),A(1),N-1)	A(I)=B(I)+(C(I-1)*A(I-1))
Multiply add, recursive v2, reverse order	Triadic	CALL Q8AM021(A(2),B(2),A(1),C(1),N-1)	A(I)=B(I)+(A(I-1)*C(I-1))
Subtract multiply, recursive v1, reverse order	Triadic	CALL Q8SM011(A(2),B(2),C(2),A(1),N-1)	A(I)=B(I)-(C(I)*A(I-1))
Subtract multiply, recursive v2, reverse order	Triadic	CALL Q8SM021(A(2),B(2),A(1),C(2),N-1)	A(I)=B(I)-(A(I-1)*C(I))

res Result operand first address. A vector must be of type real.

v4 Left operand first address. A vector must be of type real.

v2 Middle operand first address. A vector must be of type real.

v1 Right operand first address. A vector must be of type real.

num The number of results to be produced. The value must be a positive integer.

The general form of a DO loop equivalent to a dyadic STACKLIB reference is:

```
    DO xx ind = first,last
 xx res(ind) = v2(ind)ⓕv1(ind)
```

The general form of a DO loop equivalent to a triadic STACKLIB reference with b=0 and m=0 is:

```
    DO xx ind = first,last
 xx res(ind) = v4(ind)ⓕv2(ind)ⓢv1(ind)
```

The ⓕ and ⓢ indicate one of the functions +, -, *, or /. In the triadic operation, the first operator is used on v4 and v2, and the second operator is used on the result of the first operation and v1. The count can be backward rather than forward, as indicated by the m part of the routine name. If the count is backward, the general form becomes:

```
    DO xx ind = first,last
    irev = last+first-ind
 xx res(irev) = v4(irev)ⓕ v2(irev)ⓢ v1(irev)
```

The order of operations can be reversed, as indicated by the m part of the routine name. In reverse order, the second operator is used on v2 and v1, and the first operator is used on v4 and the result of the first operation.

The operands can be scalar rather than vector, as indicated by the b part of the routine name.

NOTE

Since STACKLIB routines are implemented for efficiency, the validity of arguments is not checked. If the routine name indicates a certain recursive operand, an offset of 1 from the result first address is assumed, and the first address value given in the argument list is ignored.

TABLE 14-4. STACKLIB CALLS WITH BACKWARD COUNT

Description	Type	STACKLIB Call With Sample Arguments	Equivalent Statement As Contained In The Loop DO xx I = 2,N With J = (N+1)-I Included, Where J Ranges From N-1 Through 1
Multiply add, recursive v1, scalar v2	Triadic	CALL Q8MA212(A(N-2),B(N-2),S,A(N-1),N-1)	A(J)=(B(J)*S)+A(J+1)
Multiply add, recursive v1, scalar v4	Triadic	CALL Q8MA412(A(N-2),S,B(N-2),A(N-1),N-1)	A(J)=(S*B(J))+A(J+1)
Multiply add, recursive v4, scalar v1, reverse order	Triadic	CALL Q8AM143(A(N-2),A(N-1),B(N-2),S,N-1)	A(J)=A(J+1)+(B(J)*S)
Multiply add, recursive v4, scalar v2, reverse order	Triadic	CALL Q8AM243(A(N-2),A(N-1),S,B(N-2),N-1)	A(J)=A(J+1)+(S*B(J))
Subtract multiply, recursive v1, reverse order	Triadic	CALL Q8SM013(A(N-2),B(N-2),C(N-2),A(N-1),N-1)	A(J)=B(J)-(C(J)*A(J+1))
Subtract multiply, recursive v2, reverse order	Triadic	CALL Q8SM023(A(N-2),B(N-2),A(N-1),C(N-2),N-1)	A(J)=B(J)-(A(J+1)*C(J))
Divide add, recursive v2, scalar v4 and v1, reverse order	Triadic	CALL Q8DA523(A(N-2),S,A(N-1),T,N-1)	A(J)=S/(A(J+1)+T)
Divide add, recursive v1, scalar v4 and v2, reverse order	Triadic	CALL Q8DA613(A(N-2),S,T,A(N-1),N-1)	A(J)=S/(T+A(J+1))

This page left blank intentionally.

Replace Chapter 15 with the following pages.

PRECEDING PAGE BLANK NOT FILMED

A group of predefined functions is provided with the STAR FORTRAN compiler. These functions, listed and described in this section, perform the conventional manipulations such as changing the sign of a number, or frequently used mathematical computations such as logarithms and the trigonometric functions. A reference is made to one of these functions by using the function name followed by the appropriate list of arguments, as a data element in an arithmetical or logical expression. In FTN '77 certain functions may only appear in a character expression. The actual argument can be any expressions that agree in type, number, and order of arguments. Upon execution of a statement containing a reference to a predefined function, the function is executed using the values that the arguments have at the time of the reference, the function result is then made available to the expression.

The functions fall into three categories; functions when referenced:

- Cause in-line code to be generated during compilation
- Cause transfer of control to a library module during execution
- Can cause either of the above.

FTN '66 FUNCTION USAGE

When the FTN '66 option is selected the following rules apply. If the name of any function in the first category appears in an EXTERNAL specification statement, no in-line code is generated and the user must provide an entry point with that name. Any function that is to appear in an actual argument list must appear in an EXTERNAL statement in the same program unit.

The library version of a function in the third category is used if the function name appears in an EXTERNAL statement in the same program unit as the function reference; otherwise the in-line version is used. Any function in this category performs the same operations whether it is external or in-line.

FUNCTION USAGE

When the FTN '77 option is selected the following rules apply. If a function appears in an EXTERNAL statement the user must provide an entry point with that name. Any function that is to appear in an actual argument list must appear in an INTRINSIC statement in the same program unit. (FTN '66 differs from FTN '77 in function usage as specified in the appendices.)

SCALAR INTRINSIC FUNCTIONS

Scalar intrinsic functions are those intrinsic functions which produce a scalar result. The argument of these functions may be either scalar, vector, or, in some cases, a mixture of scalar and vector.

STAR FORTRAN provides a group of intrinsic functions with the prefix Q8S in their names. These functions perform more involved manipulations of data than the other scalar functions, often taking advantage of a specific STAR hardware feature. In general these functions must not appear in an INTRINSIC statement.

The scalar functions are listed in table 15-1. In this table the letter a is used for scalar arguments, the letter v for vector arguments, the letter c is used for control vectors, and the letter i for index vectors. The control vector must be of type BIT and the index vector of type INTEGER. The types of the other arguments are indicated in the table.

Scalar arguments can be general scalar expressions. Vector arguments must be arrays or dynamic variables.

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to integer	1	INT	–	Integer	Integer
	INT (a)			THINT	Half	Integer
	See Note 1			INT	Real	Integer
				IFIX	Real	Integer
				IDINT	Double	Integer
				–	Complex	Integer
	Conversion to real	1	REAL	REAL	Integer	Real
	See Note 2			FLOAT	Integer	Real
				EXTEND	Half	Real
				–	Real	Real
				SNGL	Double	Real
				–	Complex	Real
	Conversion to half precision	1	HALF	–	Integer	Half
	See Note 3			–	Half	Half
				–	Real	Half
				–	Double	Half
				–	Complex	Half
	Conversion to double	1	DBLE	–	Integer	Double
	See Note 4			–	Half	Double
				–	Real	Double
				–	Double	Double
				–	Complex	Double
	Conversion to complex	1 or 2	COMPLX	–	Integer	Complex
	See Note 5			–	Half	Complex
				–	Real	Complex
				–	Double	Complex
				–	Complex	Complex
	Conversion to integer	1		ICHAR	Character	Integer
	See Note 6					
	Conversion to character	1		CHAR	Integer	Character
	See Note 6					
Truncation	Int. (A)	1	AINT	HINT	Half	Half
	See Note 1			AINT	Real	Real
				DTNT	Double	Double

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Nearest whole numbei	Int (a+0·5) if a 0	1	ANINT	HNINT	Half	Half
	Int (a-0·5) if a 0			ANINT	Real	Real
				DNINT	Double	Double
Nearest integer	Int (a+0·5) if a 0	1	NINT	IHNINT	Half	Integer
	Int (a-0·5) if a 0			NINT	Real	Integer
				IDNINT	Double	Integer
Absolute value	/ a /	1	ABS	IABS	Integer	Integer
	See Note 7			HABS	Half	Half
	$(ar^2+ai^2)^{1/2}$			ABS	Real	Real
				DABS	Double	Double
				CABS	Complex	Real
Remaindering	a_1 - Int$(a_1/a_2)*a_2$	2	MOD	MOD	Integer	Integer
	See Note 1			HMOD	Half	Half
				AMOD	Real	Real
				DMOD	Double	Double
Transfer of sign	/ a_1 / if $a_2 \geq 0$	2	SIGN	ISIGN	Integer	Integer
	-/ a_1 / if $a_2 < 0$			HSIGN	Half	Half
				SIGN	Real	Real
				DSIGN	Double	Double
Positive Difference	$a_1 - a_2$ if $a_1 > 0$	2	DIM	IDIM	Integer	Integer
	0 if $a_1 < a_2$			HDIM	Half	Half
				DIM	Real	Real
				DDIM	Double	Double
Extended precision	$a_1 * a_2$	2		DPROD	Real	Double
				HPROD	Half	Real
Choosing largest value	Max$(a_1, a_2, \ldots)$	≥ 2	MAX	MAXO	Integer	Integer
				HMAXI	Half	Half
				AMAXI	Real	Real
				DMAXI	Double	Double
				AMAXO	Integer	Real
				MAXI	Real	Integer

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Choosing smallest value	Min$(a_1,a_2 \ldots)$	≧2	MIN	MINO HMINI AMINI DMINI	Integer Half Real Double	Integer Half Real Double
				AMINO MINI	Integer Real	Real Integer
Length	Length of a character entity See Note 12	1		LEN	Character	Integer
Index of a substring	Location of substring a_2 in substring a_1. See Note 11.	2		INDEX	Character	Integer
Imaginary part of a complex argument	ai	1		AINAG	Complex	Real
Conjugate of complex argument	(ar,-ai)	1		CONJG	Complex	Complex
Square root	$(a)^{1/2}$	1	SORT	HSORT SORT DSORT CSORT	Half Real Double Complex	Half Real Double Complex
Exponential	e**a	1	EXP	HEXP EXP DEXP CEXP	Half Real Double Complex	Half Real Double Complex
Natural logarithm	Log(a)	1	LOG	HLOG ALOG DLOG CLOG	Half Real Double Complex	Half Real Double Complex
Common logarithm	Log10(a)	1	LOG10	HLOG10 ALOG10 DLOG10	Half Real Double	Half Real Double
Sine	Sin(a)	1	SIN	HSIN SIN DSIN CSIN	Half Real Double Complex	Half Real Double Complex

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Cosine	Cos(a)	1	COS	HCOS	Half	Half
				COS	Real	Real
				DCOS	Double	Double
				CCOS	Complex	Complex
Tangent	Tan(a)	1	TAN	HTAN	Half	Half
				TAN	Real	Real
				DTAN	Double	Double
Cotangent	Cotan(a)	1	COTAN	HCOTAN	Half	Half
				COTAN	Real	Real
Arcsine	Arcsin(a)	1	ASIN	HASIN	Half	Half
				ASTN	Real	Real
				DASIN	Double	Double
Arccosine	Arccos(a)	1	ACOS	HACOS	Half	Half
				ACOS	Real	Real
				DACOS	Double	Double
Arctangent	Arctan(a)	1	ATAN	HATAN	Half	Half
				ATAN	Real	Real
				DATAN	Double	Double
	Arctan(a_1/a_2)	2	ATAN2	HATAN2	Half	Half
				ATAN2	Real	Real
				DATAN2	Double	Double
Hyperbolic sine	Sinh(a)	1	SINH	HSINH	Half	Half
				SINH	Real	Real
				DSINH	Double	Double
Hyperbolic cosine	Cosh(a)	1	COSH	HCOSH	Half	Half
				COSH	Real	Real
				DCOSH	Double	Double
Hyperbolic tangent	Tanh(a)	1	TANH	HTANH	Half	Half
				TANH	Real	Real
				DTANH	Double	Double
Lexically greater than or equal	$a_1 \geq a_2$ See Note 13.	2		LGE	Character	Logical

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Lexically greater than	$a_1 > a_2$ See Note 13.	2		LGT	Character	Logical
Lexically less than or equal	$a_1 \leq a_2$ See Note 13.	2		LLE	Character	Logical
Lexically less than	$a_1 < a_2$ See Note 13.	2		LLT	Character	Logical
Random number	Generate random number in range 0..1	0		RANF		Real
Time of day	Obtain time of day	0		TIME		Character*8
Date	Obtain the date	0		DATE		Character*8
CPU time	Obtain time in seconds since start of job	0		SECOND		Real
Insert bits	Insert bits from a_1 in a_2. See Note 14.	4	Q8SINSB	– –	Real Integer	Typeless Typeless
Extract bits	Extract bits from a. See Note 15.	3	Q8SECTB	– –	Real Integer	Typeless Typeless
Test data flag branch register	Test specified bit in data flag branch register. See Note 16.	2		Q8SDFB	Integer	Logical
Summation	Sum vector's elements. See Note 17.	1 or 2	Q8SSUM	– – –	Integer Half Real	Integer Half Real
Product	Obtain product of vector's elements. See Note 17.	1 or 2	Q8SPROD	–	Integer	Integer
Dot product	Obtain dot product of two vectors. See Note 18.	2	Q8SDOT	–	Integer	Integer
Bit count	Count number of 1 bits in bit vector	1		Q8SCNT	Bit	Integer

TABLE 15-1. SCALAR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Vector length	Obtain length of a vector or value vector	1	Q8SLEN	–	Integer	Integer
				–	Half	Integer
				–	Real	Integer
				–	Complex	Integer
Choosing largest value	Obtain maximum valued vector element. See Note 17.	1 or 2	Q8SMAX	–	Integer	Integer
				–	Half	Half
				–	Real	Real
	Count elements preceding maximum valued vector element. See Note 17.	1 or 2	Q8SMAXI	–	Half	Half
				–	Real	Real
Choosing smallest value	Obtain minimum value vector element. See Note 17.	1 or 2	Q8SMIN	–	Half	Half
				–	Real	Real
	Count elements preceding minimum valued vector element. See Note 17.	1 or 2	Q8SMINI	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Find elements	Find first pair of equal elements	2	Q8SEQ	–	Integer	Integer
				–	Half	Integer
				–	Real	Integer
	Find first pair of elements for which v_1 element $\geq v_2$ element	2	Q8SGE	–	Integer	Integer
				–	Half	Integer
				–	Real	Integer
	Find first pair of elements for which v_1 element $< v_2$ element	2	Q8SLT	–	Integer	Integer
				–	Half	Integer
				–	Real	Integer
	Find first pair of unequal elements	2	Q8SNE	–	Integer	Integer
				–	Half	Integer
				–	Real	Integer

NOTES for Table 15-1

(1) For a of type integer, int(a) = a. For a of type integer, half precision, real or double precision, there are two cases: if / a / $\cdot$ 1, int(a) = 0; if /a/ $\geq$ 1, int(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example,

int(-3.7) = -3

For a of type complex, int(a) is the value obtained by applying the above rule to the real part of a.

For a of type real, IFIX(a) is the same as INT(a).

(2) For a of type real, REAL(a) is a. For a of type integer, half or double precision, REAL(a) is as much precision of the significant part of a as a real datum can contain. For a of type complex, REAL(a) is the real part of a.

For a of type integer, FLOAT(a) is the same as REAL(a).

(3) For a of type half precision HALF(a) = a. For a of type integer, real or double precision, HALF(a) is as much precision of the significant part of a as a half precision datum can contain. For a of type complex HALF(a) is the value obtained by applying the above rule to the real part of a.

(4) For a of type double precision, DBLE(a) is a. For a of type integer, half precision or a DBLS(a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex, DBLS(a) is as much precision of the significant part of the real part of a as a double precision datum can contain.

(5) CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, half or double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, half or double precision.

For a of type complex, CMPLX(a) is a. For a of type integer, real, half or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a_1,a_2) is the complex value whose real part is REAL(a_1) and whose imaginary part is REAL(a_2).

(6) ICHAR provides a means of converting from a character to an integer, based on the position of the character in the processor collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position 255, as there are 256 characters in the collating sequence.

The value of ICHAR(a) is an integer in the range: 0 $\leq$ ICHAR(a) $\leq$ 255, where a is an argument of type character of length one. The value of a must be a character capable of representation in the processor. The position of that character in the collating sequence is the value of ICHAR.

For any characters c_1 and c_2 capable of representation in the processor, (c_1 .LE. c_2) is true if and only if (ICHAR(c_1) .LE. ICHAR(c_2)) is true, and (c_1 .EQ. c_2) is true if and only if (ICHAR(c_1) .EQ. ICHAR(c_2)) is true.

CHAR(i) returns the character in the ith position of the processor collating sequence. The value is of type character of length one. i must be an integer expression whose value must be in the range 0 $\leq$ i $\leq$ 255.

ICHAR(CHAR(i)) = i for 0 $\leq$ i $\leq$ 255.

CHAR(ICHAR(c)) = c for any character c capable of representation in the processor.

NOTES for Table 15-1 (Cont'd)

(7) A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

(8) All angles are expressed in radians.

(9) The result of a function of type complex is the principal value.

(10) All arguments in an intrinsic function reference must be of the same type.

(11) INDEX(a_1,a_2) returns an integer value indicating the starting position within the character string a_1 of a substring identical to string a_2. If a_2 occurs more than once in a_1, the starting position of the first occurrence is returned.

If a_2 does not occur in a_1, the value zero is returned. Note that zero is returned if LEN(a_1) $<$ LEN(a_2).

(12) The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(13) LGE(a_1,a_2) returns the value true if $a_1 = a_2$ or if a_1 follows a_2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LGT(a_1,a_2) returns the value true if a_1 follows a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLE(a_1,a_2) returns the value true if $a_1 = a_2$ or if a_1 precedes a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLT(a_1,a_2) returns the value true if a_1 precedes a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

(14) The argument list for this function is (a_1,m,n,a_2) where a_1 and a_2 may be of type integer or real. Arguments m and n must be of type integer. The result of the function is typeless.

(15) The argument list for this function is (a,m,n) where a may be of type integer or real. Arguments m and n must be of type integer. The result of the function is typeless.

(16) The data flag branch manager is described in Chapter 14.

(17) The argument list for this function is (v) or (v,c) where v is a vector of type integer, half precision or real, and c is vector of type bit.

(18) The argument list for this function is (v_1,v_2).

VECTOR INTRINSIC FUNCTIONS

Vector intrinsic functions are those intrinsic functions which produce a vector result. The arguments of these functions may be either scalar, vector or, in some cases, a mixture of scalar and vector.

Many of the vector intrinsic functions are the vector equivalent of a scalar function. These functions have names beginning with the letter v. The arguments of these functions are vectors and are equivalent to the application of the scalar function to each element in the vector. For example, the following are equivalent:

```
        DO 1 I = 1,64
1       M(I) = INT(A(I))

        M(1:64) = VINT(A(1:64))
```

STAR FORTRAN also provides a group of intrinsic functions whose names start with Q8V. These functions perform more involved manipulations of data than the other vector functions, often taking advantage of a specific STAR hardware feature. In general these functions must not appear in an INTRINSIC statement.

The vector functions are listed in Table 15-2. In this table the letter a is used for scalar arguments, the letter v for vector arguments, the letter c for control order vectors, the letter i for index vectors, and the letter u for results vectors. The result vector must always be specified, must always be the last argument and is separated from the other arguments by a semicolon.

Scalar arguments can be general scalar expressions, vector arguments must be arrays or dynamic variables.

TABLE 15-2. VECTOR INTRINSIC FUNCTIONS

Intrinsic Function	Definition (See Note 1)	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to integer	(v)	VINT	VIHINT	Half	Integer
				VINT	Real	Integer
				VIFIX	Real	Integer
	Conversion to real	(v)		VFLOAT	Integer	Real
				VEXTEND	Half	Real
				VSNGL	Double	Real
				VREAL	Complex	Real
	Conversion to half precision	(v)	VHALF	–	Integer	Half
				–	Real	Half
				–	Double	Half
				–	Complex	Half
	Conversion to complex	(v)		VCMPLX	Real	Complex
Truncation	Int(a)	(v)		VAINT	Real	Real
				VHINT	Half	Half
Nearest whole number	Int(a+0·5) if $a \geq 0$ Int(a-0·5) if $a < 0$	(v)		VANINT	Real	Real
				VHNINT	Half	Half
Nearest integer	Int(a+0·5) if $a \geq 0$ Int(a-0·5) if $a < 0$	(v)		VNINT	Real	Integer
				VIHNINT	Half	Integer
Absolute value	/ a /	(v)		VIABS	Integer	Integer
	$(ar^2 + ai^2)^{1/2}$			VHABS	Half	Half
				VABS	Real	Real
				VCABS	Complex	Real
Remaindering	a_1 - Int(a_1/a_2)*a_2	(v)		VMOD	Integer	Integer
				VHMOD	Half	Half
				VAMOD	Real	Real
Transfer of sign	/ a / if $a_2 \geq 0$ $-/a_1/$ if $a_2 < 0$	(v_1,v_2)		VISIGN	Integer	Integer
				VHSIGN	Half	Half
				VSIGN	Real	Real
Positive difference	$a_1 - a_2$ if $a_1 > a_2$ 0 if $a_1 \leq a_2$	(v_1,v_2)		VIDIM	Integer	Integer
				VHDIM	Half	Half
				VDIM	Real	Real

TABLE 15-2. VECTOR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition (See Note 1)	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Imaginary part of complex argument	ai	(v)		VAINAG	Complex	Real
Conjugate of a complex argument	(ar,-ai)	(v)		VCONJG	Complex	Complex
Square root	$(a)^{1/2}$	(v)		VHSORT VSORT VCSORT	Half Real Complex	Half Real Complex
Exponential	e**a	(v)		VHEXP VEXP VCEXP	Half Real Complex	Half Real Complex
Natural logarithm	Log(a)	(v)		VHLOG VALOG VCLOG	Half Real Complex	Half Real Complex
Common logarithm	Log10(a)	(v)		VHLOG10 VALOG10	Half Real	Half Real
Sine	Sin(a)	(v)		VHSIN VSIN VCSIN	Half Real Complex	Half Real Complex
Cosine	Cos(a)	(v)		VHCOS VCOS VCCOS	Half Real Complex	Half Real Complex
Tangent	Tan(a)	(v)		VHTAN VTAN	Half Real	Half Real
Arcsine	Arcsin(a)	(v)		VHASIN VASIN	Half Real	Half Real
Arccosine	Arccos(a)	(v)		VHACOS VACOS	Half Real	Half Real
Arctangent	Arctan(a)	(v)		VHATAN VATAN	Half Real	Half Real
	$\text{Arctan}(a_1/a_2)$	(v)		VHATAN2 VATAN2	Half Real	Half Real

TABLE 15-2. VECTOR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition (See Note 1)	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Find order of elements. See Note 2.	Find order of equal elements	(v_1,v_2)	Q8VEQI	–	Half	Integer
				–	Real	Integer
	Find order of greater than or equal elements	(v_1,v_2)	Q8VGEI	–	Half	Integer
				–	Real	Integer
	Find order of less than elements	(v_1,v_2)	Q8VLTI	–	Half	Integer
				–	Real	Integer
	Find order of unequal elements	(v_1,v_2)	Q8VNEI	–	Half	Integer
				–	Real	Integer
Mask vectors	Mask values in two vectors into result vector	(v_1,v_2,c)	Q8VMASK	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Merge vectors	Merge values in two vectors into result vector	(v_1,v_2,c)	Q8VMERG	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Compress vector	Delete selected elements from vector	(v,c)	Q8VCMPRS	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Expand vector	Insert zero valued elements in vector	(v,c)	Q8VXPND	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Contract vector	Select elements for result vector	(v,i)	Q8VGATHR	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Scatter vector	Scatter elements into result vector	(v,i)	Q8VSCATR	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Store selected elements	Store selected elements in result vector	(v,c)	Q8VCTRL	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Delete elements	Delete elements below threshold in sparse vector	(v_1,v_2)	Q8VARCMP	–	Integer	Integer
				–	Half	Half
				–	Real	Real

TABLE 15-2. VECTOR INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition (See Note 1)	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Reverse vector	Reverse order of elements in vector	(v)	Q8VREV	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Create an arithmetic progression	Create a vector whose elements form an arithmetic progression	(a_1,a_2) See Note 3	Q8VINTL	–	Integer	Integer
				–	Half	Half
				–	Real	Real
Compute polynomial	Compute a polynomial at several values	(v_1,v_2)	Q8VPOLY	–	Half	Half
				–	Real	Real
Compute differences	Computer differences between adjacent elements of vector	(v)	Q8VDELT	–	Half	Half
				–	Real	Real
Create a bit pattern	First group of bits are one	(a_1,a_2) See Note 3	Q8VMKO	–	Integer	Bit
	First group of bits are zero	(a_1,a_2) See Note 3	Q8VMKZ	–	Integer	Bit
Compute averages	Compute average of adjacent elements	(v)	Q8VADJM	–	Half	Half
				–	Real	Real
	Compute average of corresponding elements	(v_1,v_2)	Q8VAVG	–	Half	Half
				–	Real	Real
	Compute average difference of corresponding elements	(v_1,v_2)	Q8VAVGD	–	Half	Half
				–	Real	Real

NOTES for Table 15-2

(1) The entity a is an element of the integer, half precision, or real vector v. ar and ai are the real and imaginary parts respectively of a complex vector.

(2) Equivalent to issuing a series of Q8S<rel op > calls in which one of the arguments is a scalar equal to an element of one of the argument vectors.

(3) The arguments a_1 and a_2 are of type integer and the result is of type bit.

Function Descriptions

The following descriptions are listed in alphabetical order.

The values of some of the mathematical functions can be infinite.

The type of the result of a generic function is either predefined or depends on the type of its arguments. For example LOG(a) returns a result with the same type as a, but REAL(a) always has a real result regardless of the type of a.

A generic function name may not be passed as an actual argument, unless it corresponds to a specific function name.

ABS(a)

This function computes the absolute value of the specified argument. Its arguments may be of type integer, half precision, real, double precision or complex. It is the specific function name for obtaining the absolute value of a real argument. For a real argument x, ABS(x) computes / x /. The other specific functions which compute absolute values are CABS, DABS, HABS, and IABS.

ACOS(a)

This function computes the arccosine of a half precision, real, or double precision argument. It is the specific function name for compiling the arccosine of a real argument. The other specific functions which compute arccosines are DACOS, and HACOS. See ASIN for a description of the ACOS function.

AIMAG(a)

This returns the imaginary part of a complex number as a real number; if x+iy is the complex number, AIMAG returns y.

AINT(a)

This function computes [a], where [a] is the sign of a times the largest integer less than or equal to / a /. The type of a may be half precision, real, or double precision. It is the specific function name for truncating a real argument. The other specific function names which truncate the argument are HINT and DINT.

ALOG(a)

This computes the natural logarithm of a real number greater than zero. The result is a real number accurate to approximately 45 bits.

ALOG10(a)

This computes the logarithm of a real number. The result is a real number that is accurate to approximately 45 bits.

AMAX0(a_1,a_2,...)

This searches a list of integer numbers for the list element having the maximum value. The integer found is returned as a real number.

AMAX1(a_1,a_2,...)

This searches a list of real numbers for the list element having the maximum value and returns that value.

AMIN0(a_1,a_2,...)

This searches a list of integer numbers for the list element having the minimum value. The integer found is returned as a real number.

AMIN1(a_1,a_2,...)

This searches a list of real numbers for the list element having the minimum value and returns the number when found.

AMOD(a_1,a_2,...)

This computes one real number modulo a second real number and produces a real result. AMOD(x,y) is defined as x-[x/y]*y, where [A] is the sign of A times the largest integer less than or equal to /A/.

ANINT(a)

This function computes the nearest whole number to the specified half precision, real, or double precision argument. It is the specific function name for obtaining the nearest whole number to a real argument. The other specific function names which compute the nearest whole number are DNINT and HNINT.

Example:

```
    Given a = -3.5D+00
    ANINT(a) = AINT(-3.5D+00 - 5.0D - 01)
             = - 4.0D+00
```

ASIN(a)

This function computes the arcsine of a half precision, real, or double precision argument. It is the specific function name for computing the arcsine of a real argument. The other specific functions which compute arcsines are DASIN and HASIN.

The specific functions ASIN and ACOS compute the arcsine and the arccosine of a real number having an absolute value less than or equal to 1.0. The result is a real number expressed in radians, and is accurate to approximately 45 bits. The range of the result for ASIN is -pi/2 through pi/2, inclusive; and the range of the result for ACOS is 0 through pi, inclusive.

ATAN(a)

This function computes the arctangent of a half precision, real, or double precision argument. It is the specific function name for computing the arctangent of a real argument. The other specific function names for computing arctangents are DATAN and HATAN.

The specific function ATAN computes the arctangent of a real number. The real result is accurate to approximately 45 bits, and is in the range -pi/2 through pi/2 (not inclusive).

ATAN2(a_1,a_2)

This function computes the arctangent of the ratio of two half precision, real, or double precision arguments. It is the specific function name for computing the arctangent of the ratio of two real arguments. The other specific functions for computing the arctangent of a ratio are DATAN2 and HATAN2.

The specific function ATAN2 computes the arctangent of the ratio of two real numbers. The real result, expressed in radians, is accurate to approximately 45 bits and is in the range -pi through pi.

CABS(a)

This computes the modulus of a complex number, and produces a real result that is greater than or equal to zero which is accurate to approximately 45 bits.

CCOS(a)

This computes the cosine of a complex number. The result is a complex number whose real and imaginary parts are each accurate to approximately 45 bits.

CEXP(a)

This computes the exponential of a complex number. The result is a complex number that is accurate to approximately 45 bits.

CHAR(i)

This function returns the character in the ith position of the ASCII 256 character set. For example, CHAR(65) returns the character A and is equivalent to CHAR(X'41').

CLOG(a)

This computes the natural logarithm of any complex number except 0. + i0.. The result is a complex number that is accurate to approximately 45 bits.

CMPLX(a) or COMPLX(a_1,a_2)

This function constructs a complex number from one or two integer, half precision, real, double or complex arguments. When two arguments are given they must be of the same type.

For a of type complex, CMPLX(a) is a. For a of type integer, half, real or double precision CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a_1,a_2) is the complex value whose real part is REAL(a_1) and whose imaginary part is REAL(a_2)

There are no specific function names for constructing a complex number.

CONJG(a)

This computes the conjugate of a complex number. If the complex number is x+iy, the conjugate is x-iy; the real part, x, of the complex number is assigned to a real part of the result, and the imaginary part, y, of the complex number is negated and assigned to the imaginary part of the result.

COS(a)

This function computes the cosine of a half precision, real, double precision, or complex argument expressed in radians. It is the specific function name for computing the cosine of a real argument. The other specific functions which compute cosines are CCOS, DCOS, and HCOS. See SIN for a description of the COS function.

COSH(a)

This function computes the hyperbolic cosine for half precision, real, or double precision argument. It is the specific function name for computing the hyperbolic cosine of a real argument. The other specific functions which compute hyperbolic cosines are DCOSH and HCOSH.

The function COSH computes the hyperbolic cosine of a real number and produces a real result that is greater than or equal to 1.0 and accurate to 47 bits.

COTAN(a)

This function computes the cotangent of a half precision or real argument expressed in radians. It is the specific function name for computing the cotangent of a real argument. The other specific function for computing cotangent is HCOTAN.

The function COTAN computes the cotangent of a real number expressed in radians. The function first reduces its argument modulo 2 pi. The result is a real number that is accurate to approximately 45 bits.

CSIN(a)

This computes the sine of a complex number. The result is a complex number accurate to approximately 45 bits.

CSQRT(a)

This computes the square root of a complex number in which the real part is greater than or equal to zero, and returns a complex result that is accurate to approximately 45 bits. Whenever a result is returned in which the real part is zero, the imaginary part is greater than or equal to zero.

DABS(a)

For a double precision number x, DABS(x) computes the absolute value /x/.

DACOS(a)

See DASIN for a description of the DACOS function.

DASIN(a) and DACOS(a)

These compute the arcsine and arccosine of a double precision number having an absolute value less than or equal to 1.0. The double precision result, expressed in radians, is accurate to 94 bits.

DATAN(a) and DATAN2(a,b)

These compute the arctangent of the ratio of two double precision numbers. If the denominator is 1.0, it need not be specified (DATAN is used). The double precision result, expressed in radians, is accurate to approximately 90 bits.

DATAN2(a,b)

See DATAN for a description of the DATAN2 function.

DATE( )

This function returns the date in CHARACTER*8 format. Note that the function has no argument.

DBLE(a)

For a of type double precision DBLE(a) = a. For a of type integer, half precision, or real, this function produces a double precision result equal to a for a type complex DBLE(a) = DBLE(REAL(a)). There are no specific functions for forming a double precision result.

DCOS(a)

See DSIN for a description of the DCOS routine.

DCOSH(a)

This computes the hyperbolic cosine of a double precision number and produces a double precision result that is accurate to 94 bits.

DDIM(a_1,a_2)

This computes the positive excess of one double precision number over another double precision number. DDIM(x,y) returns the value x-y if x is greater than or equal to y, and returns a double precision value of 0.0 otherwise. The function value is accurate to 94 bits.

DEXP(a)

This computes the exponential of a double precision number. The result is double precision and is accurate to approximately 90 bits.

DIM(a_1,a_2)

This function computes the positive excess of a_1 over a_2. a_1 and a_2 must be of the same type and may be integer, half precision, real or double precision. It is the specific function name for computing the positive excess of one real number over another. The other specific functions which compute the positive excess are DDIM, HDIM, and IDIM. DIM(a_1,a_2) is equal to $a_1 - a_2$ if $a_1 > a_2$ and 0 otherwise.

DINT(a)

For a double precision number x, DINT(x) computes [x], where [A] is the sign of A times the largest integer less than or equal to /A/. DINT returns a double precision result even though its value is always integral.

DLOG(a)

This computes the natural logarithm of a double precision number. The result is a double precision number that is accurate to approximately 90 bits.

DLOG10(a)

This computes the logarithm of a double precision number. The result is a double precision number that is accurate to approximately 90 bits.

DMAX1(a_1,a_2,...)

This searches a list of double precision numbers for the list element having the maximum value and returns that value.

DMIN1(a_1,a_2,...)

This searches a list of double precision numbers for the list element having the minimum value and returns the number when found.

DMOD(a_1,a_2)

This computes one double precision number modulo a second double precision number and calculates a double precision result. Valid arguments for DMOD lie in the interval -0.476 854 057 715 93E + 8645$\leq$x $\leq$+ 0.476 854 057 715 93E + 8645 (the largest allowable argument value is half of the largest allowable real number).

DNINT(a)

This function computes the nearest whole number to a, both the argument and result are of type double precision. Note that for a double precision argument a DNINT(a) = ANINT(a).

FEASIBILITY STUDY FOR A NUMERICAL AERODYNAMIC SIMULATION FACILITY

Volume III — FMP Language Specification/User Manual

Contributions by: B. G. Kenner
N. R. Lincoln

Prepared under Contract No. NAS2-9896

CONTROL DATA CORPORATION
Research and Advanced Design Laboratory
4290 Fernwood Street
St. Paul, Minnesota 55112

for

AMES RESEARCH CENTER

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

FMP FORTRAN

This manual is intended to show the revisions and additions that Control Data proposes to make to the current STAR FORTRAN. All references to STAR FORTRAN in the manual should be considered to be a reference to FMP FORTRAN.

The revisions have generally been to show where FMP FORTRAN will be different than the currently existing FORTRAN. Places where a change is to occur in the original manual are marked and the facing page (and following pages, if necessary) of this manual shows the expected change.

This manual is a preliminary and is subject to further changes.

CONTENTS

STAR FORTRAN '77

External Reference Specification

PREFACE

This document is the STAR FORTRAN '77 External Reference Specification. It comprises Chapters 1-11 and 14-16 and Appendices D, F, and G of Revision G of the STAR FORTRAN Reference Manual, modified to reflect language changes which will be made as part of the implementation of FORTRAN '77 on STAR.

This document takes the form of the original reference manual material, plus change pages. The changes have not been integrated in order that the differences between current STAR FORTRAN and FORTRAN '77 will be clearer.

The usual format will be the reference manual page on the reader's left, followed by the changed passages which appear on the reader's right. The changes and additions are page numbered with an A following the page number.

INTRODUCTION

A The FORTRAN programming language for the STAR-100 computer contains both CDC and unique STAR extensions to the standard FORTRAN (as defined by American National Standards X3.9-1966, FORTRAN). Throughout this manual, shading is used to distinguish these extensions from the standard FORTRAN language features.

Several of the STAR FORTRAN extensions to standard FORTRAN allow the FORTRAN user to exploit the vector processing capabilities of the STAR computer. In STAR FORTRAN, vectors can be expressed with an explicit notation, functions are provided that return vector results, and special call statements enable access to any machine instruction.

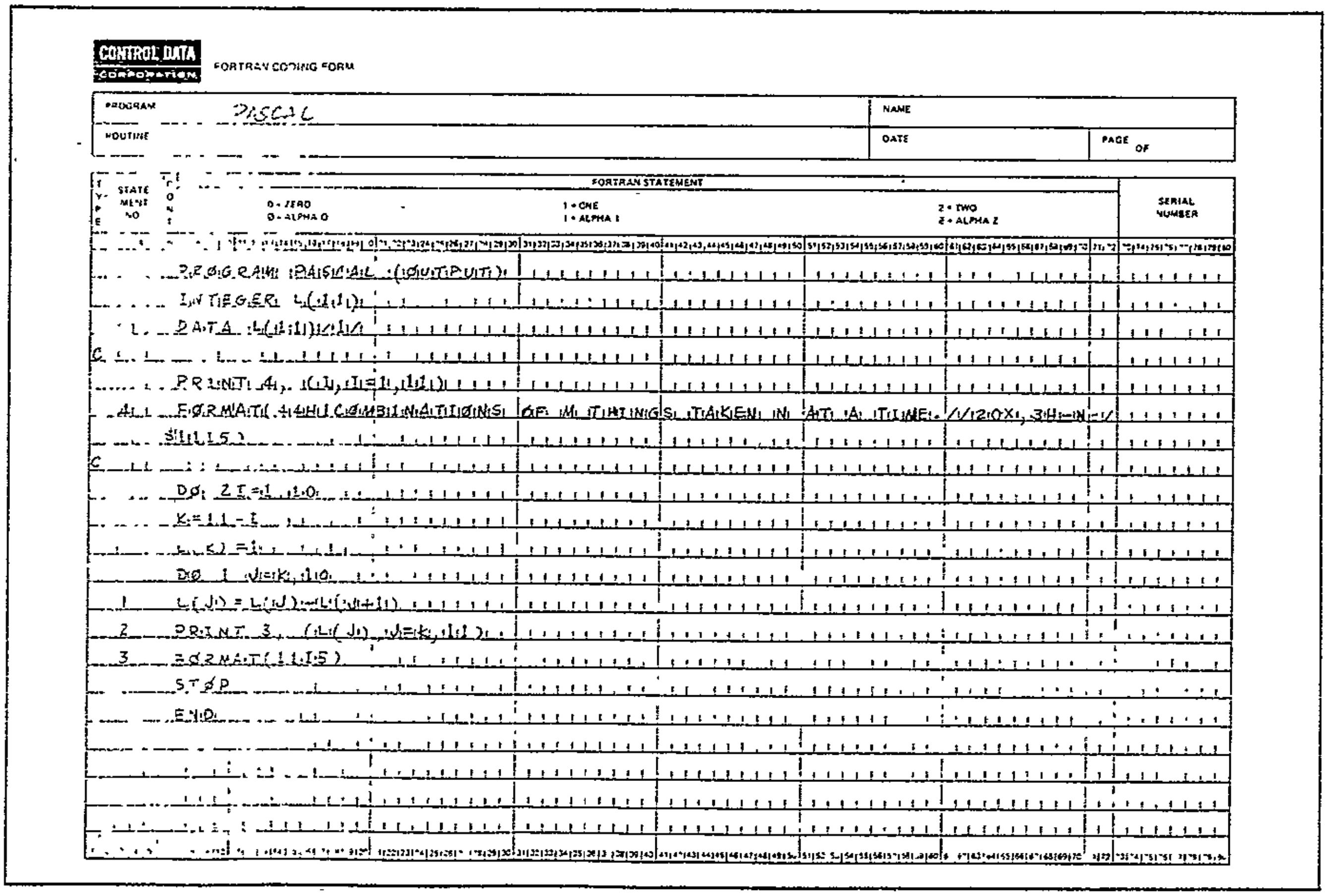
PROGRAM FORM

B A FORTRAN program consists of one or more separately defined program units. A program unit, which is either a main program or a subprogram, consists of a series of source lines that contain statements, optional comment lines, and one and only one END line. An executable FORTRAN program must contain one main program; it can also contain any number of subprograms.

If the executable program consisting of source lines aggregated as program units is accepted by the STAR FORTRAN compiler, the program is changed into a form that can be loaded and executed by the STAR operating system. The compiler executes in response to the FORTRAN system control statement. Once the program has been compiled, it can be loaded and executed in response to further system control statements.

Execution of the compiled program proceeds with one program unit having control until it relinquishes it to another program unit or stops. Values can be passed at the time that control is passed from one program unit to another. During execution, the compiled program can make use of execution-time routines that are part of the system library. Files referenced in the program are read and written by STAR Record Manager. Depending on the source program statements, other system-defined or compiler-defined procedures such as conditional interrupt routines and error processing routines might also be invoked during execution.

An example of a complete STAR FORTRAN program is provided in figure 1-1.



CONTROL DATA CORPORATION FORTRAN CODING FORM

PROGRAM PASCAL NAME

ROUTINE DATE PAGE OF

FORTRAN STATEMENT

```
      PRØGRAM PASCAL (ØUTPUT)
      INTEGER L(11)
      DATA L(1)/1/
C
      PRINT 4, (I, I=1,11)
    4 FØRMAT(44H1CØMBINATIØNS ØF N THINGS TAKEN N AT A TIME./1/20X,3HN=/
     $11I5)
C
      DØ 2 I=1,10
      K=11-I
      L(K)=1
      DØ 1 J=K,10
    1 L(J)=L(J)+L(J+1)
    2 PRINT 3, (L(J),J=K,11)
    3 FØRMAT(11I5)
      STØP
      END
```

Figure 1-1. Sample Coded FORTRAN Program

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

A: END STATEMENT

The END statement must be the last statement of each main program or subprogram. If executed in a main program, the END statement acts like

```
STOP 'END'
```

If executed in a subroutine or function subprogram, END acts like

```
RETURN
```

SPECIFICATION AND DATA INITIALIZATION STATEMENTS

Specification statements are nonexecutable statements whose purpose is to define storage requirements of variables, arrays, and function results. They define the type of a symbolic name, specify the dimensions of an array, stipulate the length of a character variable, and define how storage is to be shared.

If specification statements are used, they must appear before the first executable statement of the program unit in which they occur. Any program that refers to an array must have at least one specification statement. Otherwise, specification statements may or may not be required.

The nonexecutable data initialization statement is also described in this section.

TYPE STATEMENTS

A{ Variables, arrays, and function names that appear in a STAR FORTRAN program must each be associated with a data type. Explicit type statements and implicit typing are the two ways to make this association.

B{ The appearance of a symbolic name in a type statement informs the compiler that the name is of the specified data type in the program unit. C{ In the absence of a type statement, the type of a symbolic name is implied by the first letter of the name; unless IMPLICIT statements alter the correspondences of first letters to data types, the letters I, J, K, L, M and N imply type integer and all other letters imply type real. (This default type association is referred to as the first-letter rule.)

D{ The predefined FORTRAN function names possess predetermined data types. Implicit typing of any of these names has no effect. If the name of a FORTRAN-supplied function is explicitly associated with a type other than its predefined type, the name ceases to reference the FORTRAN-supplied function.

IMPLICIT STATEMENT

The IMPLICIT statement alters the default correspondences between first letters and data types for symbolic names. The statement can also specify length for type character. E{ IMPLICIT statements must precede all other specification statements.

Form:

IMPLICIT $typ_1(list_1), \dots, typ_m(list_m)$

typ_i	The name of a data type: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, or CHARACTER. The character variable names are assumed to be of length one unless the word CHARACTER is followed by *n, where n is an integer constant that specifies the character variable length in bytes.
$list_i$	A list of the form: $v_1, v_2, \dots, v_n$ where v_i is a range of first letters of variables to be considered of type typ. v_i is either a single alphabetic character, or two such characters separated by a minus sign to denote the first and last characters of a range. The second character in a range specification must follow the first in alphabetic sequence.

A character must not be associated with more than one data type or byte length by IMPLICIT statements. }F

An IMPLICIT statement in a function or subroutine subprogram affects the data type associated with dummy arguments and the function name, as well as with other variables in the subprogram.

Explicit typing of a variable, array, or function name in an explicit type statement or FUNCTION statement overrides any implicit type specification.

EXPLICIT TYPING

An explicit type statement is used to declare one or more entities to be of the specified data type. It overrides or confirms any implicit typing and can supply dimension and byte length information.

Forms:

INTEGER $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

REAL $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

DOUBLE PRECISION $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

COMPLEX $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

LOGICAL $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

CHARACTER $*Kv_1*k_1/d_1/, v_2*k_2/d_2/, \dots, v_n*k_n/d_n/$ }G

BIT $v_1/d_1/, v_2/d_2/, \dots, v_n/d_n/$

v_i	A variable, array, array declarator, or function name. }K
d_i	Optional. Represents the initial value for v_i. If omitted, the surrounding slashes must also be omitted. }J (Rules for initializing within a type statement are given under the heading DATA Statement later in this section.)
*K	Optional. An integer constant specifying the element length in bytes of every v. This specification is overridden by the individual *k length specifications. }H If *K is omitted, a length of one byte is implied for every v not accompanied by a *k. }I

A: There are two nearly identical FORTRAN languages for the Control Data STAR-100 and CYBER 200 series computers. One, FTN66, is based on American National Standard X3.9-1966 FORTRAN and includes that language as a subset. The other, FTN77, is based on, and includes, ANS X3.9-1978 FORTRAN.

FTN66 is composed of ANS X3.9-1966 FORTRAN together with three kinds of extensions:

1. Those which are common in FORTRAN dialects used on other Control Data computers,

2. Those designed to provide access to the vector processing capabilities of the CDC STAR-100 and CYBER 200 series computers, and

3. Those derived from ANS X3.9-1978. FTN66 includes all the features of the 1978 standard except those which are incompatible with the 1966 standard.

FTN77 is composed of ANS X3.9-1978 FORTRAN together with the first two kinds of extensions mentioned above.

This manual describes FTN77 plus extensions for FMP FORTRAN

B: . . . one and only one END statement. An executable FORTRAN . . .

A statement is written as one or more source lines, and a comment, as one source line. The first line of a statement is called an initial line and the succeeding ones are called continuation lines. Each line is a string of any characters in the 64-character ASCII subset listed in appendix A. The character positions in a line are called columns and are consecutively numbered left to right.

A FORTRAN program can be written on a coding form such as the one illustrated in figure 1-1. Each line on the coding form represents a source line that can be either keypunched on a card or typed in at a terminal. No more than one statement is permitted on a single line. The conventional significance of each column of a source line is shown in table 1-1.

TABLE 1-1. COLUMN CONVENTIONS

Columns	Significance
1	The letter C indicates that this is a comment line, and that the remainder of the line is to be ignored by the FORTRAN compiler.
1 thru 5	One to five numeric characters in this field are interpreted as a statement label.
6	Any ASCII character other than a blank or zero indicates that this is a continuation line.
7 thru 72	STAR FORTRAN statement, with blank characters ignored except in character and Hollerith constants, can appear anywhere within this field.
73 thru end of source line	Identification field, the contents of which are always ignored by the FORTRAN compiler, can contain any characters.

A

END LINES

B An END line indicates to the FORTRAN compiler the end of a program unit. Every program ~~unit~~ must have an END line as its last line.

Form:

```
    END
```

Program units are described in section 7.

COMMENTS

Comment lines are used for purposes of in-line documentation. They are not statements. Except for being printed in the output file, comment lines have no effect. The letter C in column 1 of a line indicates that this is a comment line; the comments themselves can be written anywhere after column 1. If a comment requires more than one line, each line must have a C in column 1. C

STATEMENTS

The statements in the STAR FORTRAN language fall into two classes: executable and nonexecutable (see table 1-2). In general, a FORTRAN program unit consists of nonexecutable statements followed by executable statements; however, there are a few significant exceptions to this separation.

TABLE 1-2. TYPES OF STATEMENTS

Executable	Nonexecutable
Input statements (section 8)	Procedure definition statements (sections 7 and 11)
Assignment statements (sections 4, 10, and 11)	Specification statements (sections 6 and 11)
Flow control statements (section 5)	Data initialization statements (sections 6 and 11)
Output statements (section 8)	FORMAT statements (section 9)
	NAMELIST statements (section 8)

Executable statements specify actions to be taken during program execution. Executable statements are used typically in the course of a program to request that data be input, that data be operated upon and stored, and subsequently that results are to be output.

Nonexecutable statements describe characteristics, arrangement, and format of data, as well as entry points and file requirements of the program. The first statement in a main program is, generally, the nonexecutable PROGRAM statement. A nonexecutable statement (such as a FORMAT or DATA statement) that appears in the executable portion of a program is processed once by the compiler and does not affect the flow of execution.

Statement Labels

Within a program unit, a statement label – any one- to five-digit integer – uniquely identifies a statement so that it can be identified by another statement. Labels on statement continuation lines are ignored, as are blanks and leading zeros in a label. Statements that are not referred to by other statements need not be labeled. Labels need not occur in numerical order. A statement label can be referred to as frequently as necessary, but it must not be used more than once in the same program unit to label a statement. Also, no statement can refer to the label of a statement that is contained in another program unit.

Continuation of Statements

If a statement is longer than 66 columns, it can be continued on as many as 19 continuation lines. Unless a line is a comment line, a character other than blank or zero in column 6 indicates a continuation line. Columns 2 through 5 can contain any characters in the FORTRAN character set (they are ignored), and column 1 can contain any character in the set except C. A continuation line can follow only another continuation line or the initial line of a statement. D

Ordering of Statements

The following table shows the general form of a FORTRAN program unit. Statements within a group can appear in any order (with one exception), but groups (indicated by 1, 2, . . . , 6) must be ordered as shown in figure 1-2. E Comment lines can appear anywhere within the program before the END line, except before statement continuation lines. F

A: 1 The letter C or an asterisk indicates that this is a comment line, and that the remainder of the line is ignored by the FORTRAN compiler. (In FTN66, an asterisk in column 1 does not indicate a comment line.)

B: END STATEMENT

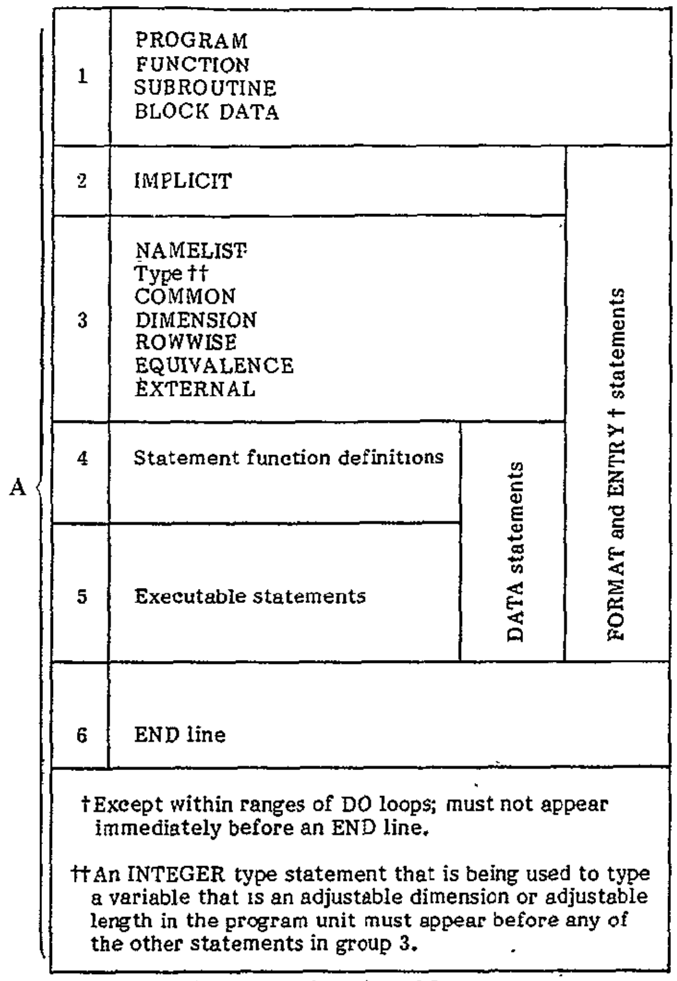
An END statement indicates the end of a program unit to the FORTRAN compiler. Every program unit must have an END statement as its last line. The END statement may be labeled but must not be continued.

C: . . . the output file, comment lines have no effect. Any line with the letter C or an asterisk in column 1 is a comment line; a blank line is also a comment line. (In FTN66, any line with the letter C in column 1 is a comment line; any blank line is also a comment line. A line with an asterisk in column 1 is not a comment line.)

D: . . . in the set except C or asterisk. (In FTN66, a continuation line may have an asterisk in column 1.) A continuation line may follow an initial line, a continuation line, or a comment line which follows an initial line or a continuation line.

E: . . . order (with one exception), but groups must be ordered as shown in Figure 1-2.

F: Comment lines may appear anywhere at all within the program unit, including before its first non-comment line.



†Except within ranges of DO loops; must not appear immediately before an END line.

††An INTEGER type statement that is being used to type a variable that is an adjustable dimension or adjustable length in the program unit must appear before any of the other statements in group 3.

Figure 1-2. Ordering of Statements

COLUMNS 73 THROUGH END OF SOURCE LINE

Any information can appear in any columns that follow column 72. The characters in these columns are copied to the output file but have no other effect. These columns might be used, for example, to order the cards in a punched deck.

PROGRAM DATA

No restrictions other than those implied in sections 8 and 9 are imposed on the format of data input to the program. Input data can appear in any of the columns of an input line and use as many input lines as required. Except on initiation of a read, or interpretation of a slash separator in the FORMAT statement associated with a READ statement, the input line boundary is not significant. Input data is not part of the source program record.

Comment Lines	PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA		
	FORMAT, ENTRY[1]	PARAMETER[2]	IMPLICIT[2]
			Other[2,3] Specification Statements
		DATA	Statement Function Definitions
			Executable Statements
END			

1. May not appear within block IFs or DO-loops

2. If the type of a constant is defined by an IMPLICIT or type statement, the IMPLICIT or type statement must precede the PARAMETER statement which defines its value.

3. If the type of integer variable used in a dimension bound expression is defined by an IMPLICIT or type statement, the IMPLICIT or type statement must precede the statement which contains the array declarator in which the variable is referenced.

STATEMENT ELEMENTS 2

The elements of a syntactically correct STAR FORTRAN statement could include any of the following:

Identifiers

Keywords

Special characters

An identifier is a name or a number. For example, a number (the statement label) is used for identifying a statement. Input and output units are also numbered. Names are used to identify data elements–such as variables and arrays–and for identifying procedures and blocks. A symbolic name consists of alphanumeric characters, the first of which must be alphabetic. STAR FORTRAN allows a symbolic name to have a length of eight characters.

In the appropriate contexts, keywords and some of the special characters (the plus sign, for example) mean that specific actions are to be taken with respect to the identified data. Other special characters (the comma, for example) serve to punctuate statements. FORTRAN does not contain reserved words, which means that a keyword out of the appropriate context is interpreted to be an identifier.

CHARACTER SET

Except for character and Hollerith constants, and character and Hollerith editing specifications in FORMAT statements, STAR FORTRAN statements are written with the 52 characters listed in table 2-1. Character and Hollerith constants and editing specifications can contain any of the 64 characters in the ASCII subset that is given in appendix A.

TABLE 2-1. FORTRAN CHARACTER SET

Character Class	Characters
Alphabetic	Letters A thru Z
Numeric	Digits 0 thru 9
Special	Blank
	= Equals sign
	+ Plus sign
	- Minus sign or hyphen
	* Multiply sign or asterisk
	/ Divide sign or slash
	(Left parenthesis
	) Right parenthesis
	, Comma
	. Decimal point or period
	& Ampersand
	' Apostrophe or single quote
	: Colon
	; Semicolon
	] Right bracket
	[Left bracket

Other than within character and Hollerith constants and in editing-specifications, the blank character is not significant within FORTRAN statements. Consequently, the user can insert blanks within a statement, even within identifiers and numeric constants, to make the program readable. The symbol ƀ is used in this manual to denote a blank character that is not optional.

DATA ELEMENTS

Data can be represented in a STAR FORTRAN program as constants, variables, and arrays.

CONSTANTS

A constant is a quantity identified by its value. The value of a constant cannot be changed at any time during execution of a program. } A

A constant has one of nine data types:

Integer

Real

Double precision

Complex

Logical

Hollerith

Character

Hexadecimal

Bit

} B

Each type of constant has its own source program form and computer internal representation. For example, if the constant 1061 appears in a source program, it represents the decimal value 1061 and has the data type integer. The full word the number occupies in memory has the 64-bit binary representation 0... 010000100101.

VARIABLES

A variable is a quantity whose value can be changed during program execution. A variable is identified by a symbolic name. A variable name is generally associated by the FORTRAN compiler with a storage location; whenever the variable is referenced in a source program, the value currently in that location is accessed.

A variable can be a simple (that is, scalar) variable, a descriptor, or a double descriptor. Descriptors and double descriptors are discussed in the vector programming section. } C

Some of the ways that the value of a variable can be changed during program execution are:

Executing an assignment statement in which the variable name occurs to the left of an equals sign

Executing an ASSIGN statement

A: A constant is a quantity identified by its value or by a symbolic name. Constants which have symbolic names are those defined in PARAMETER statements. Named constants must be defined before use and must not be redefined. A constant name has an associated type; if it is other than the default implied type, the name must be typed by an IMPLICIT or type statement before the value is assigned in a PARAMETER statement.

The value of . . .

B: A constant has one of ten data types. Named constants are restricted to the first seven types; that is, there are no named Hollerith, hexadecimal, or bit constants. The constant types are:

Integer
Real
Double Precision
Half Precision
Complex
Logical
Character
Hexadecimal
Bit
Hollerith

C: Deleted

Reading a new value into it

Using it as an argument to a subprogram that changes the argument value

Changing the value of a variable to which it has been equivalenced

The data type of a variable name is determined implicitly by the name's first letter (this is referred to as the first-letter rule) unless the name is explicitly typed by an explicit type statement. The correspondence of first letters to types is as follows, except as altered by IMPLICIT statements:

Letters	Data Type
A through H, and O through Z	Real
I through N	Integer

ARRAYS

An array is a totally ordered set of variably valued elements identified by a single symbolic name. A single element of the array can be named by suffixing the array name with a subscript that specifies the element's position within the array. Except in an EQUIVALENCE statement, when the unsubscripted array name occurs in a source program, it refers to the entire array (see Subarray References in section 10). An unsubscripted array name in an EQUIVALENCE statement or namelist input references only the first element of the array.

E An array can be a simple array, descriptor array, or double descriptor array. An array containing scalar elements is a simple array.

For each array, a DIMENSION, ROWWISE, COMMON, or type declaration statement must be used to declare the array's size. This declaration must be made once in each program unit that references or defines the array; if more than one program unit uses the array, the declaration must be the same in all of the program units.

An array declarator is used to declare the size of an array, and has the following form:

a(d)

a The array name.

d A list of the form:

$d_1, \ldots, d_n$

A where n is the number of dimensions the array is to have; and where d_i is an integer constant or simple integer variable whose magnitude indicates the maximum value that a subscript expression for the i^{th} dimension may attain in any array element name.

The dimension d_i can be a variable only when a is a dummy argument in a subprogram. Also, an augmented form of the array declarator, in which an element length specification of the form *k appears between the array name and the left parenthesis, can appear in the CHARACTER type statement. Type statements and dummy arguments are discussed later in sections 6 and 7.

The data type of an array is determined by the same explicit and implicit rules that determine the data type of a variable name. The data type of an array element is that of the array. It is possible (but not necessary) to declare the size and data type for an array with the use of a single array declarator. For example, the explicit type statement COMPLEX A(50) declares the array A to have 50 elements all of which are of type complex. In this example, no additional statement would be required (or allowed) for assigning a data type to the array.

The amount of storage reserved for an array is determined by the array's size and data type. For any array, the number of words, bytes, or bits reserved is the number required for a single element of the particular data type, times the number of elements. For example, COMPLEX A(50) reserves 100 words of storage for A, because any data element of type complex requires 2 words for its internal representation, and the array A consists of 50 of such complex data elements.

Arrays can have one to seven dimensions. A one-dimensional array can be thought of as a list or series; a two-dimensional array, as a matrix. The product of the dimension sizes equals the number of elements in the array.

Subscripts and Array Declarators

A subscript consists of a pair of parentheses enclosing one to seven subscript expressions separated by commas. Subscripted array names must not be confused with array declarators: an array declarator declares the dimensions of an array, and a subscripted array name identifies a single array element. A subscript appears in an array element name, immediately after the array name. Except in an EQUIVALENCE statement, the number of subscript expressions must always equal the number of dimensions for the array.

Each dimension in an array declarator can be an integer constant or, in a subprogram, a single integer variable. An integer variable dimension, permitted only when the array is a dummy argument, must either also be a dummy argument or else be in common. B A variable used in this way as an adjustable dimension must either be implicitly integer, or else must appear in an INTEGER type statement before it appears in any other declaration statement.

Each subscript expression in an array element name can be any scalar arithmetic expression of type integer, real, or double precision, and must never assume a value less than 1 or larger than the maximum length specified in the C declarator (the value is not checked at run time). When the value of the expression is not integer, it is truncated to integer.

Subscript Interpretation

A subscript can identify an element in the array in either of two ways, depending on whether the array declarator occurred in a ROWWISE statement or occurred in a DIMENSION, COMMON, or type declaration statement. The conventional succession of elements in an array is defined by a succession of subscripts in which the value of the leftmost subscript expression varies through its range (from 1 to the maximum value of that dimension), then the value of the D subscript expression to its right is increased by 1 and the first goes through its range again, and so on, until each subscript expression has gone throughout its entire range at least once. The subscript significance is just the reverse for an array that has been declared in a ROWWISE statement: the succession of elements is defined by a succession of subscripts in which the value of the rightmost subscript expression varies through its range, then the value of the subscript expression to its left increases by 1 and the last goes through its range again, and so on, until each subscript expression has gone through its entire range at least once.

A: . . . have; and where d_i is a dimension bound declarator. A dimension bound declarator. A dimension bound declarator for an array which is not a dummy argument is an integer constant expression or two integer constant expressions separated by a colon. A dimension bound declarator for an array which is a dummy argument is an integer expression or two integer expressions separated by a colon, except that d_n (d_1 if the array declarator appears in a ROWWISE statement) may be an asterisk, or else an integer expression followed by a colon and an asterisk in that order. Nonconstant references in a dimension bound declarator for an array which is a dummy argument are restricted to simple integer variables which are in common or else appear in every dummy argument list in which the array name appears.

If d_i consists of a single integer expression, the dimension size of the ith dimension is just the value of that expression; if d_i consists of two integer expressions separated by a colon, the dimension size of the ith dimension is 1 plus the value of the second expression minus the value of the first. If the nth dimension bound declarator (1st if the array declarator appears in a ROW-WISE statement) has an upper bound of asterisk, the dimension size of the nth (1st if ROWWISE) dimension is unknown.

If the array declarator appears in a CHARACTER type statement, it may be optionally followed by an asterisk and a length specification k, or, as a non-standard alternative, the asterisk and length specification may be inserted between the array name a, and the following left parenthesis:

```
.character array declarator.     : = a .std-declarator.
                                 : = a .non-std-declarator.

.std declarator.        : = .dimension.
                        : = .dimension. .length-spec.

.non-std-declarator.        : = .length-spec.    .dimension.

.dimension.      : = (d1l, . . . , dn)

.length-spec.    : = *k
```

The length specification, k, is a nonzero unsigned integer constant, an integer constant expression enclosed in parentheses. If it is an asterisk enclosed in parentheses, the array must be a dummy argument; in that case, the elements of the dummy array are of the same length as those of the associated actual array (8.4.2).

This page left blank intentionally

B: Each dimension of an array is defined by one or two dimension bound expressions (except the last of an assumed size array or the first of a ROWWISE assumed size array). A dimension bound expression must be an integer constant expression except when a dummy array is being declared, in which case it may involve references to integer variables in common and to integer dummy arguments which appear in every dummy argument list in which the dummy array name also appears. A variable used in this way as an . . .

C: . . . any scalar arithmetic expression of type integer, real, double precision, or half precision, and must never assume a value less than the lower or greater than the upper dimension bound specified in the . . .

D: . . . subscript expression varies through its range (from lower dimension bound to upper), then the value of the . . .

E: An array can be a simple array or a dynamic array.

A To find the location of an array element in the linear sequence in which the elements are stored given its identifying subscript, the formulas listed in table 2-2 can be used. In the table, capital letters are dimension sizes and lower case letters are the subscript expression values of a particular subscript.

A comparison is made of the ordering for conventional and rowwise subscripts for a 3-dimensional array of 24 elements in table 2-3. Interpreted geometrically, the conventional ordering is 2 rows, 3 columns, and 4 planes, as shown in figure 2-1. The rowwise ordering interpreted geometrically is 4 rows, 3 columns, and 2 planes, shown in figure 2-2.

B

DATA ELEMENT FORMS

A data element or function name must be associated implicitly or explicitly with a data type. The association applies to every occurrence of the name throughout the program unit in which the association is defined.

C The data type of a variable, array, or function name is implied by the first letter of the name or else must be specified explicitly (the data type of a FORTRAN-supplied function is predefined). The data type of a constant is implied by its form. The internal representation of a value of a particular data type is the same whether it is the value of a variable, of an array element, or of a constant.

TABLE 2-3. SUBSCRIPTING ORDER FOR A THREE-DIMENSIONAL ARRAY A(2,3,4)

ROWWISE Subscript Succession	Ordinality	Conventional Subscript Succession
A(1,1,1)	1	A(1,1,1)
A(1,1,2)	2	A(2,1,1)
A(1,1,3)	3	A(1,2,1)
A(1,1,4)	4	A(2,2,1)
A(1,2,1)	5	A(1,3,1)
A(1,2,2)	6	A(2,3,1)
A(1,2,3)	7	A(1,1,2)
A(1,2,4)	8	A(2,1,2)
A(1,3,1)	9	A(1,2,2)
A(1,3,2)	10	A(2,2,2)
A(1,3,3)	11	A(1,3,2)
A(1,3,4)	12	A(2,3,2)
A(2,1,1)	13	A(1,1,3)
A(2,1,2)	14	A(2,1,3)
A(2,1,3)	15	A(1,2,3)
A(2,1,4)	16	A(2,2,3)
A(2,2,1)	17	A(1,3,3)
A(2,2,2)	18	A(2,3,3)
A(2,2,3)	19	A(1,1,4)
A(2,2,4)	20	A(2,1,4)
A(2,3,1)	21	A(1,2,4)
A(2,3,2)	22	A(2,2,4)
A(2,3,3)	23	A(1,3,4)
A(2,3,4)	24	A(2,3,4)

TABLE 2-2. ARRAY ELEMENT SUCCESSION FORMULAS

D

Dimensionality	Declarator Dimensions	Instance of Subscript	Location of Array Element
1	(A)	(a)	a
2	(A,B) (B,A)†	(a,b) (b,a)†	a+A*(b-1)
3	(A,B,C) (C,B,A)†	(a,b,c) (c,b,a)†	a+A*(b-1) +A*B*(c-1)
4	(A,B,C,D) (D,C,B,A)†	(a,b,c,d) (d,c,b,a)†	a+A*(b-1) +A*B*(c-1) +A*B*C*(d-1)
5	(A,B,C,D,E) (E,D,C,B,A)†	(a,b,c,d,e) (e,d,c,b,a)†	a+A*(b-1) +A*B*(c-1) +A*B*C*(d-1) +A*B*C*D*(e-1)
6	(A,B,C,D,E,F) (F,E,D,C,B,A)†	(a,b,c,d,e,f) (f,e,d,c,b,a)†	a+A*(b-1) +A*B*(c-1) +A*B*C*(d-1) +A*B*C*D*(e-1) +A*B*C*D*E*(f-1)
7	(A,B,C,D,E,F,G) (G,F,E,D,C,B,A)†	(a,b,c,d,e,f,g) (g,f,e,d,c,b,a)†	a+A*(b-1) +A*B*(c-1) +A*B*C*(d-1) +A*B*C*D*(e-1) +A*B*C*D*E*(f-1) +A*B*C*D*E*F*(g-1)

†This is a subscript for an array declared in a ROWWISE statement.

A: . . . used. In the table, capital letters are dimension bounds and . . .

B: SUBSTRINGS

A substring reference is a character variable name or character array element name followed by a left parenthesis, an integer expression, a colon, an integer expression, and a right parenthesis. Both integer expressions are optional; they default to 1 and the length of the character substring references datum respectively.

C: . . . specified explicitly.

The type of a specific intrinsic function name is predefined. The type of a generic intrinsic name depends upon the type of its argument or else is predefined. See section 6 for the effect of explicitly typing an intrinsic function name. .

The data type of a constant is . . .

D:

Dimensionality	Declarator	Subscript	Displacement of Array Element
1	$(A_L:A_U)$	(a)	$a-A_L$
2	$(A_L:A_U,B_L:B_U)$	(a,b)	$a-A_L+(A_U-A_L+1)*(b-B_L)$[1] $b-B_L+(B_U\_B_L+1)*(a-A_L)$

Note 1: This is the displacement for an array declared in a ROWWISE statement.

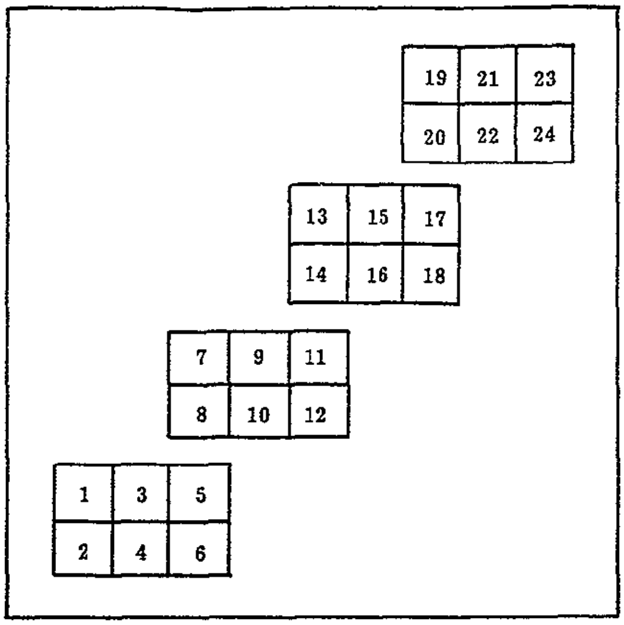


Figure 2-1. Conventional Ordering of Elements in a Three-Dimensional Array, A(2,3,4)

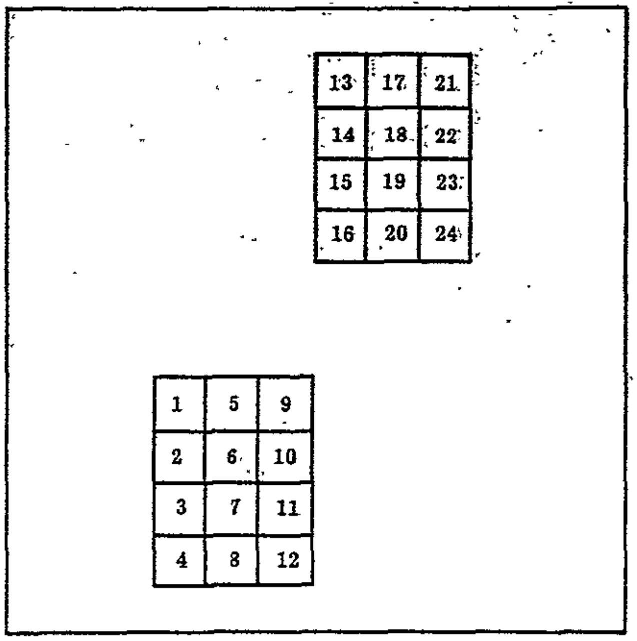


Figure 2-2. ROWWISE-Declared Array, A(2,3,4)

INTEGER ELEMENTS

An integer constant has the following form:

$d_1d_2 \ldots d_m$

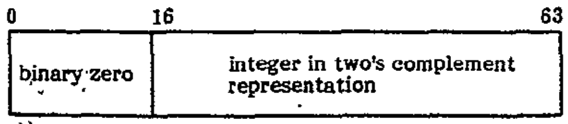
d_i A decimal digit (0 through 9); $1 \leq m \leq 14$.

It is written without a decimal point and without embedded commas.

A signed integer constant is an integer constant prefixed by a plus or minus sign. If an integer is positive, the plus sign can be omitted. If an integer is negative, a minus sign must be present. An optionally signed integer constant is an integer constant or a signed integer constant. Integer zero is neither positive nor negative but can be signed (with no significance).

The value range for an integer is -2^{47} through $2^{47}-1$. Integers used in addition, subtraction, multiplication, division, or exponentiation, as well as the results of such operations, must be within this range.

Integer data occupies one word of storage in the following format:



A variable or array can be associated with the integer data type implicitly or explicitly, as described under Variables in this section. A

Examples of integer constants:

237   0   13593569

Examples of signed integer constants:

-237   +13593569

REAL ELEMENTS

A real constant can have one of the following forms:

n
nEx
mEx

n	A string of one or more decimal digits and one decimal point. The decimal point can be placed anywhere in the string, including first or last.
m	An integer constant.
x	An optionally signed integer constant in the range -8617 through 8645.

The Ex in the real constant form expresses the exponent. Interpreted arithmetically, nEx means $n*10^x$ and mEx means $m*10^x$. An exponent of E+0 is assumed if a real constant contains no exponent. A signed real constant is a real constant prefixed by a plus sign or minus sign. The constant must be preceded by a minus sign if the real number represented is negative, but the plus sign is optional if the number is positive. An optionally signed real constant is a real constant or a signed real constant.

The absolute value range for a real number is approximately 0 through .953 708 115 431 87E+8645. The smallest positive real number that can be represented is approximately .519 211 284 565 73E-8617. The precision retained in calculations involving real numbers is approximately 14 significant decimal digits.

A: A variable, array, or named constant can be associated with the integer data . . .

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

Real data occupies one word of storage in the following format:

0	16 ... 63
exponent, a two's complement integer	mantissa, a two's complement integer

Examples of real constants:

2.5 .25E+1 .25E1 2500E-3 0E0

Examples of signed real constants:

+2.5 -.25E+1 +.25E1 -2500E-3 +0E0

Real data is always represented in normalized form in that the most significant bit of the mantissa appears in bit 17, with the value of the exponent adjusted appropriately. The STAR-100 Computer Hardware Reference Manual contains more detailed descriptions of the hardware representations for numeric data.

A{A variable or array can be associated with the real data type either implicitly or explicitly, as described under Variables in this section.

DOUBLE PRECISION ELEMENTS

A double precision constant has one of the following forms:

nDx
mDx

n A string of one or more decimal digits and one decimal point. The decimal point can be placed anywhere in the string, including first or last.

m An integer constant.

x An optionally signed integer constant in the range -8617 through 8645.

The Dx in the form expresses the exponent.

A double precision constant is written and interpreted in exactly the same way as a real constant, except that the exponent must always be used and the letter D is used in the exponent instead of an E.

The value range for double precision numbers is the same as for real numbers; however, the precision retained is approximately 28 significant digits instead of 14. The largest double precision number that can be represented is .561 194 593 766 944 619 962 041 407 3D+8645. The smallest positive double precision number that can be represented is approximately .519 211 284 565 733 055 700 413 533 9D-8617.

Double precision data occupies two contiguous words of storage. The first word is in the same format as for type real data and expresses the most significant digits. The second word is in the same format as the first, except that the exponent value is 47 less than the exponent of the first and the mantissa has not been normalized. The second word is always nonnegative (zero or positive).

B{ A variable or array can be associated with the double precision data type by means of the DOUBLE PRECISION or the IMPLICIT type declaration statement.

Examples of double precision constants:

.25D+1 .25D1 2500D-3

3.141 592 653 589 793 238 462 643 3D+0

Examples of signed double precision constants:

+.25D+1 -.25D1 +2500D-3

}E

COMPLEX ELEMENTS

A complex constant must have the following form:

(r_1,r_2)

r_i An optionally signed real constant.

A complex constant is written as an ordered pair of optionally signed real constants separated by a comma and enclosed in parentheses. The parentheses are part of the constant and must always appear. The value range for either r_1 or r_2 is the same as for type real data.

Complex data occupies two contiguous words of storage, each of which is in the format for type real data. The first word (r_1 in the form), represents the real part of the complex number. The second word (r_2 in the form), represents the imaginary part.

A variable or array can be associated with the complex data type only by means of the IMPLICIT or the COMPLEX type declaration statement. }C

Examples of complex constants:

(4.0, 5.0), which has the value of the complex number 4.0 + 5.0i, where $i = \sqrt{-1}$

(0., -1.)

(+.4E1, 5.0)

(-4., -5.)

LOGICAL ELEMENTS

A logical constant has one of the following forms:

.TRUE.

.FALSE.

The periods are part of the constants and must appear.

Logical data occupies one word of storage in the following format:

0 ... 63
0000 . . . . . . 00d

where d is a 1 bit or 0 bit for .TRUE. and .FALSE. respectively.

A variable or array is associated with the logical data type by means of the IMPLICIT or the LOGICAL type declaration statement. }D

A: A variable, array, or named constant can be associated with the real data type . .

B: A variable, array, or named constant can be associated with the double . . .

C: A variable, array, or named constant can be associated with the complex data . . .

D: A variable, array, or named constant is associated with the logical data type . . .

E: HALF PRECISION ELEMENTS

A half precision constant has the following form:

```
.half-precision-constant.    := n S x
                             := ± n S x
                             := n S ± x
                             := ± n S ± x
```

where n is a string of decimal digits including an optional decimal point and x is a string of decimal digits.

A half precision constant is interpreted as

$\pm$ n * 10 ** ($\pm$x)

Examples of half precision constants: -.1S6 3.14159S0 6.23S23 1.0S-06

The largest normalized half precision datum which can be internally represented is #6F 7F FF FF (approximately $2.177807\text{x}10^{40}$); the smallest is #6F 80 00 00 (approximately $-2.177807\text{x}10^{40}$). The smallest normalized half precision datum which is greater than zero is #90 40 00 00 (approximately $8.077936\text{x}10^{-28}$); the largest less than zero is #90 BF FF FF (approximately $-8.077938\text{x}10^{-28}$).

Half precision data occupy halfwords of storage in the following format:

0 7	8 31
exponent, a two's complement integer	mantissa, a two's complement integer

(The internally stored exponent is a power of two, not ten.)

A variable, array, or named constant of type HALF PRECISION may be declared with the HALF PRECISION or IMPLICIT type statements.

HOLLERITH ELEMENTS

A A Hollerith constant is a string composed of an (unsigned) integer constant followed by the letter H and a nonempty string of any m of the 64 characters in the ASCII subset. The blank character is an acceptable and significant character in a Hollerith constant.

Form:

B mHs

m	An (unsigned) integer constant less than or equal to 255 and nonzero.
s	A string of exactly m characters included in the 64-character ASCII subset (see appendix A).

Hollerith data uses m contiguous bytes (a byte is eight bits) to represent m characters. Eight characters fill one machine word. The word boundary generally does not effect how Hollerith data is stored; however, when used as an actual argument in a subroutine call or function reference, a Hollerith constant is aligned on a word boundary and extended with blanks on the right so that it occupies a whole number of words. C

Examples of Hollerith constants:

```
19HRESULT NUMBER THREE      5H12345

5Hb̸b̸b̸b̸b̸      1H,
```

D A Hollerith constant can be used as an actual argument, or for data initialization in a DATA or type statement. For compatibility with FORTRAN Extended, other uses of Hollerith constants are supported as described in appendix G.

It is not possible to declare a variable or array to be type Hollerith.

CHARACTER ELEMENTS

A character constant is a nonempty string of characters enclosed in single quotes. If a single quote (') is required within the string as one of the characters, it must be prefixed with another single quote. The character blank is a significant character in a character constant.

Form:

$'c_1c_2 \dots c_m'$

c_i	A character selected from the 64-character ASCII subset; m is less than or equal to 255.

Character data uses m contiguous bytes of storage to represent m characters: eight characters fill one machine word.

Examples of character constants:

```
'RESULT NUMBER THREE'   '12345'   'b̸b̸b̸b̸b̸'   ','
```

In contrast to the Hollerith data type, the character data type can be associated with a variable or array, in which case the variable or array must have length as well as type specified in an IMPLICIT or CHARACTER type declaration statement.

HEXADECIMAL ELEMENTS

A hexadecimal constant is a string composed of the letter X followed by a nonempty string of m hexadecimal digits enclosed in single quotes. The 16 hexadecimal digits are the digits 0 through 9 and the letters A through F.

Form:

$X'h_1h_2 \dots h_m'$

h_i	A hexadecimal (base 16) digit; m is less than or equal to 255.

Hexadecimal data uses as many contiguous bits of storage as are required to represent m digits: the digits 0 through F (interpreted as the hexadecimal equivalents of the decimal digits 0 through 15) each take four bits. The word boundary is not significant for hexadecimal data.

Examples of hexadecimal constants:

```
X'33'      X'1A9'   X'FFFFFFFFFFFFFFFF'
```

Hexadecimal constants are restricted to use in data initialization and special CALL statement argument lists.

It is not possible to declare a variable or array to be type hexadecimal. E

BIT ELEMENTS

A bit constant is a string composed of the letter B followed by a nonempty string of m binary digits (bits) enclosed in single quotes.

Form:

$B'b_1b_2 \dots b_m'$

b_i	A bit (0 or 1); m is less than or equal to 255.

Bit data uses m contiguous bits; the word boundary is not significant. The digits 0 and 1 each correspond to one bit in storage.

Examples of bit constants:

```
B'0'     B'10101111'     B'0000000000000001'
```

Bit constants are restricted to use in subprogram references, scalar and vector bit assignment statements, and data initialization.

A bit variable is associated with the bit data type by means of the BIT or the IMPLICIT type declaration statement. F

A: . . . integer constant m followed by the letter H or the letter R and a nonempty . . .

B: .hollerith-constant. := m H s
:= m R s

m An unsigned integer constant less than 256.

C: machine word. H constants (R constants) are stored aligned to a word boundary on the left (right) and blank (zero) filled to a word boundary on the right (left).

D: Hollerith constants are arithmetic constants and may be used wherever other arithmetic constants are legal. In particular, they may appear in arithmetic expressions, where they are typeless (that is, they assume the type of the operand with which they are combined), except that they are of type INTEGER when the arithmetic expression consists of a single Hollerith constant and no operators.

Hollerith constants are not character constants; they may not appear in character expressions.

Truncation of H (R) constants to their leftmost (rightmost) eight characters occurs whenever long H (R) constants appear in contexts other than actual argument lists. In particular, long H (R) constants in constant lists of DATA statements are truncated and initialize only a single word.

It is not possible to declare a variable or array to be type Hollerith. Hollerith constants are not permitted in PARAMETER statements.

E: It is not possible to declare a variable or array to be type hexadecimal. Hexadecimal constants may not appear in PARAMETER statements.

F: A bit variable is associated with the bit data type by means of the BIT or the IMPLICIT type declaration statements. Bit constants may not appear in PARAMETER statements. There are no named constants of type bit.

SCALAR EXPRESSIONS

A FORTRAN expression is a string of one or more operands and zero or more operators that is evaluated during program execution to yield a value. The conventional precedences for the FORTRAN arithmetic and logical operators are given later in this section.

An expression generally specifies a computation or a comparison between operands. However, in its simplest form an expression consists of a single data element (a A{single constant, variable, or array element) or a function reference. This section gives the formation and evaluation rules for the following kinds of scalar expressions:

Arithmetic	Yields numeric values; appears in arithmetic assignment statements and in relational expressions
B{ Character	Contains no operators; is used in character assignment statements and relational expressions
Relational	Yields logical values; appears in logical expressions
Logical	Yields logical values; appears in logical expressions and logical assignment statements
Bit	Yields bit values; appears in bit assignment statements

When an expression is evaluated during program execution, the result is retained in a variable, is used immediately as an operand for another operation, or is passed as an argument to a function or subroutine. An expression whose evaluation yields a result of a certain type is called an expression of that type; for example, an expression whose evaluation yields an integer result is called an integer expression.

Examples of expressions:

Expression	Value
X	Current value of the variable X
3.5	Constant real number 3.5
'CHARACTERS'	Character constant, 10 ASCII characters
DB1/DB2**2	Value of DB1 divided by the square of the value of DB2
A(C/B)	Array element A(I), where I is the value of the expression C/B
SQRT (TRUNK)	Function reference
(A+B+3*C)/2.56	The sum of the expressions A, B, and 3*C, divided by 2.56
X.LT. Y-1.0	.TRUE. if the value of X is less than the value of Y-1.0, .FALSE. otherwise
.NOT. FNLOG(B)	.TRUE. if the value of the expression FNLOG(B) is .FALSE., .FALSE. otherwise

If the value of an expression can be established without evaluating a certain part of the expression, then that part might never be evaluated. For this reason the user cannot rely on any side effects an expression might be able to produce.

Example:

During evaluation of the logical expression

```
Y .OR. F(X) .OR. Z
```

if Y has the value .TRUE., the expression has the value .TRUE. whatever the values of F(X) and Z may be. In this situation the execution of F might or might not occur as a result of the expression evaluation.

Another consideration for the user is compatibility between operand types during evaluation. The operand types that can be combined in the same arithmetic or relational expression are the following, in order of decreasing dominance:

Complex (cannot occur in relational expressions) }C

Double precision

Real

Integer }D

In general, when two operands that are to be operated upon have different types, the value of the dominated operand is converted to the type of the dominant operand before the operation is performed. For example, if the operand types of an expression (consisting of two operands and a dyadic operator) were real and integer, the effect would be as though the integer had been converted to type real data before a real operation (an operation involving only type real operands) was performed.

ARITHMETIC EXPRESSIONS

The FORTRAN arithmetic operators are:

+	Addition; unary plus
-	Subtraction; unary minus
*	Multiplication
/	Division
**	Exponentiation

Unary plus and minus are conceptually like dyadic addition and subtraction using an implied zero operand of the same type as the given unary operand.

An arithmetic expression can be a single constant, simple variable, array element, or function reference. If X is an arithmetic expression, then (X) is an arithmetic expression. Each left parenthesis must have a corresponding right

A: . . . single constant, variable, substring, or array element) or a function . . .

B: Character Is used in charac-

C: Complex

D: Half Precision

parenthesis in the same expression. Furthermore, if X and Y are arithmetic expressions, then the following are also arithmetic expressions:

```
X+Y
X*Y
X-Y
X/Y
X**Y
```

All operations must be specified explicitly. For example, to multiply two variables X and Y, the expression X*Y must be used; XY, (X) (Y), or X.Y does not result in multiplication. Also, operators in an expression must not be contiguous. A unary plus or unary minus can be separated from another operator in an expression by using parentheses around the signed element.

Examples of arithmetic expressions:

```
3.5
3.5 + N
-(3.5+N)/2**M
(XBAR+(B(I,J+I,K)/3.0))
-(C+DELTA*AERO)
(-B-SQRT(B**2-(4*A*C)))/(2.0*A)
GROSS - (TAX*0.04)
TEMP + V(M,AMAX1(A,B))*Y**C/(H-FACT(K+3))
```

EXPONENTIATION

The following types of base and exponent are permitted in exponentiation:

Type of Base	Type of Exponent
Integer	Integer, real, double precision
Real	Integer, real, double precision
Double precision	Integer, real, double precision
Complex	Integer, real

A

Also, a negative-valued base can have an exponent of type integer only and a zero-valued base can be raised to a positive exponent only.

An expression (or a subexpression delimited by parentheses) that contains only operands and the exponentiation operator is evaluated from right to left. That is, A**B**C means (A**(B**C)). This interpretation can be changed with appropriate use of parentheses, for example, (A**B)**C.

EVALUATION OF ARITHMETIC EXPRESSIONS

The value of an arithmetic expression is a close approximation to the mathematical interpretation. The sequence in which the elements of an expression are evaluated is governed by the following rules listed in descending precedence:

1. Subexpressions delimited by parentheses are evaluated beginning with the innermost subexpressions.

2. Subexpressions defined by arithmetic operators are evaluated.

3. Subexpressions containing operators of equal precedence are evaluated in effect from left to right, except for exponentiation which is evaluated from right to left (the exponent's value is calculated before the base's value).

For example, the expression

```
A/B/C-D*E**F
```

might be evaluated as follows:

1. E is raised to the power of F.
2. A is divided by B.
3. Quotient in step 2 is divided by C.
4. Result of step 1 is multiplied by D.
5. Product in step 4 is subtracted from result of step 3.

If the result of an integer division is not integral, then the fractional part is discarded. The result of an integer division is the nearest integer whose absolute value does not exceed the absolute value of the magnitude of the mathematical ratio. For example, 3/2*4 has the value 4, -3/2*4 has the value -4, and 3/(-2)*4 has the value -4.

Operators that are mathematically associative or commutative might be reordered during compilation. The user can force a definite ordering of mathematically associative operators of equal precedence by appropriate use of parentheses. Subexpressions containing integer divisions are not reordered within the division/multiplication precedence level, however, because the truncation resulting from an integer division renders these operations nonassociative.

The evaluation of an array element or function reference in an expression requires the evaluation of the subscript or actual arguments. The evaluation of the subscript or actual arguments does not affect the type of the value of the expression in which the subscript or argument list appears; neither does the expression type affect subscript or actual argument evaluation. Evaluation of a function must not alter the value of any other element within the statement in which the function reference appears.

No element can be evaluated whose value is not mathematically defined. For example, division by zero or the square root of a negative number cannot be evaluated.

TYPE OF AN ARITHMETIC EXPRESSION

The arithmetic operators +, -, *, and / can be used to combine any elements of the same numeric data type into an expression; the resultant value has the same data type as that of the operands. For example, when two real numbers are added, the data type of the result is real, and the operation is referred to as a real operation. Furthermore, a

A: The base and exponent may be any arithmetic types. The dominance of types is the same as for other arithmetic operators.

However. if the value of the base is negative, the exponent must be type integer unless one or both operands are type complex. If the value of the base is zero, the exponent must be type integer and the exponent's value must be greater than zero.

A{ complex, double precision, real, or integer element can be combined with one of these operators into an expression with an element of any of the types complex, double B{ precision, real, or integer, with the resultant value having the type possessed by the dominant operand.

CHARACTER EXPRESSIONS

A character expression consists of exactly one data element and no operators. This element can be any one of the following:

C{

- A character constant
- A Hollerith constant
- A character array element
- A character variable
- A character function reference

The value of a character expression is the value of the element. The type of a character expression is character.

RELATIONAL EXPRESSIONS

The FORTRAN relational operators are:

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

The periods are part of the operators and must appear.

A relational expression is a relational operator bracketed by two operands:

$aexpr_1$ op $aexpr_2$

$cexpr_1$ op $cexpr_2$

op	A relational operator.
$cexpr_i$	A character expression.
$aexpr_i$	An arithmetic expression.

The operands can be either two arithmetic expressions or two character expressions. As the forms above show, a relational expression cannot contain two relational operators.

Examples of relational expressions:

D{
```
5HASTER .LT. C
'ANEMONE' .EQ. FNCHAR
X+Y/3.*Z .NE. X
A(I) .GE. SQRT(R)
AMRYL .LT. 1.5D4
```

Evaluation of a relational expression consisting of arithmetic expressions proceeds as follows: each arithmetic expression is evaluated; type conversion to the dominant type takes place if the types of the arithmetic expressions differ; then the compare is made. The relational expression has the logical result .TRUE. or .FALSE. as the relation is true or false, respectively.

Arithmetic expressions in relational expressions cannot be of type complex; they can be integer, real, or double precision, however. For example, (2.0,1.0)*N is syntactically correct, but ((2.0,1.0)*N).GE.M is not. }E

When a relational expression consists of character expressions, the corresponding characters in the values of the two expressions are compared one character at a time from left to right. A character is considered greater than another character, for example, if its hexadecimal equivalent as shown in appendix A is greater than that of the other. If the two character expressions have different lengths, comparison proceeds as though the shorter had been padded on the right with blank characters until the expressions were of equal length (the hexadecimal equivalent of the blank character is less than that of any other character in the ASCII subset).

LOGICAL EXPRESSIONS

The FORTRAN logical operators are:

.AND.	Logical and
.OR.	Logical or
.XOR.	Logical exclusive or
.NOT.	Logical negation

}F

The periods must appear in any occurrence of a logical operator. The mathematical definitions of the logical operators are given in table 3-1.

TABLE 3-1. LOGICAL OPERATOR TRUTH TABLES

p	g	p.AND.g	p.OR.g	p.XOR.g	.NOT.p
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

}G

A logical expression can be a single relational expression, logical constant, logical variable, logical array element, logical function reference, or a logical expression enclosed in parentheses. Also, if X and Y are logical expressions, then .NOT.X, and X followed by a binary logical operator followed by Y, are logical expressions.

Examples of logical expressions:

```
(X).AND..NOT.Y
X*2.114 .NE.(B*22.114).AND. Z1 .AND. Z2 .AND. Z3
.NOT. (X.AND..NOT.Y) .OR. (Z.EQ.98.6)
B-C .LE. A .AND. A .LE. B+C
```

A: . . . complex, double precision, real, half precision, or integer element can be . . .

B· . . . precision, real, half precision, or integer, with the resultant value having . . .

C: CHARACTER EXPRESSIONS

The FORTRAN character operator is:

/ /	Concatenation

A character expression may be any of the following:

A character constant
A character variable
A character array element
A character function reference
A character substring
$cexpr_1$ / / $cexpr_2$
(cexpr)

where cexpr, $cexpr_1$, and $cexpr_2$ are character expressions. The concatenation operator forms a character string whose initial characters are the first operand and whose final characters are the second operand. For example, if the character variable VERB has the value 'LOOK', then the expression VERB / / 'ING' has the value 'LOOKING'.

Examples of character expressions:

```
'ING'
VERB
VERBS(K)
W(X+Y)
VERB(1:2)
VERBS(K)(L:L+1)
VERB / / 'ING'
```

where VERB is a character variable, VERBS is a character array, and W is a character function.

Character expressions may not exceed 65,535 characters.

A character substring is a character variable or character array element followed by a substring designator of one of the following forms:

(j:k)	
(:k)	meaning (1:k)
(j:)	meaning (j:n)
(.)	meaning (1:n)

where j is greater than or equal to one and less than or equal to k and k is less than or equal to n and n is the number of characters in the character ariable or array element. The character substring consists of the jth through e kth characters (inclusive) of the character variable or array element.

This page left blank intentionally

D: delete

E: Complex expressions are allowed as operands in relational expressions only when the operator is .EQ. or .NE.

F:

.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence

G: Table 3-1. LOGICAL BINARY OPERATOR TRUTH TABLES

p	q	p.AND.q	p.OR.q	p.EQV.q	p.NEQV.q
T	T	T	T	T	F
T	F	F	T	F	T
F	T	F	T	F	T
F	F	F	F	T	F

'XOR. is the same as .NEQV.

Table 3-2. LOGICAL UNARY OPERATOR TRUTH TABLE

p	.NOT. p
T	F
F	T

PRECEDING PAGE BLANK NOT FILMED

.NOT. can appear adjacent to itself only with intervening parentheses as in the following types of constructs:

```
.NOT. (.NOT.p)

.NOT. (.NOT. (.NOT.p))
```

.NOT. can appear adjacent to any other logical operator only as the operator on the right, as in the following constructs:

A

```
p.AND..NOT.q

p.OR..NOT.q

p.XOR..NOT.q
```

The operators .AND., .OR., and .XOR. cannot appear adjacent to each other; they are always flanked by relational expressions, logical elements, or any such logical expressions. (This corresponds to the mathematical usage of logical conjunction and disjunction.)

Whenever precedence is not established explicitly by parentheses, the logical, relational, and arithmetic operations that might appear in a logical expression are evaluated according to the precedences shown in table 3-2. The unparenthesized expression X.OR.Y.AND.Z.OR.W, for example, means (X.OR.((Y.AND.Z).OR.W)), and if the user had intended (X.OR.Y).AND.(Z.OR.W), then the parentheses would need to be explicit. The plus/minus category in the table applies to both unary and dyadic additive operations. The value of a logical expression is always of type logical.

BIT EXPRESSIONS

A bit expression is formed with bit data elements and the logical operators used in logical expressions. A bit expression can be a single bit constant, bit variable, bit array element, or bit expression enclosed in parentheses. Also, if B and C are bit expressions, then .NOT.B, and B followed by a binary logical operator followed by C, are bit expressions.

The operators used in bit expressions are the logical operators interpreted so that truth is the bit value 1 and falsity is the bit value 0. The mathematical definitions of the logical operators are given in table 3-1; the precedences of the operators are the same as for logical operators in logical expressions.

Bit expressions might be used to define a bit variable or bit array element, or as a more efficient use of storage for logical operations (bits instead of words).

Examples of bit expressions:

```
B'I'

C1(4)

(B).AND..NOT.C

B1.AND.B2.AND.B3.AND.B4

.NOT.(BOG.AND..NOT.BOH).OR.CO2

C1(N).XOR.C2(N)
```

TABLE 3-2. OPERATOR PRECEDENCES

Operator	Precedence	Category
**	first	Arithmetic
/ *	second	
±	third	
.EQ. .NE. .GE. .LE. .LT. .GT.	fourth	Relational
.NOT.	fifth	Logical
.AND.	sixth	
.OR. .XOR.	seventh	

B

A: p.AND..NOT.q
p.OR..NOT.q
p.X XR..NOT.q
p.EQV..NOT.q
p.NEQV..NOT.q

The binary operators .AND., .OR., .XOR., .EQV., and .NEQV. ...

B. Table 3.3 OPERATOR PRECEDENCES

Precedence	Operators	Category
1st	**	Arithmetic
2nd	*,/	Arithmetic
3rd	+,-	Arithmetic
4th	//	Character
5th	.EQ.,.NE.,.LT.,.LE.,.GT.,.GE.	Relational
6th	.NOT.	Logical
7th	.AND.	Logical
8th	.OR.	Logical
9th	.XOR.,.EQV.,.NEQV.	Logical

A scalar assignment statement initiates evaluation of the expression on the right side of the equals sign. When evaluation is complete, the variable to the left of the equals sign is assigned the value of the expression.

This section gives the formation rules for the following types of scalar assignment statements:

- Arithmetic
- Character
- Logical
- Bit

The terms left hand side and right hand side of an assignment statement refer in this manual to everything in the statement that lies to the left of and to the right of the equals sign, respectively.

Examples:

Statement	Meaning
A = A + 1	Replace the value of A with the value of A + 1
K(4) = K(1) + K(2)	Replace the value of K(4) with the sum of the array elements K(1) and K(2)
I = (-2.3, 1.5)	Replace the value of I with the truncated real part of the complex constant, -2
A = 3	Replace the value of A with 3.0

ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement has the following form:

var=expr

expr	An arithmetic expression.
var	A simple variable or array element, of type integer, real, double precision, or complex.

A

If the type of the element to the left of the equals sign differs from that of the expression on the right, type conversion takes place during assignment. The value of the expression, converted to the type of the variable on the left side, replaces the value of the variable.

The rules for conversion during arithmetic assignment are given in table 4-1. Terms used in the table are defined as follows:

Contract	Convert double precision to real.
Extend	Convert real to double precision, filling the new mantissa with zeros.
Float	Convert integer to real.
Fix	Convert real to integer, truncating the fractional part.
Real part	Real part of a complex value.
Imaginary part	Imaginary part of a complex value.

C

TABLE 4-1. CONVERSION FOR ARITHMETIC ASSIGNMENT

Variable Type (Left Side)	Expression Type			
	Integer	Real	Double Precision	Complex
Integer	No conversion	Fix	Contract and fix	Fix real part and discard imaginary part
Real	Float	No conversion	Contract	Use real part and discard imaginary part
Double Precision	Float and extend	Extend	No conversion	Extend real part and discard imaginary part
Complex	Float and use for real part; zero imaginary part	Use for real part; zero imaginary part	Contract and use for real part; zero imaginary part	No conversion

B

A. var A simple variable or array element of type integer, real, double or half precision, or complex.

B:

Left Side (L)	Right Side (R)				
	Integer	Real	Double	Half	Complex
Integer	L=R	L=INT(R)	L=INT(R)	L=INT(R)	L=INT(R)
Real	L-REAL(R)	L=R	L=REAL(R)	L=REAL(R)	L=REAL(R)
Double	L=DBLE(R)	L=DBLE(R)	L=R	L=DBLE(R)	L=DBLE(R)
HALF	L=HALF(R)	L=HALF(R)	L=HALF(R)	L=R	L=HALF(R)
Complex	L=CMPLX(R)	L=CMPLX(R)	L=DMPLX(R)	L=CMPLX(R)	L=R

C: . . . given in Table 4-1. In the table, L stands for the left hand side of an assignment statement; R stands for the right. The functions referenced (INT, REAL, DBLE, HALF, and CMPLX) are just the generic intrinsic functions for type conversion; their behavior is described in Chapter 15 of this manual.

CHARACTER ASSIGNMENT STATEMENT

The character assignment statement has the following form:

var=expr

expr	A character expression.
var	A character variable or a character array element.

A

When the length of the entity var and the length of the character value of the expression expr are the same, execution of the character assignment statement causes the value of the character expression to be assigned to the character entity to the left of the equals sign.

The elements var and expr can have different lengths. When var is longer than expr, expr is extended to the right with blank characters until it matches the length of var, and then is assigned. If var is shorter than expr, expr is truncated from the right until it matches the length of var, and then is assigned.

Examples:

Given the declarations

```
CHARACTER*10 C
CHARACTER*5 VOWELS, CARRAY (50)
```

Statement	Meaning
VOWELS = 'AEIOU'	Replace the value of VOWELS with the value of 'AEIOU'
C = CARRAY (N)	Replace the value of C with the value of CARRAY (N) left-justified in C and padded on the right with five blank characters

LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement has the following form:

var=expr

expr	A logical expression.
var	A logical variable or a logical array element.

Execution of the logical assignment statement causes the value of the logical expression to be assigned to the logical entity specified to the left of the equals sign.

Examples:

```
LOGICAL LOG2     LOG2 is assigned the value .FALSE.
I=1              because I does not equal 0.
LOG2 = I..EQ. 0

LOGICAL NSUM, VAR
BIG = 200.                                NSUM is assigned
VAR = .TRUE.                              the value .TRUE.
NSUM = BIG .GT. 200. .XOR. VAR

LOGICAL A,B,C,D,E,LGA,LGB,LGC
REAL F,G,H
LGB=B.AND.C.AND.D
A=F.GT.G.OR.F.GT.H
A=.NOT.(A.AND..NOT.B).AND.(C.OR.D)
LGA=.NOT.LGB
LGC=E.OR.LGC.OR.LGB.OR.LGA.OR.(A.AND.B)
```

BIT ASSIGNMENT STATEMENT

The bit assignment statement has the following form:

var=expr

expr	A bit expression
var	A bit variable or bit array element

Execution of the bit assignment statement causes the bit value of the bit expression to be assigned to the bit entity to the left of the equals sign.

Examples:

Given the declaration

BIT B2, AI(3000)

Statement	Meaning
B2 = AI(N).OR.B'0'	Assign to B2 the value 1 if AI(N) is a 1 bit, the value 0 otherwise.
B2 = B'1'	Replace the value of B2 with a value of 1.

A:	var	A character variable or a character array element, or a substring of either of these. (No part of var may be part of expr — that is, may be part of any operand of expr.)

FLOW CONTROL STATEMENTS

The statements of a STAR FORTRAN program are in effect executed consecutively except when flow is altered by a control statement or by an exceptional condition (for example, end-of-file on input, or a data flag branch interrupt). The execution of a control statement alters, interrupts, terminates, or otherwise modifies the normal sequential flow of program execution.

Some control statements indicate where control is to be transferred by referring to a statement label. The transfer of control must not be made to a nonexecutable statement such as a FORMAT statement. It can be made to the dummy executable statement CONTINUE (which is used for no other purpose than to be labeled) or to any other labeled executable statement.

A Besides the CONTINUE statement, STAR FORTRAN contains four kinds of control statements:

Unconditional branch (GO TO statement; assigned GO TO statement)

Conditional branch (computed GO TO; arithmetic and logical IF)

Loop (DO statement)

Program control (PAUSE; STOP; CALL; RETURN)

Only the fourth kind does not involve labels.

GO TO STATEMENT

The three types of GO TO statements are unconditional, assigned, and computed.

UNCONDITIONAL GO TO

The unconditional GO TO statement has the following form:

```
GO TO n
```

n	The statement label of an executable statement.

Control is transferred on execution of the GO TO so that the statement labeled n is the next statement to be executed. The statement labeled n must be in the same program unit.

ASSIGNED GO TO

An ASSIGN statement is used in conjunction with the assigned GO TO statement. This ASSIGN statement is not related to the descriptor and double descriptor ASSIGN statements described in the vector programming section.

ASSIGN Statement

The ASSIGN statement initializes a variable for subsequent use in an assigned GO TO statement. It has the following form:

```
ASSIGN n TO var
```

n	The statement label of an executable statement.
var	A simple integer variable.

n is the label of the executable statement to which control is transferred by an assigned GO TO statement that contains the variable var. The statement labeled n must be in the same program unit in which the ASSIGN statement appears.

Use of the ASSIGN statement does not have the same effect as use of an assignment statement; for instance, an arithmetic assignment cannot be used interchangeably with an ASSIGN. Once a variable var is associated with a labeled statement by means of an ASSIGN, it must be used exclusively in ASSIGN statements and in assigned GO TO statements until it is defined by means of an assignment statement. Similarly, once it has been defined by an assignment statement, it must be used exclusively in statements other than the assigned GO TO statement until it is associated with a labeled statement by means of an ASSIGN. That is, results are unpredictable in either of the following cases:

use of the variable var in an assigned GO TO statement when var's current value was defined by other than an ASSIGN statement

use of the variable var in an arithmetic expression when var is currently associated with a labeled statement as a result of an ASSIGN

Assigned GO TO Statement

The assigned GO TO statement has the following form:

GO TO var,(n_1,n_2, . . . ,n_m)

GO TO var

var	A simple integer variable.
n_i	The statement label of an executable statement.

The comma separating var from the label list is optional. Control is transferred so that the labeled statement associated with var is the next statement to be executed. The statement labeled n_i must be in the same program unit in which the GO TO statement referencing it appears.

At the time of execution of an assigned GO TO, the variable var must have been associated with a labeled statement by prior execution of an ASSIGN statement. In the first form of the statement, var must be associated with one of the labels in the parenthesized list, while in the second form var must be associated with a label in the program unit.

Examples:

```
ASSIGN 100 TO LSWICH
GO TO LSWICH (500,100,150,200)
```

Control transfers to statement 100 upon execution of the GO TO statement.

A: The following statements are classified as control statements:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- Logical IF
- Block IF
- ELSE IF
- ELSE
- END IF
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL
- RETURN

```
ASSIGN 110 TO LSWICH
GO TO LSWICH (500,100,150,200)
```

Results of executing the GO TO statement are unpredictable because 110 is not one of the labels in the list.

COMPUTED GO TO

The computed GO TO statement has the following form:

GO TO(n_1,n_2, . . . ,n_m),sel

sel	A simple integer variable.
n_i	The statement label of an executable statement.

The comma separating sel from the label list is optional. The statement labeled n_i must be in the same program unit. The computed GO TO statement transfers control to a statement whose label is in the parenthesized list. If the selecting variable sel has the value 1, then the statement labeled n_1 is the next statement to be executed; if sel has the value i, the statement labeled n_i is the next statement to be executed. If the value of sel is not in the range 1 to m, the first executable statement following the computed GO TO is executed next.

Example:

Given the statements:

```
GO TO (200,100,400,200),L
CAT = FUR + GRIN
```

the label of the next statement executed is:

200 if L = 1

100 if L = 2

400 if L = 3

200 if L = 4

If $L \geq 5$ or if $L < 0$, control falls through to the statement immediately following the GO TO statement, in this case CAT = FUR + GRIN.

IF STATEMENT

The two types of IF statements provide for transfer of control on sign and on truth value conditions.

ARITHMETIC IF

The arithmetic IF statement has the following form:

IF (expr) n_1,n_2,n_3

expr	Any arithmetic expression of type integer, real, or double precision.
n_i	The statement label of an executable statement.

The statement labeled n_i must be in the same program unit. On execution of the IF statement, the arithmetic expression expr is evaluated and control transfers to one of the statement labels n_1, n_2, or n_3 according to whether the value of expr is less than zero, zero, or greater than zero, respectively.

LOGICAL IF

The logical IF statement has the following form:

IF (expr) s

expr	Any logical expression.
s	Any executable statement, except a DO statement or logical IF statement.

Upon execution of this statement, the logical expression expr is evaluated. Then, if the value of expr is false, statement s is not executed and control passes to the next executable statement following the logical IF statement. If the value of expr is true, statement s is executed; then the next executable statement following the IF statement is executed, unless s caused a transfer of control.

The K compile option controls how .EQ. and .NE. compares are performed in evaluation of the logical expression in this statement. If the K option has not been selected, only the bits 16-63 are compared. Selection of the K option causes a full word compare to take place during evaluation of the expression.

DO STATEMENT

Execution of a group of statements can be repeated a specified number of times through use of the DO statement. The range of a DO statement is the set of executable statements beginning with the first executable statement following the DO and ending with the terminal statement associated with the DO. A DO statement along with its range is referred to as a DO loop.

DEFINING A DO LOOP

The DO statement has the following form:

DO n i = m_1,m_2,m_3

n	The label of the terminal statement.
i	The control variable, a simple integer variable.
m_1	The initial value parameter of i, an integer constant or a simple integer variable with a value greater than zero.
m_2	The terminal value parameter of i, an integer constant or a simple integer variable with a value greater than zero.
m_3	Optional. The incrementation value parameter for i, an integer constant or a simple integer variable with a value greater than zero. Default value is 1.

The terminal statement of a DO loop can be any assignment statement and almost any input or output statement. However, any control statement other than a CONTINUE is either highly restricted or must not appear as the terminal statement of a DO. The terminal statement must not be any of the following:

A RETURN, STOP, or PAUSE statement

A GO TO statement of any form

A: s Any executable statement except DO, logical IF, block IF, ELSE IF, ELSE, END IF, or END.

B: Block IF, ELSE IF, ELSE, and END IF as described in ANSI X3.9-1978 sections 11.6-11.9.

C: DO n, i = e_1, e_2, e_3

n	The label of the terminal statement. The comma after n is optional.
i	The control variable. The type of i may be any arithmetic type except complex.
e_1	The initial value. e_1 may be any non-complex arithmetic expression.
e_2	The terminal value. e_2 may be any non-complex arithmetic expression.
e_3	Optional incrementation value. If e_3 is omitted, the incrementation value is 1. e_3 may be any non-complex arithmetic expression.

D: delete

E: sel An integer expression

F: . . . selecting expression sel has the value 1, then the statement . . .

G: The IF statements provide for transfer of . . .

H: expr An arithmetic expression of any type other than complex.

I: A RETURN or STOP statement
An unconditional GO TO or assigned GO TO
A block IF, ELSE IF, ELSE, or END IF
An END statement

A special call statement

A DO statement

E A READ statement containing an ERR or END branch

A CALL statement that passes a return label

An arithmetic IF statement

F A logical IF statement containing any of these restricted forms

The terminal statement must physically follow and be in the same program unit as the DO statement that refers to it.

Example:

```
     DO 10 I=1,11,3
     IF(ALIST(I)-ALIST(I+1))15,10,10
  15 ITEMP=ALIST(I)
  10 ALIST(I)=ALIST(I+1)
 300 WRITE(6,200)ALIST
```

The statements following DO up to and including statement 10 are executed four times. The DO loop is executed with I equal to 1, 4, 7, 10. Statement 300 is then executed.

A DO loop can be initially entered only through the DO statement. That is, the group of statements in figure 5-1 are incorrect: The GO TO statement in figure 5-1 transfers control into the range of the DO before the DO statement has been executed.

```
     GO TO 100
     DO 100 I=1,50
 100 A(I)=I
```

Figure 5-1. Incorrect: Entering Range of DO Before DO Execution

Execution of a DO statement causes the following sequence of operations:

A
1. i is assigned the value of m_1.
2. The range of the DO statement is executed.
3. i is incremented by the value of m_3.
4. i is compared with m_2. If the value of i is less than or equal to the value of m_2, the sequence of operations starting at step 2 is repeated. If the value of i is greater than the value of m_2 then the DO is said to have been satisfied, the control variable becomes undefined (has an unpredictable value), and control passes to the statement following the statement labeled n. If m_1 is greater than m_2, the range of the DO is still executed once.

A transfer out of the range of a DO loop is allowable at any time. When such a transfer occurs, the control variable remains defined at its most recent value in the loop. If control eventually is returned to the same range without entering at the DO statement, the statements executed while control is out of the range are said to define the extended range of the DO. The extended range of a DO must not contain a DO that has its own extended range.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO must not be redefined during the execution of the range of that DO. However, the group of statements in figure 5-2 are correct. If ever an element of the array RA is zero or negative, it is set to 1 and the DO statement is reentered, which reinitializes the control variable I.

```
     K=0
     GO TO 300
 200 RA(I)=1.
 300 DO 100 I=1,50
     K=K+1
     IF (RA(I).LE.0.)GO TO 200
 100 RA(I)=K
```

Figure 5-2. DO Control Variable Reinitialization D

NESTING DO LOOPS

When a DO loop contains another DO statement, the grouping is called a DO nest. DO loops can be nested to any number of levels. The range of a DO statement can include other DO statements only if the range of each inner DO is entirely within the range of the containing DO statement. When DO loops are nested, each must have a different control variable.

The terminal statement of an inner DO loop must be either the same statement as the terminal statement of the containing DO loop or must occur before it. If more than one DO loop has the same terminal statement, a branch to that statement can be made only from within the range or extended range of the innermost DO. Figure 5-3 gives an example of an incorrect transfer into the range of an inner DO. Since statement 500 in figure 5-3 is the terminal statement for more than one DO loop, if the first element of any row in array A is less than or equal to zero, the consequent branch to the CONTINUE statement will be an entrance into the range of the inner DO.

If the nested loops in figure 5-3 did not share a terminal statement, or if the outer loop did not reference the terminal statement, the loops would be correctly nested.

```
     DO 500 I=1,5
     IF (A(I,1).LE.0.) GOTO 500
     DO 500 K=1,10
     A(I,K)=SQRT(A(I,K))
 500 CONTINUE
```

Figure 5-3. Example of Incorrect Sharing of Terminal Statement C

CONTINUE STATEMENT

The CONTINUE statement has the following form:

```
CONTINUE
```

The CONTINUE statement performs no operation. It is an executable statement that can be placed anywhere in a program without interrupting the flow of control. The CONTINUE statement is generally used to carry a statement label. For example, it can provide DO loop termination when a GO TO or IF would otherwise be the last statement of the range of the DO. B

A: 1. The initial, terminal, and incrementation values m_1, m_2, m_3 are determined by evaluating the expressions e_1, e_2, e_3 and converting them to the type of i.

2. The DO-variable i is initialized to the value of m_1.

3. The iteration count is determined as

   k = MAX (0, INT $((m_2 + m_3 - m_1)/m_3)$).
   m_3 may not be zero. The iteration count is zero if $m_1 > m_2$ and $m_3 > 0$, or if $m_1 < m_2$ and $m_3 < 0$.

4. The range of the DO is executed k (possibly 0) times. After each execution of the range, i is incremented by m_3.

5. Execution then continues at the first executable statement after the statement labeled n (unless that statement is also the terminal statement of another DO containing this one). Unlike ANSI 66 FORTRAN, the DO-variable i remains defined; its value is now $m_1 + k * m_3$.

B: NESTING OF DO loops AND BLOCKS

A DO loop range, IF block, ELSE IF block, or ELSE block which contains a DO statement must contain all statements in the range of that DO. Likewise, a range or block which contains a block IF, ELSE IF, or ELSE statement must contain all statements in the IF block, ELSE IF block, or ELSE block.

Note that the last statement in the range of a DO may not be a block IF, ELSE IF, ELSE, or END IF statement.

C: NO ZERO TRIP OPTION

The compiler may generate more efficient object code for a DO loop if the No-Zero-Trip option is selected on the FORTRAN control statement. In this case, the test for loop termination will be made at the bottom of the loop rather than the top. Thus a loop will always be executed at least once. This is the way the STAR FORTRAN system functioned prior to the upgrade to ANSI 77.

The 66 option implies the No-Zero-Trip option.

D. The control variable may not be redefined in the range or extended range of the DO. However, variables which appear in the expressions e_1, e_2, e_3 may be redefined without any effect on the DO.

E: delete

F: delete

PAUSE STATEMENT

The PAUSE statement has the following form:

PAUSE n

C n Optional. A string of one to five decimal digits, or a character constant.

B If a string is given, it is displayed in the job dayfile or at the terminal. The string is also placed in the output file for the job. Program execution then continues with the next executable statement following the PAUSE statement. If no string is given, instead of n being displayed and output, the string PAUSE is displayed and output before program execution continues.

STOP STATEMENT

The STOP statement has the following form:

STOP n

n Optional. A string of one to five decimal digits, or a character constant.

Upon execution of the STOP statement, program execution unconditionally terminates and control is returned to the operating system. If a string is given, it is displayed in the job dayfile or at the terminal. The string is also placed in the output file for the job. If no string is given, instead of n being displayed and output, the string STOP is displayed and output.

RETURN STATEMENT

Subroutine and function subprograms contain one or more RETURN statements that when executed cause immediate return of control to the referencing program unit. The RETURN statement must not appear in a main program.

Form:

RETURN n

D n Optional in subroutine subprograms, prohibited in function subprograms. An integer constant or simple integer variable that specifies the nth dummy argument asterisk in the SUBROUTINE or ENTRY statement.

In a function subprogram, execution of a RETURN causes the function value to be returned to the referencing program unit and to be substituted for the most recently executed function reference in that program unit. Evaluation of the expression that contained the function reference continues. The integer n must not appear after a RETURN statement in a function subprogram.

In a subroutine subprogram, when n is not given, execution of a RETURN returns control to the first executable statement following the CALL statement last executed in the calling program unit. When n is given, control returns instead to a statement indicated in the argument list of the CALL statement. The statement label to which control returns is given by the actual argument corresponding to the nth asterisk dummy argument in the SUBROUTINE or ENTRY statement of the called subroutine. If there are fewer than n such statement label arguments or if $n \le 0$, the return is as if n had not been specified (that is, control returns to the first executable statement following the appropriate CALL statement).

CALL STATEMENT

The CALL statement is used to transfer control to a subroutine subprogram, STAR Record Manager module, META subroutine, or any other external subroutine. The execution of a CALL statement is not complete until the subroutine designated in the statement completes execution and returns control to the calling program unit.

Form:

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

s The symbolic name of a subroutine, or an entry point name in a subroutine.

a_i Optional. An actual argument which can be an expression, vector, descriptor, double descriptor, array, external procedure name, or the label of an executable statement in the same program unit (the label is prefixed by an ampersand). When the argument list is omitted, the parentheses and commas must also be omitted. n must equal the number of dummy arguments in the SUBROUTINE or ENTRY statement for s. A

Execution of the CALL statement transfers control to entry point name s. See the heading Passing Arguments Between Subprograms in section 7 for a further description of actual arguments in CALL statements.

Control normally returns to the first executable statement following the CALL statement. However, control can be made to return to some other statement in the program unit by appropriate selection of the CALL statement's actual arguments. If the dummy argument list in the called subroutine contains at least n asterisks, and if the called subroutine contains a RETURN n statement, then upon execution of the RETURN n statement, control returns to the statement having the nth statement label in the CALL statement actual argument list.

For example, the program in figure 5-4 uses both the RETURN n and the RETURN statement formats. If the data read with the READ statement in the subroutine is less than 1.0 or greater than 10.0, then control transfers back to the main program statement having the label 100. A message is printed out and the program terminates. On the other hand, if the data is within the appropriate range, then the subroutine continues executing until the RETURN statement is reached, at which time control transfers back to the main program statement that immediately follows the call to the subprogram.

A: a_i An actual argument. Each actual argument may be an expression, array, external function, intrinsic function, dummy procedure, SHADE subroutine, dynamic variable, dynamic array, dynamic array element, or alternate return specifier. An alternate return specifier is a statement label prefixed by an asterisk or an ampersand.

If there are no actual arguments, the parentheses in the CALL statement are optional.

B: n is displayed in the job dayfile or at the terminal. If n is omitted, PAUSE is displayed. The same x is also displayed at the operator console, and the program waits for operator response. When the operator responds, the response is displayed in the job dayfile or at the terminal. The program execution continues.

C: n One to five decimal digits or . . .

D: . . . in function subprograms. An integer that specifies the . . .

```
      PROGRAM P(INPUT)
          .
          .
          .
      CALL S(A, &100,B)
          .
          .
          .
      STOP
  100 PRINT 2
    2 FORMAT (1X, 'BAD DATA')
      STOP
      END

      SUBROUTINE S (D1,*,D2)
          .
          .
          .
      READ 3,X
    3 FORMAT (F4.1)
      IF (X.LT.1.0 .OR. X.GE.10.0) RETURN 1
          .
          .
          .
      RETURN
      END
```

Figure 5-4. Example of RETURN Statements

A{

DPROD(a_1,a_2)

This computes the double precision product of two real numbers. Valid arguments for DPROD lie in the interval -0.476 854 057 715 93E + 8645 $\leq$ x $\leq$ + 0.476 854 057 715 93E + 8645 (the largest allowable argument value is half of the largest allowable real number). The double precision equivalents of the real numbers are multiplied and a double precision result obtained that is accurate to 94 bits.

DSIGN(a_1,a_2)

This combines the absolute value of one double precision number with the sign of another double precision number; DSIGN(x,y) = 1 x 1 if y $\geq$ 0; DSIGN(x,y) = -1 x 1 if y $<$ 0.

DSIN(a) and DCOS(a)

These compute the sine and cosine of a double precision number expressed in radians. The double precision number modulo 2 pi is used by the functions. The results are double precision numbers in the range -1 to 1, inclusive, and are accurate to approximately 90 bits.

DSINH(a)

This computes the hyperbolic sine of a double precision number and produces a double precision result that is accurate to approximately 90 bits.

DSQRT(a)

This computes the square root of a double precision number greater than or equal to zero and returns a double precision result that is accurate to approximately 90 bits.

DTAN(a)

This computes the tangent of a double precision number expressed in radians. The double précision number modulo 2 pi is used by DTAN. The result is a double precision number that is accurate to approximately 90 bits. Allowable arguments for the DTAN function are in the range -.110 534 964 875 444 D+15 $<$ x $<$ .110 534 964 875 444 D+15.

DTANH(a)

This computes the hyperbolic tangent of a double precision number and returns a double precision result that is accurate to 90 bits.

EXP(a)

This function computes the exponential of a half precision, real, or double precision argument. It is the specific name for computing the exponential of a real argument. The other functions for computing the exponential and CEXP, DEXP, and HEXP. The specific function EXP computes the exponential of a real number. The result, accurate to approximately 45 bits, is a real number greater than or equal to zero.

DSINH(a)

This computes the hyperbolic sine of a double precision number and produces a double precision result that is accurate to approximately 90 bits.

DSQRT(a)

This computes the square root of a double precision number greater than or equal to zero and returns a double precision result that is accurate to approximately 90 bits.

DTAN(a)

This computes the tangent of a double precision number expressed in radians. The double precision number modulo 2 pi is used by DTAN. The result is a double precision number that is accurate to approximately 90 bits. Allowable arguments for the DTAN function are in the range -.110 534 964 875 444 D+15 $< x <$ .110 534 964 875 444 D+15.

DTANH(a)

This computes the hyperbolic tangent of a double precision number and returns a double precision result that is accurate to 90 bits.

EXTEND(a)

This function converts the half precision argument a into a real result.

FLOAT(a)

This converts an integer number to a real number by normalizing the integer number.

HABS(a)

For a half precision argument a, HABS(a) computes the absolute value /a/.

HACOS(a)

This function computes the arccosine of a half precision argument.

HACOS(a) = HALF(ACOS(EXTEND(A)))

HALF(a)

For a of type half precision HALF(a) = a. For a of type integer, real, or double precision this function produces a half precision result equal to a. For a of type complex

HALF(a) = HALF(REAL(a))

There are no specific functions for forming a half precision result.

HASIN(a)

This function computes the arcsine of a half precision argument.

HASIN(a) = HALF(ASIN(EXTEND(a)))

HATAN(a)

This function computes the arctangent of a half precision argument.

HATAN(a) = HALF(ATAN(EXTEND(a)))

HATAN2(a_1,a_2)

This function computes the arctangent of the ratio of two half precision arguments.

HATAN2(a_1,a_2) = HALF(ATAN2(EXTEND(a_1),EXTEND(a_2)))

HCOS(a)

This function computes the cosine of a half precision argument.

HCOS(a) = HALF(COS(EXTEND(a)))

HCOSH(a)

This function computes the hyperbolic cosine of a half precision argument.

HCOSH(a) = HALF(COSH(EXTEND(a)))

HCOTAN(a)

This function computes the cotangent of a half precision argument.

HCOTAN(a) = HALF(COTAN(EXTEND(a)))

HDIM(a_1,a_2)

This function computes the positive excess of one half precision number over another half precision number. HDIN(a_1,a_2) is $a_1 - a_2$ if $a_1 > a_2$, otherwise it is zero.

HEXP(a)

This function computes the exponential of a half precision argument.

HEXP(a) = HALF(EXP(EXTEND(a)))

HINT(a)

This function computes [a], where [a] is the sign of a times the largest integer less than or equal to /a/.

HLOG(a)

This function computes the natural logarithm of a half precision argument.

HLOG(a) = HALF(ALOG(EXTEND(a)))

HLOG10(a)

This function computes the common logarithm of a half precision argument.

HLOG10(a) = HALF(ALOG10(EXTEND(a)))

HMAXI(a_1,a_2,...)

This function searches the list of half precision arguments for the element having the maximum value and returns this value.

HMINI(a_1,a_2,...)

This function searches the list of half precision arguments for the element having the minimum value and returns this value.

HMOD(a_1,a_2)

This function computes one half precision number modulo a second half precision number. For a_1 and a_2 of type half precision.

HMOD(a_1,a_2) = a_1 - a_2 *HINT(a_1/a_2)

HNINT(a)

This function computes the nearest whole number to a. Both the argument and result are of type half precision. Note that for a half precision argument a

HNINT(a) = ANINT(a)

HPROD(a_1,a_2)

This computes the single precision product of two half precision numbers.

HSIGN(a_1,a_2)

This function combines the absolute value of one half precision number with the sign of another half precision number.

HSIGN(a_1,a_2) = $|a_1|$ if $a_2 \geq 0$,
HSIGN(a_1,a_2) = $-|a_1|$ if $a_2 < 0$.

HSIN(a)

This function computes the sine of a half precision argument.

HSIN(a) = HALF(SIN(EXTEND(a)))

HSINH(a)

This function computes the hyperbolic sign of a half precision argument.

HSINH(a) = HALF(SINH(EXTEND(a)))

HSORT(a)

This function computes the square root of a half precision number using the machine instruction Q8SORT.

HTAN(a)

This function computes the tangent of a half precision argument.

HTAN(a) = HALF(TAN(EXTEND(a)))

HTANH(a)

This function computes the hyperbolic tangent of a half precision argument.

HTANH(a) = HALF(TANH(EXTEND(a)))

IABS(a)

For an integer number x, IABS(x) computes the absolute value /x/.

ICHAR(a)

For a of type character, this function returns the integer corresponding to the ASCII code for a. For example

ICHAR('A') = X'41'

IDIM(a_1,a_2)

This computes the positive excess of one integer number over another integer number. IDIM(x,y) returns the value x-y if x is greater than or equal to y, and returns a value of 0 otherwise.

IDINT(a)

For a double precision number x, IDINT(x) computes [x], where [A] is the sign of A times the largest integer less than or equal to /A/.

IDNINT(a)

This function computes the integer nearest to the double precision argument a. For example

IDNINT(4.1D+00) = 4
and
IDNINT(-4.1D+00) = -4

IFIX(a)

This function converts the real argument a into an integer and is an alternative name for the specific function usage of INT.

IHINT(a)

For a half precision number a, IHINT(a) computes [a], where [a] is the sign of a times the largest integer less than or equal to /a/.

IHNINT(a)

This function computes the integer nearest to the half precision argument a.

INDEX(a_1,a_2)

This function returns an integer value indicating the starting position within the character string a_1 of a substring identical to a_2. If a_2 occurs more than once in a_1, the starting position of the first occurrence is returned.

If a_2 does not occur in a_1, the value zero is returned. Note that zero is returned if LEN(a_1) $<$ LEN(a_2).

INT(a)

For a of type integer INT(a) = a. For a of type half precision, real, or double precision, there are two possible results. If /a/ $<$ 1 then INT(a) = 0. If /a/ $\geq$ 1, INT(a) is the integer whose magnitude does not exceed the magnitude of a and whose sign is the same as the sign of a. For example

INT(-3·7) = -3

For a of type complex, INT(a) is the value obtained by applying the above rule to the real part of a. INT is the specific name for converting a real argument to an integer. The other specific functions which convert their argument to an integer are IDINT, IFIX, and IHINT.

ISIGN(a_1,a_2)

This combines the absolute value of one integer number with the sign of another integer number.

ISIGN(a_1,a_2) = /a_1/ if $a_2 \geq 0$
ISIGN(a_1,a_2) = -/a_1/ if $a_2 < 0$

LEN(a)

This function returns the number of characters in the character argument a.

LGE(s_1,s_2)

This function returns the result .TRUE. or .FALSE. depending on the relation between the character entities s_1 and s_2. If a_1 and a_2 are corresponding characters in s_1 and s_2 then

LGE(s_1,s_2) = .TRUE. if ICHAR(a_1) $\geq$ ICHAR(a_2) for all a_1 and a_2

$LGT(s_1,s_2)$

This function returns the result .TRUE. or .FALSE. depending on the relation between the character entities s_1 and s_2. If a_1 and a_2 are corresponding characters in s_1 and s_2 then

$LGT(s_1,s_2)$ = .TRUE. if $ICHAR(a_1) > ICHAR(a_2)$ for all a_1 and a_2

$LLE(s_1,s_2)$

This function returns the result .TRUE. or .FALSE. depending on the relationship between the character entities s_1 and s_2. If a_1 and a_2 are corresponding characters in a_1 and a_2 then

$LLE(s_1,s_2)$ = .TRUE. if $ICHAR(a_1) \leq ICHAR(a_2)$ for all a_1 and a_2

$LLT(s_1,s_2)$

This function returns the result .TRUE. or .FALSE. depending on the relationship between the character entities s_1 and s_2. If a_1 and a_2 are corresponding characters in s_1 and s_2 then

$LLT(s_1,s_2)$ = .TRUE. if $ICHAR(a_1) < ICHAR\ (a_2)$ for all a_1 and a_2

LOG(a)

This function computes the natural logarithm for a half precision, real, double precision or complex argument. The specific function names are ALOG, CLOG, DLOG, and HLOG.

LOG10(a)

This function computes the common logarithm of a half precision, real, or double precision argument. The specific function names are ALOG10, DLOG10, and HLOG10.

$MAX(a_1,a_2,...)$

This function searches the list of integer, half precision, real, or double precision numbers for the list element having the maximum value and returns this value. The type of the result is the same as the type of the arguments. The specific function names are AMAX1, DMAX1, HMAX1, and MAX0.

$MAX0(a_1,a_2,...)$

This searches a list of integer numbers for the list element having the maximum value and returns that value.

$MAX1(a_1,a_2,...)$

This searches a list of real numbers for the list element having the maximum value. The selected real number is converted with IFIX before being returned.

$MIN(a_1,a_2,...)$

This function searches the list of integer, half precision, real, or double precision numbers for the list element having the minimum value and returns this value the type of the result is the same as the type of the arguments. The specific function names are AMIN1, DMIN1, HMIN1, and MIN0.

MIN0(a_1,a_2,...)

This searches a list of integer numbers for the list element having the minimum value and returns the integer when found.

MIN1(a_1,a_2,...)

This searches a list of real numbers for the list element having the minimum value. The selected real number is converted with IFIX before being returned.

MOD(a_1,a_2)

This function computes one number modulo a second number. Both numbers must must be of the same type which may be integer, half precision, real, or double precision. It is the specific function name for compiling one integer modulo another integer. The other specific names for this function are AMOD, DMOD, and HMOD.

NINT(a)

This function compiles the nearest integer to the specified half precision, real, or double precision argument. It is the specific function name for computing the nearest integer to a real number. Examples:

NINT(3·5) = INT(3·5 + 0·5) = 4
NINT(-0·1) = INT(-0·1 -0·5) = 0

The other specific function names for computing the nearest integer are IDNINT and IHNINT.

Q8SCNT(v)

This counts the number of 1 bits in a bit vector. The result returned is an integer.

Q8SDFB(a,b)

This tests the bits in the data flag branch register, given a pair of integer constants (x,y), where x indicates the bit to be tested, and y is an indicator that can assume one of the following values:

0 means the bit tested is not to be altered.
1 means the bit tested is to be set to 0.
2 means the bit tested is to be set to 1.
3 means the bit tested is to be toggled (that is, if 1, set to 0, and if 0, set to 1).

Bit x in the data flag branch register is tested and a logical result of .TRUE. or .FALSE. returned, depending on whether bit x is 1 or 0. Action is also taken according to the indicator y.

Example:

Given

the 10th bit in the DFB register is 1
x = 9
y = 3

the value of Q8SDFB(x,y) is .TRUE., and the 10th bit in the DFB register, since it is 1, is set to 0.

$\underline{\text{Q8SDOT}(v_1,v_2)}$

This calculates the dot product of two vectors having the same length and data type. Q8SDOT produces a scalar result that has the same data type as its arguments.

For given vectors x and y, the procedure for calculating the dot product is as follows. Corresponding elements in x and y are multiplied together, and the sum of the resulting products is taken.

Example:

Given

```
      x = 0 1 3 200
      y = 2 2 2 0
the value of Q8SDOT(x,y) is
      (0 * 2) + (1 * 2) + (3 * 2) + (200 * 0) = 8
```

$\underline{\text{Q8SEQ}(v_1,v_2)}$

From among the pairs of corresponding elements in two real, half precision, or integer vectors, Q8SEQ selects the first pair of elements that are equal. (A vector and a scalar is an alternative to the two vectors.) The result is an integer scalar.

Q8SEQ(x,y) compares the corresponding elements of vectors x and y, beginning with the first element of x and the first element of y, until a pair is found that has equal elements, or until all elements in the vectors have been compared. The value returned is the number of unsuccessful compares that were made. A scalar x or y is considered to be a vector of the appropriate length with every element being the scalar value.

Example:

Given

```
      x = 0 1 4 5 4
      y = 1 0 3 5 4
the value of Q8SEQ(x,y) is 3.
```

Q8SEXTB(a,m,n)

This extracts m bits, beginning with bit n of a. The result if right-justified in a 64-bit word with zero fill. The m and n values are integer. Bits in the word are numbered from left to right, beginning with zero.

$\underline{\text{Q8SGE}(v_1,v_2)}$

This is identical to Q8SEQ, except that Q8SGE searches for an element in x that is greater than or equal to the corresponding element in y.

Q8SINSB(a,m,n,b)

This produces a word into which bits have been inserted. The result is equal to b, except that m bits, beginning with bit n, are replaced by the m rightmost bits of a. The argument b is not altered. The m and n values are integer. Bits in the word are numbered from left to right, beginning with zero.

Q8SLEN(v)

This counts the number of elements in a half precision, real, integer, or complex vector, or the number of elements in the value vector part of a real or integer sparse vector. The result returned is an integer. For a complex vector, the number of elements is half the number of words.

Q8SLT(v_1,v_2)

This is identical to Q8SEQ, except that Q8SLT searches for an element in x that is less than the corresponding element in y.

Q8SMAX(v) or Q8SMAX(v,c)

This selects the maximum from among the elements in a half precision, real or integer vector, or only those elements selected by an optional bit control vector. The result is a scalar that has the same data type as the function argument.

For a given vector x and a bit control vector c, the procedure for selecting the element having the maximum value is the same as for Q8SMIN, except that the maximum rather than the minimum is selected.

Example:

The elements in x, as presented to Q8SMAX, might be:

x = 2 3 19 6 -1

When only x is presented to Q8SMAX for evaluation, the function selects the element from among all of the elements of x:

Q8SMAX = 19

A bit mask presented as argument c might appear as:

c = 0 1 0 1 1

When a bit in c is zero, it inhibits the inclusion of the corresponding element of x in the evaluation of the function. Therefore, if the argument list for Q8SMAX includes c, the function result would be:

Q8SMAX = 6

Q8SMAX(v) or Q8SMAXI(v,c)

Like Q8SMAX, this finds the maximum from among the elements in a half precision or real vector or only those elements selected by an optional bit control vector. However Q8SMAXI returns not the value itself but, instead, a count of the number of elements preceding, but not including, the element having the maximum value.

The procedure for selecting the element having the maximum value is the same for Q8SMAXI as for Q8SMAX. The control vector bits that are set to zero (when the control vector is present) have no effects on the count returned by Q8SMAXI. The action of the control vector is the same for both functions in all other respects.

Example:

> The example given for Q8SMAX is an example for Q8SMAXI as well, except that where Q8SMAX equals 19 or 6, depending on the presence of the bit control vector argument, Q8SMAXI would return 2 and 3 respectively.

Q8SMIN(v) or Q8SMIN(v,c)

This selects the minimum from among the elements in a half precision, real or integer vector, or from among only those elements selected by an optional bit control vector. The result is a scalar that has the same data type as the vector.

For a given vector x and a bit control vector c, the procedure for selecting the element having the minimum value is as follows. When c is not present, the minimum value in x is selected. If c is present, it acts as a binary mask; each element in c that is set to 1 permits the corresponding element in x to be included in the function evaluation, whereas each element in c that is set to 0 causes the corresponding element in x to be excluded from the evaluation.

Example:

The elements in x, as presented to Q8SMIN, might be:

x = 2 3 19 6 -1

When only x is presented to Q8SMIN for evaluation, the function selects the element from among all of the elements of x:

Q8SMIN = -1

A bit mask presented as argument c might appear as:

c = 1 0 1 1 0

When a bit in c is zero, it inhibits the inclusion of the corresponding element of x in the evaluation of the function. Therefore, if the argument list for Q8SMIN includes c, the function result would be:

Q8SMIN = 2

Q8SMINI(v) or Q8SMINI(v,c)

Like Q8SMIN, this finds the minimum from among the elements in a real vector or only those elements selected by an optional bit control vector. However, Q8SMINI returns not the value itself but, instead, a count of the number of elements preceding, but not including, the element having the minimum value.

The procedure for selecting the element having the minimum value is the same for Q8SMINI as for Q8SMIN. The control vector bits that are set to zero (when the control vector is present) have no effect on the count returned by Q8SMINI. Otherwise, the action of the control vector is the same for both functions

Example:

> The example given for Q8SMIN is an example for Q8SMINI as well, except that where Q8SMIN equals -1 or 2, depending on the presence of the bit control vector argument, Q8SMINI would return 4 and 0 respectively.

<u>Q8SNE(v_1,v_2)</u>

This is identical to Q8SEQ, except that Q8SNE searches for an element in x that is not equal to the corresponding element in y.

<u>Q8SPROD(v) or Q8SPROD(v,c)</u>

This calculates the product of the elements in a half precision or real or integer vector, or only those elements selected by an optional bit control vector. A scalar result is produced that has the same data type as the vector.

For a given vector x and a bit control vector c, the procedure for calculating the product is as follows. When c is not present, the product of all of the elements in x is computed. If c is present, it acts as a binary mask; each element in c that is set to 1 permits the corresponding element in x to be included in the product, while each element in c that is set to 0 causes the corresponding element in x to be excluded from the computation. If c is all zero, the result of Q8SPROD is one.

Example:

```
    The elements in x, as presented to Q8SPROD, might be:
        x = 2 1 4 3
    When only x is given to Q8SPROD for evaluation, the function calculates the product of all
    the elements to obtain the evaluation:
        Q8SPROD = 2 * 1 * 4 * 3 = 24
    A bit mask presented as argument c might appear as:
        c = 0 0 1 1
    When a bit in c is zero, it inhibits the inclusion of the corresponding element of x in the
    function evaluation. Therefore, the function result if c is present would be:
        Q8SPROD = 4 * 3 = 12
```

<u>Q8SSUM(v) or Q8SSUM(v,c)</u>

This sums the elements in a half precision, real or integer vector, or only those elements selected by an optional bit control vector. A scalar result is produced that has the same data type as the vector.

For a given vector x and a bit control vector c, the procedure for calculating the sum is as follows. When c is not present, the arithmetic sum of all of the elements in x is taken. If c is present, it acts as a binary mask; each element in c that is set to 1 permits the corresponding element in x to be included in the sum, while each element in c that is set to 0 causes the corresponding element in x to be excluded from the summation. If c is all zero, the result of Q8SSUM(x,c) is zero.

Example:

```
    The elements in x, as presented to Q8SSUM, might be:
        x = 2 1 4 3
```

When only x is presented to Q8SSUM for evaluation, the function sums all of the elements to obtain the evaluation:

Q8SSUM = 2 + 1 + 4 + 3 = 10

A bit mask presented as argument c might appear as:

c = 1 0 1 1

When a bit in c is zero, it inhibits the inclusion of the corresponding element of x in the evaluation of the function. Therefore, the function result, if c is present, would be:

Q8SSUM = 2 + 4 + 3 = 9

<u>Q8VADJM(v)</u>

This computes the averages of adjacent elements of the half precision or real input vector. For a given real vector x, Q8VADJM(x;r) forms the nth element of the result vector r by adding the nth and (n+1)th elements of x and dividing the sum by 2. That is, $r_n = (x_n + x_{n+1})/2$, where the result vector r is one element shorter than the input vector x.

Example:

Given

x = 5. 3. 5. 3. 5. 4. 5. 3.

the result vector r for Q8VADJM(x) is

r = 4. 4. 4. 4. 4.5 4.5 4.

<u>Q8VARCMP(v_1,v_2)</u>

This deletes from a half precision, real or integer vector any element having a value below the threshold value provided by the corresponding element of another vector. The lengths and data types of the real or integer vector and the threshold vector must be the same. The result is a sparse vector.

Q8VARCMP(x,t;r) creates the result sparse vector as follows. For each element of x, if the element value is less than the value of the corresponding element in the threshold vector 5, a 0 bit is placed in the order vector of the result sparse vector. If the element value is greater than or equal to the value of the corresponding element in the threshold vector, the element is placed in the result value vector and a 1 bit is placed in the result order vector. Evaluation proceeds from first to last element of the vector x.

The length of x governs the operation. If t is shorter than x, t is in effect extended with zeros. If t is longer than x, the excess elements are ignored.

The initial lengths of the value vector and order vector components of r are ignored. Upon completion of the operation, the length of the order vector component of r is that of x, and the length of the value vector component of r is that if the number of 1 bits in the order vector.

Example:

Given

x = 10 11 44 11 9 -1 0 50

t = 10 10 10 8 10 10 10 10

the value of Q8VARCMP(x,t) is the sparse vector

r = value vector: 10 11 44 11 50

order vector: 1 1 1 1 0 0 0 1

Q8VAVG(v_1,v_2)

This computes the averages of corresponding elements of two half precision or real input vectors. A vector and a scalar is an alternative to a pair of vector arguments.

For given real vectors x and y, Q8VAVG(x,y;r) forms the n^{th} element of the result vector r by adding the n^{th} element of x and the n^{th} element of y, then dividing the sum by 2 (that is $r_n = (x_n + y_n)/2$). The vectors x, y, and r all have the same length. A scalar x or y is considered to be a vector of the appropriate length with every element being the scalar value.

Example:

Given

x = 1.

y = 9.3 10.4 18. 8.91 0.1

the value of Q8VAVG(x,y) is the vector

f = 5.15 5.7 9.5 4.955 0.55

Q8VAVGD(v_1,v_2)

This computes the average differences of corresponding elements of the two input vectors. A vector and a scalar is the alternative to the two input vectors.

For given real vectors x and y, Q8VAVGD(x,y;r) forms the n^{th} element of the result vector r by subtracting the n element of y from the n^{th} element of x, then dividing the difference by 2 (that is, $r_n = (x_n - y_n(/2)$. The vectors x, y, and r all have the same length. A scalar x or y is considered to be a vector of the appropriate length with every element being the scalar value.

Example:

Given

x = 100. 100. 100. 100. 100.

y = 4. 9. 9. 15. 14.

the value of Q8VAVGD(x,y) is the vector

r = 48. 45.5 45.5 42.5 43.

Q8VCMPRS(v,c)

This deletes selected elements from a half precision or real or integer vector under control of a bit control vector.
For a given real vector x and control vector c, the deletion procedure is as follows: every value in the vector x whose position corresponds to that of a 0 in the bit vector c is deleted, leaving for the result vector only those values in the vector x whose positions correspond to those of 1s in the bit vector c. The length of the result vector will be the number of 1s in c.

Example:

Given

x = 4 5 5 4 4 0

c = 0 1 1 0 0 0

the value of Q8VCMPRS(x,c) is the vector

r = 5 5

Q8VCTRL(v,c)

This changes the values of only selected elements in a half precision or real or integer result vector, using the elements in another vector of the same data type to provide the new values. Selection of values is performed with a bit control vector.

For a given real or integer vector y (the result vector), a vector x of the same data type as y, and a control vector c, the procedure for modifying y is as follows. Any element in the vector x that corresponds to a 1 in the control vector c is directly assigned to the corresponding element in the result vector y. All other elements in y (the elements that correspond to 0s in c) retain whatever values they had before.

Example:

Given

x = 5 55 19 9 40

c = 0 0 0 1 0

y = 9 9 9 10 9

the value of Q8VCTRL(x,c) is the vector

y = 9 9 9 9 9

Q8VDELT(v)

This computes the differences between the adjacent elements of the input vector. For a given real vector x, Q8VDELT(x) computes the nth element of the result vector r by subtracting the nth element of x from the (n+1)th element of x. That is, $r_n = (x_{n+1} - x_n)$, where the result vector r is one element shorter than the input vector x.

Example:

Given

x = 5. 3. 5. 3. 5. 4. 5. 3.

the result vector r for Q8VDELT(x) is

r = 2. 2. -2. 2. -1. 1. -2.

Q8VEQI(v_1,v_2)

The effect of a call to Q8VEQI is identical to that of issuing a series of Q8SEQ calls in which one of the arguments for Q8SEQ is a half precision or real scalar. For given real vectors x and y, Q8VEQI(x,y) performs a search iteration for each element of x, beginning with the first element of x. A search iteration consists of comparisons of the element of x with successive elements of y, beginning with the first element of y, until an element of y is found which is equal to the element of x or until the element of x has been compared with every element of y. The result of the nth iteration, which is performed using the nth element of x and which is a count of the number of unsuccessful compares that were made on this iteration, is placed in the nth element of r.

Example:

```
     Given
          x = 0. 1. 4. 5. 4.
          y = -1. 0. 3. 5. 4.
     the value of Q8VEQI(x,y) is the vector
          r = 1 5 4 3 4
```

<u>Q8VGATHR(v,i)</u>

This creates a half precision or real or integer vector, using the elements in another vector of the same data type to provide the values. Selection of values is performed with an integer index vector.

For a given real or integer vector x and an index vector i, the procedure for constructing the result vector is as follows. A 1 in i indicates that the corresponding element in the result vector is to be assigned the value of the first element in x, a 2 in i indicates that the corresponding element in the result vector is to be assigned the value of the second element in x, and so on. The value of any one element in x can be assigned to more than one element in the result vector, and not every element in x need be used. The index vector and the result vector must be the same length.

Example:

```
     Given
          x = 10 19 11 15 0 9 3
          i = 7 6 5 6 3 1 1
     the value of Q8VGATHR(x,i) is the vector
          r = 3 9 0 9 11 10 10
```

<u>Q8VGEI(v_1,v_2)</u>

This is identical to Q8VEQI, except that Q8VGEI searches for an element in y that is greater than or equal to the element in x which is of concern for the current iteration.

<u>Q8VINTL(a_1,a_2)</u>

This forms a half precision or real or integer vector whose adjacent elements have values differing by a specified interval. For given constant scalars x and y, both integer or both real, Q8VINTL(x,y) creates the vector r as follows. The first element of r is assigned the value x. Each succeeding element of r is assigned a value arrived at by adding the constant y to the preceding element's value (that is, $r_n = r_{n-1} + y$). When r is filled the calculations cease.

Example:

Given

x = 0.0

y = 6.7

length of r = 12

the value of Q8VINTL(x,y) is the vector

r = 0.0 6.7 13.4 20.1 26.8 33.5 40.2 46.9 53.6 60.3·67.0 73.7

Q8VLTI(v_1,v_2)

This is identical to Q8VEQI, except that Q8VGEI searches for an element in y that is less than the element in x which is of concern for the current iteration.

Q8VMASK(v_1,v_2,c)

Q8VMASK(x,y,c) creates a result vector, each element of which is the corresponding element of one of the vectors x and y (one or both of x and y can alternatively be scalar). The arguments (x and y only) and the result vector must all have the same data type.

For given vectors x and y, and a bit control vector c, the result vector is created as follows. If an element is c is 1, then the corresponding element in vector x is placed in the corresponding position in the result vector. If an element in c is 0, then the corresponding element in vector y is placed in the corresponding position in the result vector. A scalar x or y is considered to be a vector of the appropriate length with every element being the scalar value.

The length of c governs the operation; the lengths of x and y are ignored and the length of r is set to that of c.

Example:

Given

x = 1 2 3 1 2 3 1 2 3

y = 19

c = 1 1 0 1 1 0 1 1 0

the result vector r for Q8VMASK(x,y,c) is

r = 1 2 19 1 2 19 1 2 19

Q8VMERG(v_1,v_2,c)

This merges the elements in two half precision, two real or two integer vectors, under control of a bit control vector, into a single result vector. Q8VMERG(x,y,c) merges x and y as follows. If an element in c is 1, then the corresponding position in the result vector is assigned the first element from x that has not already been selected. If an element in c is 0, then the corresponding position in the result vector is assigned the first element from x that has not already been selected. If an element in c is 0, then the corresponding position in the result vector is assigned the first element from y that has not already been selected. Control vector c is scanned in this way from first to last element. The merge stops when the result vector is full, even when there are unmerged elements remaining in x and y.

The length of c governs the operation; the lengths of x and y are ignored and the length of r is set to that of c.

Example:

Given

x = 10 11 12 14 13

y = 5 4 3 2 1

c = 1 1 0 0 1

the value of Q8VMERG(x,y,c) is the vector

r = 10 11 5 4 12

Q8VMKO(a_1,a_2)

This forms a bit vector whose elements are either all zeros or else a repeated pattern of ones and zeros, beginning with a one. For given integer constants x and y, Q8VMKO(x,y;r) creates the elements of the vector r as follows. The pattern, which consists of a string of x ones followed by a string of y-x zeros, is repeated until the result vector r has been filled. The length of r need not be divisible by y.

Example:

Given

x = 3

y = 6

length of r = 10

the value of Q8VMKO(x,y) is the bit vector

r = 1110001110

Q8VMKZ(a_1,a_2)

This forms a bit vector whose elements are either all ones or else a repeated pattern of ones and zeros, beginning with a zero. For given integer constants x and y, Q8VMKZ(x,y) creates the elements of the vector r as follows. The pattern, which consists of a string of x zeros followed by a string of y-x ones, is repeated until the result vector r has been filled. The length of the result vector r need not be divisible by y.

Example:

Given

x = 7

y = 25

length of r = 10

the value of Q8VMKZ(x,y) is the bit vector

r = 0000000111

Q8VNEI(v_1,v_2)

This is identical to Q8VEQI, except that Q8VNEI searches for an element in y that is not equal to the element in x which is of concern for the current iteration.

Q8VPOLY(v_1,v_2)

This computes a polynomial at several values. For given half precision or real vectors x and y, Q8VPOLY(x,y) is evaluated as follows (x can also be a scalar). The input vector y contains the coefficients of the polynomial: the first element of the vector y contains the coefficient of the highest order term of the polynomial and the last element of the vector y contains the lowest order term of the polynomial (the constant). The length of the vector y determines the order of the polynomial: if n is the length of y, the order of the polynomial is n-1. The polynomial is evaluated for each element of x and the result is placed in the corresponding element in the result vector r. If y is a scalar rather than a vector, the result r must be referenced as a vector with length equal to 1, not as a scalar.

Example:

Given

x = -2 -1 1 2 3

y = 10 3 2

the value of Q8VPOLY(x,y) is the vector

r = 36 9 15 48 101

The elements of r are computed as follows:

$r(1) = 10(-2^2) + 3(-2) + 2 = 36$

$r(2) = 10(-1^2) + 3(-1) + 2 = 9$

$r(3) = 10(1^2) + 3(1) + 2 = 15$

$r(4) = 10(2^2) + 2(2) + 2 = 48$

$r(5) = 10(3^2) + 3(3) + 2 = 101$

Q8VREV(v)

This reverses the order of the elements in a half precision or real or integer vector, by transmitting the elements of the input vector in reverse order to the result vector.

Example:

Given

x = 4 3 5 6 9 10

the value of Q8VREV(x) is the vector

r = 10 9 6 5 3 4

Q8VSCATR(v,i)

This changes the values of only selected elements in a half precision, real or integer result vector, using the elements in another vector of the same data type to provide the new values. Selection of values is performed with an integer index vector.

For a given real or integer vector y (the result vector), a vector x of the same data type as y, and an index vector i, the procedure for modifying y is as follows. A 1 in i indicates that the corresponding element in x is to be assigned to the first position in y, a 2 in i indicates that the corresponding element in x is to be assigned to the second position in y, and so on. More than one value assignment can be made to be so defined. Elements in y that are not given a value retain the values they already had.

If x is shorter than i, then x is extended with zeros to match the length of i.

Example:

Given

```
    x = 0 50 -1 60 70
    i = 1 2 1 5 5
    y = 9 9 9 9 9
```

the vector y passes through the following five stages during the computation of Q8VSCATR(x,i)

```
    y = 0 9 9 9 9
    y = 0 50 9 9 9
    y = -1 50 9 9 9
    y = -1 50 9 9 60
    y = -1 50 9 9 70
```

and the result is the vector

```
    y = -1 50 9 9 70
```

<u>Q8VXPND(v,c)</u>

This inserts additional elements having the value 0 (or 0.0) into a half precision or real or integer vector, under control of a bit control vector. The effect of the procedure is as though a Q8VMERG(x,n,c) had been performed, where n is a vector of zeros, and x, c, and y are the real or integer vector, the control vector, and the result vector respectively.

The length of c governs the operation; the length of x is ignored and the length of y is set to that of c.

Example:

Given

```
    x = 5 5
    c = 0 1 1 0 0 0
```

the value of Q8VXPND(x,c) is the vector

```
    r = 0 5 5 0 0 0
```

<u>RANF</u>

This returns a random number. It has no argument. The multiplicative congruential method modulo 2**47 is used to generate the next random number in the sequence.

$$x_{n+1} = (a * x_n) \bmod 2**47$$

The value of the multiplier a is X'0000 4C65 DA2C 866D'. The seed can be obtained and reset with the subroutines RANGET and RANSET, respectively. The default value of the seed is X'0000 54F4 A3B9 33BD'. A vector of random numbers can be returned with the subroutine VRANF.

REAL(a)

For a of type real REAL(a) = a. For a of type integer, half or double precision, REAL(a) is as much precision of the significant part of a as a real number can obtain. For a of type complex, REAL(a) is the real part of a. This function is the specific name for conversion of an integer to real and for an integer argument REAL(a) = FLOAT(a). The other specific functions for conversion to real are EXTEND and SNGL.

SECOND

This queries the system as to how much CPU time in seconds has elapsed since the job started. The result is a real number expressing the time in seconds, accurate to within one microsecond. This function has no argument.

SIGN(a_1,a_2)

This function combines the sign of one argument with the absolute value of the other. Both arguments must be of the same type which may be integer, half precision, real, or double precision. It is the specific function name for transferring the sign between two real numbers.

Examples:

```
SIGN(-2.0,2.5) = 2.0
SIGN(-10.0,0.0) = 10.0
SIGN(3.4,-7.0) = -3.4
```

The other specific function names for transferring sign are DSIGN, HSIGN, and ISIGN.

SIN(a)

This computes the sine of a half precision, real, double precision or complex argument. It is the specific function name for computing the sine of a real argument. The other specific functions which compute sines are CSIN, DSIN and HSIN.

The specific functions SIN and COS calculate the sine and cosine of a real argument.

SINH(a)

This function computes the hyperbolic sine of a half precision, real, or double precision argument. It is the specific name for computing the hyperbolic sine of a real argument. The other specific functions which compute hyperbolic sines are DSINH and HSINH.

The specific function SINH computes the hyperbolic sine of a real number and produces a real result that is accurate to 47 bits.

SNGL(a)

This converts a double precision number to a real number by retaining only the most significant part (the first word) of the double precision number.

SORT(a)

This function computes the square root of a half precision, real, double precision or complex argument. It is the specific name for computing the square root of a real argument and the machine instruction SQRT, is used in this case. The other specific functions which compute square roots are CSQRT, DSQRT, and HSQRT.

TAN(a)

This function computes the tangent of a half precision, real, or double precision number. It is the specific name for computing the tangent of a real argument. The other specific functions which compute tangents are DTAN and HTAN.

The specific function TAN computes the tangent of a real number expressed in radians. The function first reduces its argument modulo 2 pi. The result is a real number that is accurate to approximately 45 bits. The valid arguments for TAN lie in the interval

$$-0.276\ 334\ 121\ 886E + 14 \leq x \leq + 0.276\ 334\ 121\ 886E + 14$$

Note that

$$(2^{46}-1) * pi/8 = 0.276\ 334\ 121\ 886E \pm 14$$

TANH(a)

This function computes the hyperbolic tangent of a half precision, real, or double precision argument. It is the specific function name for computing the hyperbolic tangent of a real argument. The other specific functions which compute hyperbolic tangents are DTANH and HTANH.

The specific function TANH computes the hyperbolic tangent of a real number expressed in radians. It produces a result that is in the range -1 through 1, inclusive, and which is accurate to approximately 45 bits.

TIME

This queries the system as to the time of day, and returns a result of type CHARACTER *8 in the following format:

hh:mm:ss

hh	Pair of decimal digits expressing the hour.
mm	Pair of decimal digits expressing the minute.
ss	Pair of decimal digits expressing the second.

This function has no argument.

VABS(v)

For each element x in a real vector, VABS computes the absolute value (x). The real result is accurate to 47 bits.

VACOS(v)

This computes the arccosine of each element in a real vector. The result real vector contains elements that are accurate to approximately 45 bits.

VAIMAG(v)

This constructs a real vector from the imaginary parts of a complex vector. For each element of the complex vector, if x+iy is the complex element, y is assigned to the result vector. Accuracy of the result is 47 bits.

VAINT(v)

For each element x in a real vector, VAINT computes (x) and converts it to real before assigning it to a real vector. (A) is the sign of A times the largest integer less than or equal to (A). The real results are accurate to 47 bits. The effect of VAINT on each x is that of the expression AINT(x).

VALOG(v)

This computes the natural logarithm of each element in a real vector. VALOG returns a result vector of real numbers that are each accurate to approximately 45 bits.

For a given real number x, VALOG(x) is computed as described for the function ALOG.

VALOG10(v)

This computes the logarithm of each element in a real vector, returning a result vector of real numbers accurate to approximately 45 bits.

VAMOD(v_1,v_2)

For each pair of corresponding elements in two real vectors, this computes one real number modulo the second real number to produce a real result that is assigned to the real result vector. For each pair of elements (x,y),x-(x/y) * y is computed, where (A) is the sign of A times the largest integer less than or equal to (A).

VANINT(v)

For each element x of the real vector v, VANINT computes ANINT(x).

VASIN(v)

This computes the arcsine of each element in a real vector. The magnitude of the error that is introduced into the results because a table lookup technique is used for fast computation of VASIN is approximately 2^{-45}.

VATAN(v)

This computes the arctangent of each element in a real vector. The magnitude of the error that is introduced into the results because a table lookup technique is used for fast computation of VATAN is approximately 2^{-45}.

VATAN2(v_1,v_2)

This computes the arctangent of the ratio of two real elements in corresponding positions in two real vectors. The result is a real vector having elements that are accurate to approximately 45 bits.

VCABS(v)

This computes the modulus of each element in a complex vector, and places the results in a real result vector. Each result is accurate to approximately 45 bits.

VCCOS(v)

See VCSIN for a description of the VCCOS function.

VCEXP(v)

This computes the exponential of each element in a complex vector, and produces a complex vector of results.

VCLOG(v)

This computes the natural logarithm of each element in a complex vector, returning a complex result vector.

VCMPLX(v_1,v_2)

This constructs a complex vector from two real vectors. For each pair of corresponding elements (x,y) in the two real vectors, x is assigned to the real part and y is assigned to the imaginary part of the corresponding element in the complex result vector. Accuracy of the result is 47 bits for each part of the complex value.

VCONJG(v)

This constructs a vector of conjugates from a complex vector. For each element x+iy of the complex vector, x-iy is assigned to the result vector. The function sets up a control vector of ones and zeros, copies the real parts of the complex vector and negates the imaginary parts before assigning them.

VCOS(v)

See VSIN for a description of the VCOS function.

VCSIN(v) and VCCOS(v)

These compute the sine and cosine of each element in a complex vector. Each complex result is accurate to approximately 45 bits.

VCSQRT(v)

This computes the square root of each element in a complex vector, and places the results in a complex result vector. For a given complex vector x, VCSQRT(x) is computed exactly as for the function CSQRT.

VDBLE(v)

This constructs a double precision vector from a real vector. For each element of the real vector, the element value is assigned to the most significant part (the first word) in the double precision result vector; the least significant parts are real zero. Accuracy of the result is 94 bits.

VDIM(v_1,v_2)

For each pair of corresponding elements in two real vectors, this computes the positive excess of one real number over the other real number; for a pair (x,y), the value x-y is assigned to the result vector if x is greater than or equal to y, and the value 0.0 is assigned otherwise. Accuracy of the result is 47 bits.

VEXP(v)

This computes the exponential of each element in a real vector. VEXP returns a result vector of real numbers.

VEXTEND(v)

For each element of the half precision vector v, VEXTEND computes EXTEND(x). The result is a real vector.

VFLOAT(v)

This constructs a real vector from an integer vector. Each integer vector element is normalized and assigned to the real vector.

VHABS(v)

For each element x of the half precision vector v, VHABS computes HABS(x). The result is a half precision vector.

VHACOS(v)

For each element of the half precision vector v, VHACOS computes HACOS(x). The result is a half precision vector.

VHALF(v)

For each element x of the input vector v, VHALF computes HALF(x). The input vector may be of type integer, real, double precision or complex. The result is a half precision vector.

VHASIN(v)

For each element x of the half precision vector, VHASIN computes HASIN(x). The result is a half precision vector.

VHATAN(v)

For each element x of the half precision vector v, VHATAN computes HATAN(x). The result is a half precision vector.

VHATAN2(v_1.v_2)

For each corresponding pair of elements x_1 and x_2 of the input vectors v_1 and v_2, VHATAN2 computes HATAN2(x_1,x_2). The arguments and result are half precision vectors.

VHCOS(v)

For each element x of the half precision vector v, VHCOS computes HCOS(x). The result is a half precision vector.

VHDIM(v_1,v_2)

For each corresponding pair of elements x_1 and x_2 of the input vectors v_1 and v_2, VHDIM computes HDIM(x_1,x_2). The arguments and result are half precision vectors.

VHEXP(v)

For each element x of the half precision vector v, VHEXP computes HEXP(x). The result is a half precision vector.

VHINT(v)

For each element x of the half precision vector v, VHINT computes HINT(x). The result is a half precision vector.

VHLOG(v)

For each element x of the half precision vector v, VHLOG computes HLOG(x). The result is a half precision vector.

VHLOG10(v)

For each element x of the half precision vector v, VHLOG10 computes HLOG10(x). The result is a half precision vector.

VHMOD(v_1,v_2)

For each corresponding pair of elements x_1 and x_2 the input vectors v_1 and v_2. VHMOD computes HMOD(x_1.x_2). The arguments and result are half precision vectors.

VHNINT(v)

For each element x of the half precision vector v, VHNINT computes HNINT(x). The result is a half precision vector.

VHSIGN(v_1,v_2)

For each corresponding pair of elements x_1 and x_2 of the input vectors v_1 and v_2, VHSIGN computes HSIGN(x_1,x_2). The arguments and result are half precision vectors.

VHSIN(v)

For each element x of the half precision vector v, VHSIN computes HSIN(v). The result is a half precision vector.

VHSQRT(v)

For each element x of the half precision vector v, VHSQRT computes HSQRT(x). The result is a half precision vector.

VHTAN(v)

For each element x of the half precision vector v, VHTAN computes HTAN(x). The result is a half precision vector.

VIABS(v)

For each element x in an integer vector, VIABS computes the absolute value (x).

VIDIM(v_1,v_2)

For each pair of corresponding elements in two integer vectors, this gives the positive excess of one integer number over the other integer number; for a pair (x,y), the value x-y is assigned to the result vector if x is greater than or equal to y, and the value 0 is assigned otherwise.

VIFIX(v)

This constructs an integer vector from a real vector. VIFIX, which is an alternative name for VINT, computes (x) for each element x in a real vector. (A) is the sign of A times the largest integer less than or equal to (A).

VIHINT(v)

For each element x of the half precision vector v, VIHINT computes IHINT(x). The result is an integer vector.

VIHNINT(v)

For each element x of the half precision vector v, VIHNINT computes IHNINT(x). The result is an integer vector.

VINT(v)

For each element x in a real vector, VINT computes (x) and assigns the resulting value to an integer vector. (A) is the sign of A times the largest integer less than or equal to (A).

VISIGN(v_1.v_2)

For each pair (x,y) of corresponding elements in two integer vectors, this combines the sign of x with the absolute value of y; the effect of VISIGN on each pair (x,y) is that of the expression ISIGN(x,y).

VMOD(v_1,v_2)

For each pair of corresponding elements in two real vectors, this computes one integer number modulo the second integer number to produce an integer result that is assigned to the integer result vector. For each pair of elements (x,y), x-(x/y) * y is computed, where (A) is the sign of A times the largest integer less than or equal to (A).

VNINT(v)

For each element x of the real vector v, VNINT computes NINT(x). The result is an integer vector.

VREAL(v)

This constructs a real vector from the real parts of a complex vector. For each element of the complex vector, if x+iy is the complex element, x is assigned to the result vector. Accuracy of the result is 47 bits.

VSIGN(v_1,v_2)

For each pair (x,y) of corresponding elements in two real vectors, this combines the sign of x with the absolute value of y; the effect of VSIGN on each pair (x,y) is that of the expression SIGN(x,y). Accuracy of each result is 47 bits.

VSIN(v) and VCOS(v)

These compute the sine and cosine of each element in a real vector. The magnitude of the error that is introduced into the results by use of the table lookup technique for fast computation of VSIN and VCOS is approximately 2^{-45}.

VSNGL(v)

This converts a double precision vector to a real vector. The most significant part (the first word) of each double precision element is assigned to the result vector. Accuracy of each result is 47 bits.

VSQRT(v)

This computes the square root of each element in a real vector. The real result vector contains elements that are accurate to approximately 47 bits.

For a given real element x of the vector argument, the appropriate element of the result vector is indefinite if $x < 0.0$. For each $x \geq 0.0$, a result is computed.

VTAN(v)

This computes the tangent of each element in a real vector. A table lookup technique is used for fast computation of VTAN; consequently, the error for small results has a magnitude of approximately 2^{-45}.

PROGRAM COMPILATION

The system control statements accompanying a STAR FORTRAN program must include a call to the FORTRAN compiler. The parameters for this call optionally declare files for input and output, and optionally include instructions to the compiler to (for example) output storage maps. Additional control statements are required to load and to execute the compiled program, and can be used to change at run time the file declarations made in a PROGRAM statement.

FORTRAN STATEMENT

The FORTRAN system control statement is used to execute the STAR FORTRAN compiler. In the statement parameter descriptions that follow, underlining indicates the minimum number of characters that can be used in specifying the parameter.

Forms:

FORTRAN.

FORTRAN(INPUT=f_1,BINARY=f_2/l_2,LIST=f_3/l_3/d_3, OPTIONS=olist)

INPUT=f_1	Optional. f_1 is the name of the file containing the FORTRAN source program to be compiled. When the parameter is omitted, the default file name INPUT is used.
BINARY=f_2/l_2	Optional. f_2 is the name of the file that is to receive the compiler-generated object modules. l_2 is a specification of the length of f_2, and can be either an integer constant or a hexadecimal number prefixed with a #. l_2 can be omitted along with the slash. When the entire parameter is omitted, the default file name BINARY is used. When l_2 or the entire parameter is omitted, the default file length of 16 small pages is used.
LIST=f_3/l_3/d_3	Optional. f_3 is the name of the file that is to receive the compiler-generated listings and program output. l_3 is a specification of the length of f_3. Like l_2,l_3 can be either an integer constant or a hexadecimal number prefixed with a #. d_3 is the routing disposition of f_3 and must be PR (the line printer) or can be omitted (in which case no routing is performed). l_3 and d_3 can occur in either order. When l_3 is omitted, the default file size of 336 small pages is used. When the entire parameter is omitted, the default is OUTPUT.
OPTIONS=olist	Optional. olist is some logical combination of the compile option letters ABCEIKLMORSUVYZ12, with the restriction that Y must not occur with any other option except L. Default olist is B.

Alternative delimiters for the parameter list are a comma or blank instead of the left parenthesis along with a period replacing the right parenthesis. When communicating interactively with the system, the user can replace a period with a carriage return.

The FORTRAN system control statement parameters must be separated by commas or blanks. Partial parameter lists are acceptable, with default values used for the omitted parameters. The first form of the FORTRAN statement selects all defaults for the parameters. The I=, B=, and L= parameters can be interchanged without consequence; the O= parameter must occur last.

The object and output files (specified by the B= and L= parameters of the FORTRAN system control statement) may or may not exist when the control statement is executed. If the file does not exist, it is automatically created on a unit assigned by the operating system and with the length specified in the control statement. If the file does exist and has write access, it is automatically destroyed and recreated on the same unit with the length specified in the control statement. If the file does exist but does not have write access, a request is made to interactive users for permission to destroy the file. If permission is granted, the procedure followed is the same as for files that exist with write access. If permission is not granted, or if the user is in batch mode, the job is aborted.

When a compile option letter appears in the O=olist parameter, certain actions are performed during compilation that would not be performed otherwise. The L option is an exception in that the listing of the source program is inhibited rather than initiated by its appearance in olist.

When O=olist is omitted, or when B is included in olist, the object file for the program is built. The only time when the object file is not built is when the O=olist parameter, with B not in olist, appears in the parameter list for the FORTRAN system control statement.

A — ASSEMBLY LISTING

An assembly listing of the object code can be placed in the output file by selecting the A option.

B — BUILD OBJECT FILE

An object file is required for the loading and execution of the FORTRAN program. A request that the file be built is made by selecting the B option.

C — CROSS REFERENCE LISTING

All mentions in the source program to labels and symbolic names are listed in tabular form in the output file by selecting the C option.

E — EXTENDED BASIC BLOCK OPTIMIZATION

The E option selects optimization of extended basic blocks. Optimization involves redundant code elimination and instruction scheduling. The E option is included in the O option. The E option effectively selects options R and I.

I — INSTRUCTION SCHEDULING

The I option selects optimization of object instructions according to the results of a critical path analysis. The I option is included in the O and E options.

K — 64-BIT COMPARE

This option enables full word (64-bit) integer compares for .EQ. and .NE. operators in logical IF statements. Otherwise, 48-bit compares are performed for the .EQ. and .NE. operations (integers are 48 bits).

L — SOURCE LISTING SUPPRESSION

The first part of the output file for a STAR FORTRAN program is normally the source program listing. This can be omitted from the file by selecting the L option.

M — MAP OF REGISTER FILE AND STORAGE ASSIGNMENTS

A listing in the output file of all variables, constants, externals, arrays, and descriptors, along with a map of the contents of the register file, is produced when the M option is selected.

O — OPTIMIZATION

The O option selects all available optimization of scalar object code. More efficient object code is produced at the expense of increased compilation time. The O option effectively selects options Z, E, R, and I.

R — REDUNDANT CODE ELIMINATION

The R option selects elimination of redundant code. The R option is included in the O and E options.

S — CREATE DEBUG SYMBOL TABLES

The effect of this option is to generate in the binary output a debug symbol table for each program unit. The symbol table makes it possible for the system-provided debugging utility DEBUG to recognize names in the FORTRAN program. The user must select this option if DEBUG is going to have to interpret variables, names, and symbolic addresses; if only absolute addresses will be used in commands to DEBUG, the S option need not be selected.

U — USAFE VECTORIZATION

The U option enables unsafe vectorization of certain DO loops. If the terminal value of a DO loop is variable and the loop contains any references to dummy arrays, then the compiler cannot determine the number of iterations of the loop. Vectorization of such loops is considered unsafe because the loop count might exceed 65 535, which is the maximum length of a vector.

V — VECTORIZATION

Vectorization of certain STAR FORTRAN language constructs is requested with the V compile option. The language constructs that produce vector machine instructions in the object code are described in section 11.

Y — SYNTAX CHECK

A partial compilation can be performed to check the syntax of a FORTRAN program, and output any resulting diagnostics, by selecting the Y compile option. The Y option can appear alone or with the L option only (as LY or YL); all other option combinations using Y, such as CMY or SY, are invalid compile option lists and produce an error accompanied by a dayfile message.

Z — DO LOOP OPTIMIZATION

The Z option selects optimizations of DO loops and loop nests. Optimization involves invariant code removal and strength reduction of subscript calculations. The Z option is included in the O option.

1 — STAR-100 OPTIMIZATION

The 1 option selects optimization for the STAR-100. The 1 option conflicts with the 2 option. When 1 or 2 is not selected, optimization is for the mainframe on which compilation is performed.

2 — STAR-100A OPTIMIZATION

The 2 option selects optimization for the STAR-100A. The 2 option conflicts with the 1 option. When 1 or 2 is not selected, optimization is for the mainframe on which compilation is performed.

COMPILER-GENERATED LISTINGS

As a result of requesting compilation of a FORTRAN program with a FORTRAN system control statement, a variety of information is placed in the output file. The compile options A, C, and M directly request such information.

A header line at the top of each page of printed compiler output contains the compiler version, the compile options selected, the type of listing, and the time, date, and page number.

Unless the L compile option has been selected, the source program (including comments) is the first item to be placed on the file. The source program is listed 58 lines per printed page (excluding headers); the output lines are numbered on the right and the FORTRAN statements are numbered on the left. The statement numbers are used in the cross-reference maps.

Diagnostics are collected and listed at the end of each program unit. When no compile options have been selected, any error diagnostics immediately follow the source listing; or, if the syntax of the program is acceptable to the compiler, the message NO ERRORS appears instead. Listed with each diagnostic is the line number of the source line during the processing of which the error was detected, as well as the error number (see appendix B) and the severity level of the error.

The order in which the assembly listing, cross-reference maps, and storage maps appear on the output file following the source listing is:

Cross-reference map

Assembly listing

Storage map and register map

This order can be seen in the sample output in figure 16-1. Any diagnostics follow the storage and register maps.

CROSS-REFERENCE TABLES

When the C compile option is selected, either one or two cross-reference tables appear in the output for the program compilation. These tables appear immediately following the source program listing or, when the L compile option was also selected, as the first listings in the output.

Any statement labels in the source program are itemized in the first cross-reference table. For each statement label, the statement where the label was defined is given, followed by any statements that reference the label. Statements are indicated by source listing statement line numbers.

The cross-reference table itemizing all symbolic names in the source program appears after the statement label cross-reference table. For each symbolic name, the source listing statement numbers of any statements containing the name are listed.

ASSEMBLY LISTING

When the A compile option is selected, a listing of the assembly representation of the FORTRAN program appears after any cross-reference tables. Given are the location counter (the offset from the code area base address), the machine instruction in hexadecimal (either half- or full-word instruction), the source listing line number of the associated source program statement, the instruction mnemonic, instruction qualifiers, and operands. Refer to the Assembler Reference Manual for an interpretation of META assembler language.

REGISTER MAP AND STORAGE MAPS

When the M compile option is selected, a listing of the contents of the 256-register register file is produced, appearing after any assembly listing. The STAR FORTRAN register usage conforms to standard STAR operating system register conventions, which are described in volume 2 of the STAR Operating System Reference Manual. Also produced under this option is a storage map, giving the following information:

Start address and size of data area copy of the register file

Name, location, class, and data type of all scalars, constants, and externals assigned to registers

Name, location, and class of descriptors assigned to registers

Length and start address of the object code

Length and start address of character constants, literals, and format segments

Length and start address of argument vectors

Length and start address of constants, externals, descriptors, variables (not in COMMON), namelist groups, and character scalars not assigned to registers

Quantity of temporary storage

Common blocks

Entry points

Externals

EXECUTION-TIME FILE REASSIGNMENT

The PROGRAM statement declarations for files can be entirely or partially overridden at program execution time (run time). The alternative to having the files opened as declared in the PROGRAM statement is to call the controllee file (default controllee file is GO) followed by one of the following forms:

(**message)

(message)

message	File declarations in the same forms as for the PROGRAM statement (described in section 7).

With use of the first form, the file declarations in the PROGRAM statement are ignored and the file declarations in the message are used. With use of the second form, any logical unit assignment in the message overrides the assignment made to the same logical unit in the PROGRAM statement. If a unit was given in the message but was not given in the PROGRAM statement, it is opened in addition to those declared in the statement.

All file declarations in the message must be presented in exactly the same form as used for file information parameters in a PROGRAM statement. If files are partially reassigned, the original PROGRAM statement declaration string is still processed. Therefore, it is not possible to get around syntax, file name, or parameter errors in the PROGRAM statement by attempting partial run-time reassignment.

The effect of partial run-time reassignment is the same as if the run-time declaration of a particular unit had appeared in the PROGRAM statement declaration instead of the original declaration. After the original PROGRAM statement is processed, the original data for a unit is overwritten with run-time data taken from the file tables. However, the user must consider the effect of run-time changes on other declarations. For example, if the original unit declarations in the PROGRAM statement were:

```
TAPE6[7,800,1]=DATA1,TAPE7=DATA1
```

and the run-time reassignment specified was:

```
TAPE6=MYFILE
```

then the explicit parameters for TAPE6 in the PROGRAM statement would be lost, and DATA1 would become an implicit disk file.

When a program is executed interactively under DEBUG, the user is prompted for file reassignment. As the prompt indicates, the user must then either enter a period for no file reassignment or a file reassignment enclosed in parentheses.

CONTROL OF DROP FILE SIZE

If a DROP FILE OVERFLOW run-time error message is issued, the user can increase the size of the drop file and rerun the program. The CDF parameter of the LOAD system control statement or the D parameter of the SWITCH system control statement can be used to make the drop file size larger. Increasing the size of the drop file can usually solve the overflow problem, but a program error (especially an infinite loop) might be the cause.

```
STAR FORTRAN 2.0 CYCLE 115                    SOURCE LISTING      13:49 HRS. 15MAR77                      PAGE 0001
   00001          PROGRAM PASCAL (OUTPUT)                                                                 0001/00001
   00002          INTEGER L(11)                                                                           0001/00002
   00003          DATA L(11) /1/                                                                          0001/00003
         C                                                                                                0001/00004
   00004          PRINT 4, (I,I=1,11)                                                                     0001/00005
   00005    4     FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/                      0001/00006
                 111I5)                                                                                   0001/00007
   00006          DO 200  I=1,10                                                                          0001/00008
   00007          K=11-I                                                                                  0001/00009
   00008          L(K)=1                                                                                  0001/00010
   00009          DO  100  J=K,10                                                                         0001/00011
   00010   100    L(J)=L(J)+L(J+1)                                                                        0001/00012
   00011   200    PRINT 3,(L(J),J=K,11)                                                                   0001/00013
   00012   3      FORMAT (11I5)                                                                           0001/00014
           C                                                                                              0001/00015
   00013          STOP                                                                                    0001/00016
   00014          END                                                                                     0001/00017




STAR FORTRAN 2.0 CYCLE 115                    CROSS REF LISTING   13:49 HRS. 15MAR77        PASCAL        PAGE 0002


          CROSS REFERENCE TABLE

       LABEL   DEFINED   REFERENCES
100        000010   000009
200        000011   000006
3          000012   000011
4          000005   000004




STAR FORTRAN 2.0 CYCLE 115                    CROSS REF LISTING   13:49 HRS. 15MAR77        PASCAL        PAGE 0003


          CROSS REFERENCE TABLE

       SYMBOL     REFERENCES
I          000004  000004  000006  000007
J          000009  000010  000010  000010  000010  000011  000011
K          000007  000007  000008  000008  000009
L          000002  000003  000008  000008  000010  000010  000010  000010  000011
PASCAL     000001
```

Figure 16-1. Sample Output (Sheet 1 of 5)

```
STAR FORTRAN 2.0 CYCLE 115                    ASSEMBLY LISTING     13:49 HRS. 15MAR77                    PASCAL        PAGE 0004

  LOCATION     MACHINE             LINE    SOURCE              ASSEMBLY REPRESENTATION
  COUNTER      INSTRUCTION         NUMBER  LABEL

                                           PASCAL   IDENT
                                                    ENTRY      PASCAL
  0000000      7000151C            00001   PASCAL   SWAP       0,C_#1A,CUR_STACK
  0000020      781C001D                             RTOR       CUR_STACK,PREV_STACK
  0000040      781B001C                             RTOR       DYN_SPACE,CUR_STACK
  0000060      3F1B1400                             IS         DYN_SPACE,#1400
  0000080      2A1C0050                             ELEN       CUR_STACK,#50
  00000A0      3E040580                             ES         PR_4,#580
  00000C0      631E0404                             ADDX       CALLEDATA,PR_4,PR_4
  00000E0      2A04004A                             ELEN       PR_4,#4A
  0000100      70041400                             SWAP       PR_4,C_#20,0
  0000120      3021FB20                             SHIFTI     CODEADRB,#FB,CODEADRH
  0000140      2A204000                             ELEN       CODEADRH,#4000
  0000160      78660003                             RTOR       L_C00001_DESCR,PR_3
  0000180      7861001E                             RTOR       FT_INIT_DB,CALLEDATA
  00001A0      361A0060                             BSAVE      RETURN,FT_INIT_ADR
  00001C0      781B004C                             RTOR       DYN_SPACE,PI_DYNSP
  00001E0      78670004            00004            RTOR       L_F4_DESCR,PR_4
  0000200      785D001E                             RTOR       FT_HTIPR_DB,CALLEDATA
  0000220      361A005C                             BSAVE      RETURN,FT_HTIPR_ADR
  0000240      78540059                             RTOR       C_#1,I
  0000260      78590003                    000002   RTOR       I,PR_3
  0000280      785F001E                             RTOR       FT_HTIE_DB,CALLEDATA
  00002A0      361A005E                             BSAVE      RETURN,FT_HTIE_ADR
  00002C0      8406595400035559                     IBXLE,BRB  I,C_#1,000002,C_#B,I
  0000300      785B001E                             RTOR       FT_HTTPR_DB,CALLEDATA
  0000320      361A005A                             BSAVE      RETURN,FT_HTTPR_ADR
  0000340      78540059            00006            RTOR       C_#1,I
  0000360      67555957            00007   000003   SUBX       C_#B,I,K
  0000380      7F655754            00008            STO        [L_18_DESCR,K],C_#1
  00003A0      78570058            00009            RTOR       K,J
                                           000004
  00003C0      7E655803            00010   S100     LOD        [L_18_DESCR,J],PR_3
  00003E0      7E645804                             LOD        [L_20_DESCR,J],PR_4
  0000400      63030405                             ADDX       PR_3,PR_4,PR_5
  0000420      7F655805                             STO        [L_18_DESCR,J],PR_5
  0000440      8406585400045658                     IBXLE,BRB  J,C_#1,000004,C_#A,J
  0000480      78680004            00011   S200     RTOR       L_F3_DESCR,PR_4
  00004A0      785D001E                             RTOR       FT_HTIPR_DB,CALLEDATA
  00004C0      361A005C                             BSAVE      RETURN,FT_HTIPR_ADR
  00004E0      78570058                             RTOR       K,J
  0000500      7E655803                    000005   LOD        [L_18_DESCR,J],PR_3
  0000520      785F001E                             RTOR       FT_HTIE_DB,CALLEDATA
  0000540      361A005E                             BSAVE      RETURN,FT_HTIE_ADR
  0000560      8406585400035558                     IBXLE,BRB  J,C_#1,000005,C_#B,J
  00005A0      785B001E                             RTOR       FT_HTTPR_DB,CALLEDATA
  00005C0      361A005A                             BSAVE      RETURN,FT_HTTPR_ADR
  00005E0      8406595400145659                     IBXLE,BRB  I,C_#1,000003,C_#A,I
  0000620      3E030000            00013            ES         PR_3,0
  0000640      7863001E                             RTOR       FT_STOP_DB,CALLEDATA
  0000660      361A0062                             BSAVE      RETURN,FT_STOP_ADR
                                                    END
```

Figure 16-1. Sample Output (Sheet 2 of 5)

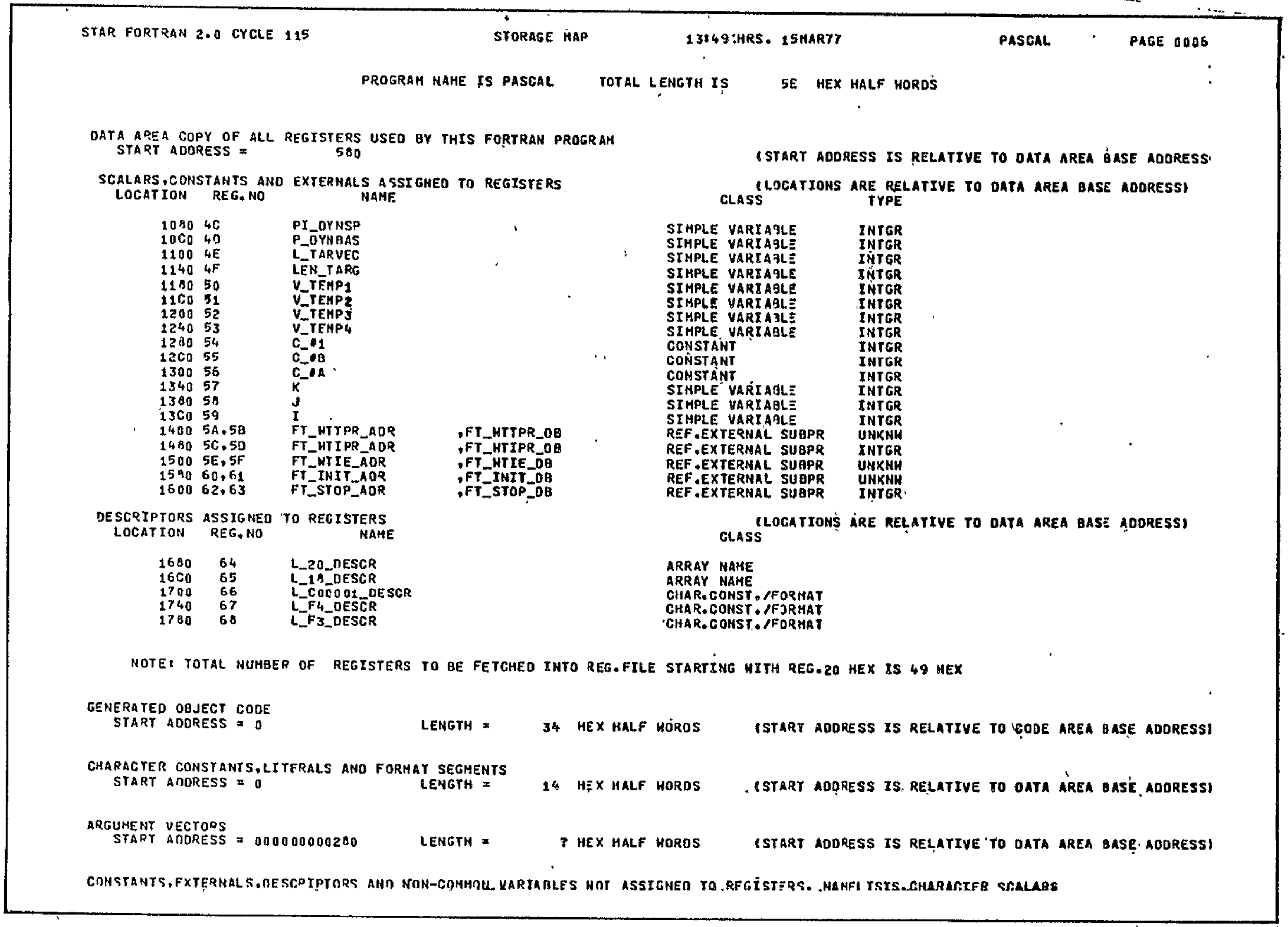
```
STAR FORTRAN 2.0 CYCLE 115                     REGISTER MAP        13:49 HRS. 15MAR77                  PASCAL       PAGE 0005

REG.      NAME               REG.      NAME               REG.      NAME               REG.      NAME          REG.      NAME
 NO                           NO                           NO                           NO                      NO

 00  0    (MACHINE ZERO)      33  TFR_33                   66  L_C00001_DESCR           99  FR_99               CC  FR_CC
 01  DATA_FLAG_RETURN         34  TFR_34                   67  L_F4_DESCR               9A  FR_9A               CD  FR_CD
 02  TM_INTERUPT_ENTRY        35  TFR_35                   68  L_F3_DESCR               9B  FR_9B               CE  FR_CE
 03  PR_3                     36  TFR_36                   69  FR_69                    9C  FR_9C               CF  FR_CF
 04  PR_4                     37  TFR_37                   6A  FR_6A                    9D  FR_9D               D0  FR_D0
 05  PR_5                     38  TFR_38                   6B  FR_6B                    9E  FR_9E               D1  FR_D1
 06  PR_6                     39  TFR_39                   6C  FR_6C                    9F  FR_9F               D2  FR_D2
 07  PR_7                     3A  TFR_3A                   6D  FR_6D                    A0  FR_A0               D3  FR_D3
 08  PR_8                     3B  TFR_3B                   6E  FR_6E                    A1  FR_A1               D4  FR_D4
 09  PR_9                     3C  TFR_3C                   6F  FR_6F                    A2  FR_A2               D5  FR_D5
 0A  PR_A                     3D  TFR_3D                   70  FR_70                    A3  FR_A3               D6  FR_D6
 0B  PR_B                     3E  TFR_3E                   71  FR_71                    A4  FR_A4               D7  FR_D7
 0C  PR_C                     3F  TFR_3F                   72  FR_72                    A5  FR_A5               D8  FR_D8
 0D  PR_D                     40  TFR_40                   73  FR_73                    A6  FR_A6               D9  FR_D9
 0E  PR_E                     41  TFR_41                   74  FR_74                    A7  FR_A7               DA  FR_DA
 0F  PR_F                     42  TFR_42                   75  FR_75                    A8  FR_A8               DB  FR_DB
 10  PR_10                    43  TFR_43                   76  FR_76                    A9  FR_A9               DC  FR_DC
 11  PR_11                    44  TFR_44                   77  FR_77                    AA  FR_AA               DD  FR_DD
 12  TM_SCRATCH               45  TFR_45                   78  FR_78                    AB  FR_AB               DE  FR_DE
 13  TM_REQUEST_ENTRY         46  TFR_46                   79  FR_79                    AC  FR_AC               DF  FR_DF
 14  C_#20                    47  TFR_47                   7A  FR_7A                    AD  FR_AD               E0  FR_E0
 15  C_#1A                    48  TFR_48                   7B  FR_7B                    AE  FR_AE               E1  FR_E1
 16  C_1                      49  TFR_49                   7C  FR_7C                    AF  FR_AF               E2  FR_E2
 17  C_PARM_DESCR             4A  TFR_4A                   7D  FR_7D                    B0  FR_B0               E3  FR_E3
 18  F_RET1                   4B  TFR_4B                   7E  FR_7E                    B1  FR_B1               E4  FR_E4
 19  F_RET2                   4C  PI_DYNSP                 7F  FR_7F                    B2  FR_B2               E5  FR_E5
 1A  RETURN                   4D  P_DYNBAS                 80  FR_80                    B3  FR_B3               E6  FR_E6
 1B  DYN_SPACE                4E  L_TARVEC                 81  FR_81                    B4  FR_B4               E7  FR_E7
 1C  CUR_STACK                4F  LEN_TARG                 82  FR_82                    B5  FR_B5               E8  FR_E8
 1D  PREV_STACK               50  V_TEMP1                  83  FR_83                    B6  FR_B6               E9  FR_E9
 1E  CALLEDATA                51  V_TEMP2                  84  FR_84                    B7  FR_B7               EA  FR_EA
 1F  ON_UNIT                  52  V_TEMP3                  85  FR_85                    B8  FR_B8               EB  FR_EB
 20  CODEADRH                 53  V_TEMP4                  86  FR_86                    B9  FR_B9               EC  FR_EC
 21  CODEADRB                 54  C_#1                     87  FR_87                    BA  FR_BA               ED  FR_ED
 22  PARM_DESCR               55  C_#8                     88  FR_88                    BB  FR_BB               EE  FR_EE
 23  DATABASE                 56  C_#A                     89  FR_89                    BC  FR_BC               EF  FR_EF
 24  TFR_24                   57  K                        8A  FR_8A                    BD  FR_BD               F0  FR_F0
 25  TFR_25                   58  J                        8B  FR_8B                    BE  FR_BE               F1  FR_F1
 26  TFR_26                   59  I                        8C  FR_8C                    BF  FR_BF               F2  FR_F2
 27  TFR_27                   5A  FT_HTTPR_ADR             8D  FR_8D                    C0  FR_C0               F3  FR_F3
 28  TFR_28                   5B  FT_HTTPR_DB              8E  FR_8E                    C1  FR_C1               F4  FR_F4
 29  TFR_29                   5C  FT_HTIPR_ADR             8F  FR_8F                    C2  FR_C2               F5  FR_F5
 2A  TFR_2A                   5D  FT_HTIPR_DB              90  FR_90                    C3  FR_C3               F6  FR_F6
 2B  TFR_2B                   5E  FT_HTIE_ADR              91  FR_91                    C4  FR_C4               F7  FR_F7
 2C  TFR_2C                   5F  FT_HTIE_DB               92  FR_92                    C5  FR_C5               F8  FR_F8
 2D  TFR_2D                   60  FT_INIT_ADR              93  FR_93                    C6  FR_C6               F9  FR_F9
 2E  TFR_2E                   61  FT_INIT_DB               94  FR_94                    C7  FR_C7               FA  FR_FA
 2F  TFR_2F                   62  FT_STOP_ADR              95  FR_95                    C8  FR_C8               FB  FR_FB
 30  TFR_30                   63  FT_STOP_DB               96  FR_96                    C9  FR_C9               FC  FR_FC
 31  TFR_31                   64  L_20_DESCR               97  FR_97                    CA  FR_CA               FD  FR_FD
 32  TFR_32                   65  L_18_DESCR               98  FR_98                    CB  FR_CB               FE  FR_FE
                                                                                                                FF  FR_FF
```

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

Figure 16-1. Sample Output (Sheet 3 of 5)



```
STAR FORTRAN 2.0 CYCLE 115                    STORAGE MAP          13:49:HRS. 15MAR77                PASCAL           PAGE 0006

                                 PROGRAM NAME IS PASCAL     TOTAL LENGTH IS      5E  HEX HALF WORDS


DATA AREA COPY OF ALL REGISTERS USED BY THIS FORTRAN PROGRAM
   START ADDRESS =          580                                              (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

 SCALARS,CONSTANTS AND EXTERNALS ASSIGNED TO REGISTERS                        (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
   LOCATION   REG.NO             NAME                                        CLASS                TYPE

        1080 4C        PI_DYNSP                                    SIMPLE VARIABLE       INTGR
        10C0 4D        P_DYNBAS                                    SIMPLE VARIABLE       INTGR
        1100 4E        L_TARVEC                                    SIMPLE VARIABLE       INTGR
        1140 4F        LEN_TARG                                    SIMPLE VARIABLE       INTGR
        1180 50        V_TEMP1                                     SIMPLE VARIABLE       INTGR
        11C0 51        V_TEMP2                                     SIMPLE VARIABLE       INTGR
        1200 52        V_TEMP3                                     SIMPLE VARIABLE       INTGR
        1240 53        V_TEMP4                                     SIMPLE VARIABLE       INTGR
        1280 54        C_#1                                        CONSTANT              INTGR
        12C0 55        C_#B                                        CONSTANT              INTGR
        1300 56        C_#A                                        CONSTANT              INTGR
        1340 57        K                                           SIMPLE VARIABLE       INTGR
        1380 58        J                                           SIMPLE VARIABLE       INTGR
        13C0 59        I                                           SIMPLE VARIABLE       INTGR
        1400 5A,5B     FT_WTTPR_ADR        ,FT_WTTPR_DB            REF.EXTERNAL SUBPR    UNKNW
        1480 5C,5D     FT_WTIPR_ADR        ,FT_WTIPR_DB            REF.EXTERNAL SUBPR    INTGR
        1500 5E,5F     FT_WTIE_ADR         ,FT_WTIE_DB             REF.EXTERNAL SUBPR    UNKNW
        1580 60,61     FT_INIT_ADR         ,FT_INIT_DB             REF.EXTERNAL SUBPR    UNKNW
        1600 62,63     FT_STOP_ADR         ,FT_STOP_DB             REF.EXTERNAL SUBPR    INTGR

 DESCRIPTORS ASSIGNED TO REGISTERS                                            (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
   LOCATION   REG.NO             NAME                                        CLASS

        1680    64     L_20_DESCR                                  ARRAY NAME
        16C0    65     L_18_DESCR                                  ARRAY NAME
        1700    66     L_C00001_DESCR                              CHAR.CONST./FORMAT
        1740    67     L_F4_DESCR                                  CHAR.CONST./FORMAT
        1780    68     L_F3_DESCR                                  CHAR.CONST./FORMAT


     NOTE: TOTAL NUMBER OF  REGISTERS TO BE FETCHED INTO REG.FILE STARTING WITH REG.20 HEX IS 49 HEX


GENERATED OBJECT CODE
   START ADDRESS = 0                   LENGTH =      34  HEX HALF WORDS      (START ADDRESS IS RELATIVE TO CODE AREA BASE ADDRESS)


CHARACTER CONSTANTS,LITERALS AND FORMAT SEGMENTS
   START ADDRESS = 0                   LENGTH =      14  HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)


ARGUMENT VECTORS
   START ADDRESS = 000000000280        LENGTH =       7 HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)


CONSTANTS,EXTERNALS,DESCRIPTORS AND NON-COMMON VARIABLES NOT ASSIGNED TO REGISTERS. NAMELISTS.CHARACTER SCALARS
```

Figure 16-1. Sample Output (Sheet 4 of 5)

```
  STAR FORTRAN 2.0 CYCLE 115                         STORAGE MAP         13:49:HRS. 15MAR77              PASCAL         PAGE 0007

    START ADDRESS = 000000000280       LENGTH =     16  HEX HALF WORDS     (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS

    LOCATION      SYMBOLIC NAME OR HEX VALUE                CLASS          TYPE  (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

         280       L                                   ARRAY VARIABLE      INTGR

  TEMPORARY STORAGE
                                               LENGTH =        7 HEX HALF WORDS     (STORAGE IS SCATTERED THROUGHOUT DATA AREA)

  COMMON BLOCKS

    NO COMMON BLOCK IS SPECIFIED

  LIST OF ALL ENTRY POINTS

    LOCATION      SYMBOLIC NAME                                                  (LOCATIONS ARE RELATIVE TO CODE AREA BASE ADDRESS)

                    PASCAL

  LIST OF ALL EXTERNALS

                  SYMBOLIC NAME

                    FT_WTTPR
                    FT_WTIPR
                    FT_WTIE
                    FT_INIT
                    FT_STOP
     NO ERRORS
```

Figure 16-1. Sample Output (Sheet 5 of 5)

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

SPECIAL CALL STATEMENTS D

This appendix describes the available special call statements. Each special call statement directly generates a machine instruction. Special calls are described in general terms in section 14. Each special call name is a mnemonic preceded by Q8. The mnemonics are identical to the STAR Assembler mnemonics in most cases. Certain special calls use an abbreviated mnemonic because the name is limited to 6 characters following the Q8.

The first field of each machine instruction is the op code (F), indicating which function is to be performed. The special call name supplies the op code (F) in the generated instruction. Other operands are specified as arguments in the special call. The operand designators are explained in table D-1.

The special call formats are shown in table D-2. The G bits that can be set either to 0 or 1 are indicated with the marking x. In the table, the following additional notations are used:

f	Indicates a fullword register containing an operand.
h	Indicates a halfword register containing an operand.
a	Indicates a fullword register containing an address; length field is ignored.
i	Indicates a fullword register containing an index.
d	Indicates a fullword register containing a descriptor.
e	Indicates a fullword register with an exponent field that contains a length operand.
eh	Indicates a halfword register with an exponent field that contains a length operand.
FP	Is an abbreviation for floating point.
OV	Is an abbreviation for order vector.
RJ	Is an abbreviation for right-justified.
SE	Is an abbreviation for sign extended.
.OP.	Indicates one of the logical operators .EQ., .NE., .GE. or .LT.
U	Indicates upper result.
L	Indicates lower result.
N	Indicates normalized upper result.
S	Indicates significant result.

The instruction format is one of the twelve possible instruction formats shown in figure D-1. Additional information about any machine instruction, including the G bit settings, can be found in the STAR-100 Computer Hardware Reference Manual.

TABLE D-1. OPERAND DESIGNATORS

Designator	Format Type	Definition
A	1 and 3	Specifies a register that contains a field length and base address for the corresponding source vector or string field.
	2	Specifies a register that contains the base address for a source sparse vector field.
	C	Specifies a register that contains, based on bit 12 of the instruction (G-bit-4), either a two's complement or unsigned integer in the rightmost 48 bits.
B	1 and 3	Specifies a register that contains a field length and base address for the corresponding source vector or string field.
	2	Specifies a register that contains the base address for a source sparse vector field
	C	Specifies a register that contains the branch base address in the rightmost 48 bits
C	1, 2, and 3	Specifies a register that contains the field length and base address for storing the result vector, sparse vector, or string field.
	C	Specifies the register that will contain, based on bit 12 of the instruction (G-bit-4), either a two's complement or unsigned sum of (A) + (X) in the rightmost 48 bits. The leftmost 16 bits are cleared.
C + 1	1	Specifies a register containing the offset for C and Z vector fields. If the C + 1 designator is used, the C designator must specify an even-numbered register.

TABLE D-1. OPERAND DESIGNATORS (Contd)

Designator	Format Type	Definition
G	1, 2, 3, 9, B and C	8-bit designator specifies certain subfunction conditions. Subfunctions include length of operands (32- or 64-bit), normal or broadcast source vectors, etc. The number of bits used in the G designator varies with instructions.
I	5	48-bit index used to form the branch address in a B6 branch instruction. In BE and BF index instructions, I is a 48-bit operand.
	6	In 3E and 3F index instructions, I is a 16-bit operand.
	B	In the 33 branch instruction, the 6-bit I is the number of the DFB object bits used in the branching operation.
R	4	In the register and 3D instructions, R is the register containing an operand to be used in an arithmetic operation.
	5 and 6	In the 3E, 3F, BE, and BF index instructions, R is a destination register for the transfer of an operand or operand sum. In the B6 branch instruction, this register contains an item count used to form the branch address.
	7, 8, and A	R specifies registers and branching conditions given in the individual instruction descriptions.
S	4	In the register and 3D instructions, S is a register containing an operand to be used in an arithmetic operation.
	7, 8, and 9	S specifies registers and branching conditions given in the individual instruction descriptions.
T	4	T specifies a destination register for the transfer of the arithmetic results.
	7, 8, 9, and B	T specifies a register that contains the base address and, in some cases, the field length of the corresponding result field or branch address.
	A	T specifies a register containing the old state of a register, DFB register, etc.; in an index, branch, or inter-register transfer operation.
X	1 and 3	Specifies a register that contains the offset or index for vector or string source field A.
	2	Specifies a register that contains length and base address for order vector corresponding to source sparse vector field A.
	C	In the B0-B5 Branch instructions; this register contains, based on bit 12 of the instruction (G-bit-4), either a two's complement or unsigned integer in the rightmost 48 bits used as an operand in the branching operation.
Y	1 and 3	Specifies a register that contains the offset or index for vector or string field B.
	2	Specifies a register that contains the length and base address for the order vector corresponding to source sparse vector field B.
	C	In the B0-B5 Branch instructions, Y specifies a register that contains an index used to form the branch address.
Z	1	Z specifies a register that contains the base address for the order vector used to control the result vector in field C.
	2	Z specifies a register that contains the length and base address for the order vector corresponding to result sparse vector field C.
	3	Z specifies a register that contains the index for result field C.
	C	In the B0-B5 Branch instructions, Z specifies a register that contains, based on bit 12 of the instruction (G-bit-4), either a two's complement or unsigned integer in the rightmost 48-bits. It is used as the comparison operand in determining whether the branch condition is met.

TABLE D-2. SPECIAL CALL FORMATS

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8ABS(R_f, ,T_f)	79	A	Absolute, fullword FP: ABS(R_f)->T_f	
CALL Q8ABSH(R_h, ,T_h)	59	A	Absolute, halfword FP: ABS(R_h)->T_h	
CALL Q8ABSV(G,X,A, , ,Z,C)	99	1	Absolute, vector: ABS(A)->C	xxxx oooo
CALL Q8ACPS(G,X,A,Y,B,Z,C)	CF	1	A_n.GE.B_n->C_n,set Z_n,OV length->Z_{0-15}	xooo xxxx
CALL Q8ADDB(,X,A,Y,B,Z,C)	E0	3	Add binary: A+B->C	
CALL Q8ADDD(,X,A,Y,B,Z,C)	E4	3	Add decimal: A+B->C	
CALL Q8ADDL(R_f,S_f,T_f)	61	4	Add lower, fullword FP: ((R_f)+(S_f))$_L$ ->T_f	
CALL Q8ADDLEN(R_e,S_f,T_e)	2B	4	Add to length, R_{0-15}+S_{48-63} ->T_{0-15},R_{16-63}->T_{16-63}	
CALL Q8ADDLH(R_h,S_h,T_h)	41	4	Add lower, halfword FP: ((R_h)+(S_h))$_L$ ->T_h	
CALL Q8ADDLS(G,X,A,Y,B,Z,C)	A1	2	Add lower, sparse vector: (A+B)$_L$ ->C	xoox xxxx
CALL Q8ADDLV(G,X,A,Y,B,Z,C)	81	1	Add lower, vector: (A+B)$_L$ ->C	xxxx xxxx
CALL Q8ADDMOD(G,X,A,Y,B,Z,C)	EC	3	Add modulo bytes: (A_n+B_n) mod (I8)->C_n	
CALL Q8ADDN(R_f,S_f,T_f)	62	4	Add normalized, fullword FP: ((R_f)+(S_f))$_N$ ->T_f	
CALL Q8ADDNH(R_h,S_h,T_h)	42	4	Add normalized, halfword FP: ((R_h)+(S_h))$_N$ ->T_h	
CALL Q8ADDNS(G,X,A,Y,B,Z,C)	A2	2	Add normalized, sparse vector: (A+B)$_N$ ->C	xoox xxxx
CALL Q8ADDNV(G,X,A,Y,B,Z,C)	82	1	Add normalized, vector: (A+B)$_N$ ->C	xxxx xxxx
CALL Q8ADDU(R_f,S_f,T_f)	60	4	Add upper, fullword FP: ((R_f)+(S_f))$_U$ ->T_f	
CALL Q8ADDUH(R_h,S_h,T_h)	40	4	Add upper, halfword FP: ((R_h)+(S_h))$_U$ ->T_h	
CALL Q8ADDUS(G,X,A,Y,B,Z,C)	A0	2	Add upper, sparse vector: (A+B)$_U$ ->C	xoox xxxx
CALL Q8ADDUV(G,X,A,Y,B,Z,C)	80	1	Add upper, vector: (A+B)$_U$ ->C	xxxx xxxx
CALL Q8ADDX(R_f,S_f,T_f)	63	4	Add index, fullword: R_{16-63}+S_{16-63}->T_{16-63},R_{0-15}->T_{0-15}	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8ADDXV(G,X,A,Y,B,Z,C)	83	1	Add index, vector: $A_{16-63}+B_{16-63} \rightarrow C_{16-63}, A_{0-15} \rightarrow C_{0-15}$	oxxx xooo
CALL Q8ADJE(R_f,S_f,T_f)	75	4	Adjust exponent, fullword FP: (R_f) per $S \rightarrow T_f$	
CALL Q8ADJEH(R_h,S_h,T_h)	55	4	Adjust exponent, halfword FP: (R_h) per $S \rightarrow T_h$	
CALL Q8ADJEV(G,X,A,Y,B,Z,C)	95	1	Adjust exponent, vector: A per B→C	xxxx xooo
CALL Q8ADJM(G,X,A, , ,Z,C)	D1	1	Adjacent mean: $(A_{n+1}+A_n)/2 \rightarrow C_n$	xxxo oooo
CALL Q8ADJS(R_f,S_f,T_f)	74	4	Adjust significance, fullword FP: (R_f) per $S \rightarrow T_f$	
CALL Q8ADJSH(R_h,S_h,T_h)	54	4	Adjust significance, halfword FP: (R_h) per $S \rightarrow T_h$	
CALL Q8ADJSV(G,X,A,Y,B,Z,C)	94	1	Adjust significance, vector: A per B→C	xxxx xooo
CALL Q8AND(,X,A,Y,B,Z,C)	F1	3	Logical AND: A•B→C	
CALL Q8ANDN(,X,A,Y,B,Z,C)	F6	3	Logical AND NOT: $A \bullet \bar{B} \rightarrow C$	
CALL Q8AVG(G,X,A, , ,Z,C)	D0	1	Vector average: $(A_n+B_n)/2 \rightarrow C_n$	xxxx xooo
CALL Q8AVGD(G,X,A, , ,Z,C)	D4	1	Vector average difference: $(A_n-B_n)/2 \rightarrow C_n$	xxxx xooo
CALL Q8BAB(G,S_a,T_a)	32	9	Branch and alter bit: (S_a) is bit to be altered, (T_a) is branch address	xxxx oxxo
CALL Q8BADF(G,I6,T_a)	33	B	D.F. reg. bit branch and alter: I6 is bit altered, (T_a) is branch address	xxxx oxxo
CALL Q8BARB(G,S,T)	2F	9	Branch to [S] on condition of bit 63 of register T	xxxx oooo
CALL Q8BEQ(R_f,S_f,T_a)	24	8	Branch to (T_a) if (R_f).EQ.(S_f), fullword FP compare	
CALL Q8BGE(R_f,S_f,T_a)	26	8	Branch to (T_a) if (R_f).GE.(S_f), fullword FP compare	
CALL Q8BHEQ(R_h,S_h,T_a)	20	8	Branch to (T_a) if (R_h).EQ.(S_h), halfword FP compare	

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8BHGE(R_h,S_h,T_a)	22	8	Branch to (T_a) if (R_h).GE.(S_h), halfword FP compare	
CALL Q8BHLT(R_h,S_h,T_a)	23	8	Branch to (T_a) if (R_h).LT.(S_h), halfword FP compare	
CALL Q8BHNE(R_h,S_h,T_a)	21	8	Branch to (T_a) if (R_h).NE.(S_h), halfword FP compare	
CALL Q8BIM(R_i,I48)	B6	5	Branch immediate to (R_i)+I48	
CALL Q8BKPT(R_a)	04	4	Breakpoint: R_{16-63}->breakpoint register	
CALL Q8BLT(R_f,S_f,T_a)	27	8	Branch to (T_a) if (R_f).LT.(S_f), fullword FP compare	
CALL Q8BNE(R_f,S_f,T_a)	25	8	Branch to (T_a) if (R_f).NE.(S_f), fullword FP compare	
CALL Q8BSAVE(R_f,S_i,T_a)	36	7	Set (R_f) to next instruction address, branch to [T_a+S_i]	
CALL Q8BTOD(R_f, ,T_f)	11	A	Convert binary R to packed BCD T, fixed length	
CALL Q8CLG(R_f, ,T_f)	72	A	Ceiling, fullword FP: nearest integer .GE.(R_f)->T_f	
CALL Q8CLGH(R_h, ,T_h)	52	A	Ceiling, halfword FP: nearest integer .GE.(R_h)->T_h	
CALL Q8CLGV(G,X,A, , ,Z,C)	92	1	Ceiling, vector: nearest integer .GE.A->C	xxxx oooo
CALL Q8CLOCK(, ,T_f)	39	A	Transmit (real time clock)->T_{16-63},0->T_{0-15}	
CALL Q8CMPB(,X,A,Y,B)	E8	3	Compare binary,set: DFB 53 operands equal DFB 54 1st operand high DFB 55 1st operand low	
CALL Q8CMPD(,X,A,Y,B)	E9	3	Compare decimal, set: DFB 53 operands equal DFB 54 1st operand high DFB 55 1st operand low	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8CMPEQ(G,X,A,Y,B,Z)	C4	1	Vector compare, form order vector: if (A_n).OP.(B_n), set bit Z_n in order vector	xoox xooo
CALL Q8CMPGE(G,X,A,Y,B,Z)	C6	1		xoox xooo
CALL Q8CMPLT(G,X,A,Y,B,Z)	C7	1		xoox xooo
CALL Q8CMPNE(G,X,A,Y,B,Z)	C5	1		xoox xooo
CALL Q8CNTEQ(R_d,S_i,T_f)	1E	7	Count: # of leading bits equal to bit at $[R+S] \rightarrow T_{48-63}$	
CALL Q8CNTO(R_d,S_i,T_f)	1F	7	Count 1's in field R: # of 1's in field $[R+S] \rightarrow T_{48-63}$	
CALL Q8CON(R_f, ,T_h)	76	A	Contract, fullword FP: $R_{64} \rightarrow T_{32}$	
CALL Q8CONV(G,X,A, , ,Z,C)	96	1	Contract, vector: $A_{64} \rightarrow C_{32}$	oxxx oooo
CALL Q8CPSB(R_d,S_e,T_d)	14	7	Compress bit string: every R_n substring from R_n+S_n pattern$\rightarrow$T	
CALL Q8CPSV(G, ,A, , ,Z,C)	BC	2	Compress vector: vector A$\rightarrow$sparse C, controlled by OV Z	xxoo oooo
CALL Q8DBNZ(R_f,S_i,T_a)	35	7	$(R_f)-1 \rightarrow (R_f)$, if $(R_f) \neq 0$ branch to $[T_a+S_i]$	
CALL Q8DELTA(G,X,A, , ,Z,C)	D5	1	Vector delta: $(A_{n+1}-A_n) \rightarrow C_n$	xxxo oooo
CALL Q8DIVB(,X,A,Y,B,Z,C)	E3	3	Divide binary: A/B$\rightarrow$C	
CALL Q8DIVD(,X,A,Y,B,Z,C)	E7	3	Divide decimal: A/B$\rightarrow$C	
CALL Q8DIVS(R_f,S_f,T_f)	6F	4	Divide significant, fullword FP: $((R_f)/S_f))_S \rightarrow T_f$	
CALL Q8DIVSH(R_h,S_h,T_h)	4F	4	Divide significant, halfword FP: $((R_h)/(S_h))_S \rightarrow T_h$	
CALL Q8DIVSS(G,X,A,Y,B,Z,C)	AF	2	Divide significant, sparse vector: $(A/B)_S \rightarrow C$	xoox xxxx
CALL Q8DIVSV(G,X,A,Y,B,Z,C)	8F	1	Divide significant, vector: $(A/B)_S \rightarrow C$	xxxx xxxx
CALL Q8DIVU(R_f,S_f,T_f)	6C	4	Divide upper, fullword FP: $((R_f)/(S_f))_U \rightarrow T_f$	
CALL Q8DIVUH(R_h,S_h,T_h)	4C	4	Divide upper, halfword FP: $((R_h)/(S_h))_U \rightarrow T_h$	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8DIVUS(G,X,A,Y,B,Z,C)	AC	2	Divide upper, sparse vector: $(A/B)_U \rightarrow C$	xoox xxxx
CALL Q8DIVUV(G,X,A,Y,B,Z,C)	8C	1	Divide upper, vector: $(A/B)_U \rightarrow C$	xxxx xxxx
CALL Q8DOTS(G,X,A,Y,B, ,C)	DD	2	Sparse vector dot product: A•B→C,C+1	xooo xxxx
CALL Q8DOTV(G,X,A,Y,B,Z,C)	DC	1	Dot product vector: A•B→C, C+1	xxoo oooo
CALL Q8DTOB(R_f, ,T_f)	10	A	Convert packed BCD to binary T, fixed length	
CALL Q8DTOZ(G,X,A, , ,Z,C)	FC	3	Unpack BCD to zoned: A→C	xxoo oooo
CALL Q8ELEN(R_e,I16)	2A	6	Enter length: I16→R_{0-15},R_{16-63} unchanged	
CALL Q8EMARK(G,X,A,Y,B,Z,C)	EB	3	Edit and mark: A per pattern B→C, G=1st significant result address	
CALL Q8ES(R_f,I16)	3E	6	Enter short, fullword: I16→R_{16-63},RJ,SE,0→R_{0-15}	
CALL Q8ESH(R_h,I16)	4D	6	Enter short, halfword: I16→R_{8-31},RJ,SE,0→R_{0-7}	
CALL Q8EX(R_f,I48)	BE	5	Enter index, fullword: I48→R_{16-63},0→R_{0-15}	
CALL Q8EXH(R_h,I24)	CD	5	Enter index, halfword: I24→R_{8-31},0→R_{0-7}	
CALL Q8EXIT	09	4	Exit force, job mode to monitor mode	
CALL Q8EXP(R_e, ,T_f)	7A	A	Exponent, fullword: R_{0-15}→T_{16-63},SE,0→T_{0-15}	
CALL Q8EXPH(R_{eh}, ,T_h)	5A	A	Exponent, halfword: R_{0-7}→T_{8-31},SE,0→T_{0-7}	
CALL Q8EXPV(G,X,A, , ,Z,C)	9A	1	Exponent vector: A_{0-15}→C_{48-63},SE,0→C_{0-15}	xxxx oooo
CALL Q8EXTB(R_f,S_d,T_f)	6E	4	Extract bits from R_f to T_f per S_d	
CALL Q8EXTH(R_h, ,T_f)	5C	A	Extend halfword FP. R_{32}→T_{64}	
CALL Q8EXTV(G,X,A, , ,Z,C)	9C	1	Extend vector: A_{32}→C_{64}	oxxx oooo

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8EXTXH(R_h, ,T_f)	5D	A	Extend index, halfword FP: $R_{8-31} \rightarrow T_{16-63}$,SE,$R_{0-7} \rightarrow T_{0-15}$,SE	
CALL Q8FAULT(G)	06	7	Simulate fault	0000 xxxx
CALL Q8FILLC(I8,S_i,T_d)	1A	7	Fill field T with byte: repeat I8 for field [T+S]	
CALL Q8FILLR(R_f,S_i,T_d)	1B	7	Fill field T with byte: repeat (R_{56-63}) for field [T+S]	
CALL Q8FLR(R_f, ,T_f)	71	A	Floor, fullword FP: nearest integer .LE.(R_f)$\rightarrow T_f$	
CALL Q8FLRH(R_h, ,T_h)	51	A	Floor, halfword FP: nearest integer .LE. (R_h)$\rightarrow T_h$	
CALL Q8FLRV(G,X,A, , ,Z,C)	91	1	Floor, vector: nearest integer .LE.A$\rightarrow$C	xxxx 0000
CALL Q8IBNZ(R_f,S_i,T_a)	31	7	(R_f)+1$\rightarrow$(R_f), if (R_f) $\neq$0 branch to [T_a,S_i]	
			Increment and branch index:	
CALL Q8IBXEQ(G,X,A,Y,B,Z,C)	B0	C	$A_{16-63}+X_{16-63} \rightarrow C_{16-63}$,$A_{0-15} \rightarrow C_{0-15}$,	0000 xxxx
CALL Q8IBXGE(G,X,A,Y,B,Z,C)	B2	C		0000 xxxx
CALL Q8IBXGT(G,X,A,Y,B,Z,C)	B5	C	if $A_{16-63}+X_{16-63}$.OP.Z_{16-63}	0000 xxxx
CALL Q8IBXLE(G,X,A,Y,B,Z,C)	B4	C		0000 xxxx
CALL Q8IBXLT(G,X,A,Y,B,Z,C)	B3	C	then branch to Y or relative from current	0000 xxxx
CALL Q8IBXNE(G,X,A,Y,B,Z,C)	B1	C	location	0000 xxxx
CALL Q8IDLE	00	4	Idle: enable external interrupts and idle	
CALL Q8INSB(R_f,S_d,T_f)	6D	4	Insert bits from R_f to T_f per S_d	
CALL Q8INTVAL(G, ,A, , B,Z,C)	DF	1	Interval vector: A+((n-2)*B)$\rightarrow$C	xxx0 0000
CALL Q8IOR(,X,A,Y,B,Z,C)	F2	3	Logical inclusive OR: A+B$\rightarrow$C	
CALL Q8IS(R_f,I16)	3F	6	Increase short, fullword: R_{16-63}+I16$\rightarrow R_{16-63}$,R_{0-15} unchanged	
CALL Q8ISH(R_h,I16)	4E	6	Increase short, halfword: R_{8-31}+I16$\rightarrow R_{8-31}$,R_{0-7} unchanged	
CALL Q8IX(R_f,I48)	BF	5	Increase index, fullword: I48+R$\rightarrow$R	
CALL Q8IXH(R_h,I24)	CE	5	Increase index, halfword: I24+R$\rightarrow$R	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8LOD(R_a,S_i,T_f)	7E	7	Load fullword: load [R_a+S_i]→T_f	
CALL Q8LODAR	0D	4	Load associative registers: beginning at 400xx$_8$→AR	
CALL Q8LODC(R_a,S_i,T_f)	12	7	Load byte: [R_a+S_i]→T_{56-63}, 0→T_{0-55}	
CALL Q8LODH(R_a,S_i,T_h)	5E	7	Load halfword: load [R_a+S_i]→T_h	
CALL Q8LODKEY(R_f,S_a,T_a)	0F	4	Load key from (R_f), translate virtual (S_a) to absolute T_a	
CALL Q8LSDFR(R_f, ,T_f)	3B	A	Load and store data flag register: (DFR)→T_f,(R_f)→DFR	
CALL Q8LTOL(R_e, ,T_e)	38	A	Transmit length R_{0-15} to length T_{0-15}, T_{16-63} unchanged	
CALL Q8LTOR(R_e, ,T_f)	7C	A	Length to register, fullword FP: R_{0-15}→T_{48-63},0→T_{0-47}	
CALL Q8MASKB(R_d,S_d,T_d)	16	7	Mask bit strings: alternate (R_d) string and (S_d) string → T string	
CALL Q8MASKO(R_e,S_e,T_d)	1D	7	Form bit mask: repeat (R_n) ones and (S_n)-(R_n) zeros → T string	
CALL Q8MASKV(G, ,A, ,B,Z,C)	BB	2	If Z_n=1, A_n→C_n; if Z_n=0, B_n→C_n; result length→C_{0-15}	xoox xooo
CALL Q8MASKZ(R_e,S_e,T_d)	1C	7	Form mask: repeat (R_n) zeros and (S_n)-(R_n) ones→ T string	
CALL Q8MAX(G,X,A, ,B,Z,C)	D8	1	Vector maximum: A_{max}→C, item count→B	xxoo oxoo
CALL Q8MCMPC(G,X,A,Y,B,Z,C)	FD	3	Find A_n=B_n per mask C, A and B index incremented by # of bytes	xxoo oxxo
CALL Q8MIN(G,X,A, ,B,Z,C)	D9	1	Vector minimum: A_{min}→C, item count →B	xxoo oxoo
CALL Q8MMRGC(I8,X,A,Y,B,Z,C)	EA	3	Merge bits per byte mask: A or B per I8=0 or 1→C	xxxx xxxx
CALL Q8MOVL(G,X,A, ,B,Z,C)	F8	3	Move bytes left: A→C (left to right)	xxxx oxox

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8MOVLC(G,X,A, ,B,Z,C)	F9	3	Move bytes left, ones complement: A→C (left to right)	xxxx oxox
CALL Q8MOVR(R_i,S_i,T_d)	18	7	Move bytes right: $(T_d)+(R_i) \to (T_d)+(R_i)+(S_i)$	
CALL Q8MOVS(,X,A, ,B,Z,C)	FA	3	Move and scale: A→C, scale (B) decimal places	
CALL Q8MPYB(,X,A,Y,B,Z,C)	E2	3	Multiply binary: A*B→C	
CALL Q8MPYD(,X,A,Y,B,Z,C)	E6	3	Multiply decimal: A*B→C	
CALL Q8MPYL(R_f,S_f,T_f)	69	4	Multiply lower, fullword FP: $((R_f)*(S_f))_L \to T_f$	
CALL Q8MPYLH(R_h,S_h,T_h)	49	4	Multiply lower, halfword FP: $((R_h)*(S_h))_L \to T_h$	
CALL Q8MPYLS(G,X,A,Y,B,Z,C)	A9	2	Multiply lower, sparse vector: $(A*B)_L \to C$	xoox xxxx
CALL Q8MPYLV(G,X,A,Y,B,Z,C)	89	1	Multiply lower, vector: $(A*B)_L \to C$	xxxx xxxx
CALL Q8MPYS(R_f,S_f,T_f)	6B	4	Multiply significant, fullword FP: $((R_f)*(S_f))_S \to T_f$	
CALL Q8MPYSH(R_h,S_h,T_h)	4B	4	Multiply significant, halfword FP: $((R_h)*(S_h))_S \to T_h$	
CALL Q8MPYSS(G,X,A,Y,B,Z,C)	AB	2	Multiply significant, sparse vector: $(A*B)_S \to C$	xoox xxxx
CALL Q8MPYSV(G,X,A,Y,B,Z,C)	8B	1	Multiply significant, vector: $(A*B)_S \to C$	xxxx xxxx
CALL Q8MPYU(R_f,S_f,T_f)	68	4	Multiply upper, fullword FP: $((R_f)*(S_f))_U \to T_f$	
CALL Q8MPYUH(R_h,S_h,T_h)	48	4	Multiply upper, halfword FP: $((R_h)*(S_h))_U \to T_h$	
CALL Q8MPYUS(G,X,A,Y,B,Z,C)	A8	2	Multiply upper, sparse vector: $(A*B)_U \to C$	xoox xxxx
CALL Q8MPYUV(G,X,A,Y,B,Z,C)	88	1	Multiply upper, vector: $(A*B)_U \to C$	xxxx xxxx
CALL Q8MPYX(R_f,S_f,T_f)	3D	4	Multiply index, fullword: $R_{16-63}*S_{16-63} \to T_{16-63}, 0 \to T_{0-15}$	

REPRODUCIBILITY OF THE
ORIG

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G-Bits
CALL Q8MPYXH(R_h,S_h,T_h)	3C	4	Multiply index, halfword: $R_{8-31}*S_{8-31} \rightarrow T_{8-31}, 0 \rightarrow T_{0-7}$	
CALL Q8MRGB(R_d,S_d,T_d)	15	7	Merge bit strings: interleave (R_d) string with (S_d) string $\rightarrow T_d$ string	
CALL Q8MRGC(R_d,S_d,T_d)	17	7	Merge byte strings: (R_d):(S_d),lesser $\rightarrow T_d$	
CALL Q8MRGV(G, ,A, ,B,Z,C)	BD	2	Merge vector: if Z_n=1, $A_n \rightarrow C_n$; if Z_n=0, $B_n \rightarrow C_n$; result length $\rightarrow C_{0-15}$	xoox xoox
CALL Q8MTIME(R_f)	0A	4	Transmit (R_f) $\rightarrow$ monitor interval timer	
CALL Q8NAND(,X,A,Y,B,Z,C)	F3	3	Logical NAND: $\overline{A \bullet B} \rightarrow C$	
CALL Q8NOR(,X,A,Y,B,Z,C)	F4	3	Logical NOR: $\overline{A+B} \rightarrow C$	
CALL Q8ORN(,X,A,Y,B,Z,C)	F5	3	Logical OR NOT: $A+\overline{B} \rightarrow C$	
CALL Q8PACK(R_f,S_f,T_f)	7B	4	Pack, fullword FP: R_{48-63} and $S_{16-63} \rightarrow T_f$	
CALL Q8PACKH(R_h,S_h,T_h)	5B	4	Pack, halfword FP: R_{24-31} and $S_{8-31} \rightarrow T_h$	
CALL Q8PACKV(G,X,A,Y,B,Z,C)	9B	1	Pack, vector: A_{48-63} and $B_{16-63} \rightarrow C$	xxxx xooo
CALL Q8POLYEV(G,X,A,Y,B,Z,C)	DE	1	Polynomial evaluation: A_n per $B \rightarrow C_n$	xxxx oooo
CALL Q8PRODCT(G,X,A, , ,Z,C)	DB	1	Vector product: Product(A_0,A_1,...A_n)$\rightarrow$C	xxoo oooo
CALL Q8RAND(R_f,S_f,T_f)	2D	4	Logical AND: R,S$\rightarrow$T	
CALL Q8RCON(R_f, ,T_h)	77	A	Rounded contract, fullword FP: $R_{64} \rightarrow T_{32}$	
CALL Q8RCONV(G,X,A, , ,Z,C)	97	1	Rounded contract, vector: A_{64} rounded $\rightarrow$32	oxxx oooo
CALL Q8RIOR(R_f,S_f,T_f)	2E	4	Logical inclusive OR: R,S$\rightarrow$T	
CALL Q8RJTIME(, ,T_f)	37	A	Read job interval-timer to (T_f)	
CALL Q8RTOR(R_f, ,T_f)	78	A	Register to register fullword transmit: (R_f)$\rightarrow T_f$	
CALL Q8RTORH(R_h, ,T_h)	58	A	Register to register halfword transmit: (R_h)$\rightarrow T_h$	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8RXOR(R_f,S_f,T_f)	2C	4	Logical exclusive OR: R,S→T	
CALL Q8SCNLEQ(I8,S_i,T_d)	28	7	Scan left to right from [T_d,S_i] for byte equal to I8, index S_i	
CALL Q8SCNLNE(I8,S_i,T_d)	29	7	Scan left to right from [T_d,S_i] for byte not equal to I8, index S_i	
CALL Q8SCNRNE (I8,S_i,T_d)	19	7	Scan right to left from [T_d,S_i] for byte not equal to I8, decrement S_i	
CALL Q8SELEQ(G,X,A,Y,B,Z,C)	C0	1	Vector select: if A_n.OP.B_n, then count up to the condition met →C	xxox xooo
CALL Q8SELGE(G,X,A,Y,B,Z,C)	C2	1		xxox xooo
CALL Q8SELLT(G,X,A,Y,B,Z,C)	C3	1		xxox xooo
CALL Q8SELNE(G,X,A,Y,B,Z,C)	C1	1		xxox xooo
CALL Q8SETCF(R_f)	08	4	Input/output: set channel (R_f) channel flag	
CALL Q8SHIFT (R_f,S_f,T_f)	34	4	Shift R_f by (S_f) →T_f	
CALL Q8SHIFTI(R_f,I8,T_f)	30	7	Shift R_f by I8→T_f	
CALL Q8SKEYB(G,X,A,Y,B,Z,C)	D6	3	Search A for B per C, A_{index}= #no match (bits)	
CALL Q8SKEYC(G,X,A,Y,B,Z,C)	FE	3	Search A for B per C, A_{index}= #no match (bytes)	
CALL Q8SKEYW(G,X,A,Y,B,Z,C)	FF	3	Search A for B per C, A_{index}= #no match (words)	
CALL Q8SQRT(R_f, ,T_f)	73	A	Significant square root, fullword FP: $(SQRT(R_f))_S$→T_f	
CALL Q8SQRTH(R_h, ,T_h)	53	A	Significant square root, halfword FP: $(SQRT(R_h))_S$→T_h	
CALL Q8SQRTV(G,X,A, , ,Z,C)	93	1	Significant square root, vector: $SQRT(A)_S$→C	xxxx oxxo
CALL Q8SRCHEQ(G, ,A, ,B,Z,C)	C8	1	Vector search from indexed list: each (A_n).OP.(B_n), count →C_n	xxxo oooo
CALL Q8SRCHGE(G, ,A, ,B,Z,C)	CA	1		xxxo oooo
CALL Q8SRCHLT(G, ,A, ,B,Z,C)	CB	1		xxxo oooo
CALL Q8SRCHNE(G, ,A, ,B,Z,C)	C9	1		xxxo oooo
CALL Q8STO(R_a,S_i,T_f)	7F	7	Store, fullword: store (T_f)→address [R_a+S_i]	

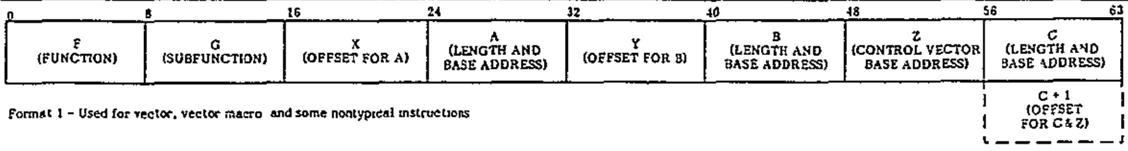
REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

TABLE D-2. SPECIAL CALL FORMATS (Contd)

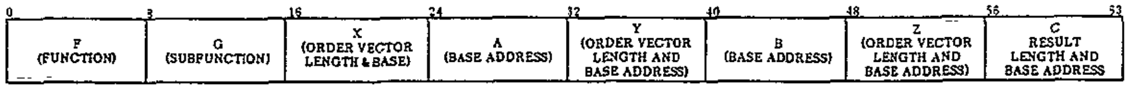
Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8STOAR	0C	4	Store associative registers: AR$\rightarrow$400xx$_8$ and higher addresses	
CALL Q8STOC(R_a,S_i,T_f)	13	7	Store byte (character): $T_{56-63}\rightarrow$address $[R_a+S_i]$	
CALL Q8STOH(R_a,S_i,T_h)	5F	7	Store, halfword: $(T_h)\rightarrow$address $[R_a+S_i]$	
CALL Q8SUBB(,X,A,Y,B,Z,C)	E1	3	Subtract binary: A-B$\rightarrow$C	
CALL Q8SUBD(,X,A,Y,B,Z,C)	E5	3	Subtract decimal: A-B$\rightarrow$C	
CALL Q8SUBL(R_f,S_f,T_f)	65	4	Subtract lower, fullword FP: $((R_f)-(S_f))_L\rightarrow T_f$	
CALL Q8SUBLH(R_h,S_h,T_h)	45	4	Subtract lower, halfword FP: $((R_h)-(S_h))_L\rightarrow T_f$	
CALL Q8SUBLS(G,X,A,Y,B,Z,C)	A5	2	Subtract lower, sparse vector: $(A-B)_L\rightarrow C$	xoox xxxx
CALL Q8SUBLV(G,X,A,Y,B,Z,C)	85	1	Subtract lower, vector: $(A-B)_L\rightarrow C$	xxxx xxxx
CALL Q8SUBMOD(I8,X,A,Y,B,Z,C)	ED	3	Modulo subtract bytes: (A_n-B_n)mod(I8)$\rightarrow C_n$	
CALL Q8SUBN(R_f,S_f,T_f)	66	4	Subtract normalized, fullword FP: $((R_f)-(S_f))_N\rightarrow T_f$	
CALL Q8SUBNH(R_h,S_h,T_h)	46	4	Subtract normalized, halfword FP: $((R_h)-(S_h))_N\rightarrow T_f$	
CALL Q8SUBNS(G,X,A,Y,B,Z,C)	A6	2	Subtract normalized, sparse vector: $(A-B)_N\rightarrow C$	xoox xxxx
CALL Q8SUBNV(G,X,A,Y,B,Z,C)	86	1	Subtract normalized, vector: $(A-B)_N\rightarrow C$	xxxx xxxx
CALL Q8SUBU(R_f,S_f,T_f)	64	4	Subtract upper, fullword FP: $((R_f)-(S_f))_U\rightarrow T_f$	
CALL Q8SUBUH(R_h,S_h,T_h)	44	4	Subtract upper, halfword FP: $((R_h)-(S_h))_U\rightarrow T_h$	
CALL Q8SUBUS(G,X,A,Y,B,Z,C)	A4	2	Subtract upper, sparse vector: $(A-B)_U\rightarrow C$	xoox xxxx
CALL Q8SUBUV(G,X,A,Y,B,Z,C)	84	1	Subtract upper, vector: $(A-B)_U\rightarrow C$	xxxx xxxx
CALL Q8SUBX(R_f,S_f,T_f)	67	4	Subtract index: $R_{16-63}-S_{16-63}\rightarrow T_{16-63}$,$R_{0-15}\rightarrow T_{0-15}$	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

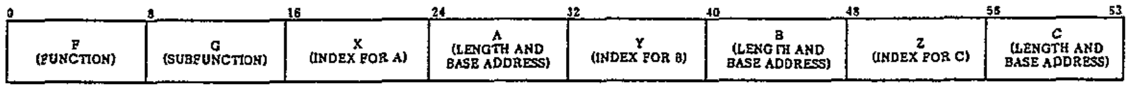
Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8SUBXV(G,X,A,Y,B,Z,C)	87	1	Subtract index, vector: $A_{16-63}-B_{16-63} \rightarrow C_{16-63}, A_{0-15} \rightarrow C_{0-15}$	oxxx xooo
CALL Q8SUM(G,X,A, , ,Z,C)	DA	1	Vector sum: $Sum(A_0,A_1,...A_n) \rightarrow C,C+1$	xxoo oooo
CALL Q8SWAP(R_d,S_f,T_d)	7D	7	Swap registers: start with S_f, storing at T_d and loading from R_d	
CALL Q8TL(G,X,A,Y,B,Z,C)	EE	3	Translate bytes: $B_n \rightarrow C_n$	xxxx oxox
CALL Q8TLMARK(G,X,A,Y,B,Z,C)	D7	3	Translate and mark: A per B→ vector C	xxoo xxoo
CALL Q8TLTEST(G,X,A,Y,B,Z,C)	EF	3	Translate and test: $B_n \rightarrow C, A_n \rightarrow Z$ if B_n.NE.O	xxoo oxoo
CALL Q8TLXI(R_a,S_i,T_f)	0E	4	Translate external interrupt: (T_f)=priority, branch to $R_a[S_i]$	
CALL Q8TPMOV(G,X,A,Y,B,Z,C)	B9	1	Transpose and move 8 by 8 matrix	xoxx xooo
CALL Q8TRU(R_f, ,T_f)	70	A	Truncate, fullword FP: nearest integer .LE.$(R_f) \rightarrow T_f$	
CALL Q8TRUH(R_h, ,T_h)	50	A	Truncate, halfword FP: nearest integer .LE.$(R_h) \rightarrow T_h$	
CALL Q8TRUV(G,X,A, , ,Z,C)	90	1	Truncate, vector: nearest integer .LE.(A)→C	xxxx oooo
CALL Q8VREVV(G,X,A, , ,Z,C)	B8	1	Transmit vector reversed to vector: $A_{rev} \rightarrow C$	xxxo oooo
CALL Q8VTOV(G,X,A, , ,Z,C)	98	1	Vector to vector transmit: A→C	xxxx oooo
CALL Q8VTOVX(G, ,A, ,B, ,C)	B7	1	Vector to vector indexed transmit: B→C indexed by A	xooo xoxx
CALL Q8VXTOV(G, ,A, ,B, ,C)	BA	1	Vector to vector indexed transmit: B indexed by A→C	xooo ooxx
CALL Q8WJTIME(R_f)	3A	A	Transmit $(R_f) \rightarrow$ job interval timer	
CALL Q8XOR(,X,A,Y,B,Z,C)	F0	3	Logical exclusive OR: A-B→C	
CALL Q8XORN(,X,A,Y,B,Z,C)	F7	3	Logical equivalence (exclusive OR NOT): $A-\overline{B} \rightarrow C$	
CALL Q8ZTOD(G,X,A, , ,Z,C)	FB	3	Pack zoned to BCD: A→C	xxoo oooo


0 8 16 24 32 40 48 56 63
F (FUNCTION)
G (SUBFUNCTION)
X (OFFSET FOR A)
A (LENGTH AND BASE ADDRESS)
Y (OFFSET FOR B)
B (LENGTH AND BASE ADDRESS)
Z (CONTROL VECTOR BASE ADDRESS)
C (LENGTH AND BASE ADDRESS)
C + 1 (OFFSET FOR C & Z)

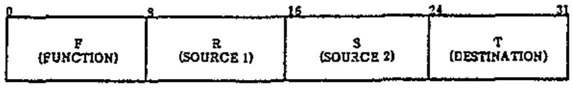
Format 1 - Used for vector, vector macro and some nontypical instructions


0 8 16 24 32 40 48 56 63
F (FUNCTION)
G (SUBFUNCTION)
X (ORDER VECTOR LENGTH & BASE)
A (BASE ADDRESS)
Y (ORDER VECTOR LENGTH AND BASE ADDRESS)
B (BASE ADDRESS)
Z (ORDER VECTOR LENGTH AND BASE ADDRESS)
C RESULT LENGTH AND BASE ADDRESS

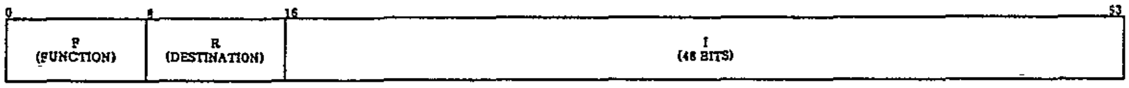
Format 2 - Used for sparse vector and some nontypical instructions.



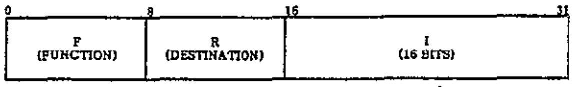
Format 3 - Used for the logical string and string instructions



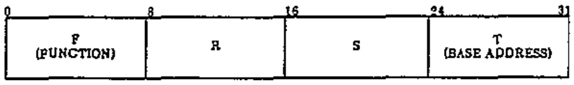
Format 4 - Used for some register instructions, for all monitor instructions, and for the 3D and 04 nontypical instructions



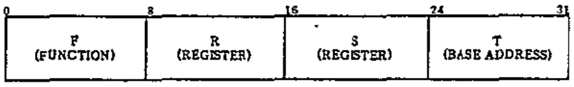
Format 5 - Used for the BE, BF, CD and CE index instructions, and for the B6 branch instruction



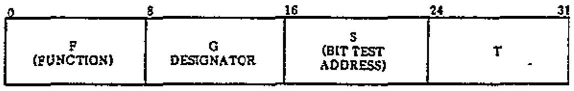
Format 6 - Used for the 3E, 3F, 4D and 4E index instructions, and for the 2A register instruction



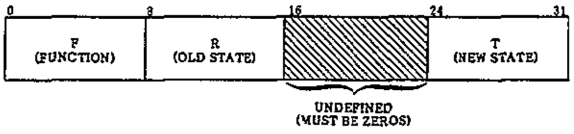
Format 7 - Used for some branch and nontypical instructions



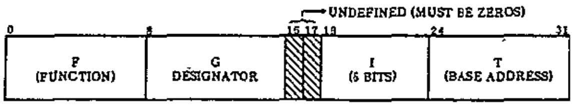
Format 8 - Used for some branch instructions



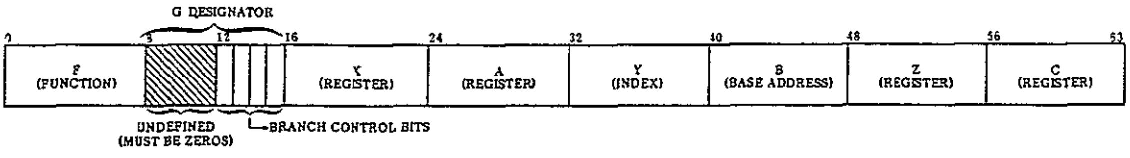
Format 9 - Used for the 32 branch instruction



Format A - Used for some index, branch, and register instructions



Format B - Used for the 33 branch instruction



Format C - Used for the 80-B5 branch instructions

Figure D-1. Instruction Formats

As a convenience for the user of special calls, the special calls are listed by op code in table D-3.

TABLE D-3. SPECIAL CALLS LISTED BY OP CODE

Op code	Special Call	Op code	Special Call	Op code	Special Call	Op code	Special Call	Op code	Special Call
00	Q8IDLE	34	Q8SHIFT	67	Q8SUBX	99	Q8ABSV	D0	Q8AVG
04	Q8BKPT	35	Q8DBNZ	68	Q8MPYU	9A	Q8EXPV	D1	Q8ADJM
06	Q8FAULT	36	Q8BSAVE	69	Q8MPYL	9B	Q8PACKV	D4	Q8AVGD
08	Q8SETCF	37	Q8RJTIME	6B	Q8MPYS	9C	Q8EXTV	D5	Q8DELTA
09	Q8EXIT	38	Q8LTOL	6C	Q8DIVU	A0	Q8ADDUS	D6	Q8SKEYB
0A	Q8MTIME	39	Q8CLOCK	6D	Q8INSB	A1	Q8ADDLS	D7	Q8TLMARK
0C	Q8STOAR	3A	Q8WJTIME	6E	Q8EXTB	A2	Q8ADDNS	D8	Q8MAX
0D	Q8LODAR	3B	Q8LSDFR	6F	Q8DIVS	A4	Q8SUBUS	D9	Q8MIN
0E	Q8TLXI	3C	Q8MPYXH	70	Q8TRU	A5	Q8SUBLS	DA	Q8SUM
0F	Q8LODKEY	3D	Q8MPYX	71	Q8FLR	A6	Q8SUBNS	DB	Q8PRODCT
10	Q8DTOB	3E	Q8ES	72	Q8CLG	A8	Q8MPYUS	DC	Q8DOTV
11	Q8BTOD	3F	Q8IS	73	Q8SQRT	A9	Q8MPYLS	DD	Q8DOTS
12	Q8LODC	40	Q8ADDUH	74	Q8ADJS	AB	Q8MPYSS	DE	Q8POLYEV
13	Q8STOC	41	Q8ADDLH	75	Q8ADJE	AC	Q8DIVUS	DF	Q8INTVAL
14	Q8CPSB	42	Q8ADDNH	76	Q8CON	AF	Q8DIVSS	E0	Q8ADDB
15	Q8MRGB	44	Q8SUBUH	77	Q8RCON	B0	Q8IBXEQ	E1	Q8SUBB
16	Q8MASKB	45	Q8SUBLH	78	Q8RTOR	B1	Q8IBXNE	E2	Q8MPYB
17	Q8MRGC	46	Q8SUBNH	79	Q8ABS	B2	Q8IBXGE	E3	Q8DIVB
18	Q8MOVR	48	Q8MPYUH	7A	Q8EXP	B3	Q8IBXLT	E4	Q8ADDD
19	Q8SCNRNE	49	Q8MPYLH	7B	Q8PACK	B4	Q8IBXLE	E5	Q8SUBD
1A	Q8FILLC	4B	Q8MPYSH	7C	Q8LTOR	B5	Q8IBXGT	E6	Q8MPYD
1B	Q8FILLR	4C	Q8DIVUH	7D	Q8SWAP	B6	Q8BIM	E7	Q8DIVD
1C	Q8MASKZ	4D	Q8ESH	7E	Q8LOD	B7	Q8VTOVX	E8	Q8CMPB
1D	Q8MASKO	4E	Q8ISH	7F	Q8STO	B8	Q8VREVV	E9	Q8CMPD
1E	Q8CNTEQ	4F	Q8DIVSH	80	Q8ADDUV	B9	Q8TPMOV	EA	Q8MMRGC
1F	Q8CNTO	50	Q8TRUH	81	Q8ADDLV	BA	Q8VXTOV	EB	Q8EMARK
20	Q8BHEQ	51	Q8FLRH	82	Q8ADDNV	BB	Q8MASKV	EC	Q8ADDMOD
21	Q8BHNE	52	Q8CLGH	83	Q8ADDXV	BC	Q8CPSV	ED	Q8SUBMOD
22	Q8BHGE	53	Q8SQRTH	84	Q8SUBUV	BD	Q8MRGV	EE	Q8TL
23	Q8BHLT	54	Q8ADJSH	85	Q8SUBLV	BE	Q8EX	EF	Q8TLTEST
24	Q8BEQ	55	Q8ADJEH	86	Q8SUBNV	BF	Q8IX	F0	Q8XOR
25	Q8BNE	58	Q8RTORH	87	Q8SUBXV	C0	Q8SELEQ	F1	Q8AND
26	Q8BGE	59	Q8ABSH	88	Q8MPYUV	C1	Q8SELNE	F2	Q8IOR
27	Q8BLT	5A	Q8EXPH	89	Q8MPYLV	C2	Q8SELGE	F3	Q8NAND
28	Q8SCNLEQ	5B	Q8PACKH	8B	Q8MPYSV	C3	Q8SELLT	F4	Q8NOR
29	Q8SCNLNE	5C	Q8EXTH	8C	Q8DIVUV	C4	Q8CMPEQ	F5	Q8ORN
2A	Q8ELEN	5D	Q8EXTXH	8F	Q8DIVSV	C5	Q8CMPNE	F6	Q8ANDN
2B	Q8ADDLEN	5E	Q8LODH	90	Q8TRUV	C6	Q8CMPGE	F7	Q8XORN
2C	Q8RXOR	5F	Q8STOH	91	Q8FLRV	C7	Q8CMPLT	F8	Q8MOVL
2D	Q8RAND	60	Q8ADDU	92	Q8CLGV	C8	Q8SRCHEQ	F9	Q8MOVLC
2E	Q8RIOR	61	Q8ADDL	93	Q8SQRTV	C9	Q8SRCHNE	FA	Q8MOVS
2F	Q8BARB	62	Q8ADDN	94	Q8ADJSV	CA	Q8SRCHGE	FB	Q8ZTOD
30	Q8SHIFTI	63	Q8ADDX	95	Q8ADJEV	CB	Q8SRCHLT	FC	Q8DTOZ
31	Q8IBNZ	64	Q8SUBU	96	Q8CONV	CD	Q8EXH	FD	Q8MCMPC
32	Q8BAB	65	Q8SUBL	97	Q8RCONV	CE	Q8IXH	FE	Q8SKEYC
33	Q8BADF	66	Q8SUBN	98	Q8VTOV	CF	Q8ACPS	FF	Q8SKEYW

Appendix F

Replaced with the following page

Appendix F

STAR FORTRAN '77

STATEMENT LIST

The following statement lis is intended only to suggest the scope of the STAR dialect of FORTRAN '77. See the body of the manual for details concerning the correct construction and use of the various statements.

Statement function definition statement

Assignment statements:

- arithmetic scalar – arithmetic expression
- character entity – character expression
- logical entity – logical expression
- array or dynamic variable – arithmetic expression or vector arithmetic expression
- bit scalar or bit vector – bit scalar or bit vector

Keyword statements:

ASSIGN (descriptor)	ENDFILE	PAUSE
ASSIGN (statement label)	END IF	PRINT
BACKSPACE	ENTRY	PROGRAM
BIT	EQUIVALENCE	PUNCH
BLOCK DATA	EXTERNAL	READ
BUFFER IN	FORMAT	REAL
BUFFER OUT	FREE	RETURN
CALL	FUNCTION	REWIND
CHARACTER	GO TO (assigned)	ROWWISE
CLOSE	GO TO (computed)	SAVE
COMMON	GO TO (simple)	STOP
COMPLEX	HALF PRECISION	SUBROUTINE
CONTINUE	IF (arithmetic)	WRITE
DATA	IF (block)	
DECODE	IF (logical)	
DIMENSION	IMPLICIT	
DO	INQUIRE	
DOUBLE PRECISION	INTEGER	
DYNAMIC	INTRINSIC	
ELSE	LOGICAL	
ELSE IF	NAMELIST	
ENCODE	OPEN	
END	PARAMETER	
ENF		

COMPATIBILITY FEATURES

Certain features of STAR FORTRAN are provided only for compatibility with FORTRAN Extended. The compatibility features are described in this appendix.

NOTE

> The features described in this appendix should not be used for new programs and are intended only for the conversion of existing programs.

HOLLERITH CONSTANT COMPATIBILITY

Hollerith elements are described in section 2, Statement Elements. For compatibility, Hollerith constants are supported in relational and arithmetic expressions.

A A Hollerith constant used in an arithmetic or relational expression is limited to 8 characters. A Hollerith constant is left-justified with blank fill in a full word. A Hollerith constant that is too long is truncated on the right hand side, and a warning diagnostic is issued.

The Hollerith constant is considered typeless. A typeless constant is not converted for use as an argument or for assignment. If Hollerith constants are the only operands in an arithmetic expression, the result is type integer.

BUFFER IN AND BUFFER OUT COMPATIBILITY

Input, output, and memory transfer statements are described in section 8. The BUFFER IN and BUFFER OUT statements are provided for compatibility with FORTRAN Extended. The UNIT and LENGTH functions are also provided for compatibility.

B The BUFFER IN and BUFFER OUT statements are used to transmit binary data between SRM-structured files and main memory. The length of the buffer area in which the data is contained should be an even number of bytes for tape files, or a multiple of pages for disk files. Ordering the data in this manner provides for the most economical use of storage.

C A file referenced in a BUFFER statement must be declared in the PROGRAM statement to be an explicit file. The file cannot be referenced in any other input or output statement; however, it can be referenced in the unit positioning statements BACKSPACE, REWIND, and ENDFILE. Once buffered input/output is established for a logical unit in a FORTRAN program, all input and output for that unit must be buffered.

After a BUFFER IN or BUFFER OUT, the error status of the logical unit involved should be checked using the UNIT function before another operation with the unit is initiated. The unit status should also be checked before the buffered data is used. After the unit check, the number of bytes read by a BUFFER IN can be obtained with the LENGTH function.

BUFFER IN STATEMENT

Execution of the BUFFER IN statement causes transfer of data from the logical unit specified, in the mode given, to the buffer defined in this statement as storage locations first to last. Only one record is read for each BUFFER IN statement.

Form:

```
BUFFER IN(u,mode)(first,last)
```

u	The logical unit number.
mode	An integer constant or simple integer variable that specifies the recording mode of the data being read. The permitted values are: 0 = 7-track tape, BCD mode, even parity 1 = 7-track or 9-track tape, binary mode, odd parity 2 = 7-track tape, CDC 64-character ASCII subset, odd parity 4 = Disk
first	A variable or array element name that can be type character, integer, real, double precision, complex, or logical, and which defines the first location in the buffer into which data is to be transmitted.
last	A variable or array element name that can be type character, integer, real, double precision, complex, or logical, and which defines the location in the buffer into which the last data item is to be transmitted.

D

The location of last cannot precede first in memory. The quantity (last-first+1) must be less than or equal to 24 small pages.

BUFFER OUT STATEMENT

The execution of the BUFFER OUT statement transfers data to the logical unit specified in the mode given, from the buffer defined in this statement as storage locations first to last.

Form:

```
BUFFER OUT(u,mode)(first,last)
```

u	The logical unit number.
mode	An integer constant or simple integer variable that specifies the mode in which the data record is to be written: 0 = 7-track tape, BCD mode, even parity 1 = 7-track or 9-track tape, binary mode, odd parity

E

A: A Hollerith constant used in an arithmetic or relational expression is limited to 8 characters. An H-constant is left justified with blank fill in a full word. An R-constant is right justified with zero fill in a full word. An H-constant that is too long is truncated on the right hand side and a warning diagnostic is issued. An R-constant that is too long is truncated on the left hand side and a warning diagnostic is issued.

C: A file referenced in a BUFFER I/O statement must be preconnected or connected for sequential access. The specified unit must not be referenced in any other data transfer input/output statement while connected. However, the unit may be closed and opened again. The unit may be referenced in the file positioning statements BACKSPACE, ENDFILE, and REWIND. The unit may also be referenced in an INQUIRE by unit statement and the file in an INQUIRE by file statement.

B: . . . transmit binary data between files connected for sequential access and internal storage. The length of the buffer area in which the data is . . .

D:

u	An external unit identifier.
mode	Ignored.

E:

u	An external unit identifier.
mode	Ignored.

A 2 = 7-track tape, CDC 64-character ASCII subset, odd parity

4 = Disk

first	A variable or array element name that can be type character, real, integer, double precision, complex, or logical, and which defines the first location in the buffer from which data is to be transmitted.
last	A variable or array element name that can be type character, real, integer, double precision, complex, or logical, and which defines the location in the buffer from which the last data item is to be transmitted.

One logical record is written for each BUFFER OUT statement. The parameters first and last must refer to the same array, and last cannot precede first in memory.

UNIT FUNCTION

The UNIT function checks to see whether or not data transmission was completed without error. After a BUFFER IN or BUFFER OUT, the UNIT should be referenced before any further operations are performed on the file.

The UNIT function is suitable for evaluation in an arithmetic IF statement that causes branching to appropriate statements, as directed by the value returned.

Form:

UNIT(u)

The function returns one of the following real values:

-1.0 =	Unit ready
0.0 =	Unit ready; end-of-file encountered
1.0 =	Unit ready; parity error encountered

B

LENGTH FUNCTION

The length of the physical record read from the logical unit by the previous BUFFER IN statement can be determined by the LENGTH function.

Form:

LENGTH(u)

C u The logical unit number.

The function returns an integer value that represents the number of bytes actually read. If the buffer area is larger than the physical record, the excess buffer space is undefined. If the physical record is larger than the buffer, the remainder of the record is lost.

* SPECIFICATION COMPATIBILITY

Input/output lists and data formatting is described in section 9. For compatibility with FORTRAN Extended, the * specification is supported; the * specification is identical to the ' specification, except that asterisks replace the apostrophes.

SUPPLIED FUNCTION COMPATIBILITY

Supplied functions are described in section 15, STAR FORTRAN-Supplied Functions. For compatibility, a number of additional functions are supplied. The functions are shown in table G-1.

TABLE G-1. FUNCTIONS SUPPLIED FOR COMPATIBILITY

Function	Function Reference	Type of	
		Arguments	Result
Masking Functions	MASK(n)	Integer	Typeless
	SHIFT(a,n)	Real or Integer	Typeless
	COMPL(a)	Real or Integer	Typeless
	AND(a_1,a_2, . . .)	Real or Integer	Typeless
	OR(a_1,a_2, . . .)	Real or Integer	Typeless
	XOR(a_1,a_2, . . .)	Real or Integer	Typeless

D

A typeless function generates a result that is typeless. A typeless result is not converted for use as an argument or for assignment. For example, the statement

```
X = Y + SHIFT(I,5)
```

does not involve conversion of the SHIFT result from integer to real. The result is typeless and is used without conversion.

AND (a_1,a_2, . . .)

This computes the bit-by-bit logical product of a_1 through a_n.

COMPL (a)

This computes the bit-by-bit Boolean complement of a.

MASK (n)

This forms a mask of n bits set to 1 starting at the left of the word. The n value must be in the range $0 \le n \le 64$. The result is undefined for an argument outside the range.

OR (a_1,a_2, . . .)

This computes the bit-by-bit logical OR of a_1 through a_n.

A: delete

B: Note that the significance of the signs of the values returned by the UNIT function is different from that for the input/output status specifiers described in Chapter 8.

C: u An external unit identifier.

D:	Type Conversion	DFLOAT(i)	Integer	Double Precision

SHIFT (a,n)

This produces a shift of n bit positions in a. If n is positive, the shift is left circular. If n is negative, the shift is right end-off with sign extension from bit zero. The n value must be in the range $-64<n<64$. The result is undefined if n is outside the range. The n value is integer.

XOR ($a_1,a_2,\ldots$)

This computes the bit-by-bit exclusive OR of a_1 through a_n.

The supplied function list in appendix E indicates the type of code generated by the function and the fast call name, if any. The information about functions described in this appendix is shown in table G-2.

TABLE G-2. COMPATIBILITY FUNCTIONS LIST

Function	Category	Fast Call Name
AND	N	–
COMPL	N	–
MASK	N	–
OR	N	–
SHIFT	N	–
XOR	N	–

N = In-line
X = External
NX = In-line and external

A

A: DFLOAT(i)

This function converts an integer number to a double precision number. The result is accurate to 94 bits.